1st Summit on Advances in Programming Languages

SNAPL'15, May 3–6, 2015, Asilomar, California, US

Edited by

Thomas Ball
Rastislav Bodík
Shriram Krishnamurthi
Benjamin S. Lerner
Greg Morrisett

LIPICS

Editors

Thomas Ball
Microsoft Research
United States of America
tball@microsoft.com

Rastislav Bodík
University of California Berkeley
United States of America
bodik@cs.berkeley.edu

Shriram Krishnamurthi
Brown University
United States of America
sk@cs.brown.edu

Benjamin S. Lerner
Northeastern University
United States of America
blerner@ccs.neu.edu

Greg Morrisett
Harvard University
United States of America
greg@eecs.harvard.edu

ACM Classification 1998
D.3 Programming Languages

ISBN 978-3-939897-80-4

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-80-4.

Publication date
May, 2015

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.SNAPL.2015.i

LIPIcs – Leibniz International Proceedings in Informatics

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Preface

SNAPL is a new venue for the programming languages community. The goal of SNAPL is to complement existing conferences by discussing big-picture questions. We hope that the inaugural Summit oN Advances in Programming Languages (SNAPL) will grow into a meeting where our community comes to enjoy talks with inspiring ideas, fresh insights, and lots of discussion. Open to perspectives from both industry and academia, SNAPL values innovation, experience-based insight, and vision. Not affiliated with any other organization, SNAPL is organized by the PL community for the PL community. We plan to hold SNAPL every two years in early May, at Asilomar, California.

SNAPL seeks to draw on the best elements of many successful meeting formats, including the database community's CIDR conference; the various Hot* conferences in systems; the practitioner-leaning Strange Loop; Seminars hosted at Dagstuhl; Working Groups run by IFIP; and *PLS regional programming language events. SNAPL will certainly develop its own unique character over time.

Authors were asked to submit a five-page paper on a programming languages topic. Appropriate topics included the following:

- a visionary idea requiring years of exploration and evaluation,
- progress on an ongoing, long-term research program,
- lessons from a completed project, including design mistakes,
- well-argued challenges to accepted ideas and methods,
- an unexpected connection between two areas of programming languages,
- a new foundation for a well-explored area of programming languages, and
- a new line of research that exploits results from other areas of Computer Science or other disciplines.

This list was not meant to be exclusive. A good SNAPL paper may resemble a grant proposal, mini-keynote, or other format that would not find a home at traditional conferences. A good submission should also convince the Program Committee that the talk will lead to a stimulating, thoughtful, and perhaps (gently) provocative discussion.

List of Authors

Umut Acar
Carnegie Mellon University
United States of America
umut@cs.cmu.edu

Amal Ahmed
Northeastern Universtiy
United States of America
amal@ccs.neu.edu

Nada Amin
EPFL
Switzerland
nada.amin@epfl.ch

Katerina Argyraki
EPFL
Switzerland
katerina.argyraki@epfl.ch

Joshua Auerbach
IBM Research
United States of America

David F. Bacon
IBM Research
United States of America

Eli Barzilay
PLT Design Inc.
United States of America

Pavol Bielik
ETH Zurich
Switzerland
pavol.bielik@inf.ethz.ch

Stephen M. Blackburn
Australian National University
Australia
steve.blackburn@anu.edu.au

Guy Blelloch
Carnegie Mellon University
United States of America
blelloch@cs.cmu.edu

James Bornholt
University of Washington
United States of America
bornholt@cs.washington.edu

John Boyland
University of Wisconsin
United States of America
boyland@uwm.edu

Kevin Brown
Stanford University
United States of America
kjbrown@stanford.edu

Luis Ceze
University of Washington
United States of America
luisceze@cs.washington.edu

Perry Cheng
IBM Research
United States of America

Alvin Cheung
University of Washington
United States of America
akcheung@cs.washington.edu

Brian Chin
Google
United States of America

Matteo Cimini
Indiana University
United States of America
mcimini@indiana.edu

Mohammad Dashti
EPFL DATA
Switzerland

Zachary DeVito
Stanford University
United States of America
zdevito@stanford.edu

Cezara Dragoi
IST Austria
Austria
cezara.dragoi@ist.ac.at

Vuk Ercegovac
Google
United States of America

Michael D. Ernst
Computer Science and Engineering,
University of Washington
United States of America
mernst@cs.washington.edu

Matthias Felleisen
PLT Design Inc.
United States of America
matthias@ccs.neu.edu

Robert Bruce Findler
PLT Design Inc.
United States of America

Stephen J. Fink
IBM Research
United States of America

Kathleen Fisher
Tufts University
United States of America
kfisher@eecs.tufts.edu

Matthew Flatt
PLT Design Inc.
United States of America

Matthew Fluet
Rochester Institute of Technology
United States of America
mtf@cs.rit.edu

Marco Gaboardi
University of Dundee
United Kingdom of Great Britain and
Northern Ireland
m.gaboardi@dundee.ac.uk

Michael Greenberg
Princeton University
United States of America
michael.m.greenberg@gmail.com

Dan Grossman
Computer Science and Engineering,
University of Washington
United States of America
djg@cs.washington.edu

Pat Hanrahan
Stanford University
United States of America
phanrahan@cs.stanford.edu

Peter Hawkins
Google
United States of America
phawkins@google.com

Thomas Henzinger
IST Austria
Austria
tah@ist.ac.at

Antony L. Hosking
Purdue University
United States of America
hosking@purdue.edu

Justin Hsu
University of Pennsylvania
United States of America
justhsu@cis.upenn.edu

Jon Jacky
Radiation Oncology, University of
Washington
United States of America
jon@washington.edu

Manohar Jonnalagedda
EPFL
Switzerland
manohar.jonnalagedda@epfl.ch

Shoaib Kamil
MIT
United States of America
skamil@csail.mit.edu

Yannis Klonatos
EPFL
Switzerland
yannis.klonatos@epfl.ch

Christoph Koch
EPFL
Switzerland
christoph.koch@epfl.ch

Shriram Krishnamurthi
PLT Design Inc.
United States of America

Hyoukjoong Lee
Stanford University
United States of America
hyouklee@stanford.edu

Yi Lin
Australian National University
Australia
yi.lin@anu.edu.au

Benjamin Livshits
Microsoft Research
United States of America
livshits@microsoft.com

Calvin Loncaric
Computer Science and Engineering,
University of Washington
United States of America
loncaric@cs.washington.edu

Jay McCarthy
PLT Design Inc.
United States of America

Daniel Marino
Symantec Research
United States of America
daniel_marino@symantec.com

Milo M K Martin
University of Pennsylvania
United States of America
milom@cis.upenn.edu

Mark Miller
Google
United States of America

Todd Millstein
University of California, Los Angeles
United States of America
todd@cs.ucla.edu

Stefan Muller
Carnegie Mellon University
United States of America
smuller@cs.cmu.edu

Madanlal Musuvathi
Microsoft Research
United States of America
madanm@microsoft.com

Todd Mytkowicz
Microsoft Research
United States of America
toddm@microsoft.com

Santosh Nagarakatte
Rutgers University
United States of America
santosh.nagarakatte@cs.rutgers.edu

Satish Narayanasamy
University of Michigan, Ann Arbor
United States of America
nsatish@umich.edu

Michael Norrish
NICTA
Australia
michael.norrish@nicta.com.au

Franz Och
Human Longevity, Inc.
United States of America

Georg Ofenbeck
ETH Zurich
Switzerland

Christopher Olston
Google
United States of America

Kunle Olukotun
Stanford University
United States of America
kunle@stanford.edu

Aurojit Panda
University of California Berkeley
United States of America
apanda@cs.berkeley.edu

Fernando Pereira
Google
United States of America

Stuart Pernsteiner
Computer Science and Engineering,
University of Washington
United States of America
spernste@cs.washington.edu

Francois Pottier
INRIA
France
francois.pottier@inria.fr

Jonathan Protzenko
Microsoft Research Redmond
United States of America
jonathan.protzenko@ens-lyon.org

Markus Püschel
ETH Zurich
Switzerland

Rodric Rabbah
IBM Research
United States of America
rabbah@us.ibm.com

Ram Raghunathan
Carnegie Mellon University
United States of America
ram.r@cs.cmu.edu

Veselin Raychev
ETH Zurich
Switzerland
veselin.raychev@inf.ethz.ch

Tiark Rompf
Purdue University
United States of America
tiark@purdue.edu

Mooly Sagiv
Tel Aviv University
Israel
msagiv@acm.org

Adrian Sampson
University of Washington
United States of America
asampson@cs.washington.edu

Michael Schapira
Hebrew University
Israel
schapiram@huji.ac.il

Scott Shenker
UC Berkeley and ICSI
United States of America
shenker@icsi.berkeley.edu

Sunil Shukla
IBM Research
United States of America

Jeremy Siek
Indiana University
United States of America
jsiek@indiana.edu

Abhayendra Singh
University of Michigan, Ann Arbor
United States of America
ansingh@umich.edu

Armando Solar-Lezama
MIT
United States of America
asolar@csail.mit.edu

Friedrich Steimann
Fernuniversität in Hagen, Germany
Germany
steimann@acm.org

Alen Stojanov
ETH Zurich
Switzerland

Arvind Sujeeth
Stanford University
United States of America
asujeeth@stanford.edu

Zachary Tatlock
Computer Science and Engineering,
University of Washington
United States of America
ztatlock@cs.washington.edu

Sam Tobin-Hochstadt
PLT Design Inc.
United States of America

Emina Torlak
Computer Science and Engineering,
University of Washington
United States of America
emina@cs.washington.edu

Martin Vechev
ETH Zurich
Switzerland
martin.vechev@inf.ethz.ch

Michael Vitousek
Indiana University
United States of America
mvitouse@indiana.edu

Daniel von Dincklage
Google
United States of America

Philip Wadler
University of Edinburgh
United Kingdom of Great Britain and
Northern Ireland
wadler@inf.ed.ac.uk

David Walker
Princeton University
United States of America
dpw@cs.princeton.edu

Xi Wang
Computer Science and Engineering,
University of Washington
United States of America
xi@cs.washington.edu

Steve Zdancewic
University of Pennsylvania
United States of America
stevez@cis.upenn.edu

Damien Zufferey
MIT CSAIL
United States of America
zufferey@csail.mit.edu

Coupling Memory and Computation for Locality Management

Umut A. Acar[1,2], Guy Blelloch[1], Matthew Fluet[3], Stefan K. Muller[1], and Ram Raghunathan[1]

1 **Carnegie Mellon University, Pittsburgh, PA, USA,** `{umut,blelloch,smuller,ram.r}@cs.cmu.edu`
2 **Inria, Paris, France**
3 **Rochester Institute of Technology, Rochester, NY, USA,** `mtf@cs.rit.edu`

Abstract

We articulate the need for managing (data) locality automatically rather than leaving it to the programmer, especially in parallel programming systems. To this end, we propose techniques for coupling tightly the computation (including the thread scheduler) and the memory manager so that data and computation can be positioned closely in hardware. Such tight coupling of computation and memory management is in sharp contrast with the prevailing practice of considering each in isolation. For example, memory-management techniques usually abstract the computation as an unknown "mutator", which is treated as a "black box".

As an example of the approach, in this paper we consider a specific class of parallel computations, nested-parallel computations. Such computations dynamically create a nesting of parallel tasks. We propose a method for organizing memory as a tree of heaps reflecting the structure of the nesting. More specifically, our approach creates a heap for a task if it is separately scheduled on a processor. This allows us to couple garbage collection with the structure of the computation and the way in which it is dynamically scheduled on the processors. This coupling enables taking advantage of locality in the program by mapping it to the locality of the hardware. For example for improved locality a heap can be garbage collected immediately after its task finishes when the heap contents is likely in cache.

1998 ACM Subject Classification D.3.3. Dynamic Storage Management

Keywords and phrases Parallel computing, locality, memory management, parallel garbage collection, functional programming, nested parallelism, thread scheduling

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.1

1 Introduction

A confluence of hardware and software factors have made the cost of memory accesses and thus management of data locality an important theoretical and practical challenge.

On the hardware side, we have witnessed, over the past two decades, an increasing performance gap between the speed of CPU's and off-chip memory, a.k.a., the memory wall [59]. Starting with sequential architectures, the growing CPU-memory gap led to the development of deep memory hierarchies. With the move from sequential to parallel (multi- and many-core) architectures, the CPU-memory gap has only grown larger due to parallel components competing for limited bandwidth, the larger latency caused by the increased scale, and the potential need for memory coherence [15]. To help close the increasing gap, modern parallel architectures rely on complex memory hierarchies with multiple levels of caches, some shared among cores, some not, with main memory banks associated with specific processors leading to non-uniform memory access (NUMA) costs, and complicated coherence schemes that can cause various performance anomalies. To obtain good performance from such hardware, programmers can employ very low-level machine-specific optimizations that carefully control memory allocation and the mapping of parallel tasks to specific processors so as to minimize communication between a processor and remote memory objects. While such a low-level approach might work in specific instances, it is known that this approach leads to vast inefficiencies in programmer productivity, and leads to software that is error-prone, difficult to maintain, and non-portable. It may appear that given these facts of hardware, we are doomed to designing and coding low-level machine-specific algorithms.

Developments on the software side limit the effectiveness of even such heroic engineering and implementation efforts: with the broad acceptance of garbage-collection frameworks and managed runtime systems, it may not even be possible for the programmer to exert much control over memory allocation, confining the applicability of locality optimizations to low-level languages such as C. Even in low-level languages, the challenges of locality management are amplified in the context of parallel programs, where opportunities for parallelism are expressed by the programmer and utilized by the run-time system by creating parallel threads as needed and mapping the threads to processors by using a thread scheduler. Such thread schedulers often make non-deterministic decisions in response to changes in the work load, making it very difficult if not impossible for the programmer to determine where and when a piece of computation may be performed.

Based on these observations, we conclude that locality should be managed by the compiler and the run-time system of the programming languages. Given the importance of locality in improving efficiency and performance, automatic management of locality in parallel programs can have significant scientific and broad impacts. But, such automatic management may seem hard: after all, how can the compiler and the run-time system identify and exploit locality? Fortunately, it is well known that programs exhibit natural locality. According to Denning, the reason for this is the "human practice of divide and conquer – breaking a large problem into parts and working separately on each." [24]. Research in the last decade shows that this natural locality of programs extends to nested parallelism as supported by languages such as Cilk [31], Fork/Join Java [41], NESL [14], parallel Haskell [39], parallel ML [29, 37, 30, 53], TPL [42], and X10 [16], Habanero Java [36]. This is because these languages take advantage of the same divide-and-conquer problem-solving methodology that leads to good locality. As a result, the compiler or runtime does not need to find the locality, just take advantage of it.

For example, consider the matrix multiply code in Figure 1. The eight recursive calls as well as the additions (recursively), can be performed in parallel, leading to abundant parallelism. Furthermore, the computation has abundant natural "temporal" locality: an $m \times m$ matrix multiply deep in the recursion will do $O(m^3)$ work but only need to load $O(m^2)$ memory allowing for significant reuse. There is also high "spatial" locality because quadrants can be laid out in a spatially coherent array. While this simple example considers only a highly structured algorithm, more irregular parallel algorithms also exhibit similar locality properties.

Previous work on (thread) scheduling has already shown how such natural locality in parallel software can automatically be exploited, and has proved bounds on a variety of machine models, such as multiprocessors with private caches using work stealing [2, 32], with shared caches using DFS scheduling [12, 18], or with trees of caches using space bounded schedulers [20, 11, 23]. A limitation of the previous results, however, is that they make very restrictive assumptions about memory allocation. The work-stealing results, for example,

```
function MM($A, B$) =
    **if** $A$ and $B$ are small
        **return** SmallMM($A, B$)
    **else**
        $R_{TL} = \text{MM}(A_{TL}, B_{TL}) + \text{MM}(A_{TR}, B_{BL})$
        $R_{BL} = \text{MM}(A_{BL}, B_{TL}) + \text{MM}(A_{BR}, B_{BL})$
        $R_{TR} = \text{MM}(A_{TL}, B_{TR}) + \text{MM}(A_{TR}, B_{BR})$
        $R_{BR} = \text{MM}(A_{BL}, B_{TR}) + \text{MM}(A_{BR}, B_{BR})$
        **return** compose($R_{TL}, R_{BL}, R_{TR}, R_{BR}$)
```

Figure 1 Parallel Matrix Multiply; X_{YZ} refers to the four quadrants of the matrix X with Y and Z denoting the left/right and bottom/top parts. The $+$ indicates matrix addition.

assume all memory is allocated on the stack, and the space bounded schedulers assume all space is preallocated and passed in. The results therefore do not apply to systems with automatic memory management and remain low-level in how the user has to carefully lay out their memory, making fine-grained dynamically allocated data structures particularly difficult to implement efficiently.

We believe that it is possible (and desirable) to manage locality automatically within the run-time system – specifically the memory manager and the scheduler – of a high-level garbage-collected parallel language. Why? At a fundamental level, such automatic management of locality is feasible because of the natural locality of certain software, and because all the information needed to manage it, such as the structure of the machine memory, the position of individual objects in memory, is readily available to and possibly controlled by the runtime system. But how?

In this paper, we suggest and briefly explore the following two ideas that we believe are important for taking advantage of locality for parallel programs.

1. Memory management and scheduling should be integrated as part of the same runtime system, such that management decisions are tightly linked with scheduling decisions.
2. The runtime heap structure should match the structure of the computation itself making it possible to position data and computation closely in hardware.

We describe an application of these ideas by considering purely functional nested-parallel programs executed on shared memory architectures such as modern multicore machines. Nested parallel computations, represented by fork-join programs is the mainstay of many parallel languages such as Cilk [31], Fork/Join Java [41], Habanero Java [36], NESL [14], parallel Haskell [39], parallel ML [29, 37, 30, 53], TPL [42], and X10 [16]. In this paper, we further limit ourselves to purely functional programs. While purely functional programming may seem like a significant restrictions, it often comes at no or little cost. For example, at Carnegie Mellon University, we teach the introductory undergraduate algorithms class [1], which covers a whole range of algorithms and data structures, including algorithms on sequences (arrays), binary search trees, and graphs by using the purely functional, nested-parallelism paradigm. The work presented in this paper was partly motivated from a desire to develop a compiler that generates fast parallel executables for modern multicore architectures from nested parallel programs written in functional languages.

Our approach to managing locality is to structure memory in a way to reflect the structure of the computation, as shaped by the scheduling decisions, and to couple memory management and computation via memory, on which they both operate. A key observation behind our approach is that many parallel programs, both purely functional and impure, observe a *disentanglement* property, where parallel threads avoid side-effecting memory objects that might be accessed by other threads. Disentanglement is often quite natural, because entanglement often leads to race conditions. Taking advantage of disentanglement, we structure the memory as a tree (hierarchy) of heaps, also by observing scheduling decisions, thus coupling the computation, scheduling, and the memory. Taking advantage of disentanglement, we also couple garbage collection with the computation, completing the circle.

While we assume purely functional code here, there are relaxations to purity that appear to remain compatible with the techniques proposed here. We discuss some possible directions for future research in Section 7.

2 Preliminaries

We consider shared-memory, nested parallel computations expressed in a language with managed parallelism, where the creation and balancing of parallelism is performed automatically by the run time system of the host language. We allow arbitrary dynamic nesting of fork-join (a.k.a., "par-synch" or "spawn-synch") constructs (including parallel loops), but no other synchronizations. A nested-parallel computation can be represented as a Directed Acyclic Graph (*DAG*) of *threads* (a.k.a., "strands"), each of which represent a sequential computation uninterrupted by parallel fork and join operations. The vertices of the DAG represent threads and the edges represent control dependencies between them. The DAG can be thought of as a "trace" of the computation, in that all threads evaluated during a computation along with the dependences between them are represented in the DAG. We call a vertex a *fork-vertex* if it has out-degree two and a *join-vertex* if it has in-degree two. Much of the literature on parallel computing uses this characterization.

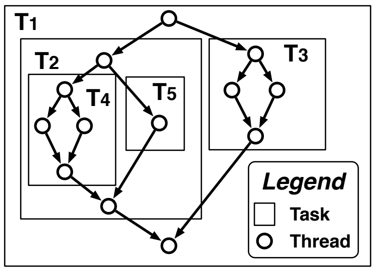


Figure 2 Tasks and threads.

We exploit an important hierarchical structure of nested-parallel computations. To see this structure, note first that in a nested-parallel computation, each fork-vertex matches with a unique join-vertex where the subcomputations started by the fork come together (join). We refer to such a subcomputation that starts at the child of a fork-vertex and completes at the parent of the matching join-vertex as a *task*. We also consider the whole computation as a task. We say that two threads are *concurrent* if there is no path of dependencies (edges) between them. We say that two tasks are concurrent if all their threads are pairwise concurrent. Figure 2 illustrates the DAG of an example fork-join computation along with its threads and some tasks. Tasks T_2 and T_3 are concurrent, whereas T_2 and T_4 are not.

At any given time during the computation we have a *task tree*, in which each node corresponds to a task. We can also associate with each leaf of the tree the thread that is currently active within that task. All these threads are said to be *ready* and a computational step can be taken on any subset of them. As expected, a fork operation executed on a leaf tasks T will create some number of new tasks that are children of T, and a join operation executed on a leaf task will remove the task from the tree. When the last child of a task T is removed, T becomes a leaf and hence has a *ready* thread. Note that although the DAG represents the trace of a computation when done, the task tree represents a snapshot in time

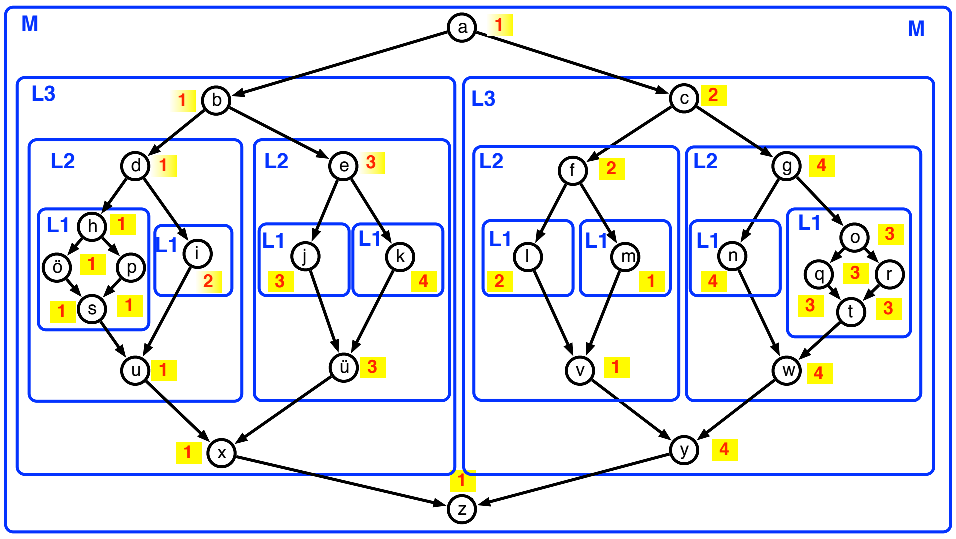


Figure 3 An illustration of how a parallel computation can be mapped processors and levels in the memory hierarchy. The label next to each thread (vertex) illustrates the processors, numbered from 1 to 4, executing the thread. Boxes illustrate parallel tasks. Each task is mapped to the lowest (fastest) level in the hierarchy that is large enough to hold its live data set; M denotes a memory bank and L1,L2, and L3 denote the levels of the cache.

of the computation. Assuming the computation is deterministic, the DAG is deterministic and does not depend on the schedule, but the task tree very much depends on the schedule.

3 Observations

We start by making some observations, using the following (hypothetical) example to help motivate them. Consider a nested-parallel program written with fork-join constructs. Suppose now that we run the program on an 4-processor machine, where each processor has its own L_1 and L_2 cache and four processors share an L_3 cache, by using a thread scheduler that performs load balancing so as to minimize completion time. Assume that when run, the program produces the computation DAG shown in Figure 3. Each thread is labeled with a letter, a through z (also with accents). We illustrate the schedule by labeling each thread with the number of the processor that executes it. For example, processor 1 executes thread a; processor 3 executes threads o, q, r and t. We illustrate a task by drawing a rectangular box around it. For the purposes of discussion, we assign each task a level in the memory hierarchy based on the size of the maximum memory needed by the live objects in that task, i.e., *its memory footprint*. For example, the task with root d is assigned to $L2$ because its live set does not fit into a level-1 cache but it does fit into a level-2 cache.

Disentanglement

Our approach to automatic locality management is based on a key property of nested-parallel computations, which we call *(memory) disentanglement.* In a nested-parallel computation, memory objects allocated by concurrent tasks are often *disentangled*: they don't reference each other. In purely functional languages such as NESL and pure Parallel ML, because of the lack of side effects, all objects allocated by concurrent tasks are disentangled. For example, in Figure 3, the memory object allocated by tasks rooted at b and c would be disentangled (they can point to objects allocated by their parent a, but not to objects allocated by each other). In the presence of side effects, disentanglement is also the common form because side effects naturally harm parallelism and therefore are rarely used in a way to cause entanglement. For example, in Cilk, where imperative programming is readily available, disentanglement is encouraged, is the common case, and can be checked with a race detector.

Task Local Heaps

A key property of generational collection with sequential garbage collection is that the generations can be sized so they take advantage of the various levels of the cache. If the newest/smallest generation fits within the L1 cache, for example, and is flushed or compacted whenever it fills, then most accesses to that space will be L1 cache hits. Short lived data will therefore rarely cause an L1 miss. Similarly if the second generation fits within L2, slightly longer lived data will rarely cause an L2 miss. Our goal is to take advantage of this property in the context of parallel computations in which caches might be local or might be shared among subsets of processors. In our example, all the tasks that have L1 around them are scheduled on a single processor so as long as we use a generation within each one, most accesses will be L1 hits even if the total memory allocated is much larger than L1 (recall that the size refers to the live data at any given time). More interestingly the task rooted at b fits within L3 (which is shared). If we allocate a heap proportionally and share that heap among the subtasks, then their footprint will fit within L3. However this only works if we do not simultaneously schedule any of the tasks under the root c since together they would not (necessarily) fit into L3. We therefore would want to link the scheduling decision to heap sizes.

Independence

With disentanglement it is always safe to garbage collect any task in the task graph by only synchronizing descendant tasks. This allows garbage collection to proceed independently without the complications of a concurrent collector at various levels of the hierarchy. This is true even for a moving (copying or compacting) collector. It is even possible to collect an ancestor task with a non-moving collector when a descendant task is still running as long as the root set is properly accounted for.

GC Initiation

Since scheduling is performed at task boundaries, pieces of computation move at those points. Thus, if we coordinate the garbage collector with the scheduler, we can initiate garbage collections at scheduling points. For example when finishing the task rooted at o all the data that was accessed in that task (nodes o, q, r and t) is presumably still in the L1 cache. It would now be a good time to run a GC on the L1 heap to compact it so a smaller footprint has to be flushed from the cache, and later reloaded into another cache when used.

4 Hierarchical Memory Management

Based on the aforementioned observations we now outline a particular set of techniques for managing memory for locality. This is meant to be a concrete example and there are surely many variants. As mentioned previously, the key components are integrating the memory management with scheduling, and using a hierarchy (tree) of heaps. Organizing the memory as a hierarchy of heaps will enable two key outcomes: 1) it will enable performing scalable, parallel and concurrent garbage collection, and 2) it will make it possible to map the hierarchy of heaps onto the caches of a hierarchical memory system, and 3) it allows the memory manager to collaborate with the scheduler to make effective use of shared caches by associating heaps with shared caches in the hierarchy.

Hierarchical, task- and scheduler-mapped heaps

We take advantage of the invariant that parallel tasks are disentangled – they create memory graphs that are disjoint (don't point to each other) – and partition memory into *subheaps* or *heaps*, which will be managed independently. Heaps can be of any size and can match the sizes in the cache hierarchy. For good locality, heaps are associated with *separately scheduled tasks* that start executing at a processor due to a scheduler task migration action, and heaps are merged to reflect the structure of the computation. Each heap contains all the live data allocated by its associated task, except for live data living in the heap of a migrated descendant task. The heaps are therefore properly nested, becoming smaller towards the leaves of the task tree. Each child heap is newer than the parent and therefore references only go up the heap hierarchy. A heap at the leaf corresponds to a computation that runs sequentially and thus can be garbage collected independently. We note that such a computation might have nested parallel tasks within it as long as the scheduler has not migrated them. When a task terminates its heap can be merged into its parent's heap. This is a logical merge and might or might not involve actually moving data.

Knot sets

The heaps in the hierarchy are "tied together" with knots; a *knot* is a pointer that points from the heap of a separately scheduled task H_c into its parent heap H_p. The knots can be used as a root set into H_p to allow for separate collection of H_p. Initially the knot set from H_c consists of only the closure for the task that created H_c. While the computation in the child runs this is a conservative estimate of what the child can reach. Whenever the child does a collection on its heap H_c the knot set to its parent can be updated, allowing more to be collected in the parent. Maintaining the knots therefore requires no read barriers (special code for every read), and allows garbage collection to proceed independently within each heap as long as internal heaps do not move their data. The only required synchronization is when accessing the knots, which is easy to implement with an atomic switch by the child.

An example

As an example, let's consider a hypothetical task A that starts as shown in Figure 4. We assume the task is migrated by the scheduler (perhaps to start the computation) and has a heap H_A of a size that fits in the L2 cache. The roots of the task point to two trees that are stored in the heap. For illustrative convenience, we draw the task (A) within its heap. We draw memory objects as squares and references between them as edges. We draw unreachable objects that can be reclaimed in green (appears gray in grayscale). After it starts executing, A performs some computation and forks two new tasks B and C that take as argument subtrees from H_A (Figure 5). Supposing that these tasks are executed in parallel by a scheduling action, we will create two separate heaps H_B and H_C for them. The arguments to B and C create the initial knot-set of H_A (depicted as dark squares). When forking, A creates the suspended join thread D (continuation) for when B and C complete,

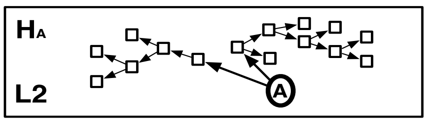


Figure 4 Start task A and its heap H_A.

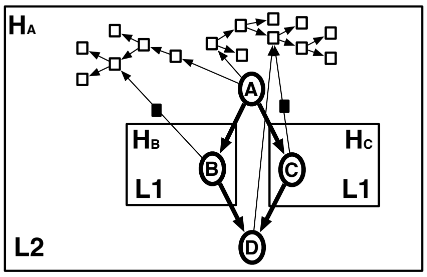


Figure 5 Task A forks two new tasks B and C. Their heaps H_B and H_C can be managed and collected independently.

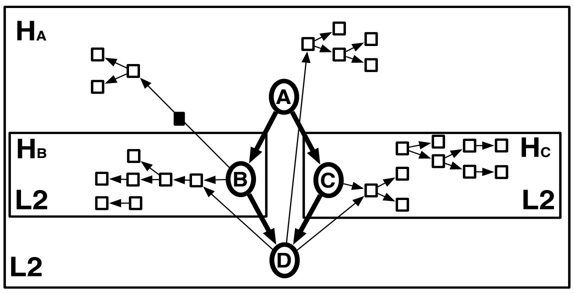


Figure 7 By the time tasks B and C complete, they have grown their heap by adding new objects.

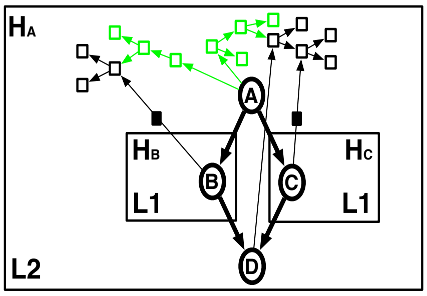


Figure 6 The suspended heap H_A is collected some time after B and C starts running.

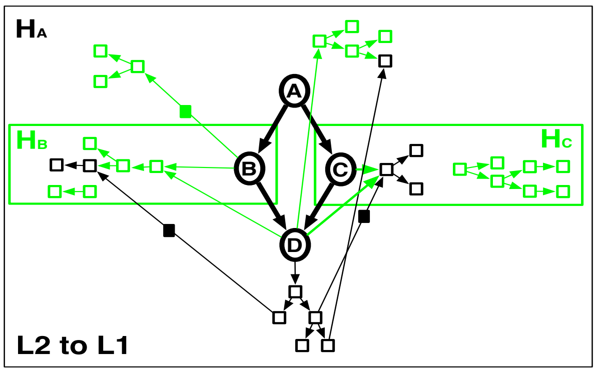


Figure 8 When B and C join D, their heaps are merged with H_A, the heap of A. A garbage-collection of H_A is shown some time after the merge.

which can also refer to memory in H_A. As B and C run, if needed, the suspended heap H_A can be collected by performing a non-moving garbage collection using the knot sets as roots. Figure 6 illustrates the reclaimed memory objects in green (light in grayscale).

The heaps H_B and H_C start out empty (trivially fitting into an L_1 cache) and grow as B and C allocate objects, triggering garbage collection. Since B (C) is independent from all other tasks running in parallel and its knot-set is empty, its heap is completely independent from other heaps. It can therefore collect anytime using any algorithm it chooses. In particular it can use a copying or compacting collector to ensure the footprint of the remaining live data is minimized. A good time for garbage collection is when the heap size approaches L_1; this would help keep the working set in L_1. If significant live memory remains after a collection, the heap size might be promoted to the next cache size as illustrated in Figure 7, where the tasks have been promoted to fit into the level-2 cache. When B and C complete (not necessarily at the same time), they pass their results to the join thread D. At this point, the heap H_B (or H_C) can be merged with its parent heap H_A and its knot sets can be dropped. In the example, after thread D starts running it creates a pair of the results returned to it by B and C. As D runs, its heap can be collected (Figure 8). Such a collection can reclaim many unreachable objects (essentially any object that does not contribute to the final result), possibly making the heap fit into L1.

5 Implementation and Evaluation

We are in the process of developing a compiler and run-time system for an extension of the Standard ML language that supports nested parallelism and the presented hierarchical memory management techniques (Section 4). The starting point for our implementation effort is the MLton compiler infrastructure [48, 58] and the shared-heap-multicore branch developed by Daniel Spoonhower as part of his dissertation work [55, 54]. MLton is an open-source, whole-program, optimizing compiler for Standard ML (SML) [47]. The current release of MLton includes a self-tuning garbage collector that uses copying, mark-compact, and generational collection algorithms, automatically switching between them at run time based on both the amount of live data and the amount of physical memory. MLton's combination of garbage collection algorithms is inspired by Sansom's dual-mode garbage collection [51]. The current release of MLton does not include support for using multiple processors or processor cores to improve the performance of SML programs.

Using our implementation, we plan to evaluate the quantitative effectiveness of our proposed techniques by developing a purely-functional nested-parallelism parallel benchmark suite modeled after the recently announced Problem Based Benchmark Suite (PBBS) [52], which include standard parallelism benchmarks as well as less traditional benchmarks using graph algorithms. We also plan to evaluate the approach more quantitatively by using it in our teaching.

6 Related Work

Scheduling for Locality

There have been thousands of papers that deal with the issue of locality in some way. In the following discussion we focus on techniques that (1) apply to parallel computations where tasks are created dynamically and mapped to processors by the runtime, (2) are useful for reasonably general purpose programs and data structures (e.g. not just dense arrays), (3) are suited for shared memory (or at least shared address space) cache-based architectures, and (4) for which at least something theoretical can be said about their properties. There has been plenty of work on models where the user maps their algorithms directly to processors (e.g. [56, 4, 57]), or for specific domains such as dense matrices (e.g. [17, 9, 7]).

There has been significant work over the past decade on trying to model and understand locality in a way that is independent of how a computation is mapped to processors, and then have a runtime scheduler somehow preserve this locality [2, 12, 32, 21, 10, 22, 13, 20, 23, 50, 11]. By relying an specific scheduling policies, the results from this work can asymptotically bound cache misses on a concrete parallel machine model in terms of abstract costs derived from the program. For example, earlier work showed that for a nested parallel computation with depth (span, critical path length) D and sequential cache complexity Q_1, a work-stealing scheduler on P processors with private caches will incur at most $Q_1 + O(PDM/B)$ cache misses [2], where M is the total size of each cache and B is the block size. Similarly a PDF scheduler on the same computation with a shared cache of size $M + O(PDB)$ will incur at most Q_1 cache misses [12]. A key point of these results is that cost metrics for a program that have nothing to do with the particular machine, Q_1 and D, can be used to bound costs on those machines. In addition to theory the approaches have been shown to work well in practice [18].

More recent work has considered deeper hierarchies of caches, involving levels of private and shared caches, and has shown that a class of space-bounded schedulers [20, 11, 23] are well suited for such hierarchies. These schedulers try to match the memory footprint of a subcomputation with the size of a cache/cluster in the hierarchy and then run the computation fully on that cluster. Under certain conditions these schedulers can guarantee cache miss bounds at every level of the cache hierarchy that are comparable to the sequential misses at the same level. However in this case the abstract cache complexity is not the sequential complexity, but a relatively natural model [11] that considers every access a miss for subcomputations that don't fit in the cache and every access a hit for those that do. Cole and Ramachandran [23] describe a scheduler with strong asymptotic bounds on cache misses and runtime for highly balanced computations. Recent work [11] describes a scheduler that generalizes to unbalanced computations. Several heuristic techniques have also been suggested for improving locality with dynamic scheduling on such cache hierarchies [28, 40, 50].

Our work builds on ideas developed in this previous work, but as far as we know, none of this work has considered linking memory management with scheduling to improve locality, and the previous work assumes very limited memory allocation schemes.

Memory Management

There has also been significant work on incorporating locality into memory management schemes. Several explicit memory allocators, such as Hoard [8], TCMalloc [33], and SSMalloc [43], have been designed to be multithreading-friendly. These schemes create local pools of memory on each processor so that lock contention is reduced and so that freed memory is reused while still in the cache. All the approaches are heuristic; in our work, we have found that they can do exactly the wrong thing in certain contexts. A recent blog discussion [46] shows just how subtle memory allocation and management can be for a multicore system. The PIs have independently experienced many of the same issues discussed in the blog as well as several others. The problem is that the schemes know nothing about the scheduler or even programming methodology in which they are being embedded. The only information they have is the allocation request sequence at each thread.

For implicit memory allocation with garbage collection, maintaining locality becomes even more complicated, although it does give the runtime more flexibility because of the ability to move data. There have been dozens of proposed techniques for parallel garbage collection ([38]), and many of these suggest the use of processor or thread local heaps [34, 35, 25, 19, 27, 49, 3, 5, 45, 60]. Some of this work uses the idea of creating a nursery for each thread that handles recent allocations, and then a global heap for shared data [25, 27, 3, 5, 45]. If the nursery is appropriately sized, this organization helps keep recently generated and accessed data in local cache. The problem is that once promoted to the global heap, locality is lost. Furthermore since the work uses no knowledge of the scheduler, when a new task is scheduled on a processor, what is left in the cache will likely be evicted while still fragmented and not collected. Finally, as far as we know, none of this work has looked at multi-level hierarchies with private and shared caches nor has it tried to show any theoretical bounds on cache misses.

Another property of almost all memory management systems is that they try to be independent of the computation that is running on them and are only accessed through a minimal interface. Most work on garbage collection has treated the mutator (the program) as a black box that runs on a fully general purpose GC, perhaps with some heuristics that work well with typical programs (e.g. generations, more reads than writes) [38]. Although this might be a benefit for porting a GC across different programming languages, in practice most GCs are designed for a specific language. We believe that by making them generic is giving up significant benefit that might be achieved by tying them more directly to the program or other aspects of the runtime such as the scheduler.

7 Discussions

In this paper, we only considered purely functional, nested parallel programs. While probably not straightforward, it appears possible to extend these ideas to more relaxed models of parallelism to include impure programs with side effects. This is because the primary assumptions that we make, disentanglement, is guaranteed by purely functional programs but it is, in general, a weaker condition. For example, side effects that remain local to a thread do not violate disentanglement. More generally, it appears possible to extend the work presented here to allow for arbitrary side effects by using techniques such as those used by two-level garbage collectors employed in memory managers of functional languages [26, 6, 44, 53]. For example, all objects reachable by mutable memory locations can be kept in a separate region of memory and treated specifically to ensure that memory management does not alter program invariants.

8 Conclusion

In this paper, we propose techniques for automatic locality and memory management in nested parallel programs. The basic idea behind these techniques is to take advantage of a key invariant of much parallel computations – their independence from each other, which then leads to disentangled memory regions – and structure memory hierarchically to reflect the structure of the computation and hardware, both of which are also usually hierarchical.

Acknowledgments. This research is partially supported by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940, and by the European Research Council under grant number ERC-2012-StG-308246.

References

1 Umut A. Acar and Guy Blelloch. 15210: Algorithms: Parallel and seqential, 2015. `http://www.cs.cmu.edu/~15210/`.

2 Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

3 Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *ISMM*, pages 21–30, 2010.

4 Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, 2008.

5 Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage collection for multicore NUMA machines. In *MSPC'11: Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 51–57. ACM Press, June 2011.

6 Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI'11, San Jose, CA, USA, June 5, 2011*, pages 51–57, 2011.

7 Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Graph expansion and communication costs of fast matrix multiplication: regular submission. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 1–12, 2011.

8 Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS*, pages 117–128, 2000.

9 Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–57, 2006.

10 Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *In the Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 501–510, 2008.

11 Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'11, pages 355–366, 2011.

12 Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, 2004.

13 Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low-depth cache oblivious algorithms. In *SPAA*, 2010.

14 Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM, 1996.

15 S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel processor and platform evolution for the next decade., 2005.

16 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'05, pages 519–538. ACM, 2005.

17 Siddhartha Chatterjee. Locality, communication, and code generation for array-parallel languages. In *PPSC*, pages 656–661, 1995.

18 Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *ACM Symposium on Parallel Algorithms and Architectures*, SPAA'07, pages 105–115, 2007.

19 Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *PLDI*, pages 125–136, 2001.

20 R.A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

21 Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *SPAA*, 2007.

22 Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *SPAA*, 2008.

23 Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. In *ICALP*, 2010.

24 Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, July 2005.

25 Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL*, pages 70–83, 1994.

26 Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 70–83, 1994.

27 Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In *ISMM*, pages 76–87, 2002.

28 Kayvon Fatahalian, Timothy Knight, Mike Houston, Mattan Erez, Daniel Horn, Larkhoon Leem, Ji Park, Manman Ren, Alex Aiken, William Dally, and et al. Sequoia: Programming the memory hierarchy. *ACMIEEE SC 2006 Conference SC06*, 0(November):4–4, 2006.

29 Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP'08: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*, pages 119–130. ACM Press, September 2008.

30 Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. *The Journal of Functional Programming*, 20(5–6):537–576, November 2010.

31 Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

32 Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA*, 2006.

33 Sanjay Ghemwat and Paul Menage. TCMalloc : Thread-caching malloc, 2010.

34 Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

35 Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

36 Shams Mahmood Imam and Vivek Sarkar. Habanero-java library: a java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ'14, Cracow, Poland, September 23-26, 2014*, pages 75–86, 2014.

37 Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. The design rationale for Multi-MLton. In *ML'10: Proceedings of the ACM SIGPLAN Workshop on ML*. ACM, 2010.

38 Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook : The Art of Automatic Memory Management*. CRC Press, 2012.

39 Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP'10, pages 261–272, 2010.

40 Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 217–228, 2008.

41 Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA'00, pages 36–43, 2000.

42 Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA'09, pages 227–242, 2009.

43 Ran Liu and Haibo Chen. SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Third ACM SIGOPS Asia-Pacific conference on Systems*, APSys'12, pages 15–15, Berkeley, CA, USA, 2012. USENIX Association.

44 Simon Marlow and Simon L. Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, pages 21–32, 2011.

45 Simon Marlow and Simon L. Peyton Jones. Multicore garbage collection with local heaps. In *ISMM*, pages 21–32, 2011.

46 Apurva Mehta and Cuong Tran. Optimizing linux memory management for low-latency / high-throughput databases. `http://engineering.linkedin.com/performance/optimizing-linux-memory-management-low-latency-high-throughput-databases`, 2013.

47 Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised)*. The MIT Press, 1997.

48 MLton web site. `http://www.mlton.org`.

49 Takeshi Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *OOPSLA*, pages 377–390, 2009.

50 Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag.

51 Patrick Sansom. Dual-mode garbage collection. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, 1991.

52 Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA'12, pages 68–70, New York, NY, USA, 2012. ACM.

53 K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Multimlton: A multicore-aware runtime for standard ML. *J. Funct. Program.*, 24(6):613–674, 2014.

54 Daniel Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009.

55 Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008.

56 Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

57 Leslie G. Valiant. A bridging model for multicore computing. In *Proc. 16th European Symposium on Algorithms*, 2008.

58 Stephen Weeks. Whole-program compilation in MLton. In *ML'06: Proceedings of the 2006 workshop on ML*, pages 1–1. ACM, 2006.

59 Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

60 Jin Zhou and Brian Demsky. Memory management for many-core processors with software configurable locality policies. In *ISMM*, pages 3–14, 2012.

Verified Compilers for a Multi-Language World[*]

Amal Ahmed

Northeastern University
amal@ccs.neu.edu

Abstract

Though there has been remarkable progress on formally verified compilers in recent years, most of these compilers suffer from a serious limitation: they are proved correct under the assumption that they will only be used to compile *whole* programs. This is an unrealistic assumption since most software systems today are comprised of components written in different languages – both typed and untyped – compiled by different compilers to a common target, as well as low-level libraries that may be handwritten in the target language.

We are pursuing a new methodology for building verified compilers for today's world of multi-language software. The project has two central themes, both of which stem from a view of *compiler correctness as a language interoperability problem.* First, to specify correctness of component compilation, we require that if a source component s compiles to target component t, then t linked with some arbitrary target code t' should behave the same as s interoperating with t'. The latter demands a formal semantics of interoperability between the source and target languages. Second, to enable safe interoperability between components compiled from languages as different as ML, Rust, Python, and C, we plan to design a gradually type-safe target language based on LLVM that supports safe interoperability between more precisely typed, less precisely typed, and type-unsafe components. Our approach opens up a new avenue for exploring sensible language interoperability while also tackling compiler correctness.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, D.3.1 Formal Definitions and Theory, D.3.4 Processors

Keywords and phrases verified compilation, compositional compiler correctness, multi-language semantics, typed low-level languages, gradual typing

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.15

1 Landscape of Verified Compilation

The field of compiler verification has witnessed considerable progress in the last decade following the pioneering work on CompCert [29, 30]. The latter uses the Coq proof assistant to both implement and verify a multi-pass optimizing compiler from C to assembly, proving that the compiler preserves semantics of source programs. Several other compiler verification efforts have successfully followed CompCert's lead and basic methodology to verify increasingly sophisticated compilers for increasingly realistic languages, for instance, focusing on just-in-time compilation [37], multithreaded Java [31], C with relaxed memory concurrency [48], LLVM passes [62], and imperative functional languages [15, 28].

Unfortunately, all of the above projects prove correctness assuming that the compiler will only be used to compile whole programs. But this assumption contradicts the reality of how we use these compilers. The whole programs that we actually run are almost never the output of a single compiler: they are composed by linking code from various places, including

[*] This work is supported by the National Science Foundation and a Google Faculty Research Award.

the runtime system, libraries, and foreign functions, all potentially compiled using different compilers and written in different languages (e.g., Java, Python, C, and even handcrafted assembly). For today's world of multi-language software, we need verified compilers that guarantee correct compilation of *components.*

Formally verifying that components are compiled correctly – often referred to as *compositional compiler correctness* – is a difficult problem. A key challenge is how to state the compiler correctness theorem for this setting. CompCert's compiler correctness theorem is easy to state thanks to the whole program assumption: informally, it says that if a source program P_S compiles to a target program P_T, then running P_S and P_T results in the same trace of observable events. The same sort of theorem does not make sense when we compile a component e_S to a component e_T: we cannot "run" a component since it is not a complete program.

Part of the challenge is that any correct-component-compilation theorem should satisfy two important properties: (1) it should allow compiled components to be linked with target components of arbitrary provenance, including those compiled from other languages (dubbed *horizontal compositionality* in the literature); and (2) it should support verification of multi-pass compilers by composing proofs of correctness for individual passes (dubbed *vertical compositionality*). These are nontrivial requirements. There have been several notable efforts at compositional compiler verification in recent years but none offer a proof methodology that fully addresses the dual challenges of horizontal and vertical compositionality. The earliest work, by Benton and Hur [12, 13, 26], specifies compiler correctness in a way that does not scale to multi-pass compilers. Both that work and the recent work on Pilsner [41] only supports linking with separately compiled components from the *same* source language. The recent work on Compositional CompCert [51] supports linking across the languages in CompCert's compilation pipeline, but all of these share the same memory model; it is not clear how to extend the approach to support linking across languages with different memory models. (We discuss related work in detail in §3.2.)

Let us look at why compiler correctness becomes challenging in the context of multi-language software. The issue is that real software systems often link together components from multiple languages with *different expressive power* and *different guarantees.* When compiling source language S, what does it mean for the compiler to "preserve the semantics of source code" when a compiled component e_T may be linked with some e'_T from a language S' that is *more expressive* or provides *fewer guarantees* as compared to S?

- Suppose S does not support first-class continuations (call/cc) but S' does. For instance, e'_T, compiled from language S', might be a continuation-based web server that provides call/cc-based primitives that a web programmer can use for creating client-server interaction points [46, 27, 47, 42]. The programmer, meanwhile, might use language S to develop her web application – making use of the afore-mentioned primitives which allow her to write her program in a more direct style [46, 27] – and compile it to e_T. When we link these components and run, e'_T disrupts the control flow of e_T in a way that is useful and desirable but does not mirror any whole-program behavior in the source language S. How, then, do we specify that the compiler "preserves semantics of source code"?
- Suppose S supports strong static typing (e.g., ML) while S' is dynamically typed (e.g., Scheme) or weakly typed (e.g., C). Again, linking e'_T with e_T might result in code that does not mirror any whole-program behavior in the source language S. For instance, (1) the C component may try to access memory it has already freed; (2) the C component may write past the end of a C array, but in doing so overwrite a memory location owned by an ML module; or (3) the Scheme component may apply an ML function that expects

an argument of type $\forall\alpha.\alpha \to \alpha$ to the boolean negation function, thus violating ML's guarantee that a function of this type will behave parametrically with respect to its input. The first interaction (with C) seems entirely reasonable since the error is in C code and does not affect the behavior of the ML component. The second interaction (with C) is clearly unacceptable as it can violate ML's type safety guarantee. The third interaction (with Scheme) may or may not be considered reasonable: it does not violate ML's type safety guarantee, but it does violate ML's parametricity guarantee. In what sense, then, can we say that the compiler "preserves semantics of source code"?

The above examples suggest that we need a novel way of understanding and specifying what we mean by compiler correctness in the context of multi-language software.

2 Our Approach and Research Goals

My research group at Northeastern is working on techniques for building realistic, multi-pass *verified compositional compilers* that guarantee correct compilation of components and formally support linking with arbitrary target code, no matter its source. We follow the tenet that ***compositional compiler correctness is a language interoperability problem***. We are in the early stages of a project that will require years of investigation and evaluation. Our long-term vision, depicted in Figure 1, is to have verified compositional compilers from languages as different as C, Python, ML, Rust, Coq's Gallina, and Hoare Type Theory (HTT) [38, 40, 39] to a common low-level target language that supports safe interoperability between more precisely typed, less precisely typed, and type-unsafe components. Our current focus is on laying the groundwork for realizing this vision through two central contributions:

1. Development of a proof methodology for verifying compositional compilers. The defining feature of our approach is that the specification of compositional compiler correctness relies on a formal semantics of interoperability between source components and target code using multi-language semantics in the style of Matthews and Findler [33].(We have already done extensive work on developing this methodology [45]; we give further details in §3.) To demonstrate the viability of this approach, we plan to verify compositional type-preserving compilers for subsets of ML and Rust.
2. Design of a *gradually type-safe* LLVM-like target language (dubbed GTVM, see Figure 1) that supports safe interoperability between components that are statically type-safe (e.g., produced by our type-preserving ML and Rust compilers), dynamically type-safe (e.g., compiled from Python or Scheme), or potentially unsafe (e.g., compiled from C). We plan to build on Vellvm (verified LLVM) [57, 61, 62] by first developing TTVM (*tightly typed* LLVM), a statically type-safe version of the LLVM IR formalized in Vellvm, and then design GTVM as a gradually typed extension of TTVM. GTVM will make use of casts (think: coercions or contracts) to ensure safe interoperability between the more precisely typed, less precisely typed, and type-unsafe parts of the language – the latter unsafe part is just standard LLVM IR which has types but is not type-safe. We will compile GTVM to the LLVM IR, inserting wrappers that perform dynamic checks to ensure safe coercion. Thus, compilers targeting our gradually type-safe LLVM can continue to leverage the optimizations provided by the LLVM compiler infrastructure [54] or the verified optimizations provided by Vellvm [57, 62].

Specifying compositional compiler correctness for a multi-language world. Informally, if a component e_S compiles to a component e_T then compiler correctness should require that

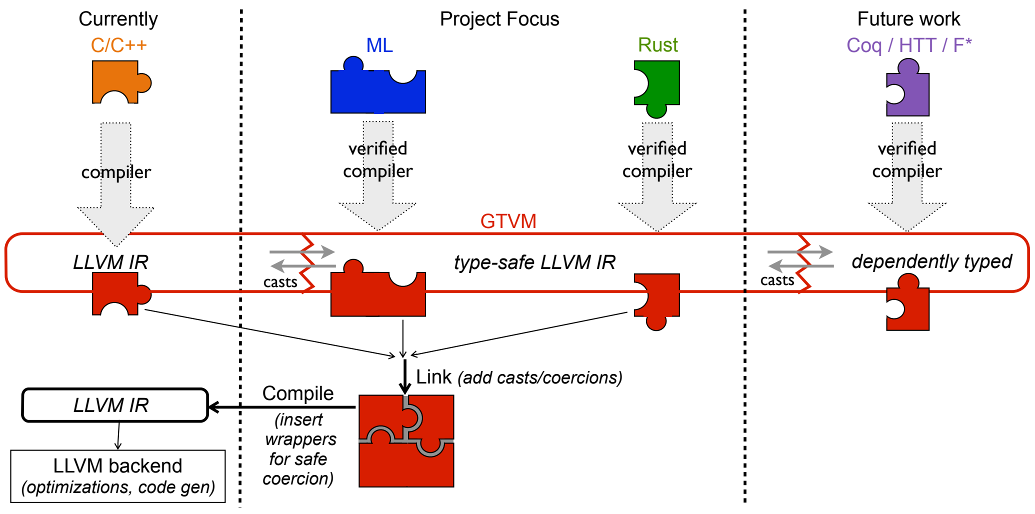


Figure 1 Research planned as part of this project and potential future work.

e_S is "equivalent" to e_T. But how can we *formalize* this notion of "equivalence" between source and target components? Observe that to *use* a compiled component e_T, we will link it with some other target-level component e'_T to obtain a whole program that can be run. Intuitively, therefore, compiler correctness should guarantee that the operational behavior of this resulting target program is the same as the operational behavior of e_S linked with e'_T. Therefore, to formally state that "a component is compiled correctly," we need to formalize the semantics of interoperability between source and target code. For a multi-pass compiler we propose to do this in a modular fashion. For instance, if the compiler consists of two passes, from source language S to intermediate language I to target language T, we define a combined language SIT that embeds these three languages and formalizes the semantics of interoperability between each pair of adjacent languages using boundaries in the style of Matthews and Findler's multi-language semantics [33]. We can stack these boundaries to allow interoperability between the source and target of the compiler, e.g., $\mathcal{SI}(\mathcal{IT}(e_T))$, which we abbreviate to $\mathcal{SIT}(e_T)$, allows a target component e_T to be used from within an S-language expression. Compiler correctness can now be stated as observational equivalence in the combined language: if e_S compiles to e_T, then e_S is observationally equivalent to $\mathcal{SIT}(e_T)$. Direct proofs of observational equivalence – also known as contextual equivalence – are known to be intractable. Therefore, we define a *logical relation* for the combined language that corresponds to contextual equivalence and use that to carry out the proof of compiler correctness. Note that we do not use the multi-language semantics for running actual multi-language programs; its purpose is to serve as a *specification* of the desired source-target relationship, allowing us to state and prove compiler correctness. This specification also enables reasoning about the whole-program behavior of e_T linked with e'_T in terms of the whole-program behavior of e_S linked with $\mathcal{SIT}(e'_T)$. Most importantly, note that we have not imposed any restrictions on the provenance of e'_T. We give further details in §3.

Why ML and Rust? All of the existing work on compositional compiler correctness has either (a) focused on unsafe C-like source languages [51, 59] or (b) assumed that code produced by a verified compiler from a type-safe language will only be linked with code produced by another verified compiler from the *same* source language [12, 13, 26, 41]. We instead focus on compiling the statically typed languages ML and Rust while allowing linking with code of arbitrary provenance. This is a real-world scenario with interesting semantic challenges for interoperability – specifically, how to maximize interoperability with less precisely typed and type-unsafe components while ensuring that those interactions respect the ML or Rust type system. We believe that these interoperability challenges make compositional compiler correctness harder to establish for these languages. Languages that provide fewer guarantees (e.g., C) are less picky in terms of what they can interoperate with, which makes it less semantically challenging to ensure correct compositional compilation.

Rust is a systems programming language with type-system support for safe memory management. In essence, it supports *affine* types which indicate that a resource – in this case, memory cells – can be used at most once. Since ML does not support affine types, we will have to ensure that ML respects Rust's affine typing guarantees.

GTVM: a gradually type-safe LLVM IR. As the above discussion suggests, to prove compiler correctness in the context of multi-language software, we must specify a formal semantics of interoperability (1) between source and target code, and (2) between more precisely typed, less precisely typed and type-unsafe code. For instance, for our ML compiler, we will have to specify how an ML source component e_S interoperates with a target component e'_T that may have been compiled from Scheme or C. We believe that safe interoperability between more and less typed and type-unsafe should be investigated at the level of the target language so that the benefits of this effort can be reaped by all compilers that target the language. That is the philosophy underlying our investigation of GTVM.

To understand the design of GTVM, it is useful to think of it in two layers: (1) a statically type-safe core, TTVM (tightly typed LLVM), and (2) a gradual typing extension, GTVM, that extends TTVM with dynamically type-safe code of type `dyn` and unsafe, standard LLVM IR code of type `un`. Our goal is a gradual type system that ensures that the typing (and parametricity) guarantees provided by more precise types cannot be violated by the less precisely typed parts of the language.

The statically type-safe core language, TTVM, should provide a rich enough type system to adequately serve as a target for type-preserving compilers from different statically typed languages. Since we wish to compile ML and Rust, our TTVM will need to support at least type abstraction and affine types in addition to the standard LLVM types. Even within this TTVM core, we will need a system of casts (contracts) to mediate between more precise and less precise types to support interoperability between code compiled from different source languages. For instance, we wish to allow code compiled from ML and Rust to interoperate, which means we will need to design casts (probably along the lines of Tov and Pucella [55]) that protect an affinely typed resource (from Rust) from being used more than once, even if it is passed to a function (from ML) that knows nothing about affine types and might freely duplicate the resource.

GTVM will let compiler writers choose whether to target the statically type-safe, dynamically type-safe, or unsafe parts of the language (or some mix of the three) depending on the nature of their source language, and *depending on how restrictive a form of interoperability they want*. ***The lever that provides control over interoperability is the compiler's type translation.*** For instance, when compiling ML, picking the most informative type translation (relative to source types) will guarantee that interoperability with other languages respects the source type system – including parametricity guarantees, as long as the compiler doesn't monomorphize. At the other extreme, translating all ML code to the type `un` says that interoperability with other code need not respect ML's type system guarantees.

Our longer-term goal is to enrich the statically typed core of GTVM with dependent types in the style of Hoare Type Theory (HTT) [38, 40]. HTT incorporates specifications – in the style of Hoare logic or separation logic – into types and makes it possible to formally specify and reason about effects. A type system based on HTT would allow us to express rich invariants about memory layout, separation, and resource usage, which will be important for specifying low-level conventions that affect interoperability at the LLVM level.

Enabling secure compilation via target-level types. Compiler correctness is about preservation of dynamic semantics, but we are interested in an architecture that can also support the development of secure (or fully abstract) compilers. Informally, secure compilation guarantees that compiled components will remain as secure as their source-level counterparts when executed within target contexts of arbitrary origin. More formally, a fully abstract compiler guarantees that if two source components have the same observable behavior in all well-typed source contexts then their compiled versions must have the same observable behavior in all appropriately-typed target contexts. In prior work [4], we studied how compilers can be made fully abstract by changing the compiler's type translation to ensure that compiled code is never linked with target contexts whose behavior does not match that of some source context. By equipping GTVM with an expressive type system, we wish to offer compiler writers the facility to pick different degrees of *protection* of compiled components from their target contexts – ranging from no protection at all (when compiled code has type `un` and the verified compiler only guarantees preservation of dynamic semantics), all the way through fully abstract compilation – via their choice of type translation.

The rest of this paper explains our approach to specifying compiler correctness (§3) and then outlines our research plans and anticipated challenges (§4).

3 Proof Methodology for Verified Compilation of Components

In recent work [45], we have demonstrated the viability of our multi-language-semantics approach, using it to prove correctness of a two-pass compiler that performs closure conversion and heap allocation, translating a polymorphic source language with recursive types to a target that also features dynamically allocated mutable memory. In particular, we support linking of compiled code with code that performs state effects that cannot be expressed in the source. This work was the first multi-pass, compositional compiler-correctness result. We believe that it is, to date, the only approach that supports linking with code that cannot be expressed in the verified compiler's own source language. We plan to use this methodology to build verified compositional compilers for ML and Rust. Below, we explain our approach in more detail, compare it to related work, and illustrate our methodology using typed closure conversion [34, 36, 3] as a case study.

3.1 Specifying compiler correctness using multi-language semantics

The compiler correctness theorem should say that if a component e_S compiles to e_T, then some desired relationship $e_S \simeq e_T$ holds between e_S and e_T – intuitively, they should "behave the same." But how do we formally specify $e_S \simeq e_T$? To answer this question, consider how the compiled component is actually used: it needs to be linked with some e'_T, creating a whole program that can be run. This e'_T may have been handwritten in the target language or produced by another compiler, possibly from a different source language. Of course, it doesn't make sense to link with *any* e'_T: at the very least, e'_T should adhere to the same calling conventions that e_T does. Moreover, since our target language is typed – with types `dyn` and `un` in additional to the more standard types – that means that e_T can only be linked with components of a certain type because we want the resulting whole program to be well typed. Informally, then, the compiler correctness theorem should guarantee that if we link e_T with an appropriately typed e'_T then the resulting target-level program should correspond to the source component e_S linked with e'_T. Formally speaking, what does it mean to "link a source component with a target component" and what are the rules for running the resulting source-target hybrid? We have argued that these questions demand a *semantics of interoperability* between the source and target languages. Next, we explain how to specify such semantics.

Consider a two-pass compiler from a source language S (in blue) to intermediate language I (in red) to target language T (in purple). The first pass translates S components $\mathsf{e_S}$ of type τ to I components $\mathrm{e_I}$ of type $\tau^{\mathcal{I}}$, where $\tau^{\mathcal{I}}$ denotes the type translation of τ. As is usual with type-preserving compilation, the type $\tau^{\mathcal{I}}$ provides a simple means of expressing any compiler invariants about representation and/or layout of the transformed term $\mathrm{e_I}$ that are useful to keep track of as we compile. The second pass analogously translates I components $\mathrm{e_I}$ of type τ to T components $\mathsf{e_T}$ of type $\tau^{\mathcal{T}}$, where $\tau^{\mathcal{T}}$ is the type translation of τ.

To define the semantics of interoperability between these languages, we *embed* them all into one language, SIT, and add syntactic boundary forms between each pair of adjacent languages in the style of Matthews and Findler [33] and our own prior work [4, 45]. For instance, the term $\mathcal{IS}^{\tau}(\mathsf{e_S})$ allows an S component $\mathsf{e_S}$ of type τ to be used as an I component of type $\tau^{\mathcal{I}}$, while ${}^{\tau}\mathcal{SI}(\mathrm{e_I})$ allows an I component $\mathrm{e_I}$ of translation type $\tau^{\mathcal{I}}$ to be used as an S component of type τ. Similarly, we have boundary forms $\mathcal{TI}$ and $\mathcal{IT}$ for the next language pair. Non-adjacent languages can interact by stacking up boundaries: for example, $\mathcal{SI}(\mathcal{IT}\,\mathsf{e_T})$ (abbreviated $\mathcal{SIT}(\mathsf{e_T})$) allows a T component $\mathsf{e_T}$ to be embedded in an S term.

Design principles for multi-language system. Our goal is for the SIT interoperability semantics to give us a useful specification of when a component in one of the embedded languages should be considered equivalent to a component in another language. But how do we know if that specification is correct? There are three essential properties that the combined language must satisfy.

First, the operational semantics of SIT must be designed so that the original languages are embedded into SIT unchanged: running an SIT program that's written solely in one of the embedded languages is identical to running it in that language alone. For instance, execution of the T program $\mathsf{e_T}$ proceeds in exactly the same way whether we use the operational semantics of T or the augmented semantics for SIT. Second, the typing rules must be similarly embedded: a component that contains syntax from only one underlying language should typecheck under that language's individual type system if and only if it typechecks under SIT's type system. The final property we need is *boundary cancellation* which says that wrapping two opposite language boundaries around a component yields the same behavior as the underlying component with no boundaries: for example, any $\mathsf{e_S} : \tau$ must be contextually equivalent to ${}^{\tau}\mathcal{SI}(\mathcal{IS}^{\tau}\mathsf{e_S})$, and any $\mathrm{e_I} : \tau^{\mathcal{I}}$ must be equivalent to $\mathcal{IS}^{\tau}({}^{\tau}\mathcal{SI}\mathrm{e_I})$. Note that two components e_1 and e_2 are considered contextually equivalent in language SIT (written $e_1 \approx^{ctx}_{SIT} e_2$) if there is no well-typed SIT program context that can tell them apart.

Compiler correctness. We state the correctness criterion for our compiler as a contextual equivalence: if $\mathsf{e_S} : \tau$ compiles to $\mathrm{e_I}$, then $\mathsf{e_S} \simeq \mathrm{e_I}$, where: $\mathsf{e_S} \simeq \mathrm{e_I} \overset{\text{def}}{=} \mathsf{e_S} \approx^{ctx}_{SIT} {}^{\tau}\mathcal{SI}(\mathrm{e_I}) : \tau$ and similarly for the next pass:

If $\mathrm{e_I} : \tau$ compiles to $\mathsf{e_T}$, then $\mathrm{e_I} \simeq \mathsf{e_T}$, where: $\mathrm{e_I} \simeq \mathsf{e_T} \overset{\text{def}}{=} \mathrm{e_I} \approx^{ctx}_{SIT} {}^{\tau}\mathcal{IT}(\mathsf{e_T}) : \tau$.

Since contextual equivalence is transitive, our framework achieves vertical compositionality immediately: it is easy to combine the two correctness proofs for the individual compiler passes, to get the correctness result for the entire compiler:

If $\mathsf{e_S}$ compiles to $\mathsf{e_T}$, then $\mathsf{e_S} \approx^{ctx}_{SIT} {}^{\tau}\mathcal{SIT}(\mathsf{e_T}) : \tau$.

All of the above properties are stated as contextual equivalences but direct proofs of contextual equivalence are usually intractable. We use the standard technique of defining a logical relation for the combined language SIT that is sound and complete with respect to contextual equivalence. Defining the logical relation becomes more challenging as the demands of interoperability increase: e.g., when all the interoperating languages support type abstraction. (See the paper [45] for details.)

Reasoning about linking. Our approach enjoys a strong horizontal compositionality property: we can link with any target component $\mathsf{e'_T}$ that has an appropriate type, with no requirement that $\mathsf{e'_T}$ was produced by any particular means or from any particular source language. Specifically, if $\mathsf{e_S} : \tau' \to \tau$ expects to be linked with a component of type τ' and compiles to $\mathsf{e_T}$, then $\mathsf{e_T}$ will expect to be linked with a component of type $((\tau')^{\mathcal{I}})^{\mathcal{T}}$. If $\mathsf{e'_T}$ has this type, then using our compiler correctness theorem, we can conclude that: $(\mathsf{e_S}\ {}^{\tau'}\mathcal{SIT}(\mathsf{e'_T})) \approx^{ctx}_{SIT} \mathcal{SIT}(\mathsf{e_T}\ \mathsf{e'_T}) : \tau$.

3.2 Comparison with related work

The literature on compiler verification spans almost five decades but is mostly limited to whole-program compilation. We refer the reader to the bibliography by Dave [16] for compilation of first-order languages, and to Chlipala [15] for compilation of higher-order functional languages. Here we discuss only recent work on compositional compiler correctness.

The approach advocated by Benton and Hur [12, 13] involves formalizing correct compilation of components using a logical relation that specifies when a source term e_S semantically approximates target code e_T and vice versa (written $e_S \simeq e_T$). Using this approach, they verifed a compiler from STLC with recursion (and later from System F) to an SECD machine, proving that if source component e_S compiles to target code e_T, then e_S and e_T are logically related. Later, Hur and Dreyer [26] used essentially the same strategy to verify a compiler from an idealized ML to assembly. However, this strategy of setting up a logical relation between source and target has two serious drawbacks. First, the approach does not scale to multi-pass compilers because the source-target logical relations defined for each pass do not compose. Second, if we compile an S component e_S to e_T and then wish to link e_T with some arbitrary e'_T, the only way to check if it's okay to link with e'_T (i.e., if the compiler correctness theorem allows it) is to *come up with a source-level component* e'_S and show that $e'_S \simeq e'_T$. It may be possible to come up with e'_S when e'_T is a few lines long, but it would be infeasible when e'_T consists of hundreds of lines of assembly. Worse, the question of whether such an e'_S exists in language S is undecidable: for instance, there is no e'_S in the case of the continuation-based web server example discussed in §1. By comparison, our approach simply requires type-checking e'_T to ensure that it can be sensibly linked with t.

Neis *et al.* [41] recently developed Pilsner, a verified separate compiler from a typed, higher-order imperative language to an untyped target. Pilsner also suffers from the second drawback discussed above for the Benton-Hur and Hur-Dreyer work – put another way, Pilsner assumes that compiled code will only be linked with code compiled by a (possibly different) verified compiler from the *same source language.* Instead of a logical relation between source and target code, Pilsner uses parametric inter-language simulations (PILS) between the source and target of each compiler pass. Unlike Kripke logical relations, PILS do compose transitively and can be used to verify a multi-pass compiler.

Stewart *et al.* [51] recently reported on Compositional CompCert, a verified separate compiler for C. (This generalizes their prior work [14] which allowed shared-memory system calls from C, but could not accommodate mutually recursive inter-module dependencies.) Whereas we define a multi-language semantics for interoperability between source and target, Stewart *et al.* have devised an *interaction semantics*, a protocol-oriented operational semantics that accommodates interoperation between different languages. They have shown that CompCert's intermediate languages – all of which share the same C-like memory model – can be "plugged into" this interaction semantics. It is not clear how to extend interaction semantics to support interoperability between C and high-level, strongly typed languages like ML, or more generally, to accommodate compilers whose source and target languages use different memory models.

3.3 Our proof methodology applied to closure conversion

As an example, we show how our methodology can be used to prove compositional correctness of typed closure conversion [34, 36] for the simply-typed lambda calculus (STLC). (We elide many details; see [45].)

Step 0: Specify the source language. The source language S is call-by-value STLC with booleans. The syntax of the language is as follows:

$$\begin{array}{llll}
\textit{Types} & \sigma & ::= & \mathsf{bool} \mid \sigma_1 \rightarrow \sigma_2 \\
\textit{Values} & \mathsf{v} & ::= & \mathsf{x} \mid \mathsf{true} \mid \mathsf{false} \mid \lambda \mathsf{x} : \sigma.\, \mathsf{e} \\
\textit{Expressions} & \mathsf{e} & ::= & \mathsf{v} \mid \mathsf{if\ e\ then\ e_1\ else\ e_2} \mid \mathsf{e_1\ e_2}
\end{array}$$

Step 1. Pick appropriate target language and define translation. Closure conversion is a compiler transformation that collects a function's free variables in a tuple called a *closure environment* that is passed as an additional argument to the function, thus turning the function into a closed term. The closed function is paired with its environment to create a *closure*. The basic idea of *typed* closure conversion goes back to Minamide *et al.* [34], whom we follow in using an existential type to abstract the type of the environment. This ensures that two functions with the same type, but different free variables still have the same type after closure conversion: the abstract type hides the fact that the closures' environments have different types. Thus our target language for closure conversion must support existential types as well as tuples. It also supports multi-argument functions. We define the target language T as follows:

$$\begin{array}{llll}
\textit{Types} & \boldsymbol{\tau} & ::= & \mathbf{bool} \mid (\overline{\boldsymbol{\tau}}) \rightarrow \boldsymbol{\tau}' \mid \langle \boldsymbol{\tau}_1, \ldots, \boldsymbol{\tau}_n \rangle \mid \boldsymbol{\alpha} \mid \exists \boldsymbol{\alpha}.\, \boldsymbol{\tau} \\
\textit{Values} & \mathbf{v} & ::= & \mathbf{x} \mid \mathbf{true} \mid \mathbf{false} \mid \boldsymbol{\lambda}(\overline{\mathbf{x} : \boldsymbol{\tau}}).\, \mathbf{e} \mid \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle \mid \mathbf{pack}(\boldsymbol{\tau}, \mathbf{v})\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\, \boldsymbol{\tau}' \\
\textit{Expressions} & \mathbf{e} & ::= & \mathbf{v} \mid \mathbf{if\ e\ then\ e_1\ else\ e_2} \mid \mathbf{e}_1(\overline{\mathbf{e}}) \mid \langle \mathbf{e}_1, \ldots, \mathbf{e}_n \rangle \mid \boldsymbol{\pi}_i\, \mathbf{e} \mid \\
 & & & \mathbf{pack}(\boldsymbol{\tau}, \mathbf{e})\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\, \boldsymbol{\tau}' \mid \mathbf{unpack}(\boldsymbol{\alpha}, \mathbf{x}) = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2
\end{array}$$

The typing rules are standard (with judgments $\boldsymbol{\Delta}; \boldsymbol{\Gamma} \vdash \mathbf{e} : \boldsymbol{\tau}$), with one exception: since language T is the target of closure conversion, we ensure via the function typing rule that functions contain no free term variables. Thus, the typing rule for functions is as follows:

$$\frac{\cdot; \overline{\mathbf{x} : \boldsymbol{\tau}} \vdash \mathbf{e} : \boldsymbol{\tau}'}{\boldsymbol{\Delta}; \boldsymbol{\Gamma} \vdash \boldsymbol{\lambda}(\overline{\mathbf{x} : \boldsymbol{\tau}}).\, \mathbf{e} : (\overline{\boldsymbol{\tau}}) \rightarrow \boldsymbol{\tau}'}$$

$$\begin{array}{llll}
\textit{Types} & \varphi & ::= & \sigma \mid \boldsymbol{\tau} \\
\textit{Terms} & \mathsf{e} & ::= & \ldots \mid {}^{\sigma}\mathcal{ST}\,\mathbf{e} \\
 & \mathbf{e} & ::= & \ldots \mid \mathcal{TS}^{\sigma}\,\mathsf{e} \\
 & e & ::= & \mathsf{e} \mid \mathbf{e} \\
\textit{Values} & v & ::= & \mathsf{v} \mid \mathbf{v}
\end{array}
\qquad
\begin{array}{llll}
\textit{Type Environments} & \Delta & ::= & \cdot \mid \Delta, \boldsymbol{\alpha} \\
\textit{Value Environments} & \Gamma & ::= & \cdot \mid \Gamma, \mathsf{x} : \sigma \mid \Gamma, \mathbf{x} : \boldsymbol{\tau}
\end{array}$$

$$\boxed{\Delta; \Gamma \vdash e : \varphi}$$

$$\ldots \quad \frac{\Delta; \Gamma \vdash \mathbf{e} : \sigma^{+}}{\Delta; \Gamma \vdash {}^{\sigma}\mathcal{ST}\,\mathbf{e} : \sigma} \quad \frac{\Delta; \Gamma \vdash \mathsf{e} : \sigma}{\Delta; \Gamma \vdash \mathcal{TS}^{\sigma}\,\mathsf{e} : \sigma^{+}}$$

$$\boxed{e \longmapsto e'}$$

$$\begin{array}{rcl}
{}^{\mathsf{bool}}\mathcal{ST}\,\mathbf{true} & \longmapsto & \mathsf{true} \\
{}^{\mathsf{bool}}\mathcal{ST}\,\mathbf{false} & \longmapsto & \mathsf{false} \\
{}^{\sigma_1 \to \sigma_2}\mathcal{ST}\,\mathbf{v} & \longmapsto & \lambda \mathsf{x} : \sigma_1.\, {}^{\sigma_2}\mathcal{ST}\,(\mathbf{unpack}(\boldsymbol{\alpha}, \mathbf{y}) = \mathbf{v}\ \mathbf{in}\ \boldsymbol{\pi_1}\,\mathbf{y}\ (\boldsymbol{\pi_2}\,\mathbf{y}, \mathcal{TS}^{\sigma_1}\,\mathsf{x})) \\
 & & \\
\mathcal{TS}^{\mathsf{bool}}\,\mathsf{true} & \longmapsto & \mathbf{true} \\
\mathcal{TS}^{\mathsf{bool}}\,\mathsf{false} & \longmapsto & \mathbf{false} \\
\mathcal{TS}^{\sigma_1 \to \sigma_2}\,\mathsf{v} & \longmapsto & \mathbf{pack}(\langle\rangle, \langle \mathbf{v}, \langle\rangle\rangle)\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\langle\langle \boldsymbol{\alpha}, {\sigma_1}^{+}\rangle \to {\sigma_2}^{+}, \boldsymbol{\alpha}\rangle \\
 & & \quad \text{where } \mathbf{v} = \boldsymbol{\lambda}(\mathbf{z} : \langle\rangle, \mathbf{x} : {\sigma_1}^{+}).\mathcal{TS}^{\sigma_2}\,(\mathsf{v}\ {}^{\sigma_1}\mathcal{ST}\,\mathbf{x})
\end{array}$$

Figure 2 $\mathcal{ST}$: Extensions to S and T syntax, static semantics, dynamic semantics.

Closure conversion maps source terms of type σ to target terms of type σ^{+}. The definition of the type translation (σ^{+}) and environment translation (Γ^{+}) is as follows:

$$\begin{array}{rcl}
(\mathsf{bool})^{+} & = & \mathbf{bool} \\
(\sigma_1 \to \sigma_2)^{+} & = & \exists \boldsymbol{\alpha}.\, \langle\langle \boldsymbol{\alpha}, {\sigma_1}^{+}\rangle \to {\sigma_2}^{+}, \boldsymbol{\alpha}\rangle
\end{array}
\qquad
\begin{array}{rcl}
(\cdot)^{+} & = & \cdot \\
(\Gamma, \mathsf{x} : \sigma)^{+} & = & \Gamma^{+}, \mathbf{x} : \sigma^{+}
\end{array}$$

We then define a type-directed term translation $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{e}$ that translates a term e such that $\Gamma \vdash \mathsf{e} : \sigma$ into a target term $\mathbf{e}$ such that $\cdot; \Gamma^{+} \vdash \mathbf{e} : \sigma^{+}$. For instance, x translates to $\mathbf{x}$ and below we show how functions are translated into closures. (We elide the rest of the translation as it is standard, e.g., see [34, 36, 45].)

$$\frac{\begin{array}{ccc}
\mathsf{y}_1, \ldots, \mathsf{y}_m = \text{free-vars}(\lambda \mathsf{x} : \sigma.\, \mathsf{e}) & \Gamma \vdash \mathsf{y}_1 : \sigma_1 \ldots \Gamma \vdash \mathsf{y}_m : \sigma_m & \boldsymbol{\tau}_{env} = \langle {\sigma_1}^{+}, \ldots, {\sigma_m}^{+} \rangle \\
\Gamma, \mathsf{x} : \sigma \vdash \mathsf{e} : \sigma' \rightsquigarrow \mathbf{e} & \multicolumn{2}{c}{\mathbf{v} = \boldsymbol{\lambda}(\mathbf{z} : \boldsymbol{\tau}_{env}, \mathbf{x} : \sigma^{+}).\, \mathbf{e}[\boldsymbol{\pi_1}\,\mathbf{z}/\mathbf{y}_1] \ldots [\boldsymbol{\pi_m}\,\mathbf{z}/\mathbf{y}_m]}
\end{array}}{\Gamma \vdash \lambda \mathsf{x} : \sigma.\, \mathsf{e} : \sigma \to \sigma' \rightsquigarrow \mathbf{pack}(\boldsymbol{\tau}_{env}, \langle \mathbf{v}, \langle \mathbf{y}_1, \ldots, \mathbf{y}_m \rangle\rangle)\ \mathbf{as}\ \exists \boldsymbol{\alpha}.\, \langle\langle \boldsymbol{\alpha}, \sigma^{+}\rangle \to \sigma'^{+}, \boldsymbol{\alpha}\rangle}$$

Step 2: Define interoperability semantics. The language $\mathcal{ST}$ embeds the languages S and T so that both languages have natural access to foreign values (i.e., values from the other language). They receive foreign boolean values as native values, and can call foreign functions as native functions. We extend the original core languages with boundary terms ${}^{\sigma}\mathcal{ST}\,\mathbf{e}$ and $\mathcal{TS}^{\sigma}\,\mathsf{e}$ which mediate between the types σ on the source side and σ^{+} on the target side. Figure 2 presents the new syntax, typing rules and reduction rules. Typing judgments for $\mathcal{ST}$ have the form $\Delta; \Gamma \vdash e : \varphi$ where the environment Γ now tracks both source variables of type σ and target variables of type $\boldsymbol{\tau}$. The typing rules include all the S typing rules, but augmented with the additional environment Δ; all the T typing rules, unchanged; and rules for the two boundary constructs, shown in Figure 2.

We evaluate under a boundary until we have a value. The reduction rules for boundaries annotated bool convert boolean values from one language to the other. To convert functions across languages, we use native proxy functions. We represent a target function $\mathbf{v}$ in the source at type $\sigma_1 \to \sigma_2$ by a new function that takes an argument of type σ_1 and first translates this argument from source to target, then unpacks the closure $\mathbf{v}$ and applies the code to its environment and the translated argument, and finally translates the result back to source at type σ_2. Notice that the direction of the conversion (and the boundary used) reverses for function arguments.

Converting source functions to target functions is a bit more subtle. To represent a source function v in the target at type $(\sigma_1 \rightarrow \sigma_2)^+$ we have to construct a closure. Since these are reduction rules, and since we only run closed programs, we know that v is closed. Hence, we simply use an empty tuple type $\langle\rangle$ for the closure environment. The underlying function $\mathbf{v}$ for this closure takes an environment of type $\langle\rangle$ and an argument of type $\sigma_1{}^+$, translates the argument to source, applies v to the translated argument, and finally translates the result back to target. The use of an empty environment in this reduction rule illustrates the difference between the translation done by boundaries in the multi-language and that done by the compiler itself.

Step 3: Define logical relation for combined language ($\approx^{log}_{ST}$) and prove $\approx^{log}_{ST} \equiv \approx^{ctx}_{ST}$. Next, we define a logical relation for the combined language ST and prove it sound and complete with respect to contextual equivalence ($\approx^{ctx}_{ST}$). In the process, we must prove the *boundary cancellation* property described in §3.1 – that is, this is the stage at which we are required to prove that the combined language can sensibly serve as an interoperability semantics.

Step 4: Prove translation is semantics preserving.
If $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \mathbf{e}$ then $\cdot; \Gamma \vdash \mathsf{e} \approx^{log}_{ST} {}^{\sigma}\mathcal{ST}\, \mathbf{e'} : \sigma$, where $\mathbf{e'}$ is $\mathbf{e}$ with all $\mathbf{x}$ that are translations of $\mathsf{x} : \sigma' \in \Gamma$ replaced with $\mathcal{TS}^{\sigma'}\, \mathsf{x}$. Since $\approx^{log}_{ST}$ exactly captures $\approx^{ctx}_{ST}$, this lemma immediately yields the compiler correctness theorem we want.

4 Research Plan and Central Challenges

We plan to carry out this work in three stages, with the target language GTVM growing in features and functionality across the three stages. Below we discuss the main tasks and the challenges we anticipate, and describe some of the work done to date.

Verified compiler: ML to TTVM. In the first phase, we plan to develop a verified compiler from an idealized ML to a statically type-safe LLVM IR. The LLVM IR is a platform-independent, static single assignment (SSA) language [54]. An LLVM compiler normally translates a high-level language into LLVM IR. This can then be optimized using a series of IR to IR transformations. LLVM provides a collection of such transformations which perform optimizations and static analyses. The resulting LLVM IR can then be translated to a target architecture using LLVM's code generator or JIT-compiler.

LLVM programs consist of modules, which in turn contain function definitions and declarations. A function consists of a sequence of basic blocks, which as usual are a sequence of commands ending with a branch or return (`ret`) instruction. Commands include `load`, `store`, `malloc`, and `free` instructions; `alloca` for stack allocation; and a function `call` instruction. To be well formed, an LLVM program must be in valid SSA form. All components in the language are annotated with types, but the LLVM IR is not a type-safe language – like C, it allows arbitrary casts, invalid memory access, and so on.

Vellvm [57] provides a mechanized formal semantics of LLVM's IR, its static semantics, and SSA form, all formalized in Coq, as well as a proof of static safety (modulo reaching known stuck states) via preservation and progress theorems. It says that if the program takes a step then it continues to be well formed SSA; and if the program is well formed then either it can take a step or it is in one of the defined set of stuck states. Vellvm also provides a set of tools to extract LLVM IR from Coq so it can be processed by the standard LLVM tools. Vellvm provides us with a useful starting point; without the Vellvm formalization our research plans would not be feasible.

We will first identify a statically type-safe subset of the LLVM IR (TTVM) and modify Vellvm's static safety theorems so that the progress lemma holds without the possibility of a well formed program configuration being stuck. This will require eliminating arbitrary casts, memory deallocation (`free`), and anything that leads to undefined behavior (`undef`) from the language. We will then extend the type system with polymorphism, existential types, and any other extensions needed to do type-preserving compilation from our idealized ML (language M) to TTVM (language T).

Tentatively, the compiler will consist of four languages and three passes: a closure conversion pass from M to C, an explicit allocation pass (where the data representation strategy is made explicit) from C to A, and code generation pass from A to T. To state compiler correctness, we will embed all four of the compiler's languages into a combined language $MCAT$ by defining interoperability between the adjacent languages in the compilation pipeline. The design of the interoperability semantics between M and C (for the closure conversion pass) and between C and A (for the explicit allocation pass) is already well understood and we can adapt the multi-language and logical relation from our recent work [45] which covers closure conversion and explicit allocation for System F with recursive types. Moreover, in recent work with Phillip Mates and James Perconti, we have already proved compositional correctness of closure conversion in the presence of ML-style mutable references. The presence of mutable references required a novel extension to our logical relation for the multi-language system, which we plan to report on in the near future.

We anticipate that the design of an interoperability semantics between language A and TTVM (i.e., for the code generation pass) will be the most challenging. In the language A, code still has a compositional structure even though tuples and closures are allocated on the heap – that is, a *component* is a simply a term e_A. However, at the TTVM level, that compositional structure is lost. To define interoperability between A components and T (TTVM) components, we first have to identify what exactly constitutes a TTVM *component* – that is, since there are no "terms" in TTVM, what is the shape of an e_T that we can put under a boundary $\mathcal{AT}e_T$?

Fortunately, in preliminary work with Perconti, we have already answered this question in the context of an idealized typed assembly language (TAL), which is even lower level that TTVM. For the purpose of the multi-language semantics, a TAL (or TTVM) *component* is comprised of a number of basic blocks. Thus, e_T denotes a pair $(b_0, \overline{b})$ of the currently executing basic block b_0 and the rest of the blocks $\overline{b}$ that comprise that component (which corresponds now to a TTVM function body). The next question is how do we run the term $\mathcal{AT}e_T$? As in §3, we want to run e_T until we have a value v_T and then convert that value to the language A. But running the e_T in TTVM will ultimately end with a return instruction. How do we distinguish between a normal *return within TTVM* from a *return to language A*? The solution is to introduce a special `ret-to-A` pseudo-instruction as part of the extensions we make when defining the multi-language semantics. When $\mathcal{AT}e_T$ has reduced to $\mathcal{AT}$(`ret-to-A` v_T), we simply convert v_T to an A value in the usual type-directed manner. We are reasonably confident that we will be able to use ideas from our TAL work to design interoperability between A and TTVM. We do not yet know if LLVM's SSA form will complicate matters.

GTVM: a gradually type-safe LLVM IR. In the second phase, we aim to extend TTVM with support for dynamically type-safe code, assigned type `dyn`, and type-unsafe code, assigned type `un`. There is a significant body of work on contracts and gradual typing for high-level languages [19, 49, 50, 32, 6, 17, 23, 53, 44, 20, 25, 58, 24, 43, 11] that we can leverage when designing GTVM. The recent work on TS*, a gradual type system for JavaScript [52], deserves special mention as it also mixes static, dynamic, *and unsafe* types (though their dynamic type is called `any`).

As is usual in gradual type systems, interactions between type safe code and unsafe code must be protected by wrappers (dynamically checked contracts). However, note that unsafe code is just standard LLVM IR that may, for instance, be the output of a C program. To prevent raw LLVM IR (or raw C) from breaking internal invariants of statically type safe code (say from ML or Scheme), we need to ensure that C code cannot trample memory cells that belong to the type-safe parts of the language. This is one of the most significant challenges we face, but there are ideas that we can draw upon from the literature. One option is to use some sort of software-based fault isolation (as in Google's Native Client, NaCl [60]) to ensure that the unsafe code adheres to a strict *sandbox* policy, though we expect this to be too coarse-grained. Another option is to investigate whether we can devise wrappers for GTVM similar to those used by TS* (adopted from Fournet *et al.* [22]), which enforce a strict heap separation between unsafe and type-safe code. A third option is to devise contracts based on separation logic predicates similar to the approach of Agten *et al.* [1], who use these contracts to protect verified C modules from unverified C code, though this approach comes with considerable performance overhead.

As mentioned earlier, we plan to develop a compiler from GTVM to LLVM IR that inserts wrappers or safe coercions to ensure safe interoperability is preserved after the translation to LLVM IR. TS* similarly provides a compiler to JavaScript. To prove that the translation from TS* to JavaScript preserves properties such as memory isolation, the authors leverage a dependently typed version of JavaScript (called JS*) in which memory layout invariants can be specified. We conjecture that we should be able to carry out such a proof using our logical relation for GTVM which will be based on much recent progress in scaling up logical relations to so that they can be used to tractably prove sophisticated equivalences in the presence of state [2, 5, 18, 56].

Verified compiler: Rust to GTVM. In the third phase, we plan to extend the GTVM/TTVM type system so that it can support type-preserving compilation from Rust, in particular, adding features capable of expressing Rust's region and ownership discipline. For the design of the TTVM type system and logical relation, we can leverage ideas from our prior work on linear and affine type systems for memory management [10, 9, 35, 7, 21, 8]. However, instead of trying to shoehorn an appropriate notion of affine types or capabilities into TTVM, it may be better to extend the TTVM type system with dependent types in the style of HTT [38, 40]. This is a more challenging task, but a type system based on HTT might be a better choice as it can be designed independent of Rust considerations and yet should be expressive enough to serve as a target of typed compilation from Rust.

5 Conclusion

Practically every software system, from safety-critical software to web browsers, uses components written in multiple programming languages and stands to benefit from verified compositional compilers. We have proposed a proof architecture for verifying compositional compilers based on source-target interoperability and are working on demonstrating the viability of this approach. We also propose to extend LLVM – increasingly the backend of choice for modern compilers – to support compositional compilation from type-safe source languages and principled linking with less precisely typed languages. This ambitious research program consists of numerous technical challenges and we look forward to collaborating with other groups in the community on various facets of this project.

Acknowledgements. We thank the SNAPL anonymous reviewers, Andrew Appel, Matthias Blume, Matthias Felleisen, Robby Findler, Bob Harper, Xavier Leroy, Guy McCusker, Greg Morrisett, Aaron Turon, and members of IFIP WG 2.8 (Working Group on Functional Programming) for valuable feedback on various aspects of this project. Jamie Perconti and Phillip Mates did the bulk of the work completed to date. Current project members include William Bowman, Max New, and Nick Rioux. This research is supported in part by the NSF (grants CCF-1453796, CCF-1422133, and CCF-1203008) and a Google Faculty Research Award.

References

1 Pieter Agten, Bart Jacobs, and Frank Piessens. Sound modular verification of c code executing in an unverified context. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, January 2015.

2 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP)*, pages 69–83, March 2006.

3 Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*, pages 157–168, September 2008.

4 Amal Ahmed and Matthias Blume. An equivalence-preserving CPS translation via multi-language semantics. In *International Conference on Functional Programming (ICFP), Tokyo, Japan*, pages 431–444, September 2011.

5 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *ACM Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*, January 2009.

6 Amal Ahmed, Robert Bruce Findler, Jeremy Siek, and Philip Wadler. Blame for all. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 201–214, January 2011.

7 Amal Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In *International Conference on Functional Programming (ICFP), Tallinn, Estonia*, pages 78–91, September 2005.

8 Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3 : A linear language with locations. *Fundamenta Informaticae*, 77(4):397–449, June 2007.

9 Amal Ahmed, Limin Jia, and David Walker. Reasoning about hierarchical storage. In *IEEE Symposium on Logic in Computer Science (LICS), Ottawa, Canada*, pages 33–44, June 2003.

10 Amal Ahmed and David Walker. The logical approach to stack typing. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, pages 74–85, January 2003.

11 João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In *European Symposium on Programming (ESOP)*, March 2011.

12 Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *International Conference on Functional Programming (ICFP), Edinburgh, Scotland*, September 2009.

13 Nick Benton and Chung-Kil Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, April 2010.

14 Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. Verified compilation for shared-memory C. In *European Symposium on Programming (ESOP)*, April 2014.

15 Adam Chlipala. A verified compiler for an impure functional language. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, January 2010.

16 Maulik A. Dave. Compiler verification: A bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6), 2003.

17 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *European Symposium on Programming (ESOP)*, March 2012.

18 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4&5):477–528, 2012.

19 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP), Pittsburgh, Pennsylvania*, pages 48–59, September 2002.

20 Cormac Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2006.

21 Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *European Symposium on Programming (ESOP)*, pages 7–21, March 2006.

22 Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 371–384, 2013.

23 Kathryn E Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2005.

24 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, pages 353–364, January 2010.

25 Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop (Scheme)*, pages 93–104, September 2006.

26 Chung-Kil Hur and Derek Dreyer. A Kripke logical relation between ML and assembly. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, January 2011.

27 Shriram Krishnamurthi, Peter Walton Hopkins, Jay Mccarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.

28 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML : A verified implementation of ML. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, January 2014.

29 Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, January 2006.

30 Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

31 Andreas Lochbihler. Verifying a compiler for Java threads. In *European Symposium on Programming (ESOP)*, March 2010.

32 Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *European Symposium on Programming (ESOP)*, pages 16–31, March 2008.

33 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *ACM Symposium on Principles of Programming Languages (POPL), Nice, France*, pages 3–10, January 2007.

34 Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 271–283, January 1996.

35 Greg Morrisett, Amal Ahmed, and Matthew Fluet. L3 : A linear language with locations. In *Typed Lambda Calculi and Applications (TLCA), Nara, Japan*, pages 293–307, April 2005.

36 Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

37 Magnus O. Myreen. Verified just-in-time compiler on x86. In *ACM Symposium on Principles of Programming Languages (POPL), Madrid, Spain*, January 2010.

38 Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare Type Theory. In *European Symposium on Programming (ESOP)*, pages 189–204, March 2007.

39 Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, pages 165–179, 2011.

40 Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming (ICFP), Victoria, British Columbia, Canada*, pages 229–240, 2006.

41 Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. Available at: `http://www.mpi-sws.org/∼dreyer/papers/pilsner/paper.pdf`, February 2015.

42 Ocsigen. `http://ocsigen.org`.

43 Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *Programming Languages meets Program Verification (PLPV)*, January 2012.

44 Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, August 2004.

45 James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming (ESOP)*, April 2014.

46 Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, February 2003.

47 Seaside. `http://seaside.st`.

48 Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, 2011.

49 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (Scheme)*, pages 81–92, September 2006.

50 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007.

51 Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, 2015.

52 Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. Gradual typing embedded securely in JavaScript. In *ACM Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 425–438, 2014.

53 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2012.

54 The LLVM Development Team. The LLVM reference manual. `http://llvm.org/docs/LangRef.html`.

55 Jesse Tov. Stateful contracts for affine types. In *European Symposium on Programming (ESOP)*, March 2010.

56 Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *ACM Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 201–214, January 2013.

57 Vellvm: Verifying the llvm. `http://www.cis.upenn.edu/~stevez/vellvm/`.

58 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, March 2009.

59 Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 2014.

60 Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

61 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *ACM Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*, January 2012.

62 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, Washington*, June 2013.

Growing a Software Language for Hardware Design

Joshua Auerbach, David F. Bacon, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla

IBM Thomas J. Watson Research Center
Yorktown Heights, NY, US

Abstract

The Liquid Metal project at IBM Research aimed to design and implement a new programming language called Lime to address some of the challenges posed by heterogeneous systems. Lime is a Java-compatible programming language with features designed to facilitate high level synthesis to hardware (FPGAs). This article reviews the language design from the outset, and highlights some of the earliest design decisions. We also describe how these decisions were revised recently to accommodate important requirements that arise in networking and cryptography.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, B.5.2 Design Aids

Keywords and phrases Heterogeneous Systems, FPGA, High Level Synthesis, Dataflow, Functional Programming, Streaming, Java

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.32

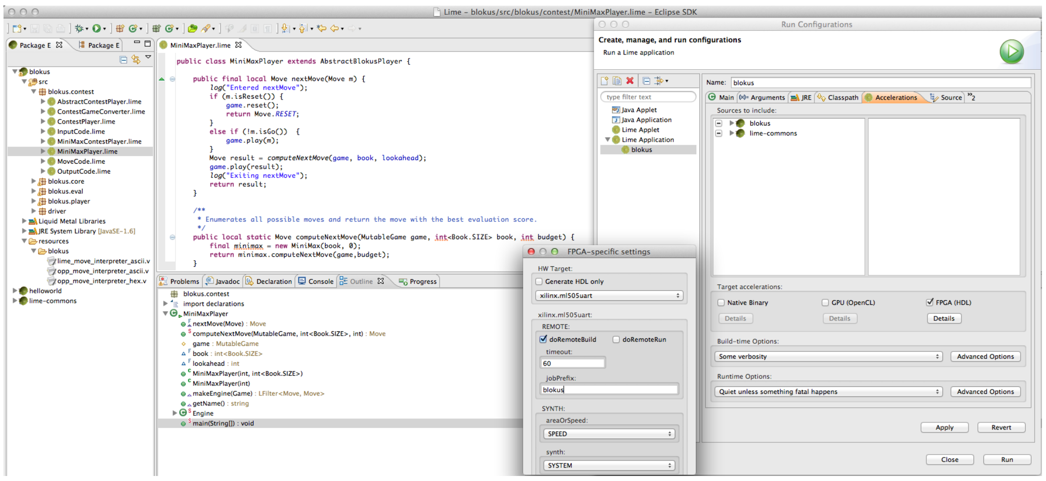
1 Introduction

Over the coming decade, the mix of computational elements that comprise a "computer" will grow increasingly heterogeneous. Already many systems exploit both general CPUs and graphics processors (GPUs) for general purpose computing. The ubiquity of GPUs, and their efficiency for certain computational problems, make them attractive hardware accelerators. Other forms of hardware accelerators include field programmable gate arrays (FPGAs) and fixed-function accelerators for cryptography, XML parsing, regular expression matching, and physics engines.

Although programming technologies exist for CPUs, GPUs, FPGAs, and various accelerators in isolation, current programming methods suffer from at least three major deficiencies. First, disparate architectures must be programmed with disparate languages or dialects, thus challenging a single programmer or programming team to work equally well on all aspects of a project. Second, current solutions provide relatively little attention to *co-execution*, the problem of orchestrating distinct computations on different architectures that must work seamlessly together. Third, current systems force an early static decision as to what will execute where, a decision that is costly to revisit as a project evolves. This is exacerbated by the fact that some of the accelerators, especially FPGAs, are notoriously difficult to program well and place a heavy engineering burden on programmers.

2 One Language, or Three

We designed Lime as a programming language for systems which connect a *host* general purpose processor (CPU) to a specialized *device* such as a GPU or FPGA. In general, the system may encompass more than one CPU, GPU or FPGA.



```
public class MiniMaxPlayer extends AbstractBlokusPlayer {

    public final local Move nextMove(Move m) {
        log("Entered nextMove");
        if (m.isReset()) {
            game.reset();
            return Move.RESET;
        }
        else if (!m.isGo())  {
            game.play(m);
        }
        Move result = computeNextMove(game, book, lookahead);
        game.play(result);
        log("Exiting nextMove");
        return result;
    }

    /**
     * Enumerates all possible moves and return the move with the best evaluation score.
     */
    public local static Move computeNextMove(MutableGame game, int<Book.SIZE> book, int budget) {
        final minimax = new MiniMax(book, 0);
        return minimax.computeNextMove(game,budget);
    }
```

```
blokus.contest
import declarations
MiniMaxPlayer
  nextMove(Move) : Move
  computeNextMove(MutableGame, int<Book.SIZE>, int) : Move
  game : MutableGame
  book : int<Book.SIZE>
  lookahead : int
  MiniMaxPlayer(int, int<Book.SIZE>)
  MiniMaxPlayer(int)
  makeEngine(Game) : LFilter<Move, Move>
  getName() : string
  Engine
  main(String[]) : void
```

Create, manage, and run configurations
Run a Lime application

Name: blokus

Sources to include:
- blokus
- lime-commons

Target accelerations:
[ ] Native Binary [ ] GPU (OpenCL) [✓ FPGA (HDL)]

Build-time Options: Some verbosity

Runtime Options: Quiet unless something fatal happens

HW Target:
[ ] Generate HDL only
xilinx.ml505uart

xilinx.ml505uart:
REMOTE:
[✓ doRemoteBuild] [ ] doRemoteRun
timeout: 60
jobPrefix: blokus

SYNTH:
areaOrSpeed: SPEED
synth: SYSTEM

Figure 1 The Lime IDE is an Eclipse plugin available at `http://lime.mybluemix.net`.

Consider a program as a collection of components, some of which run on the CPU, and some which run on the device. Clearly we need to describe the host program, the device program, and how to coordinate these two components and their communication. Lime allows a developer to program all three aspects using a single language and semantic rules which are neither host or device specific. This is convenient for the developer, provided the language implementation can deliver reasonable performance compared to using a plurality of languages.

Central to the Lime programming language lies the notion of dataflow graphs, where nodes encapsulate computation and edges describe data communication between nodes. Lime provides features for constructing planar and irreducible dataflow graphs which we call *task graphs*.

A task graph may execute entirely on the host or entirely on the device. Alternatively, the graph may be partitioned into subgraphs, some of which run on the host and some on the device. Thus the nodes in the graph can run on the parts of the system they are best suited for. Moreover, it may be desirable to migrate nodes between processors, in response to run time characteristics.

Lime empowers a skilled programmer to develop an application, whose parts are intended to run on different processors in the system, using a single language and semantic domain. Instead of writing code in multiple languages and using different tools to compile and assemble the source for each processor in the system, Lime makes it possible to program using a single language, a single compilation toolchain, and a single virtual runtime to execute the application on the heterogeneous system (Figure 1).

We did not always think of Lime as consisting of three languages – the host and device languages, and the task graph language – but this has proven increasingly helpful as we think about applying the salient features of Lime to other languages.

3 Salient Features

We highlight some of the Lime language features using the example code in Figure 2. The Figure shows a valid Lime program that is also a valid Java program. The class `HelloWorld` defines a method `getChar` to read a character from standard input, and a method `toLowerCase` to convert a character to its lowercase counterpart. The main method

```
 1 public class HelloWorld {
 2    public static void main(String[] args) {
 3       while (true) {
 4          System.out.print(
 5               toLowerCase(
 6                   getChar()));
 7       }
 8    }
 9    static char getChar() {
10       try {
11          return (char) System.in.read();
12       } catch (Exception e) {
13          return '\0';
14       }
15    }
16    static char toLowerCase(char c) {
17       return ('A' <= c && c <= 'Z') ?
18              (char) (c + 'a' - 'A') : c;
19    }
20 }
```

Figure 2 Lime example.

```
   public static void main(String[] args) {
      var tg =  task getChar
             => task toLowerCase
             => task System.out.print(char);
      var te = tg.create();
      te.start();
      te.finish();
   }
```

Figure 3 Task graph example.

composes `getChar`, `toLowerCase` and `System.out.print` to display the lowercased input characters to standard output.

The function composition on lines 4–6 describes a *pipeline*. Pipelining provides opportunities for parallelizing, where different stages of the pipeline may overlap in time to increase throughput. When composing functions in imperative or functional languages, the first stage of the pipeline is most deeply nested and appears last in an expression. Similarly, the last stage of the pipeline appears first. In contrast, task graph construction in Lime allows a more natural description of pipelines: left to right, and top to bottom.

The snippet in Figure 3 shows a Lime task graph that is equivalent to the function composition in Figure 2. Here we use the `task` and `=>` (connect) operators, and also demonstrate the use of local type inference using the `var` keyword. The task operator is applied to the same methods used in the previous example to create the tasks that make up the task graph[1]. This is akin to *lifting* in functional reactive programming [8]. In contrast to the main method in Figure 2, the pipeline is now more readily apparent. The design of the task graph language was initially influenced by the StreamIt programming language [13], although unlike StreamIt, task graph construction in Lime may describe richer graph topologies, including non-planar graphs.

A task consumes data from an input channel (if any), applies the method, and writes its results to an output channel (if any). The connect operator abstracts communication between

[1] A modifier is needed on the declaration of `toLowerCase` in order to apply the `task` operator. This is described later.

nodes in the task graph. If the entire graph runs on the host (device), then all communication is through the host (device) memory. However if say `getChar` and `System.out.print` run on the host, but `toLowerCase` runs on the device, the connect operator implies communication between the host and device memories.

The programmer communicates their intent for host and device partitioning using *relocation brackets* around individual tasks as in `([ task toLowerCase ])`, or task graphs in general [2]. We introduced this feature into the language because partitioning task graphs automatically between host and device presents too many practical and technical challenges. To understand why, note that Lime separates the graph construction (lines 2-4) from instantiation and execution (lines 5–7). This separation allows us to statically elaborate task graphs during compilation which is necessary for synthesis of hardware circuits. The hardware synthesis tools are time consuming to run, that even for this simple example it may take several minutes before a *bitfile* is produced for an FPGA. The bitfile is used to program the FPGA and realize the desired circuit. It is common to spend tens of minutes or hours in hardware synthesis for substantial programs.

In order to statically elaborate task graphs for hardware synthesis, while also using the expressive power of an imperative and object-oriented language for constructing task graphs (see [2] for examples), the Lime compiler statically checks the task graph construction for *repeatable* shape which guarantees it can extract the static structure needed for hardware synthesis. By repeatable, we mean compile-time constant but not limited to primitives [3].

The relocation brackets allow a program to constrain repeatable checks to only relevant parts of the program. Further, they facilitate gentle migration between host and device code since the programmer can move the brackets around in their code during development and testing. The feature is also useful for writing libraries because relocation brackets may be nested and composed.

The methods shown in the example may be used either in an imperative context or in a task graph context. In the former, the method is invoked explicitly by the programmer. In the latter, the task graph runtime invokes the methods repeatedly until one of a number of termination conditions is met. The two ways of using methods is convenient because the programmer can reuse existing code where necessary, and perform unit testing without invoking the task graph machinery. The dual-uses also informs the compiler of important characteristics that may be used in optimizing the implementation of a task graph. Specifically, the compiler will know the size of the input arguments and return values, and can infer an *I/O rate*. As an example, `toLowerCase` consumes one character and produces one character per invocation. Hence the compiler can use this information much in the same way that rate information is used to optimize synchronous dataflow programs [9].

More recently, we generalized the task programming model in Lime to permit the expression of tasks that directly manipulate input and output channels. In doing so, we increased the power of the language so that it is more natural to write tasks that consume or produce a variable number of values per invocation, such as those that arise in compression and networking applications. One may rewrite the `toLowerCase` method using an explicit `In` channel, an `Out` channel, or both as shown in Figure 4.

Note the use of the `local` keyword. This method modifier prohibits access to mutable global state, and hence limits side-effects of a method to those reachable through its input arguments. The local modifier is required on methods that have both input arguments and return values if they are to be lifted to task nodes. This is motivated by the desire to limit and confine side-effects for internal nodes in a task graph because they are usually migrated from the host to the device.

```
// local modifier prohibits access to mutable global data
static local char toLowerCase(char c) {
     return ('A' <= c && c <= 'Z') ?  (char) (c + 'a' - 'A') : c;
}

// using an In channel to read input characters
static local char toLowerCase(In<char> in) {
   var c = in.get();
   return toLowerCase(c);
}

// using an Out channel to return results
static local void toLowerCase(char c, Out<char> out) {
   out.put(toLowerCase(c));
}

// using both In and Out channels to read input characters and return results
static local void toLowerCase(In<char> in, Out<char> out) {
   var c = in.get();
   out.put(toLowerCase(c));
}
```

Figure 4 Task example using explicit channels.

In combination with *value* typed arguments, which are immutable types, the compiler can determine if a static method is pure in the functional programming sense. Primitive types in Lime are examples of values types. The language also provides a `value` keyword to modify class definitions and impart value semantics on a type declaration. The local modifier may apply to class constructors. This in turn permits the application of the task operator to instance methods of objects (rather than static methods), thereby creating tasks that may retain state across invocations.

All four versions of `toLowerCase` in Figure 4 are semantically identical when the task operator is applied. They vary only in the amount of rate of information they communicate to the compiler. In general, tasks that use explicit channel operations offer weaker compile-time guarantees about deadlock freedom because this style of programming inherits well-known problems attributed to Kahn Process Networks (KPN).

We further generalized the Lime task model to include non-determinism. This was motivated by functional requirements in networking applications where one may need to implement a timely merge operation. In our pursuit of a relaxed task model, we considered various relaxations of KPN to include non-determinism (e.g., [5]) as well as more general process models such as CSP [7] and Actors [1]. Most of these are broader than our needs for FPGA programming purposes. A summary of relevant literature by Lee and Parks [10] lists five ways of relaxing KPN to include non-determinism. One of these is to add the ability to test input channels for readiness. The analysis shows that this single relaxation is expressively equivalent to several others. Since any one relaxation is sufficient, we preferred the input testing relaxation because we found it to be the simplest to incorporate into our programming language.

The relaxation of the task model from its initial inception implied we could elide some of the specialized tasks in Lime in favor of library implementations. There is a trade-off however in that library functions in general may not have intrinsic guarantees and place a greater burden on the compiler to analyze their behavior for the sake of optimization.

There are many more features of the language that are not described here. We refer the interested reader to the Lime language manual [12]. The features described here highlight some of the earliest and latest design choices of the language.

4 Language Interoperability

Lime is an extension of the Java programming language and aimed for interoperability with Java from the outset, so that we may reuse existing ecosystems and libraries. Inter-language operability also made it possible to apply a gentle migration strategy to port existing Java code to Lime. In our own work, we quite often applied this strategy by starting with working Java source programs, and introducing Lime features to realize the promise of hardware synthesis.

Java was a logical choice because it is supported by mature development environments and programming tools, it executes in a managed runtime, is often optimized with a just-in-time (JIT) compiler. We thought we would get to the point of "JITing the hardware" – that is, the Lime virtual machine and JIT can exploit runtime information to further specialize the generated hardware designs, or to dynamically migrate code through a heterogeneous system in order to run code where it is best suited at a particular time.

Interoperability between Lime and Java presented a number advantages, but also some challenges particularly in the context of generics and arrays.

- The Lime generic type system does not elide all types by erasure because doing so implies the hardware backend cannot specialize the control and data paths where appropriate. In hardware design, better performance is achieved through specificity rather than genericity. Lime generic types do not attempt to be fully compatible with existing Java generics. Rather, the Lime programmer must mark particular generic types as Java-compatible where desired.
- Arrays are a important data structure in Lime, as in many programming languages. Lime revises array semantics compared to Java with two goals in mind. First, it is possible to declare arrays that are immutable (as in `int[[]]` is a read only array of integers). Second, one may establish static bounds on the sizes of arrays (as in `int[5]` or `int[[5]]` for mutable and immutable integer arrays with five elements). Array bounds guarantee that array indexing exceptions will not happen. This in turn permits simpler and more efficient hardware design. It is not feasible to achieve the expanded semantics of Lime arrays while using the identical representation to that used in Java, hence arrays (like generic types) are an area of tension between the expressivity requirements of Lime and its Java compatibility goals. This requires the programmer to make a type distinction between Java arrays and Lime arrays using designated syntax.

In retrospect, we might have used a different base language to build upon, or in an even greater departure, we could have aimed for Java interoperability but designed a minimalist language from a clean-slate. Chisel [4] for example supports hardware design as an embedded language in Scala [11]. There are of course many endeavors to raise the level of abstraction used to program hardware and commercial synthesis tools are increasingly offering viable C, C++ and OpenCL programming options for FPGAs.

5 Auto-tuning shapes

The task graph construction in Lime may use the full power of the language. That is, graph construction may use generic types and methods, recursive or iterative constructs, etc. An example is shown in Figure 5. This example introduces a few more Lime features so we will define these first. The `task` keyword appears in the method declaration to indicate this is a method that creates a task graph. This enforces a number of checks by the type system that ensure graph shapes are repeatable. Among these is a requirement that the resultant graph

```
static task <N extends int, V extends Value> IsoTask repeat(IsoTask<V,V>[[N]] filters) {
   return task split V[[N]] => task [ filters ] => task join V[[N]];
}
```

Figure 5 Example of a generic graph constructor.

is isolated – that is constructed from local methods. The type of such tasks is `IsoTask<?,?>`. We did not explicitly state this before but the type of `task toLowerCase`, for all versions of `toLowerCase` shown in Figure 4 is `IsoTask<char,char>`. The first type argument is the type of the input channel, and the second type argument is the type of the output channel.

The type parameters to `repeat` in Figure 5 are `N` which is an ordinal type (an integer), and `V` which is any value type. The former is used as a bound on the array `filters`, and also to parameterize Lime system tasks for distributing and merging values with the `split` and `join` tasks. The syntax `task [ filters ]` denotes a set of tasks that are parallel. The composition of the splitter, parallel tasks, and joiner forms a split-join which describes task parallelism akin to fork-join parallelism.

In many examples which use split-joins to parallelize computation within a task graph, it is desirable to tune the width of the split-join. In other words, one may want to experiment with different values of `N`. The Lime language design lends itself well to automating such experimentation. This is desirable because FPGAs are resource-constrained and the synthesis tools are not entirely predictable, requiring a time-consuming exploration to determine suitable split-join widths.

6 An FPGA example

Applications with a high compute-to-communication ratio generally offer opportunities for accelerators as the cost of data serialization and transfer between the host and the device can be amortized against a large, and hopefully granular, computation. One such example is homomorphic encryption [6] which allows arbitrary but bounded computation on encrypted data without leaking information about the input, output, or intermediate results. Both plaintext and ciphertext are in the form of large polynomials where the degree is in the thousands and the coefficients are hundreds of bits. At the heart of the computation are costly polynomial multiplications which dominate execution time. One acceleration strategy is to offload only these operations to an FPGA while leaving the rest of the computation and higher-level application logic on the CPU.

We implemented the polynomial multiplication using a quadratic algorithm in Lime and leveraged the support of objects to express “bigint” and polynomials as datatypes with various operations. Generics allowed the reuse of lower-level implementation types. The table below shows some results from using Lime to generate the FPGA design for a Nallatech 287 FPGA card. We report the end-to-end performance (op time), the clock frequency for the circuit in the FPGA (freq), and the resource utilization for three primary ingredients – logic cells (LUT), memory (BRAM), and DSP units which are specialized floating-point units available in FPGAs – as a percentage of total resources available in the FPGA.

	Lime
Op Time	83.3 ms
Freq	100 Mhz
LUT	11.8%
BRAM	23%
DSP	1.2%

We also implemented the algorithm in C and used a commercial high-level synthesis (HLS) tool to generate a comparable FPGA design. The performance of the Lime generated design lags approximately 2× behind that of the C generated design.

However, Lime allowed us to avoid programming against a lower level of abstraction in which object inlining was performed manually. We believe that the performance gap is due to the relative immaturity of the low-level code generation in our compiler where poor scheduling led to a relatively low frequency. LUT consumption is similar in both designs although the Lime design used far more BRAM and far fewer DSPs.

BRAM usage stems from a design with many pipelining stages and the lack of aggressive FIFO sizing optimization between tasks. In addition, the Lime design contains monitoring and features for reliability, availability and serviceability that we omitted in the C-based design.

Interestingly, despite the numerical nature of the application, the Lime design used very few DSP units, suggesting that if the BRAM issues can be ameliorated, we can horizontally scale the Lime design for greater performance. However, deeper investigation into what fraction of LUTs were used to generate mathematical units are needed to confirm.

7 Status and Concluding Remarks

The Lime development environment is available as an Eclipse plug-in [12]. We integrated support for FPGA simulation and synthesis so that the entire programming experience is accessible to software developers who have little experience with hardware design and FPGA synthesis tools. A Lime program can run on any Java Virtual Machine, so the programmer can develop and debug on a standard workstation with no specialized hardware. The compiler can translate all Lime code to JVM bytecode, and can additionally compile a *subset* of the language to Verilog for execution on an FPGA or simulator. The subset of the language that is compiled for the FPGA (or GPU) is considered the device language although there is no formal distinction between the host and device languages in Lime. The Lime compiler endeavors to make clear through diagnostics when a task that is within a repeatable graph is excluded from synthesis.

Lime encourages an incremental development path starting from Java:

1. Prototype the program in Java,
2. Add Lime constructs to express types and invariants which allow the compiler to generate Verilog for selected components,
3. Deploy the program with a JVM and a Verilog simulator for performance tuning, and
4. Synthesize a bitfile and deploy on an FPGA.

We have written more than 200K lines of Lime code to date, and developed several substantial applications drawn from many domains that include low-latency messaging, financial algorithmics, encryption, compression, codecs, and game engines. For some of these, we have benchmarked the compiler against hand-tuned FPGA implementations. One of the challenges in doing this systematically is the lack of an adequate benchmark suite with tuned software and hardware implementations to use as a baseline. Such a benchmark suite would allow us to bound performance in two ways: "from below" to measure FPGA performance compared to software implementations, and "from above" to measure generated hardware designs against manually crafted designs.

Acknowledgments. Several IBMers contributed to the project since its inception. They are Erik Altman, Joshua Auerbach, David Bacon, Ioana Baldini, Perry Cheng, Stephen

Fink, Rodric Rabbah, and Sunil Shukla. The project also benefited from contributions by Christophe Dubach (University of Edinburgh) who was a visiting scientist in 2010, and the following interns: Shan Shan Huang (Georgia Tech) in 2007, Amir Hormati (University of Michigan) in 2007 and 2008, Andre Hagiescu (National University of Singapore) in 2008, Myron King (MIT) in 2009, Alina Simion Sbirlea (Rice University) in 2011, and Charlie Curtsinger (University of Massachusetts Amherst) in 2012.

References

1 Gul Abdulnabi Agha. ACTORS: A model of concurrent computation in distributed systems. Technical Report AITR-844, Massachusetts Institute of Technology, 1985.

2 Joshua Auerbach, David F. Bacon, Perry Cheng, Stephen Fink, and Rodric Rabbah. The shape of things to run. In *ECOOP 2013 Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 679–706. Springer Berlin Heidelberg, 2013.

3 Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 89–108, October 2010.

4 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC'12, pages 1216–1225, New York, NY, USA, 2012. ACM.

5 Stephen Brookes. On the Kahn principle and fair networks. Technical Report ADA356031, Carnegie Mellon University, 1998.

6 Craig Gentry and Shai Halevi. Implementing Gentry's fully-homomorphic encryption scheme. *IACR Cryptology ePrint Archive*, 2010:520, 2010.

7 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

8 Paul Hudak. Functional reactive programming. In *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999.

9 E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, January 1987.

10 Edward A. Lee and Thomas M. Parks. Readings in hardware/software co-design. In *Dataflow Process Networks*, pages 59–85. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

11 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

12 IBM Research. Liquid Metal Alpha Release. `http://lime.mybluemix.net`, 2014.

13 William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002.

Programming with "Big Code": Lessons, Techniques and Applications

Pavol Bielik, Veselin Raychev, and Martin Vechev

Department of Computer Science, ETH Zurich, Switzerland
`<firstname>.<lastname>@inf.ethz.ch`

Abstract

Programming tools based on probabilistic models of massive codebases (aka "Big Code") promise to solve important programming tasks that were difficult or practically infeasible to address before. However, building such tools requires solving a number of hard problems at the intersection of programming languages, program analysis and machine learning.

In this paper we summarize some of our experiences and insights obtained by developing several such probabilistic systems over the last few years (some of these systems are regularly used by thousands of developers worldwide). We hope these observations can provide a guideline for others attempting to create such systems.

We also present a prediction approach we find suitable as a starting point for building probabilistic tools, and discuss a practical framework implementing this approach, called Nice2Predict. We release the Nice2Predict framework publicly – the framework can be immediately used as a basis for developing new probabilistic tools. Finally, we present programming applications that we believe will benefit from probabilistic models and should be investigated further.

1998 ACM Subject Classification G.3 Probability and Statistics, I.2.2 Automatic Programming, D.2.4 Software/Program Verification

Keywords and phrases probabilistic tools, probabilistic inference and learning, program analysis, open-source software

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.41

1 Introduction

The rapid rise of open source code repositories such as GitHub [9], BitBucket [4], CodePlex [8] and others enables unprecedented access to a large corpus of high quality code. This corpus of code can be a valuable resource: much programmer effort and time has gone into developing, testing, debugging, annotating and fixing bugs in these programs.

A natural question then is: can we leverage and learn from this resource in order to create *new kinds of programming tools* that help developers accomplish tasks that were previously difficult or practically infeasible?

To address this challenge, a direction we have been exploring in the last couple of years is that of building probabilistic models of massive codebases, and then using these models as a basis for new kinds of probabilistic programming tools. For example, we developed statistical tools that are able to successfully and automatically complete API sequences [20], translate between programming languages [12], and de-obfuscate JavaScript programs [21].

Challenges. Ensuring that these tools work precisely on arbitrary general purpose programs (e.g. Android or JavaScript programs) requires solving several challenges, including selecting the right probabilistic model for the particular application at hand, finding the right program representation to be "compiled" into that model (and learned over the large corpus), as well as the appropriate learning and inference algorithms. However, it is often the case that simply taking existing algorithms is not enough as they do not scale to the large available corpus that we can learn from, taking days or even weeks to finish. Therefore, when designing learning and inference algorithms an interesting trade-off arises where one needs to trade-off precision for scalability such that realistic codebases can be handled.

This work. This paper has four objectives:

- To present the observations and insights obtained from building a variety of probabilistic programming tools,
- To describe an approach based on structured prediction that can serve as a basis for creating a variety of new probabilistic programming tools,
- To present a practical framework based on this approach, called Nice2Predict, that can be used as a foundation of new probabilistic programming tools, and
- To discuss programming applications that can benefit from probabilistic models and should be investigated further.

2 Observations and Insights

We start by describing our observations and insights accumulated from developing several probabilistic programming tools: code completion [20], language translation [12], and code de-obfuscation [21]. We hope that these insights will be useful to others who attempt to create new tools.

2.1 Probabilistic Models vs. Clustering

A useful distinction to make when discussing techniques for learning from massive codebases is that of using probability distributions (models) vs. clustering-and-retrieval methods. Clustering is a natural and popular approach: we first group “similar” programs together (based on some criteria for similarity) and then given a developer query, look into the “nearest” cluster (or some other cluster) to see if we can retrieve a solution that already solves the problem. Examples of clustering approaches are [24, 5, 23, 16]. These tools enable the developer to phrase queries and discover if other code has already solved the problem. In our experience, a key issue with the cluster-and-retrieve approach is that it is usually unable to produce new code, that is, to predict programs not seen in the training data, yet are likely. This approach also makes it difficult to provide a meaningful probability assignment to a proposed solution.

On the other side, with probabilistic models, we typically first abstract (represent) the program by breaking it down into smaller pieces (e.g. extract features from the program) over which we then learn a probabilistic model. These features (program representation) are usually quite specific to the type of programming task being solved. The advantage of probabilistic models is that they can usually provide predictions/solutions that are *not* found in the training data and can also associate probabilities with the proposed solution.

A key attribute of using probabilistic models is the presence of regularization or backoff. Regularization bounds the weights of the features to avoid creating a too complex model that only captures the properties of the training data – a problem also knows as over-fitting. A key step in achieving the precision of the JSNice code de-obfuscation [21] was to use cross-validation and pick an efficient regularization. Similarly, in our code completion system [20], we rely on a backoff technique [25] that was developed for n-gram statistical language models.

Backoff avoids over-fitting by considering only features with sufficient support in the training data and falling back to a simpler model for features with not enough support.

2.2 Probabilistic Models and Program Representations

Perhaps the most fundamental question when trying to build probabilistic models of code is deciding on the right program representation (features). What representation should be used when trying to learn a model over multiple programs? Representations that are too fine-grained and program specific or too coarse-grained and unable to differentiate between programs will not result in interesting commonalities being learned across many programs.

The program representation used is also connected with the particular model. For example, if one uses statistical language models (e.g. n-gram) [20], it may be natural to use a representation which consists of sequences of API sequences, however, it may be difficult to specify other features that are not sequence related (e.g. that the same variable is passed to multiple API invocations). Of course, it can also be the case that there are multiple probabilistic models which work with the same representation. For example, [20] uses both n-gram models and recurrent neural networks (RNNs).

Fundamentally, we believe that down the line, an important research item is finding the right probabilistic models and representations for programs. For instance, in the context of graphical models, popular models frequently used in computer vision are the Potts and the Ising models – images are usually converted to this representation and a subsequent prediction is then performed on that representation. Such models have very well studied theoretical properties and are equipped with suitable inference algorithms with provable guarantees. Can we find the corresponding fundamental representation for programs?

2.3 The Need for Clients

Developing probabilistic models of code is only part of the challenge. Even if one successfully builds such a model, it is not at all clear that this model will perform well for solving a particular programming challenge and will be able to handle realistic programs accurately. It is therefore critical to evaluate the model in the context of a particular client (e.g. code completion, etc.). In our experience, it is often the case that using a model by a particular client usually results in need to augment and fine-tune the model (e.g. by adding more features, etc.). For instance, in [12] we found out that the traditional score for measuring the quality of the learned model (the BLEU score) was not at all indicative of how well the model performs when translating new programs.

An interesting challenge here arises when one has to fine-tune the model. For instance, in the n-gram model, one needs to decide whether to use the 2-gram, the 3-gram, the particular smoothing method, etc. In natural language processing, there are key works which study these parameters and suggest "good" values [6] for certain models (e.g. the n-gram model).

Natural questions include: can we automatically come up with the model parameters based on the particular client? Are there appropriate metrics for the quality of the probabilistic model of code where the client is unknown, or do we always need to know what the client is?

2.4 The Need for Semantics

In our experience, it is important to leverage programming languages techniques (e.g. semantic program analysis) in order to extract quality information from programs upon which we learn the probabilistic model. That is, it is not advisable to learn directly over the program syntax as done in various prior works [11, 2]. We believe that it is very difficult to design useful programming tools without any semantic code analysis.

In particular, unlike the domain of images (e.g. image recognition, etc.), programs are not data (e.g. a collection of pixels), but data transformers: they generate new data and program analysis techniques enable us to get access to this data.

For example, in the case of API code completion, if instead of API sequences, one generates any syntactic sequence, then the sequence will contain a lot of “noise” leading to much sparsity in the probabilistic model and relating tokens that are unrelated to the API prediction. However, statically extracting API sequences precisely is a challenging program analysis problem requiring alias and typestate analysis. Our experience is that accurate static analysis is important for extracting quality semantic information. Note that the extracted information does not need to be an over-approximation to be useful: it is possible that the extracted information is both and under- and an over- approximation (i.e. as in symbolic execution).

Similar situation arises in statistical programming language translation in [12] where we had to incorporate knowledge about the language grammar in the translation process in order to avoid the statistical translation generating grammatically incorrect programs. Here too, more semantic information would have been useful in preserving semantic equivalence between the source and the translated program.

3 Prediction Approach

In this section we provide an overview of the proposed approach for building probabilistic tools. The main benefit of this approach is that it supports structured prediction which, instead of predicting labels in isolation, uses the provided context into account to make a joint prediction across all labels. Binary or multi-class predictors (that predict features in isolation) are a special case of this model.

In particular, we use Conditional Random Fields (CRFs) which is a discriminative log-linear classifier for structured prediction. The CRFs is a relatively new model introduced by [15] and since have been used in various domains such as natural language processing, information retrieval, computer vision and gene prediction [18, 17, 3, 10, 15]. We believe that future research will leverage and extend these ideas from CRFs and apply them in the context of programs.

3.1 Conditional Random Fields (CRFs)

For each program (i.e. a prediction instance), we consider two vectors of properties $\mathbf{x}$ and $\mathbf{y}$ (we can think of properties broadly as program elements: types, code, method names, etc.). The vector $\mathbf{x}$ includes known properties (e.g. properties which can be computed from the input with existing program analysis techniques). The vector $\mathbf{y}$ captures inferred properties for the given (potentially partial) program. Each known property has value ranges in X while unknown properties range over Y, thus $\mathbf{x} \in X^*$ and $\mathbf{y} \in Y^*$. A CRF model has the following form:

$$P(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z(\mathbf{x})} \exp(\mathit{score}(\mathbf{y}, \mathbf{x}, \boldsymbol{\lambda}))$$

where $Z(\mathbf{x})$ is an instance-specific partition function (also referred to as a normalization function), $\boldsymbol{\lambda}$ are weights learned in the training phase, and $\mathit{score}(\mathbf{y}, \mathbf{x}, \boldsymbol{\lambda})$ is a function that calculates a real valued score for given assignment $\mathbf{y}$ and $\mathbf{x}$. The partition function ensures that $P(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\lambda})$ is a valid probability distribution (i.e., sums to one) over the range of unknown properties Y and is defined as follows:

$$Z(\mathbf{x}) = \sum_{\mathbf{y} \in Y^*} \exp(score(\mathbf{y}, \mathbf{x}))$$

The scoring function $score(\mathbf{y}, \mathbf{x}, \boldsymbol{\lambda})$ assigns higher score to more likely assignments of $\mathbf{x}$ and $\mathbf{y}$ and is defined as follows:

$$score(\mathbf{y}, \mathbf{x}, \boldsymbol{\lambda}) = \sum_{i=1}^{k} \lambda_i f_i(\mathbf{y}, \mathbf{x}) = \boldsymbol{\lambda}^T \mathbf{f}(\mathbf{y}, \mathbf{x})$$

where f_i is i-th feature function $f : X^* \times Y^* \to \mathbb{R}$ with associated weight λ_i. For convenience we use vector notation $\boldsymbol{f}(x, y) = [f(x, y)_1 \ \ldots \ f(x, y)_k]$ for feature vector of k feature functions and $\boldsymbol{\lambda} = [\lambda_1 \ \ldots \ \lambda_k]$ for learned weights. The weights $\boldsymbol{\lambda}$ are learned from the training data.

Challenges. Conditional random fields define a probability distribution that is linear with respect to the learned weights $\boldsymbol{\lambda}$, but computing the actual probability with the formula $P(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{Z(\mathbf{x})} \exp(\boldsymbol{\lambda}^T \mathbf{f}(\mathbf{y}, \mathbf{x}))$ requires evaluating an expensive $Z(\mathbf{x})$ function that computes a sum over all possible assignments of labels (program properties $\mathbf{y} \in Y^*$). To avoid this limitation, we focus on applications where we do not need to report the actual probability of the prediction, but only the ability to compare probabilities of different assignments of the properties $\mathbf{y} \in Y^*$.

MAP Inference Query. The most important query in this context is that of MAP Inference (also known as Maximum APosteriori Inference):

$$\mathbf{y} = \operatorname*{argmax}_{\mathbf{y}' \in \mathcal{P}(Y)} P(\mathbf{y}' \mid \mathbf{x})$$

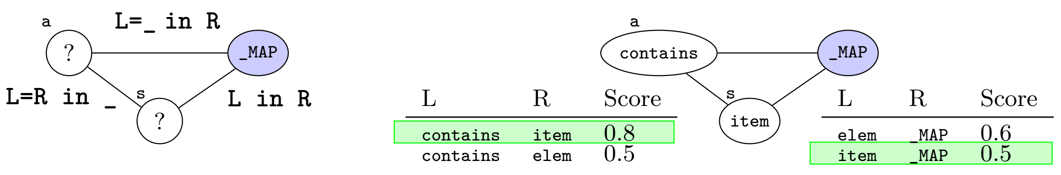
To answer this query, we do not need to evaluate the partition function since it is constant with respect to $\mathbf{y}'$ and thus $\mathbf{y} = \operatorname{argmax}_{\mathbf{y}' \in \mathcal{P}(Y)} \boldsymbol{\lambda}^T \mathbf{f}(\mathbf{y}', \mathbf{x})$. This MAP inference query requires solving an optimization problem – finding the labels that score the best. Typically, solving this problem requires leveraging properties of the feature functions $\mathbf{f}$.

Unfortunately, selecting an appropriate interference algorithm is challenging as: i) the exact inference problem is generally NP-hard and too expensive to be used in practice, and ii) even common approximate inference algorithms such as (loopy) belief propagation have shown to be at least an order of magnitude slower than greedy algorithms. Given the large-scale nature of the applications, we provide a greedy inference algorithm specifically designed to efficiently handle a massive range of the domain Y, implementation details of which can be found in [21].

Alternatives. The most important alternative discussed in literature avoids MAP inference altogether and replaces it with marginal inference. In marginal inference, we simply select for each property an assignment which maximizes its marginal probability. Such assignment however ignores dependencies between properties as it maximizes only the local probability of assignment to a single property. In contrast, optimizing the joint probability of the assignments to all the properties can lead to significant increase in accuracy as illustrated in the 9.3% improvement in the name prediction task [21].

```
var a = s in _MAP;
```

(a) Minified JavaScript



L	R	Score
contains	_MAP	0.7
has	_MAP	0.4
check	_MAP	0.2

L	R	Score
contains	item	0.8
contains	elem	0.5

L	R	Score
elem	_MAP	0.6
item	_MAP	0.5

(b) Dependency network (c) Result of MAP inference

Figure 1 a) Code snippet of minified JavaScript, b) Dependency network build using properties and features extracted from a). The known properties are shown in blue (global variable `_MAP`) and unknown properties in white (variables `a` and `s`). c) An overview of the name inference procedure.

Naturally, there is an opportunity that an alternate algorithm will provide a meaningful approximation of the exact MAP inference. For example, one could first optimize all the labels in isolation and only then optimize by taking structural features into account. However, this should be done by taking the constraints of the problem into account.

Constraints. An important extension of the greedy MAP inference is to support hard constraints on the range of possible values y. For example, in a name prediction task, one constraint is that only non-duplicate names in scope are allowed. Such constraints cannot be neglected as they guarantee that the resulting MAP assignment is a feasible program. It is worth noting that common approximate inference algorithms do not support constraints over the range of possible assignments and therefore can produce non-feasible solutions. Integrating constraints into existing algorithms is an open problem which will need to be addressed as soon as the need for more accurate inference algorithms arises.

Example. To illustrate MAP inference, we predict variable names of the code snippet in Fig. 1 a) using JSNice [21] – a system for predicting variable names and types. We start by identifying two unknown properties corresponding to variables `a` and `b` together with one known property corresponding to global variable `_MAP`. Next, the program analysis extracts three feature functions which relate the properties pairwise. An example of feature function is `L in R` between properties `s` and `_MAP`, which captures the fact that they are used together as a left-hand and right-hand side of binary operator `in`. The properties and feature functions are succinctly represented as a dependency network Fig. 1 b), which is used as a graphical representation of CRF.

Given the dependency network, we are interested in performing the MAP inference. That is, we want to find an assignment to unknown properties such that it maximizes the scoring functions learned from the data. An illustration of MAP inference is shown in Fig. 1 c). The particular assignment of properties in this case demonstrates that the MAP inference computes a global maximum of the scoring functions, which does not necessarily correspond to the best local per property assignments (i.e., for properties `s` and `_MAP` only a second best scoring assignment was selected).

Extensions. The choice of features enable fast approximate MAP inference algorithm, but it has an important implication on the system capabilities. In particular, our current implementation of the system has the following limitations:

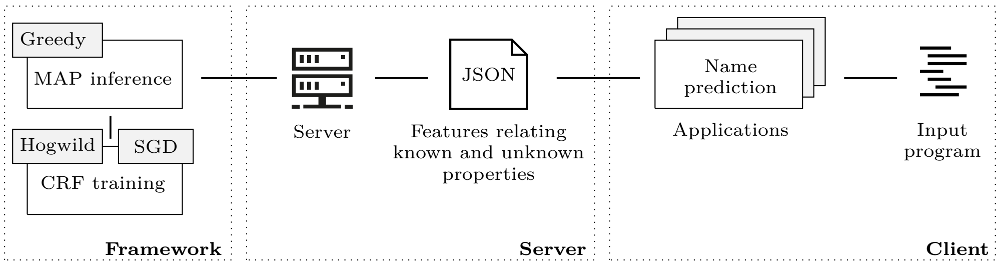


Figure 2 Overview of the framework architecture.

- Fixed domain of unknown properties. That is, only label values seen in the training data can be predicted.
- Pairwise feature functions. The feature functions can be defined only for pairs of properties and dependencies of higher cardinality are split into several pairwise feature functions.

In the context of previously discussed application of variable name prediction the above limitations are not critical to the system performance. However, they ultimately restrict the system's capabilities to predict arbitrary program properties.

4 Nice2Predict: A Prediction Framework

We believe an important part of the research on probabilistic programming tools is having an effective systems that works in practice and is shown to be effective when trained on real problems and code. Towards this, we discuss an implementation of the approach discussed above in scalable, efficient and extensible to new applications implementation available at:

`https://github.com/eth-srl/Nice2Predict`

In particular, the implementation is decoupled from the particular programming language (and from particular clients such as JSNice).

The architecture of Nice2Predict is shown in Fig. 2. The core library contains an efficient C++ implementation of both greedy MAP inference and state-of-the-art max-margin CRF training [21] which are both subject to future extensions. The CRF training currently supports two modes: parallel stochastic gradient descent (SGD) [26] and Hogwild [22] training. Further, the training is highly customizable by providing control over several important parameters such as learning rate, SVM margin and regularization. Additionally, we provide automatic cross-validation procedure which finds the best values for all parameters.

The server provides interoperation between the C++ core implementation and clients using different programming languages. Our communication format is JSON which most languages can easily write to and we provide a binding with a remote procedure call (RPC) over the HTTP server.

We chose to implement the inference engine as an external service to cover a range of clients. The clients provide the remaining steps: i) defining known and unknown properties, ii) defining features that relate properties, and iii) obtaining training data. That is, the client is mainly responsible for transforming an input program into a suitable representation which is then encoded to a JSON and sent as a query to the server.

Example client: predicting JavaScript names

We now briefly describe a client of our framework – a JavaScript deminifier implementation called UnuglifyJS which extracts properties and features as described in [21]. This client can be used as a starting point to help researchers and developers interested in using the Nice2Predict framework. The open-source implementation is available at:

`https://github.com/eth-srl/UnuglifyJS`

As discussed above, the client is responsible for three items: i) defining known and unknown properties, ii) defining features, and iii) obtaining training data. We start by describing the known and unknown properties. The known properties are program constants, objects properties, methods and global variables – that is, program parts which cannot be (soundly) renamed (e.g. the DOM APIs). The unknown properties are all local variables.

To define features, we first represent the properties in a graph where the nodes are properties, and each edge represents a relationship between them. We associate a label with each node - the value in the corresponding component from the vectors $\mathbf{x}$ and $\mathbf{y}$. Then, we define the feature functions to be indicator function for all possible labels connected by an edge. In the case of names, we introduce a feature function for all possible triples: $(z_1, z_2, type)$ where z_1 and z_2 are labels $(z_1, z_2 \in X \cup Y)$ and *type* describes the type of the relation encoded in the graph edge. Finally, specifically for JavaScript names, we create a simple filter that detects minified code and performs training on non-minified code with meaningful variable names. Based on these names, the system learns the feature functions used for code deminification.

Further, for this prediction application, we provide an interactive walk-through available at `http://nice2predict.org/`. Once the user provides JavaScript, the resulting CRF graph is visualized and a Nice2Predict server performs MAP inference.

5 Applications

We next discuss several applications that we believe would benefit from and are worth investigating in the context of probabilistic models.

- *Statistical program synthesis*: in the last few years, we have seen renewed interest in various forms of program synthesis. Usually, these techniques are applied on a per-program basis (e.g. synthesize the program from a partial specification such as assertions, input-output examples, etc.). A natural direction is to consider program synthesis based on probabilistic models. We already made first steps in prior work [20] (in the context of code completion), but much more investigation is needed to understand the limits of statistical code synthesis.
- *Statistical invariant prediction*: an interesting direction is whether we can predict properties of programs by learning models of these properties over other programs already annotated with their properties. First steps in that direction were made in [21], where we predict simple types for JavaScript (checked by the Google Closure Compiler [7]) – can this approach be applied to infer useful contracts or ownership annotations in the style of Kremenek et al. [14]? Such a system could potentially complement existing static analysis tools and make it easier to annotate programs.
- *Statistical code beautification and refactoring*: a promising direction is leveraging probabilistic models for learning natural coding and naming conventions [21, 1]. Suggesting better names and code refactorings based on probabilistic models is useful in various

settings include software engineering (better code readability) and de-obfuscation as in JSNice[1].

- *Statistical error prediction*: a useful direction we have been exploring recently is discovering pieces of program code that are unlikely and suggesting alternative code that is more likely. This has applications in regular programming but also in the context of massive open online courses (MOOCs). In the context of concurrency, can we use statistical models to predict likely thread interference and suggest likely synchronization repairs?
- *Statistical language translation*: recently, we explored the direction of using statistical models to translate programming languages [12]. An interesting observation here is that statistical translation is sometimes able to learn how to translate entire patterns of code (i.e. how to map one set of APIs to another). One challenge here is obtaining a parallel corpus of translated programs – a possible direction is using Amazon Turk for obtaining such corpus.
- *Statistical code exploration*: an interesting application of statistical models is to guide the program exploration or test case generations to exercise specific parts (e.g. suspicious) of the codebase first. Such approach is particularly interesting for large scale projects which ensure their security via a fuzzing infrastructure – an automated the test case generation and crash detection.

Thinking of applications: from Computer vision to Programs. A useful place to think about applications is to consider the analogous problems in computer vision. For example, code deobfuscation can be seen as image denoisification, where noisy pixels correspond to obfuscated program parts. Code completion corresponds to image completion and code documentation is akin to words describing an image. This view is important not only for inspiring new applications, but by formalizing these problems in similar terms as in vision, we can transfer advances such as efficient feature learning [19] and a rich arsenal of inference techniques [13] to programming language problems.

6 Conclusion

In this paper, we presented four items: insights for building probabilistic programming tools, a prediction approach that can serve as a starting point for developing such tools, a practical framework based on this approach, and a set of applications worth exploring further.

References

1 Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. Learning natural coding conventions. In *Proc. of the 22nd ACM SIGSOFT Int' Symp. on Foundations of Software Engineering (FSE'14), 2014*, pages 281–293, 2014.

2 Miltos Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *MSR*, 2013.

3 Julian Besag. On the Statistical Analysis of Dirty Pictures. *Journal of the Royal Statistical Society. Series B (Methodol.)*, 48(3):259–302, 1986.

4 Atlassian bitbucket. `https://bitbucket.org/`.

5 Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (ESEC/FSE'09)*, pages 213–222, 2009.

[1] `http://jsnice.org`

6 Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proc. of the 34th Annual Meeting on Association for Computational Linguistics (ACL'96)*, pages 310–318, 1996.

7 Google closure compiler. https://developers.google.com/closure/compiler/.

8 Atlassian bitbucket. `https://www.codeplex.com/`.

9 Github. `https://github.com/`.

10 Xuming He, Richard S. Zemel, and Miguel Á. Carreira-Perpiñán. Multiscale conditional random fields for image labeling. In *Proc. of the 2004 IEEE Conf. on Computer Vision and Pattern Recognition*, CVPR'04, 2004.

11 Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE 2012*, 2012.

12 Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proc. of the 2014 ACM Int'l Symp.on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'14)*. ACM, 2014.

13 Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques.* MIT Press, 2009.

14 Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation (OSDI'06)*, OSDI'06, pages 161–176, Berkeley, CA, USA, 2006. USENIX Association.

15 John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. of the 18th Int'l Conf. on Machine Learning (ICML'01)*, ICML'01, pages 282–289, 2001.

16 Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *OOPSLA'12*, 2012.

17 David Pinto, Andrew McCallum, Xing Wei, and W. Bruce Croft. Table extraction using conditional random fields. In *Proc. of the 26th Annual Int'l ACM SIGIR Conf. on Research and Development in Informaion Retrieval (SIGIR'03)*, pages 235–242, 2003.

18 Ariadna Quattoni, Michael Collins, and Trevor Darrell. Conditional random fields for object recognition. In *NIPS*, pages 1097–1104, 2004.

19 Nathan D. Ratliff, J. Andrew Bagnell, and Martin Zinkevich. (approximate) subgradient methods for structured prediction. In *AISTATS*, pages 380–387, 2007.

20 Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'14)*, pages 419–428. ACM, 2014.

21 Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting Program Properties from “Big Code”. In *Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'15)*, pages 111–124. ACM, 2015.

22 Benjamin Recht, Christopher Re, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proc. of Neural Information Processing Systems Conf. (NIPS'11)*, pages 693–701, 2011.

23 Steven P. Reiss. Semantics-based code search. In *ICSE'09*, 2009.

24 Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE'07*, 2007.

25 Ian H. Witten and Timothy C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, 1991.

26 Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

Bridging the Gap Between General-Purpose and Domain-Specific Compilers with Synthesis

Alvin Cheung[1], **Shoaib Kamil**[2], **and Armando Solar-Lezama**[2]

1 **University of Washington, US**
2 **MIT CSAIL, US**

Abstract

This paper describes a new approach to program optimization that allows general purpose code to benefit from the optimization power of domain-specific compilers. The key to this approach is a synthesis-based technique to raise the level of abstraction of general-purpose code to enable aggressive domain-specific optimizations.

We have been implementing this approach in an extensible system called HERD. The system is designed around a collection of parameterized *kernel translators*. Each kernel translator is associated with a domain-specific compiler, and the role of each kernel translator is to scan the input code in search of code fragments that can be optimized by the domain-specific compiler embedded within each kernel translator. By leveraging general synthesis technology, it is possible to have a generic kernel translator that can be specialized by compiler developers for each domain-specific compiler, making it easy to build new domain knowledge into the overall system.

We illustrate this new approach to build optimizing compilers in two different domains, and highlight research challenges that need to be addressed in order to achieve the ultimate vision.

1998 ACM Subject Classification I.2.2 Program Synthesis

Keywords and phrases compilers, domain-specific languages, program synthesis, cross compilation, verification

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.51

1 Introduction

Despite significant advances in compiler optimization over the last thirty years, the promise of actual optimality in generated code remains elusive; even today, clean and high-level code cannot compete with heavily optimized hand-tuned, low-level code, even if the former is compiled with the best optimizing compilers. Part of the problem is that the kind of aggressive, high-level, global optimization that a performance engineer can perform by hand often requires domain knowledge about the implementation strategies that are best for particular classes of programs. For a general-purpose compiler, it is often difficult to tell that a piece of code comes from a particular domain, and it is also difficult to incorporate the requisite domain knowledge in a traditional compiler architecture.

Domain-specific languages (DSLs) have emerged as a solution to the challenge of obtaining high performance from high-level code. Compilers for domain-specific languages leverage specialized internal representations and domain-specific transformations to encode the domain knowledge necessary to optimize a given class of programs. In a number of different domains [32, 24, 29, 23, 4], it has been shown that domain-specific compilers can outperform traditional compilers – and in some cases can even surpass hand-tuned code created by performance experts – because they can combine the domain knowledge embedded in them with exhaustive exploration to produce truly optimal implementations.

While DSLs ease the job of the compiler developer in generating efficient programs, they often come at a cost for application developers. First, given the proliferation of DSLs even within a given domain, the developers must understand the pros and cons of each DSL in order to choose the best one for their applications. Once chosen, developers might need to rewrite their existing applications into the DSL. Worse yet, developers might only want to express fragments of their computation in the DSL. As a result, the program might become a collage of code fragments written in various DSLs – database backed applications, where for years it has been customary to combine a general-purpose language like Java with query languages like SQL, provide a cautionary tale of the software engineering challenges inherent in this approach [19, 5].

In this paper, we present a system called Herd that we are building for bridging the gap between general-purpose compilers and DSLs. Given a program written in a general-purpose language, our system relies on a new technology called *Synthesis Enabled Translation* (Set) which systematically identifies code fragments that can be translated into DSLs. Set derives provably correct translations for each identified code fragment, and the translations are then given to domain-specific compilers for aggressive optimizations. We have demonstrated Set on two domains for which efficient domain-specific notations exist: stencil computations and database queries, but the generality of the Set approach should make it possible to extend the system to other domains for which high-performance DSLs are available.

With Set, compiler developers can add support for a new DSL by describing the semantics of the target language, together with high-level filtering rules to help guide the compiler towards more promising code fragments. The compiler can then leverage recent advances in program synthesis and software verification techniques [6, 13] to dynamically search for code fragments that can be rewritten into the target DSL. Using the provided specification of the target DSL, the compiler will automatically weave the generated DSL code back into the original program in order to complete the optimization process.

Overall, our approach has the following advantages over both general-purpose compilers and direct use of DSLs:

- Compared to direct use of DSLs, our approach does not require programmers to learn new formalisms, or even to know about the existence of DSLs for particular domains.
- The translations to DSLs performed by our system are provably correct, so there is less risk of introducing bugs in the program through the translation process.
- The compiler can automatically search through transformations across multiple DSLs simultaneously given an input program, and rewrite appropriate fragments into the optimal DSL according to a cost metric. This eliminates the application developers' need to rewrite their code into multiple DSLs and maintain each implementation.

In the rest of this paper, we first briefly describe constraint-based synthesis in Sec. 2 and outline the proposed architecture of Herd in Sec. 3. Then, in Sec. 4 and Sec. 5 we describe instances where we have applied Set in constructing compilers for two very different application domains, namely, applications that interact with databases for persistent storage and stencil computations. The Herd architecture is designed after our experience from the two application domains. After that, we discuss related work in Sec. 6 and conclude in Sec. 7.

2 Constraint-Based Synthesis

In this section we describe the synthesis technology that forms the basis of Set. Set uses constraint-based program synthesis [28, 1], which provides mechanisms to efficiently search a space of candidate implementations for one that satisfies a given functional specification.

Specifically, the synthesis problem is formulated as solving the Doubly Quantified Boolean Formula problem (2QBF):

$$\exists c.\forall x.Q(f_c, x)$$

Here f_c represents the unknown function parameterized by a set of control values c. Given c, the predicate Q checks to see whether f_c satisfies all the constraints under all possible inputs x. The job of the synthesizer is to find such c. Since the number of possible inputs is infinite, in practice, synthesizers perform bounded verification where the size of x is limited. For instance, if the inputs are lists, then one way to perform bounded verification is to limit the maximum number of elements in each list, or the size of the elements in each list. If the inputs are integers, then bounded verification can be done by checking constraint satisfaction for all integers up to a certain value.

In SET, we use synthesis to discover the DSL program and any program invariants that are necessary to prove equivalence to the original program. Thus, the constraints in Q consist of clauses that ensure equivalence to the semantic meaning of the original snippet.

Unlike typical compiler analysis techniques, which use semantics-preserving rewrites to transform the input code to the target language in a deductive way, the synthesis technology in SET uses inductive reasoning: it works by searching for solutions that satisfy the specification in concrete scenarios (i.e., pairs of program inputs and outputs), and then checks whether the solution generalizes for all possible program inputs. This approach is formalized in an algorithm called CEGIS [28]. CEGIS avoids the universal quantifier from the 2QBF problem shown above when searching for c by instead solving a simpler problem: given a small set of representative inputs X_i, it attempts to find c_i such that

$$\forall in \in X_i.Q(f_c^i, in)$$

CEGIS constructs the set of inputs X_i in an iterative manner. First, X_0 is constructed from a single random input, and CEGIS uses a constraint solver to solve the problem to obtain c_0 (and the resulting f_c^0). These control values are then passed to a checking procedure that checks whether the resulting program is correct for all inputs. If so, then we have found c. Otherwise, the checking procedure generates a counterexample input in_0. in_0 is then added to X_0 to construct X_1, and the process is repeated until a c is found that generalizes to all possible inputs. To speed up the search process, the search for c_i during each iteration is done in a symbolic manner: rather than enumerating and checking each c one by one, CEGIS encodes the parameterized program Q and as a constraint system, with f_c as the solution. Hence during each iteration solving the constraint system will identify the next f_c^i to use.

3 Herd Architecture

We now describe the overall architecture of HERD based on SET methodology, as shown in Fig. 1. After parsing the input code, HERD passes the AST to a series of static program analyzers to annotate the program with type, aliasing, and other structural information about the input. The language broker then takes the AST, along with analysis results, and sends them to the kernel translators registered with HERD. Each kernel translator represents a target DSL. The goal of each translator is to identify and translate code fragments from the input AST into the target DSL, and return the translated code fragment to the fragment reassembler. After receiving the translated code fragments from each of the kernel translators, the reassembler combines them to produce the final executable, potentially using a cost model to evaluate among different options in case the same code fragment has been translated by

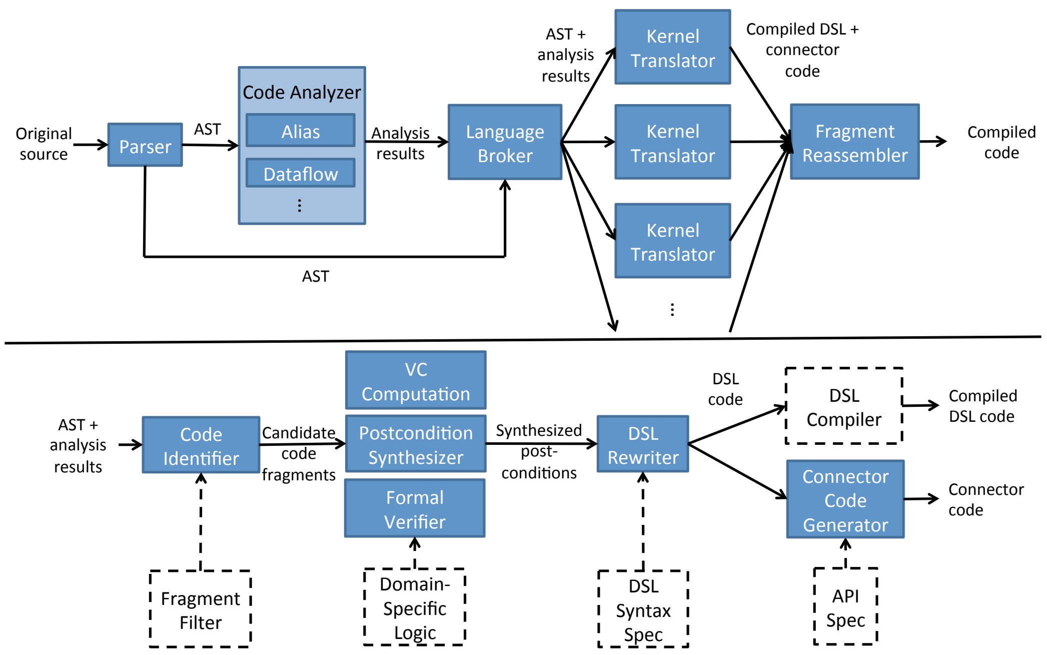


Figure 1 (a) Architecture of Herd (top); (b) kernel translator detail (bottom), with dotted components representing inputs from the DSL compiler developer when constructing a new kernel translator.

multiple translators. In the rest of this section, we describe the details of kernel translators and how compiler maintainers can use Herd to generate them for new target DSLs.

3.1 Kernel Translators

Kernel translators take in general-purpose ASTs and attempt to transform portions of the input into the given DSL. The translation consists of the steps described below.

3.1.1 Label potential code fragments

The first step in DSL translation is to identify candidate code fragments from the input AST that might be amenable to translation. To aid in fragment identification, the compiler maintainer provides Herd with filtering conditions. For instance, to compile for a DSL that targets parallel computation, the developer might specify all loop nests with array writes as candidates. Examples of such hints are discussed in Sec. 4 and Sec. 5. Using such conditions, the Herd code identifier automatically labels a number of candidate code fragments from the input program during compilation, and passes them to the next component in the kernel translator.

3.1.2 Search for rewrites

For each candidate code fragment that the code identifier found, the kernel translator then tries to express it in the target DSL. The search for semantically-equivalent constructs in the target DSL is framed as a synthesis problem, with the goal to find DSL expressions that can be proved to be semantically equivalent to the original code using Hoare-style program verification. This is done in two steps. First, compiler maintainers provide Herd with a high-level language that abstracts and formalizes the target DSL. The postcondition synthesizer then takes the formalized DSL and tries to find postcondition expressions written using that language. The search space for each postcondition expression is limited to only non-trivial expressions (i.e., no expression such as “x = x”). In addition to the postcondition, the synthesizer also searches for invariants that are needed to verify loop constructs in the candidate code fragment. However, unlike prior work in inferring loop invariants, we only need to find loop invariants that are strong enough to establish the validity of the postcondition.

3.1.3 Verify and convert

To reduce the time needed to synthesize postconditions, the kernel translator uses a bounded synthesis procedure as described in Sec. 2, where it checks validity of the found postcondition up to a fixed heap size, and only expressions up to a fixed length are considered. Other search techniques can be used as well. Due to bounded reasoning, any candidate postcondition that is discovered by the synthesizer needs to be formally verified. If the candidate fails formal verification, the kernel translator will ask the synthesizer to generate another postcondition candidate (by increasing the space of expressions considered, for instance). Otherwise, the DSL rewriter translates the postcondition into the target DSL using the syntax specification provided by the compiler maintainer.

3.1.4 Generate connector code

The translated DSL code is then compiled using the DSL compiler provided by the maintainer. At the same time, the connector code generator produces any wrapper code that is necessary to invoke the compiled DSL code fragment, for instance passing parameters and return values between the general-purpose and DSL runtimes. The DSL compiler maintainer provides an API specification of the DSL runtime to the system to facilitate the generation of wrapper code.

The final output of the kernel translator consists of the compiled DSL code along with any wrapper code, both of which are returned to the fragment reassembler. As mentioned earlier, the fragment reassembler replaces the original source code AST with translated DSL code fragments. In some cases, multiple kernel translators can find rewrites for a given code fragment. For instance, a code fragment that can be converted into SQL may also be expressible in Hadoop for parallel processing. In such cases, the reassembler chooses the translation that is most optimal. To estimate the cost of each translation, Herd executes each translation using input test cases, generating random test data if necessary. The reassembler then chooses the translation with the lowest cost (e.g., with the lowest expected execution time) and replaces the input source code with most optimal translation. Herd can use other mechanisms to estimate cost and choose the optimal translation as well (e.g., using auto-tuning [2]), and a runtime component can also be added to adaptively choose the optimal translation based on program input, and we leave that as future work.

In the next sections, we highlight two application domains where we have used the Set methodology to build DSL compilers. We constructed kernel translators for each domain to convert fragments of general-purpose code into different DSLs, and the converted code fragments achieve up to multiple orders of magnitude performance speedup compared to the original general-purpose code by utilizing domain-specific optimizations. Without translating to DSL, it would be very difficult for the original program to utilize the optimizations provided by the DSL runtime.

4 Transforming Java Code Fragments to SQL

In this section we describe our experience in building a kernel translator to enable general-purpose code to utilize specialized data retrieval functionality provided by database engines in the context of QBS (Query By Synthesis) [9]. QBS takes in Java code fragments that compute on persistently-stored data, and automatically converts them into database queries (SQL in this case).

4.1 Background

Many applications make use of database systems for data persistence. Database systems typically provide an API for applications to issue queries written in a query language such as SQL, while application frameworks [12, 26, 15, 21] encourage developers to modularize data persistence operations into libraries. This results in an intermixing of database queries and application code, making code maintenance more difficult. Moreover, as implementors of persistent data libraries cannot anticipate all possible ways that persistent data will be used in the application, this leads to application developers writing custom data manipulation code in the application rather than making use of the highly-optimized implementations provided by database engines, potentially resulting in substantial application slowdown. The goal of QBS is to address such issues by converting general-purpose application code that computes on persistent data into semantically-equivalent queries in SQL.

4.2 Kernel Translator

QBS was built as a kernel translator registered with HERD. In this section we describe the different inputs used to construct the kernel translator.

Fragment filtering. QBS considers all code fragments that can potentially use persistent data as candidate fragments for translation. To identify such code fragments, QBS first finds all statements that retrieve data from persistent storage (by identifying standard API calls [17]) and have no side-effects (since those have no semantic equivalence in standard SQL). An interprocedural taint analysis is then used to identify all statements that might take in persistent data as input, and a greedy algorithm is used to group together such statements to form candidate code fragments.

Domain-specific Logic. QBS defines its formalized DSL as shown in Fig. 2(a). The language is an imperative one that includes a number of operators derived from relational algebra (e.g., selection, projection, etc), except that they operate on ordered collections rather than relations, since the Java data persistence API (like many other popular ones) is based on ordered collections. The semantics are defined using axioms based on the theory of lists, a sample of which are shown in Fig. 2(b). Given this DSL, QBS makes use of the components included in the kernel translator to search for possible rewrites for each candidate code fragment, as described in Sec. 3.1.

Output DSL Syntax. The formalized DSL used by QBS was designed to enable easy translation to the target domain language (SQL). As such, QBS defines syntax-driven rules to transform postcondition expressions into appropriate SQL queries, the details of which are discussed in [9].

$$
\begin{array}{rcl}
\mathsf{c} \in \mathsf{lit} & ::= & \mathsf{True} \mid \mathsf{False} \\
 & \mid & \mathsf{number\ literal} \mid \mathsf{string\ literal} \\
\mathsf{e} \in \mathsf{exp} & ::= & \mathsf{c} \mid \mathsf{program\ var} \\
 & \mid & \{\mathsf{f_i} = \mathsf{e_i}\} \mid \mathsf{e_1\ op\ e_2} \mid \neg\ \mathsf{e} \\
 & \mid & [\,] \mid \mathsf{Query}(\ldots) \\
 & \mid & \mathsf{size}(\mathsf{e}) \mid \mathsf{get}(\mathsf{e_r}, \mathsf{e_s}) \\
 & \mid & \mathsf{top}(\mathsf{e_r}, \mathsf{e_s}) \mid \pi(\mathsf{e}, \mathsf{f}_\pi) \mid \sigma(\mathsf{e}, \mathsf{f}_\sigma) \\
 & \mid & \bowtie(\mathsf{e_1}, \mathsf{e_2}, \mathsf{f}_\bowtie) \mid \mathsf{append}(\mathsf{e_1}, \mathsf{e_2}) \\
 & \mid & \mathsf{sort}(\mathsf{e}, [\mathsf{e.f_i}]) \mid \mathsf{unique}(\mathsf{e}) \\
 & \mid & \mathsf{sum}(\mathsf{e}) \mid \mathsf{max}(\mathsf{e}) \mid \mathsf{min}(\mathsf{e}) \\
\mathsf{f}_\pi(\mathsf{e}) & ::= & \{\mathsf{e.f_i}\} \\
\mathsf{f}_\sigma(\mathsf{e_s}) & ::= & \wedge\ \mathsf{e.f_i\ op\ c} \mid \mathsf{e.f_i\ op\ e.f_j} \\
 & \mid & \mathsf{contains}(\mathsf{e_s}, \mathsf{e}) \\
\mathsf{f}_\bowtie(\mathsf{e_1}, \mathsf{e_2}) & ::= & \wedge\ \mathsf{e_1.f_i\ op\ e_2.f_j} \\
\mathsf{op} \in \mathsf{bop} & ::= & \wedge \mid \vee \mid > \mid =
\end{array}
$$

$$\boxed{\text{axioms for top operator}}$$

$$\mathsf{top}([\,], \mathsf{i}) = [\,]$$
$$\mathsf{i} = 0 \to \mathsf{top}(\mathsf{r}, \mathsf{i}) = [\,]$$
$$\mathsf{i} > 0 \to \mathsf{top}(\mathsf{h} : \mathsf{t}, \mathsf{i}) = \mathsf{h} : \mathsf{top}(\mathsf{t}, \mathsf{i} - 1)$$

$$\boxed{\text{axioms for projection } (\pi)}$$

$$\pi([\,], \mathsf{f}) = [\,] \qquad \pi(\mathsf{h} : \mathsf{t}, \mathsf{f}) = \mathsf{f}(\mathsf{h}) : \pi(\mathsf{t}, \mathsf{f})$$

$$\boxed{\text{axioms for selection } (\sigma)}$$

$$\sigma([\,], \mathsf{f}) = [\,]$$
$$\mathsf{f}(\mathsf{h}) = \mathsf{True} \to \sigma(\mathsf{h} : \mathsf{t}, \mathsf{f}) = \mathsf{h} : \sigma(\mathsf{t}, \mathsf{f})$$
$$\mathsf{f}(\mathsf{h}) = \mathsf{False} \to \sigma(\mathsf{h} : \mathsf{t}, \mathsf{f}) = \sigma(\mathsf{t}, \mathsf{f})$$

Figure 2 (a) Formalized DSL used by QBS (left); (b) Semantic axioms defining the language operators (right).

Connector API Spec. The Java data persistence API defines a number of methods for Java applications to issue SQL queries to the database. QBS makes use of this API to pass the inferred queries to the database. Optionally, the inferred queries can be pre-compiled by the database as stored procedures for efficient execution. In both cases, the kernel translator returns the compiled queries and rewritten code fragments back to the HERD toolchain.

4.3 Results

We used two large-scale open source web applications to evaluate QBS, with the goal of quantifying its ability to convert Java code fragments into SQL. A selection of the results are shown in Fig. 3(a), which shows that QBS can convert a majority of the candidate code fragments. The cases that failed were mostly due to lack of expressivity in the formalized DSL (e.g., the application code uses custom aggregate functions). The maximum amount of time spent in synthesis for any code fragment was 5 minutes, with the largest code fragment consisting of about 200 lines of code.

Fig. 3(b) shows a representative result comparing the execution time containing a converted code fragment. In this code fragment, the original code loops through two lists of persistent data in a nested loop and returns a subset of the objects. QBS converted this code fragment into a SQL join query. By doing so, the converted code can take advantage of the specialized join query implementations in the database system (in this case a hash join based on the object's ID was used), and that resulted in a significant performance gain.

5 Converting Stencil Codes

We now turn to describe our experience in building a kernel translator for converting fragments of serial Fortran that express stencil computations into a DSL called Halide [24]. The resulting system, STNG, generates Halide programs from general-purpose Fortran code. The translated Halide programs compile into vectorized, parallel code that outperforms code produced by general-purpose compilers.

itracker (bug tracking system)		
operation type	# benchmarks	# translated
projection	3	2
selection	3	2
join	1	1
aggregation	9	7
total	**16**	**12**

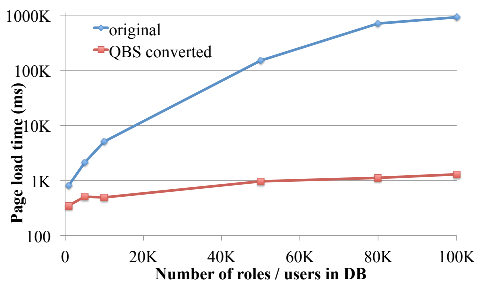


Figure 3 (a) Number of code fragments converted by QBS on one of the applications (left); (b) Performance comparison of converted code fragment (right).

5.1 Background

Stencil computations describe an important computational pattern used in many areas of computing, including image processing, linear algebra solvers, and scientific simulations. A stencil computation consists of updating each point in a multidimensional grid with a function of a subset of its neighbors. Such computations are difficult to optimize using libraries, because the optimizations depend heavily on the sequence of stencils and the particular function being applied. General compiler loop transformations based on the polyhedral method [16] can optimize some stencils, but do not usually obtain peak performance due to their limited scope – they only optimize traversals in an existing loop nest, and they cannot optimize across loop nests due to a lack of high-level knowledge about the overall computation.

A number of domain-specific languages and compilers [29, 18, 10] have been built for stencils, with excellent performance, matching or beating hand-tuned code. We choose to convert stencils to code written in the Halide [24] language, which supports empirical auto-tuning to find best-performing computation schedules for the overall stencil computation.

5.2 Kernel Translator

Fragment Filtering. We search for code fragments that could be stencil computations by examining all loop nests and finding ones that write to arrays, where the indices of the writes are dependent on loop iteration variables. In this process, we also filter out computations and structures that cannot be handled by our domain-specific logic, for instance loop nests that contain side-effect computations.

Domain-Specific Logic. STNG's formalized DSL uses a stylized representation of the space of stencil computations, using the theory of arrays. This representation includes the common constructs found in stencils, including neighbor access, and operations using neighboring points. The major difficulty in synthesis is the complexity of the verification conditions; they include universal quantifiers over arrays, resulting in very large synthesis problems. Our implementation aggressively optimizes the synthesis problem, eliminating symmetries and using domain-specific information where possible to make the problem easier to solve.

Output DSL Syntax. We translate from the formalized DSL to Halide's high-level computation language, which is embedded in C++. In addition, we output a default computation schedule which parallelizes the stencil along the outermost dimension and vectorizes the innermost. We also output utility code that can be used for auto-tuning the schedule.

App	LoC	Found	Translated
StencilMark	245	3	3
CloverLeaf	6635	45	25
NAS MG	1770	9	1

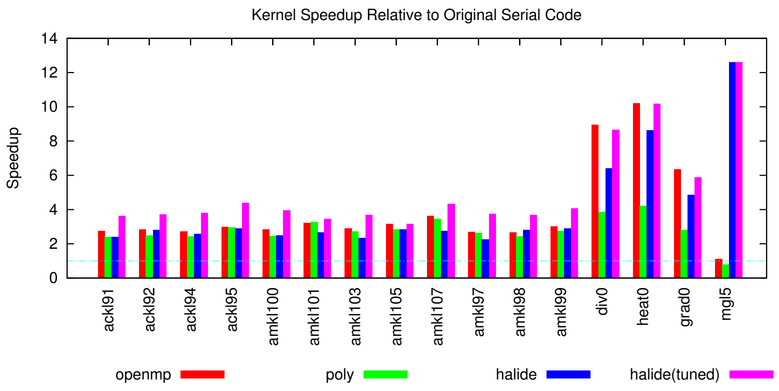


Figure 4 Summary of application results using STNG (top); Performance results (bottom).

Connector API Spec. Connector code is needed to replace the original stencil computation with one using Halide, including creating Halide buffers for each input and output grid. While buffer creation is tedious to write by hand, the automation is fairly straightforward, as it includes converting from Fortran array declarations to Halide buffers that wrap the arrays, and determining the array bounds for each Halide buffer.

5.3 Results

We evaluated STNG on two applications and a set of microbenchmarks for stencil computations, and compare against the state-of-the-art fully automated optimization method using PolyOpt/Fortran [22], a polyhedral compiler for Fortran. Selected results are shown in Fig. 4. As shown in the figure, STNG was able to translate most of the candidate code fragments, and the resulting code outperforms the version that is manually-parallelized or compiled using the polyhedral method.

The STNG prototype has some limitations that we are actively working to address; none of these are due to fundamental issues with the methodology. Currently, we do not support a number of Fortran constructs, including syntax for exponentiation, and do not support conditionals within stencil kernels. These limitations result in only a subset of kernels being translated, as shown in Fig. 4. We are extending the prototype to support these constructs and increase the effectiveness of translation.

6 Related Work

HERD builds on the successes of DSLs in improving application performance across different specialized domains. In this section we discuss other approaches in translating general-purpose to domain-specific code and structuring optimizing compilers.

The usual approach for compiler transformations is deductive: analysis and transformation rules determine which transformations are valid based on deducing characteristics of the original program text. Such approaches are fragile and tiny changes in the program can result in transformation failure [20]. In contrast, SET uses inductive techniques, by searching over combinations of possible constructs to find one with equivalent semantics to the original.

6.1 From General-Purpose to Domain-Specific Code

When converting from general-purpose code to domain-specific code, automated systems derive programmer intent in one of three ways. Pragmas or other annotations (such as those used in OpenMP [11]) require programmers to explicitly denote portions of the code to transform. Alternatively, there has been work on using language features such as types or specific classes [7, 8, 25, 31] for programmers to convey domain-specific information and intent. To our knowledge, Set is the first to use synthesis and verification technology to automatically determine domain-specific intent.

6.2 Code Generation as Search

Traditional optimizing compilers are constructed as a pipeline of code transformations, with each stage making structural changes to the code to improve its performance. Unfortunately, deciding the optimal order to execute each stage is a difficult problem [30]. Recent work has instead focused on using search techniques to find optimized code sequences. For instance, STOKE [27] uses stochastic search to solve the superoptimization problem. Like superoptimization, Herd casts the optimization problem as a search over all possible programs that can be constructed from a number of base operators (constructs in the target DSL in this case). Unlike Herd, however, STOKE performs search over low-level instructions rather than high-level logic expressions.

Meanwhile, inductive synthesis techniques have been used to search for optimal code sequences in data-parallel programs [3] and bitvector manipulation codes [14], although the search was not over multiple potential target languages, as in the case of Herd.

7 Conclusion

In this paper, we describe the Set methodology for building compilers that can automatically convert general-purpose code fragments into DSLs. Furthermore, we constructed Herd to allow DSL designers to use this methodology for building compilers by building kernel translators. We describe our experience applying Herd to build compilers to translate general-purpose code fragments into DSLs (SQL and Halide). A number of challenges remain, such as the limitations of synthesis technologies in finding postcondition expressions, and studying the relative ease of developing kernel translators in comparison to traditional syntax-driven compilers. Future work will extend Herd with new kernel translators, applying the methodology to other application domains.

References

1 Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–17, 2013.

2 Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 303–316, 2014.

3 Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. From relational verification to simd loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 123–134, 2013.

4 Gerald Baumgartner, Alexander Auer, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J. Harrison, So Hirata, Sriram Krishnamoorthy, Sandhya Krishnan, Chi chung Lam, Qingda Lu, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, P. Sadayappan, Alexander Sibiryakov, D. E. Bernholdt, A. Bibireata, D. Cociorva, X. Gao, S. Krishnamoorthy, and S. Krishnan. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. In *Proceedings of the IEEE*, page 2005, 2005.

5 Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*, OOPSLA'87, pages 441–451, 1987.

6 Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: introduction. *STTT*, 15(5-6):397–411, 2013.

7 Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Ka thy Yelick, and Armando Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*, Raleigh, NC, October 2009.

8 Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP'11, New York, NY, USA, 2011. ACM.

9 Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 3–14, 2013.

10 M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, May 2011.

11 Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

12 The django project. `http://www.djangoproject.com`.

13 Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design*, page 1, 2010.

14 Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–73, 2011.

15 Hibernate. `http://hibernate.org`.

16 F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'88, pages 319–329, New York, NY, USA, 1988. ACM.

17 Java Persistence API. `http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html`.

18 Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS*, pages 1–12, 2010.

19 David Maier. Representing database programs as objects. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, pages 377–386, New York, NY, USA, 1990. ACM.

20 Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT'11, pages 372–382, Washington, DC, USA, 2011. IEEE Computer Society.

21 Microsoft Entity Framework. `http://msdn.microsoft.com/en-us/data/ef.aspx`.

22 Mohanish Narayan. Polyopt/fortran: A polyhedral optimizer for fortran program. Master's thesis, Ohio State University, 2012.

23 Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

24 Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

25 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE'10, 2010.

26 Ruby on Rails. `http://rubyonrails.org`.

27 Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–316, 2013.

28 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

29 Yuan Tang, Rezaul Chowdhury, Bradley C. Kuszmaul, Chi keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *In SPAA*, 2011.

30 Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 147–156, 2006.

31 Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

32 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

Yedalog: Exploring Knowledge at Scale

Brian Chin[1], Daniel von Dincklage[1], Vuk Ercegovac[1], Peter Hawkins[1], Mark S. Miller[1], Franz Och*[2], Christopher Olston[1], and Fernando Pereira[1]

1 **Google, Inc.**
`{brianchin,danielvd,vuke,phawkins,erights,olston,pereira}@google.com`

2 **Human Longevity, Inc.**
`och@humanlongevity.com`

Abstract

With huge progress on data processing frameworks, human programmers are frequently the bottleneck when analyzing large repositories of data. We introduce Yedalog, a declarative programming language that allows programmers to mix data-parallel pipelines and computation seamlessly in a single language. By contrast, most existing tools for data-parallel computation embed a sublanguage of data-parallel pipelines in a general-purpose language, or vice versa. Yedalog extends Datalog, incorporating not only computational features from logic programming, but also features for working with data structured as nested records. Yedalog programs can run both on a single machine, and distributed across a cluster in batch and interactive modes, allowing programmers to mix different modes of execution easily.

1998 ACM Subject Classification D.3.2 Data-flow languages, Constraint and Logic Languages

Keywords and phrases Datalog, MapReduce

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.63

1 Introduction

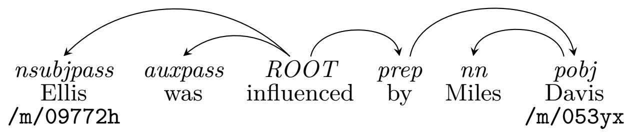
Organizations such as Google are busy assembling and exploiting vast and diverse repositories of "digital knowledge", such as corpora of natural language documents [18], the Knowledge Graph [17], and search engine query logs. Working with these repositories is largely semi-automatic: programmers assemble, curate, and learn from these data sets by repeatedly deploying simple algorithms over vast amounts of data.

With huge progress on data processing frameworks, such as MapReduce [11] or Spark [41], the main bottleneck is the human programmer. The speed at which we can conduct experiments on our data, iterate upon those experiments, and scale our code up to larger data sets, are all fundamental factors limiting progress.

As an example of a data problem, suppose we want to find influential jazz musicians. One way we might do so is to look for sentences similar to "X was influenced by Y" in a large corpus of crawled web documents. We assume standard NLP techniques have already been applied to our corpus; for example, the text has been parsed using a dependency parser (e.g., [25]) and nouns that name entities have been resolved to unique identifiers (e.g., to Freebase MIDs [4, 22, 18]). Figure 1 shows a typical parsed, annotated sentence from our corpus.

Given a corpus containing billions of such sentences, we might find influential musicians by computing statistics, such as the frequency with which a particular musician is a dependent

* Work done at Google, now at Human Longevity, Inc.



```
{url: "http://en.wikipedia.org/wiki/Cold_Sweat",
 hostname: "en.wikipedia.org", size: 200, accessed: 1418500000,
 tokens:[{text: "Ellis", kind: 'NSUBJPASS, parent: 2, entity: "/m/09772h"},
         {text: "was", kind: 'AUXPASS, parent: 2},
         {text: "influenced", kind: 'ROOT},
         {text: "by", kind: 'PREP, parent: 2},
         {text: "Miles", kind: 'NN, parent: 5},
         {text: "Davis", kind: 'POBJ, parent: 3, entity: "/m/053yx"}]}
```

Figure 1 A parsed document and its representation as a Protocol Buffer, in Yedalog syntax.

of the verb "influenced". This example is typical of the kind of experiment that programmers commonly want to conduct when working with such data sets. Each experiment may or may not work out, so we want to run many experiments casually and quickly.

Solving this task has both computational aspects, namely navigating parse trees, and data-parallel aspects, namely distributing the work of processing a corpus across a cluster. Most existing tools for data-parallel computation embed a sublanguage of data-parallel pipelines in a general purpose language (e.g., LINQ [27], DryadLINQ [40], Flume [6], Spark [41]), or vice-versa (e.g., Pig [31], Dremel [28], and SQL-based approaches [16, 13]). There are few languages where both computation and data-parallelism are first class (a notable exception is Jaql [3]).

In practice, programmers use an assortment of tools, each with different strengths. A programmer might use a batch pipeline system, like Flume [6] or Pig [31], to iterate over the corpus of documents, and use Java or C++ to navigate the parse trees. If the output is large, then a programmer will make a sequence of ad hoc queries to understand the result using a distributed SQL-like system, such as Dremel [28] or Spark [41], or even scripts running locally on a workstation.

Switching among languages and tools has both cognitive and practical costs. For our example, the programmer must mix many different kinds of code (e.g., Flume pipelines, Java, SQL, and SQL UDFs), leading to complex code that is difficult to write, extend, or debug. Because each task is in a different language, it is difficult to try out a task on small data on a local machine, or to re-deploy a batch job on a high fan-in topology if the intermediate data turns out to be small enough. The programmer must duplicate code, and write boilerplate to deal with mismatches between the data models of each language and the on-disk data.

In this paper we describe Yedalog (Knowledge Logic), a declarative programming language based on Datalog. Yedalog has two primary goals:

- Make computations over semi-structured data easy and succinct (Section 3). Data problems require lots of experimentation, and smaller code allows faster iteration. A declarative language enables programmers to solve problems quickly by focusing on the high-level logic of a task instead of low-level details. Working with semi-structured data is often laborious and error prone. Yedalog's data model is based around protocol buffers [19], which combined with unification, makes it easy to read, write, match, and transform semi-structured data. Yedalog is based on Dyna-style [14] weighted deduction,

```
Documents = Load{path: "document-parse-trees"};

# Code to execute for each document:
module PerDocument{tokens: T} = {
  # Computes parent-child relationships
  Child{p} = c :- T[c] == {parent: p, .._};

  # Computes nodes that transitively descend from "influenced"
  Descendants{t: c} :- T[p] == {text: "influenced", .._}, c == Child{p};
  Descendants{t: c} :- Descendants{t: p}, c == Child{p};

  # Counts nodes in which each entity appears under the verb "influenced".
  Influence{mid} += 1 :- Descendants{t}, T[t] == {entity: mid, .._};
};

# Each entity's influence from each hostname
Influential{mid, hostname} += count :-
  Documents{tokens, hostname, .._},
  PerDocument{tokens}.Influence{mid} == count;

# Persists Influential as protocol buffer data
? Store{data: Influential, path: "influential-entities"};
```

Figure 2 A batch Yedalog job: find influential entities using parsed web documents.

not only making it easy to compute and work with statistics, but also forming a pleasing hybrid between functional and logic languages.

- Combine data-parallel pipelines and computation in a single language (Section 4). Both are necessary for working with large corpora of rich data, such as parse trees, and combining both in a single language leads to simpler code. Yedalog can run data-parallel pipelines on three different execution platforms, each useful for different tasks: (a) on a single machine over small data, (b) on an high fan-in distributed system for interactive queries that yield small answers from big data (like Dremel [28]), and (c) as pipelines of map-reduce jobs for batch tasks (using Flume [6]). Yedalog complements purely declarative programming by allowing the programmer to override Yedalog's automatic choices using physical annotations. Physical annotations help programmers scale up declarative code without having to rewrite it in a lower-level language.

We have built a prototype implementation, in use by a handful of teams at Google (Section 5). Yedalog is an ongoing research project – this paper describes our progress to date and motivates future research (Section 6).

2 A Running Example

We introduce Yedalog using a running example. Subsequent sections explore specific features in more depth: Section 3 describes a number of language features that have proved especially valuable, and Section 4 discusses data-parallelism.

Yedalog builds on a long history of related work, notably Datalog, Dyna [14] and Jaql [3]. Our goal is not to give a formal description, but instead to survey the key features and to motivate future research.

As a running example, we use the code in Figures 2 and 3 that finds influential jazz

```
Influential = Load{path: "influential-entities"};
Freebase = Load{path: "freebase-snapshot"};

# Computes IDs of jazz artists
@small(JazzArtist); # physical annotation
JazzArtist{mid} :-
  Freebase{sub: mid, pred: "/music/artist/genre", obj: "/m/03_d0", .._};

# A jazz artist's influence, summed across documents from Wikipedia
JazzInfluence{mid} += count :-
  Influentual{hostname: "en.wikipedia.org", mid} == count,
  JazzArtist{mid};

# Prints the ID and count of each jazz artist.
? JazzInfluence{mid} == count;
```

Figure 3 A high fan-in job: find influential jazz musicians using the output of Figure 2.

musicians using a corpus of parsed web documents. We briefly walk through the example. Figure 2 loads a corpus of parse trees represented as a collection of protocol buffers (`Documents`), iterates over the corpus counting entities that are dependents of the verb "influenced" (`Influential`), and stores the result as a protocol buffer collection. Figure 3 takes as input the `Influential` data saved by the first part, joins it with a snapshot of `Freebase` [4] on `mid` to find jazz artists, and prints their mids and counts.

A Yedalog program, like Datalog, consists of *rules* and *queries*. Rules bind a value or a predicate to a variable name. For example, in Figure 2, line 17 defines a predicate named `Influential`.

Semantically, as in Datalog, a predicate is a collection of records. There are three key differences from Datalog:

- records have *named* fields. Since predicates are just bags of records, both predicate definitions and calls have *named* arguments, written `{f:x}`. There is a simple correspondence between Yedalog records and protocol buffer messages (Section 3.1).
- records can have *values*. Inspired by Dyna [14], Yedalog extends Datalog with values (Section 3.2). All predicates and expressions may have values, representing, say, statistics, weights, or probabilities.
- by default predicates are bags (multisets) of records, not sets.

There are two shorthands that we use frequently. The common case `{f:f}` where the argument and the variable have the same name can be written using the shorthand `{f}`. It is also often convenient to use a more traditional positional notation. Tuples (e.g., `(1, "foo")`) are a shorthand for records that have numbers as field names (e.g., `{0:3.1416, 1:"foo"}`). Predicates can thereby use positional notation (e.g., `Store(x, y)` as a shorthand for `Store{0: x, 1: y}`).

Core operators include unification (`==`), conjunction (`,`), disjunction (`|`), negation (`!`), if-then-else (**if** x **then** y **else** z), together with the usual arithmetic operators. The `+=` operator in the head of line 17 is an aggregator (Section 3.2). Operator `x[y]` is the list indexing operator, which looks up or scans over the elements of a list by index.

Predicates are first-class in Yedalog; the result of the `Load` predicate in Line 1 is a predicate that opens an on-disk collection of protocol buffers, binding `Documents` to a predicate that reads the data as a value. Evaluation in Yedalog is lazy, so the data will be read on-demand.

In Figure 2 line 4, `PerDocument` is a module constructor – a predicate whose value is a record representing an instance of that module. Here, the module primarily saves us from repeating the `tokens` argument. Definitions in a module's body become fields of the record; for example, the value of `PerDocument{tokens}.Influence` is a closure of `Influence` with a specific value for `tokens`.

For each document in `Documents`, predicate `Influential` instantiates the `PerDocument` module with the list of token records from that document, where the token's `parent` fields link them into parse trees. Predicate `Descendants` finds nodes that descend from the text `"influenced"` in those parse trees. `Influence` counts how many such tokens refer to any particular entity. `Influential` counts tokens for each entity (`mid`) per `hostname`.

In Figure 3, Predicate `JazzArtist` computes the mids of jazz artists in Freebase. Finally, for each influential entity that is also a jazz artist, `JazzInfluence` sums influence counts across Wikipedia documents. The query at the end starts the evaluation and displays the results.

3 Working with semi-structured data

Yedalog makes many extensions to Datalog. We focus on two that are particularly useful for working with semi-structured data. Firstly, Yedalog's data model is based on Protocol Buffers (Section 3.1); when combined with unification this gives us a convenient facility for pattern matching on structured data. Secondly, Yedalog incorporates Dyna-style weights, which is key to working with statistics over data (Section 3.2).

3.1 Data Model: Protocol Buffers and Unification

When working with data on disk or over the network, programmers write large amounts of code to convert data to and from its serialized representation. Yedalog's data model is based on Protocol Buffers [19], which Google uses as a common representation of data. The direct correspondence between the two allows us to work directly with external data in protocol buffer form, removing the need for tedious serialization code.

Yedalog *records* correspond directly with protocol buffer messages, which consist of name-value pairs. Yedalog's primitive data types correspond with the primitive protocol buffer types: `null` (representing absence), booleans, integers, floating-point numbers, and strings. Repeated fields in protocol buffers correspond to *lists* (e.g., `[1, 7]`), and protocol buffer enumerations correspond to *symbols* (e.g., `'PREP`). Records and lists may be nested (giving up the decidability of Datalog).

In Figure 2, each web document is represented as a nested protocol buffer, as shown in Figure 1. The `Documents` rule (line 1) loads a table of such data from disk as a predicate. We can work with the result like any other predicate.

3.1.1 Record Unification

A key operation when working with semi-structured data is *pattern matching*, that is, selecting records that match particular criteria and extracting data from them. Yedalog adapts the term unification of logic programming to nested records of named fields, giving a convenient way to match and build protocol buffers. Unification over records of key-value pairs dates back to Kay's work on feature structure unification [23]; we argue that it is time that the idea was revived.

A *record pattern* is a record containing free variables that unifies with a subset of records at run time. (Yedalog does not allow non-ground terms, so record patterns are a strictly syntactic construct.) Since predicates are just bags of records, both the head of a predicate's definition and the arguments to a call to a predicate are record patterns. We can use record patterns to select records matching particular values, or to extract fields:

```
? Documents{accessed:1418500000, size, .._};
# size: 200

? Documents{hostname, url:_, tokens:_, ..rest};
# hostname: "en.wikipedia.org", rest: {size: 200, accessed: 1418500000}
```

A record pattern consists of *fields* `{f:e}`, and an optional *rest* `{..e}`. Fields match the value of a field `f` against an expression `e`. A rest is a wildcard that matches all fields not explicitly named. As with argument records, we abbreviate field patterns `{p: p}`, where the argument is a variable with the same name as the field, as `{p}`. Underbar (`_`) means "don't-care".

Record patterns may be nested:

```
Menus{cafe: "Cafe A", dish: {name: "Bread", allergens: ["wheat"]}};
Menus{cafe: "Cafe B", dish: {name: "Salted peanuts", allergens: ["peanut"]}};
Menus{cafe: "Cafe B", dish: {name: "Cake", allergens: ["peanut", "wheat"]}};

# Find dishes that don't contain peanuts.
? Menus{cafe, dish: {name, allergens, .._}, .._}, !(allergens[_] == "peanut");
# cafe: "Cafe A", name: "Bread", allergens: ["wheat"]
```

Since unification is reversible, we can also use record patterns to build records. For example, the following `Token` predicate extracts the `tokens` list from each document, extracts each token record from the list (`tokens[i]` is the list indexing operator), and builds a relation consisting of the contents of token records and their indices.

```
Token{index, ..t} :- Documents{tokens, .._}, tokens[index] == t;
? Token{index:0, text, parent, .._};
# text: "Ellis", parent: 2
```

3.2 Weights, Values, and Aggregation

Yedalog attaches *values*, such as weights, probabilities, or counts, to facts and to expressions. For example, the `Child` predicate (Figure 2, line 6) has a *value*, attached using the `=` operator, in addition to its named parameter `{p}`. Facts in `Child` are pairs, where "`p`" is the parent and the value is the child.

Yedalog's aggregation syntax, which follows Dyna [14], groups the records of predicates by a key, and combines the values using an *aggregator*. Common aggregators include `+=` (sum), `*=` (product), `Min=` (minimum), `Max=` (maximum), and `Avg=` (mean). Users can define their own aggregators.

For example, the `JazzInfluence` predicate has a value defined by the aggregator "`+=`", rather than "`=`". Instead of returning all the separate `{mid} = count` pairs as facts, aggregation groups records by the fields in the head (in this case `mid`), and combines multiple values using the `+` operator. Here, the rule sums the counts for each `mid`.

Aggregation and recursion can be combined for computations over *semirings*, giving a succinct way of solving many dynamic programming problems. For example, given an `Edge` predicate associating graph edges with weights, the `Dist` predicate below computes the shortest path between pairs of nodes.

```
Edge{s: "a", t: "b"} = 0.3;
Dist{s, t} Min= Edge{s, t};
Dist{s, t} Min= Edge{s, t:u} + Dist{s:u, t};
```

The `Min=` aggregation and the `+` operator form a semiring. The semiring structure allows Yedalog to compute shortest paths efficiently and incrementally using a variant of Datalog's semi-naive evaluation over semirings [14].

3.3 Expressions and Relational Composition

Yedalog does not just attach values to facts: expressions also have values, allowing us to program in a "functional" style. However, in the absence of an aggregator, expressions may have multiple values and need not be functions.

For example, the expression `Child{p}` may succeed many times; the successive values are the children of parent `p`. Nested expressions correspond to relational composition. The grandchildren of `x` can simply be written as `Child{p: Child{p: x}}`. We have found this easy composition of relations to be very convenient [20]; any expression can act as a generator.

An example of a multi-valued operator, is the element-of operator, written `x[y]`. The element-of operator may be used in two directions; either to look up an element of a list with a specific index, or to iterate over all elements of a list, reminiscent of XPath [8]:

```
? Documents{..d}, t == d.tokens[0].text;
# t: "Ellis"

? Documents{..d}, t = d.tokens[idx].text;
# idx: 0, t: "Ellis"
# idx: 1, t: "was"
# ...
```

Like all parameters to a predicate, values can also be inputs. This is convenient for pattern matching: for example, we can find documents whose URLs start with `"http://"`:

```
HasSchema(s) = url :- String.StartsWith(url, s);
? Documents{url: HasSchema("http://"), size, .._};
```

4 Combining Data-Parallelism and Computation

Working with large corpora requires a mixture of distributed and local computations. The goal of Yedalog is to combine both in one language, allowing us to move seamlessly from local computation, to ad-hoc interactive analysis of data sets too large to fit on a single machine, to large-scale batch jobs.

Most existing tools for data-parallel computation embed a sublanguage of data-parallel pipelines in a general purpose language (e.g., LINQ [27], DryadLINQ [40], Flume [6], Spark [41]), or vice-versa (Pig [31], Dremel [28], and SQL-based approaches [16, 13]). Inspired by Jaql [3], Yedalog represents both computation and data-parallel pipelines in the same language.

Using a single language allows a planner visibility into both the data-parallel and computational parts of a pipeline, allowing more opportunities for optimization. On a purely pragmatic level, unifying both data-parallelism and computation means there is no syntactic overhead to combining the two, leading to shorter, more readable code.

In addition to running code on a single machine, Yedalog has two distributed execution platforms: a batch platform (Section 4.1) built on top of Flume [6], and an interactive service (Section 4.2) inspired by Dremel [28]. Each serves a different purpose, but code written for one can run on the others with minimal modification, subject to the constraints of the platform.

To run code locally, or within a distributed worker, Yedalog transforms predicates in the surface language into pipelines of pull-based generators that communicate via shared storage. Local evaluation is relatively standard, and we do not describe it in detail here, although Section 4.3 describes some of the more important aspects of the compilation process.

Our focus has been on programmer productivity and scalability over speed. The current Yedalog implementation is an interpreter, and its single-threaded performance is an order of magnitude slower than a compiled language such as C++. For many applications, programmer productivity and scalability are more important than speed. There are many standard optimizations that we could apply, such as JIT compilation, should it prove necessary.

4.1 Batch Backend

Yedalog can run programs in batch mode by converting them into Flume [6] pipelines, that in turn execute as sequences of map-reduce jobs. The batch backend is best suited for long-running jobs that produce large outputs or intermediate results.

For example, the `Influential` predicate (Figure 2, line 17) takes a corpus of parsed documents (`Documents`), and uses the `Influence` predicate in the `PerDocument` module to count entity `mids` that appear as dependents of the verb "influenced". The predicate sums the resulting `count`, grouping the results by the `{mid, hostname}` fields. If the corpus of documents is sufficiently small, we can simply run this rule locally. However, for a large enough corpus, or if we were doing more substantial work than simply navigating within a tree, we might want to distribute the per-document work across a cluster.

There is a natural mapping from Yedalog rules to map-reduce jobs: rules correspond to map jobs, whereas aggregators correspond to reduce tasks. For example, for the `Influential` predicate, the body of a rule corresponds to a map task – for each document, do some per-document computation, yielding key-value pairs of the form (`{mid, hostname}`, `count`). The aggregator (`+=`) corresponds to a reduce task; we are to sum the counts for each key.

We leverage this insight to compile Yedalog programs into Flume pipelines. Compilation to Flume partitions relations and operators into two coarse cardinality classes: *big*, which represent data of arbitrary size, and *small*, which are small enough to fit in a single machine's memory. Operations on big relations are compiled into Flume pipeline operators, whereas small operations are run using the single-machine execution platform, either on the head node, or by a phase of the distributed pipeline.

For example, when compiling Figure 2 to Flume, the `Documents` predicate would be inferred to be big, either automatically or using a programmer-specified annotation (Section 4.3). The planner compiles the `Influential` rule by partitioning it into two parts – a big part, whose operators are translated into distributed Flume operators, and a small per-document part, which is executed using the single-machine backend:

```
InfluentialBig{mid, hostname} += count :- Documents{tokens, hostname, .._},
                                      InfluentialSmall{tokens, mid, count};
InfluentialSmall{tokens, mid, count} :- PerDocument{tokens} == t,
                                    t.Influence{mid} == count;
```

In this particular case, `InfluentialBig` becomes a single map-reduce job, whose mapper executes the code of `InfluentialSmall` code using the single machine backend.

4.2 Interactive Backend

Although the batch backend can scale to massive datasets, we frequently want to ask sequences of short analytical queries. For example, we might want to know the average size of a document, or the frequency of a particular construction. The batch backend is not suitable because of overheads such as high start up time.

Yedalog can compile to a high fan-in distributed service, which executes programs on a tree of shared workers, modeled after Dremel [28]. Compilation to the interactive service is very similar to compiling to the batch map-reduce backend; the interactive service is limited to queries that produce small outputs and intermediate results.

For example, the `JazzInfluence` predicate in Figure 3 takes the set of `Influential` entities produced by Figure 2, and counts entities that both come from Wikipedia and are jazz artists. When executed using on the interactive backend, each leaf worker scans over a shard of the `Influential` table, accumulating a table mapping `mids` to `counts`. Non-leaf workers sum the counts for each `mid` from each of their children, and the root worker returns the grand totals to the client in seconds.

4.3 Declarative Code, Planning, and Physical Hints

Yedalog aims to let programmers focus on the logic of their program without laboring over the mechanics of how it is to be run. Given a Yedalog program, the key challenge is *planning*, that is, choosing a feasible and efficient operational strategy for running the program. For example, the planner must choose how information flows through rules (moding), whether code should run locally or distributed, whether to materialize a predicate as a table (materialization) and if so, which indexes to create (indexing), and which algorithms to use for joins and in which order (join planning).

It is not realistic to expect that any planner will always make optimal decisions about how to run every program. Many commercial relational database management systems include a hint language to guide the query optimizer to make better choices [32, 29]. Motivated by *physical transparency* [3], rather than hiding execution details and attempting fully-automatic optimization, we use a simple, best-effort planner to handle most cases adequately, and rely on monitoring and programmer-supplied physical annotations for the rest.

Users are offered a *monitor* tool that shows their (logical) Yedalog program overlaid with color-coded execution statistics (e.g., time spent in each predicate) and *physical annotations* encoding the choices made by the planner. The user can use the execution statistics to spot bottlenecks, and add manual physical annotations if necessary (e.g., force a different join order, or request an additional index).

For example, the `@small` annotation in Figure 3 line 5 marks the `JazzArtist` predicate as small (Section 4), implying that it fits in memory. There are only a few thousand jazz artists in Freebase, but the planner cannot know that without either an annotation or a good statistical model of `Freebase`.

As another example, there are two possible ways we might compute the `JazzArtist` predicate. One option is that we can scan over `Freebase`, finding all triples with the right predicate and object. However if `Freebase` had a suitable index, then we could instead take `mid` as an input, and look up a particular `mid` to determine whether it corresponds to a jazz artist. We can use an `@mode` annotation on the call to `Freebase` to select which of the two physical strategies we intend. If we do not specify, the planner will choose one automatically.

5 Experience

The Yedalog implementation is still at an early stage, but is used by a handful of teams at Google, for tasks ranging from ad hoc analysis of linguistic data, to production pipelines that use Yedalog to identify malware. For example:

- Teams working with natural language are using Yedalog to solve a range of different tasks, along the lines of the running example in this paper.
- Several teams are using Yedalog in pipelines that find and classify malware. The Yedalog program takes as input protocol buffers containing signals that describe, for example, a web page. Yedalog rules are used to match combinations of input signals and compute a numerical score for each candidate. Yedalog is used online in production to classify new candidates as they arrive. Yedalog is also used offline to test the efficacy of new rules by applying them to a corpus of historical data, using Yedalog's interactive and batch distributed backends. (Other authors have explored domain-specific languages and libraries for similar tasks [5, 26].)
- A team is using Yedalog to write data integration pipelines. Data about the same entities is currently spread across several different data sets. The team is using batch Yedalog pipelines to merge and transform data about an entity into a single data set. Yedalog is a good fit for batch pipelines that transform data in protocol buffer form.

5.1 Case Study

We conducted an informal Yedalog case study using a research task that classifies instances of certain linguistic constructions in a corpus of text. An existing implementation used a combination of C++, MapReduce and domain-specific libraries to find examples of the construction in the corpus, and used Dremel interactively to compute statistics about the relative frequency of particular examples.

An expert Yedalog programmer with limited domain knowledge reimplemented the task in Yedalog in a few days. The Yedalog programmer primarily worked from a specification, but consulted the C++ code to verify several details. The new implementation used 70% fewer lines of code than the original. The two implementations had similar scaling properties since both compiled to similar pipelines of MapReduce jobs. The Yedalog interpreter's single-threaded performance, while an order of magnitude lower than the C++ implementation, was acceptable for the task because of parallelism.

Initially, there were some puzzling discrepancies in the outputs of the two implementations, which turned out to have a number of different causes:

- part of the specification was miscommunicated to the Yedalog programmer,
- there were small logic bugs in the original C++ implementation; for example, the C++ implementation incorrectly handled matches that were supersets of other matches, and
- the original C++ implementation contained extra heuristics to work around a memory limitation in a piece of infrastructure that weren't present in the specification.

```
# The random jump probability.
Alpha = 0.15;

# Number of outlinks for every node.
Outlinks(j) += 1.0 :- Edge(j, _i);

# The (non-normalized) PageRank algorithm.
PageRank(i) += Alpha :- Node(i);
PageRank(i) += PageRank(j) * (1.0 - Alpha) / Outlinks(j) :- Edge(j, i);
```

Figure 4 PageRank in Yedalog. Assumes the existence of `Node` and `Edge` predicates that describe a directed graph.

Since the Yedalog code was much shorter, it was easier to check for correctness. After identifying and fixing these issues, the two systems produced identical output.

6 Research Challenges

6.1 Dynamic programming, Semirings, Recursion

Eisner et al.'s Dyna proposal [14] was one of the papers that inspired us to started working on Yedalog. Dyna demonstrates that a Datalog over weighted semirings can express a wide range of AI tasks elegantly, ranging from natural language parsing to constraint satisfaction. For example, Figure 4 expresses the classic PageRank algorithm [33] in just a few lines of Yedalog code.

Declaratively, this states the set of simultaneous equations whose solution is the rank of the web's pages. Operationally, this also expresses how to compute the exact answer recursively, by propagating updates along the graph until the values reach numeric convergence. Many other algorithms, such as CKY parsing and machine translation decoding can be expressed in equally elegant fashion.

However, computing the exact PageRank of the web would be both expensive and pointless. Practical approaches require significant operational control. Often we don't want precise solutions, but heuristic approximations that can be computed cheaply.

- We might stop early, well before actual convergence.
- We might opportunistically drop some updates – by unsynchronized writes to shared memory [35] or by ignoring stragglers [28] – in order to process other updates faster.
- We often need precise control over how search is done. For example, in a natural language parser, or in a machine translation decoder, we frequently use a beam search to find an approximate solution quickly, instead of an exact dynamic programming search.

Although we have expressed beam search "declaratively" in Yedalog, we found we had sacrificed the brevity and clarity benefits promised by declarative programming. A challenge for future research is to preserve the clear declarative statement of the result to be approximated, while separately stating the means of approximating it and how much distortion is implied [36].

Neither of our distributed backends are optimized for "big" recursive queries over "big" data sets. The approaches explained by [1, 37] are particularly promising.

```
# Predicate that translates English to French by making an RPC.
ToFrench(english) = french :-
  RPC.Call{
    host: "... elided ...", service: "TranslationService", method: "Translate",
    input: {text: english, src_lang: 'ENGLISH, tgt_lang: 'FRENCH},
    output: {translated_text: french, .._}};

EnglishCorpus("Shall I compare thee to a summer's day?");
EnglishCorpus("Thou art more lovely and more temperate:");

FrenchCorpus(french) :- Corpus(english), french == ToFrench(english);

? FrenchCorpus(french);
# french: "Dois-je te comparer à un jour d'été?"
# french: "Tu es plus belle et plus tempéré."
```

Figure 5 Interacting with services from Yedalog.

6.2 Interacting with Services

We often want to interact not just with tables of data on disk, but also with services running within a datacenter. Datacenter services are often provided using a remote procedure call (RPC) abstraction, via frameworks such as Thrift [15] or gRPC [12].

Since predicates are the common abstraction of both data and computation in a logic language, we represent query-only service APIs in Yedalog as predicates. Google's gRPC library [12] is based around protocol buffers; in Yedalog a remote procedure call is modeled as a predicate that inputs and outputs Yedalog records. Figure 5 shows an example of Yedalog code that uses Google Translate to translate a corpus of documents.

While it is convenient to represent services as predicates in Yedalog, it also poses a number of challenges. When interacting with remote services, it is important to hide latency via parallelism or batching. For example, if each translation call is synchronous and has, say, 10ms of network latency overhead, translating a corpus of a thousand sentences will take at least 10 seconds. If we made all the requests simultaneously, the latency overhead would be only 10ms.

It should be the obligation of the programming language to hide latency automatically. Yedalog has preliminary support for hiding RPC latency via batching, and we intend to explore this further in future work.

6.3 Incremental and Streaming Computation

A natural extension of our work would be to use Yedalog to maintain views of data incrementally and efficiently. For example, in the example of Section 2, we should be able to maintain the output statistics efficiently in the presence of additions or removals of documents from the input corpus.

There has been an immense body of work on maintaining incremental views in the database community (e.g., [21], [41], [30], [37], and many others). We believe that Datalog variants with weights over ring and semiring structures are well suited to the automatic and efficient computations of incremental views.

7 Related Work

Yedalog builds upon and incorporates many ideas from the research literature.

7.1 Datalog

Dyna's pioneering work on the elegant expression of natural language processing and machine learning algorithms [14] helped inspire our project. Yedalog uses a number of ideas from Dyna, including attaching values to facts, the syntax for aggregators, the use of aggregators for weighted deduction from noisy data, and first-class modules (Dynabases).

Yedalog builds on a long history of work on Datalog and deductive databases. Like Coral [34], Yedalog mixes top-down and bottom-up evaluation strategies. Yedalog's system of modes and compilation approach is inspired by Mercury [39]. A number of authors have also used Datalog as a language for data parallel computations. Shaw et al. [38] explored compiling Datalog programs into map-reduce jobs, focusing more on efficient recursive map-reduce computations. Socialite [37] is a distributed Datalog for recursive queries over social graphs.

7.2 Languages for Data-Parallelism

Existing tools for data-parallel computations, e.g., Flume itself [6], Pig [31], Spark [41], Dremel [28], LINQ [27], DryadLINQ [40], T-LINQ [7] and approaches that integrate MapReduce into SQL (e.g., [16, 13]) either embed a computation language into a language of data-parallel pipelines, or vice-versa. For example, in Flume, the user builds data-parallel pipelines using a library of pipeline operators, and expresses local computation by subclassing function objects. In Dremel, the large-scale structure of a query is expressed using a SQL dialect; if the user desires complex computation, they must write foreign functions in a distinct language. Jaql [3] and Yedalog represent both computation and data-parallelism in one language.

7.3 Database and Persistent Programming Languages

Object database systems seek to blur the boundary between set-oriented (readily parallelizable) database operations and general computation, along with persistence. The GemStone system [10], an early influential object-database system, extends Smalltalk's object and computational model smoothly to include shared, persistent, query-able data. Likewise, O2 [2] and ObjectStore [24], are object-database systems with a bridge to typed C++ objects. Yedalog, like most deductive database systems, naturally only expresses query-only computation, mostly sidestepping issues of persistence and consistency.

7.4 Semi-structured Data

There is a large body of work on working with semi-structured data, in particular data in XML and JSON formats. The canonical tools for working with XML data are XPath [8] and XQuery [9], which can be viewed as an adaptation of SQL-style queries to semi-structured data. Yedalog can be viewed as an adaptation of the Datalog family of database query languages to querying data in protocol buffer form. Jaql [3] is one of the more notable tools for querying large collections of JSON data and shares many goals with our work, although the two languages have many differences.

8 Conclusion

We have presented Yedalog, a declarative language for analyzing and transforming semi-structured data. Yedalog allows programmers to write both computation and data-parallel pipelines in the same formalism; both are first-class. Yedalog programs can run locally, and on interactive and batch distributed platforms. Yedalog hides operational details while allowing programmers to override the tools' choices. Despite being a young language, Yedalog's unique combination of features mean that it is already being used in production applications.

References

1 Foto N. Afrati and Jeffrey D. Ullman. Transitive closure and recursive datalog implemented on clusters. In *Proceedings of the 15th International Conference on Extending Database Technology EDBT*, pages 132–143, 2012.

2 François Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-oriented Database System: The Story of O2.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

3 Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

4 Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1247–1250, New York, NY, USA, 2008. ACM.

5 Louis Brandy. Fighting spam with pure functions. `https://www.facebook.com/notes/facebook-engineering/fighting-spam-with-pure-functions/10151254986618920`, 2013.

6 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'10, pages 363–375, New York, NY, USA, 2010. ACM.

7 James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *ACM SIGPLAN Notices*, volume 48, pages 403–416. ACM, 2013.

8 World Wide Web consortium. XML path language (XPath). `http://www.w3.org/TR/xpath-30/`, 2014.

9 World Wide Web consortium. XML query (XQuery). `http://www.w3.org/XML/Query/`, 2014.

10 George Copeland and David Maier. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD'84, pages 316–325, New York, NY, USA, 1984. ACM.

11 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

12 Google Developers. Introducing gRPC, a new open source HTTP/2 RPC framework. `http://googledevelopers.blogspot.com/2015/02/introducing-grpc-new-open-source-http2.html`, 2015.

13 David J DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. Split query processing in Polybase. In *Proceedings of the 2013 International Conference on Management of Data*, pages 1255–1266. ACM, 2013.

14 Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer Berlin Heidelberg, 2011.

15 Apache Software Foundation. Apache Thrift. `https://thrift.apache.org/`, 2015.

16 Eric Friedman, Peter Pawlowski, and John Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, 2009.

17 Google. Introducing the Knowledge Graph: things, not strings. `http://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html`, 2012.

18 Google. 11 billion clues in 800 million documents: A web research corpus annotated with Freebase concepts. `http://googleresearch.blogspot.com/2013/07/11-billion-clues-in-800-million.html`, 2013.

19 Google. Protocol buffers. `https://developers.google.com/protocol-buffers/`, 2014.

20 Ralph E Griswold and Madge T Griswold. *The Icon programming language*, volume 30. Prentice-Hall Englewood Cliffs, NJ, 1983.

21 Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD Record*, volume 22:2, pages 157–166. ACM, 1993.

22 Johannes Hoffart, Mohamed Amir Yosef, Ilaria Bordino, Hagen Fürstenau, Manfred Pinkal, Marc Spaniol, Bilyana Taneva, Stefan Thater, and Gerhard Weikum. Robust disambiguation of named entities in text. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 782–792. Association for Computational Linguistics, 2011.

23 Martin Kay. Functional grammar. In *5th Annual Meeting of the Berkeley Linguistic Society*, 1979.

24 Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, 1991.

25 Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

26 Simon Marlow and Jon Purdy. Open-sourcing Haxl, a library for Haskell. `https://code.facebook.com/posts/302060973291128/open-sourcing-haxl-a-library-for-haskell/`, 2014.

27 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD'06, pages 706–706, New York, NY, USA, 2006. ACM.

28 Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

29 Microsoft. Query hints (Transact-SQL). `https://msdn.microsoft.com/en-us/library/ms181714.aspx`, 2014.

30 Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 439–455, New York, NY, USA, 2013. ACM.

31 Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, New York, NY, USA, 2008. ACM.

32 Oracle. Database performance tuning guide. `http://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm#i8327`, 2015.

33 Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. 1999.

34 Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL deductive system. *The VLDB Journal*, 3(2):161–210, April 1994.

35 Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

36 Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 324–334. ACM, 2006.

37 Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. Distributed Socialite: a Datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

38 Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve Datalog evaluation in Hadoop. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry*, volume 7494 of *Lecture Notes in Computer Science*, pages 165–176. Springer Berlin Heidelberg, 2012.

39 Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1):17–64, 1996.

40 Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, volume 8, pages 1–14, 2008.

41 Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in Cloud Computing*, pages 10–10, 2010.

The Design of Terra: Harnessing the Best Features of High-Level and Low-Level Languages

Zachary DeVito and Pat Hanrahan

Stanford University, US
{zdevito,phanrahan}@cs.stanford.edu

Abstract

Applications are often written using a combination of high-level and low-level languages since it allows performance critical parts to be carefully optimized, while other parts can be written more productively. This approach is used in web development, game programming, and in build systems for applications themselves. However, most languages were not designed with interoperability in mind, resulting in glue code and duplicated features that add complexity. We propose a two-language system where both languages were designed to interoperate. Lua is used for our high-level language since it was originally designed with interoperability in mind. We create a new low-level language, Terra, that we designed to interoperate with Lua. It is embedded in Lua, and meta-programmed from it, but has a low level of abstraction suited for writing high-performance code. We discuss important design decisions – compartmentalized runtimes, glue-free interoperation, and meta-programming features – that enable Lua and Terra to be more powerful than the sum of their parts.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases language interoperability, meta-programming, high-performance, Lua

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.79

1 Introduction

A common approach for developing large applications is to use multiple languages that are each suited to particular tasks. This approach is used across many areas of development. In web development, high-speed servers (e.g, Apache) used for mostly static web pages are frequently written at a low-level in C/C++ for high throughput. Other parts of the web application might be written in a scripting language such as Javascript (Node.js) or Ruby (Ruby on Rails) for fast prototyping. A similar approach is used in scientific computing where Python, MATLAB, or R may be used for high-level prototyping. Since these languages are often an order-of-magnitude slower than good code in a low-level language, performance critical parts of the application are then re-written in a lower-level language (e.g., NumPy) and accessed using a foreign function interface (FFI). Video games share a similar design with most of the game engine written in a low-level language, and an embedded scripting language (commonly Lua) for describing game events.

Including both a high-level and low-level language is useful because it allows programmers to choose the level of abstraction for each task. Performance critical parts can be coded with an abstraction that is close to the machine which includes features such as manual layout and management of memory as well as access to low-level vector instructions. Other parts can be written more productively in higher-level languages (whether statically-typed like Haskell or dynamically-typed like Lua) that include features such as automatic memory management and higher-level data types which abstract the details of the machine. This design also allows people with different skills to work on the same project. A game designer might only need to understand Lua to be productive.

The approach of using multiple languages is also widely used in software development itself. A scripting language such as a shell script or Makefile control the compilation and deployment of software written in another language. This usage differs from the others because the use of multiple languages occurs during compilation and is used to organize or generate code. Nevertheless, multiple languages are still involved and must interact.

Most programming languages being used this way, however, were not designed with multi-language interoperability in mind. In applications that use multiple languages, this often manifests in two ways: as extra work in "gluing" the two languages together, and extra complexity in each language in the form of duplicated features.

Glue code can appear for many reasons. Symbols in one language need to be bound in the other. Data passed from one language to the other might need to be manually transformed. If values allocated in the runtime of one language can be seen from the other, then glue may be needed to ensure correct object lifetimes. For instance, C extensions to Python use `Py_INCREF` and `Py_DECREF` to maintain handles to Python objects and ensure they are not garbage collected.

Complexity also arises because multiple languages are often solving the same problems. Each might have its own solution for namespaces, package management, and compilation. This can produce a system that contains the worst aspects of each language. For instance, an application written in a scripting language does not require a separate compilation step, making iterative coding easier at the cost of static checking. But when a compiled language is added, the application now requires a compilation step *and* does not do any static checking on the scripting language.

We investigate an improvement to two-language programs that uses languages that were designed with this kind of interoperability in mind. For our high-level language, we use Lua since it was designed to interoperate with low-level C code [9]. C was not designed with interoperability in mind so instead we replace it with a new low-level language, Terra, that we designed from the beginning to interoperate with Lua. Terra's textual representation is embedded in Lua code. It is meta-programmed from Lua, and it relies on Lua to describe its static types. But Terra itself exposes a low-level of abstraction suited to writing high-performance code.

The approach of using two languages designed to interoperate allows each language to remain simple relative to single-language solutions, but the interaction between both languages allows for behavior that is more powerful than the sum of their parts. We can eliminate the glue code that couples two languages together and ensure that the languages do not contain redundant solutions for common problems.

In this paper we discuss important decisions we made when designing Terra to enable two-language interaction while keeping each language simple:

1. We compartmentalize the runtimes of Lua and Terra, allowing each language to focus only on features that they are good at. Terra code can execute independently from the Lua runtime, ensuring a low-level control of performance. It can also run where Lua cannot, such as on GPUs.
2. Interoperability does not require glue code since Terra type and function declarations are first-class Lua statements, providing bindings between the two languages automatically.
3. Lua is used as the meta-programming language for Terra. We provide facilities for generating code and types dynamically, removing the need for a separate offline compiler. We simplify Terra by offloading typical features of a compiled low-level language such

as class systems to the Lua meta-program. It also enables powerful behavior such as runtime generation of low-level code.

We start with background on programming in Lua and Terra, examine these design decisions in more detail, and briefly discuss our experiences using the language.

2 Lua and Terra by Example

Terra code itself is syntactically embedded in a Lua program. The keyword `terra` introduces a Terra function (`function` is used for Lua). Here is an example that declares both a Lua function and a Terra function:

```
  function addLua(a,b) -- Lua function
      return a + b
  end
  terra addTerra(a : int, b : int) : int  -- Terra function
5     return a + b
  end
  print(addTerra(1,1)) -- 2
```

Both share a similar syntax, but have different semantics. Lua is a dynamically-typed language with semantics that are similar to Javascript, including automatic memory management and high-level data types based on associative tables. In contrast, Terra is a low-level language with semantics that are analogous to C. It has pointer arithmetic, manual memory management, and low-level data types like C's. Terra functions include type annotations and are statically typed in the sense that types are checked at compile time. Compilation of Terra itself occurs dynamically as the Lua program executes.

Lua code can call Terra functions directly, as illustrated on line 7. This call causes the Terra function to be compiled to machine code. The arguments are converted from Lua values to Terra values, and the result is converted back to a Lua value (Section 3.2 has more details about this process).

Terra types are also declared using statements in Lua:

```
struct FloatArray {
    data : &float
    N : int
}
-- method declaration:
terra FloatArray:get(i : int) : float
    return self.data[i]
end
```

Terra entities (e.g., types, functions, expressions, symbols) are all first class values in Lua. They can be held in Lua variables and passed through Lua functions. For instance, we can get a result similar to C++ templating by simply nesting our type and function definitions inside a Lua function that takes a type as its argument:

```
function Array(ElemType)
  local struct ArrayType {
    data : &ElemType
    N : int
  }
  terra ArrayType:get(i : int) : ElemType
    return self.data[i]
  end
  return ArrayType
end
```

Both code (the method `get`) and types (the definition of `ArrayType`) can be defined dynamically in this way. When defined they capture the values in the local environment that they reference such as `ElemType`. We also allow the generation of arbitrary code using an approach based on multi-stage programming [3], and the generation of arbitrary types using a customizable API that runs during typechecking based on meta-object protocols [4]. We describe this functionality in more detail in Section 3.3.

3 Design Decisions

Our goal when designing Terra was to make a low-level language that complements the design of Lua and supports two-language programming as used in practice. Other approaches to two-language interoperability have focused on retrofitting existing languages [14, 5, 7, 18], and while these approaches can reduce glue code, the combination of both languages ends up more complicated than if the languages themselves were originally designed with interoperability in mind. Lua's design values simplicity as its most important aspect [8], so it is important that Terra is simple as well. Our design decisions focus on providing the benefits of multi-language programming while minimizing the complexity added to either language to support it.

3.1 Compartmentalized Runtimes

Though Terra programs are embedded inside Lua, we compartmentalize the runtimes of Lua and Terra. Each can run independently from the other, and only interact through fixed channels that we describe in the next section. This is one reason we refer to Terra as a separate language from Lua rather than an extension to it. This design decision allows both Lua and Terra to thrive for the tasks they are suited towards. We limit interaction to functionality that does not impose additional complexity on either runtime.

Each language has its own separate approach to code execution and memory management. For Lua, we use LuaJIT [14], a just-in-time trace-based compiler for Lua code. It handles the dynamic features of the Lua language well. Lua has its own heap where its objects are managed using garbage collection. In contrast, Terra code goes through a different pathway. It is compiled directly to machine code and optimized using LLVM [13]. This approach is less suited to dynamic features and takes longer than JIT compilation, but produces high quality low-level code. To further control the performance of Terra, memory is manually managed with `malloc` and `free`.

Alternative designs that combine high- and low-level programming attempt to augment higher-level languages with constructs that express lower-level ideas. For instance, in certain dialects of LISP, annotations allow the compiler to elide dynamic type safety checks to get higher numeric performance [20, 25]. Other dynamically-typed languages allow the optional annotation of types [1, 2, 5, 19, 27]. For instance, Cython [1] allows the programmer to optionally tag certain types in a Python program with C types, replacing them with a higher-performance implementation. Other types, however, go through the normal Python API. Type annotation can help eliminate boxing of certain values which often helps improve performance in local areas.

However, there is a difference between improving performance of a high-level program and obtaining near optimal performance that is possible when programming at a low level. Near optimal performance is often a composition of global factors including the specific memory layout of objects, careful instruction selection for inner loops, and use of advanced features such as vector instructions.

Intermixing high-level features with low-level features makes this careful composition harder to control. For instance, a missed annotation on a higher-level type may cause insertion of guards that will ruin the performance of an inner loop. The complexity of this approach makes the result harder to debug compared to a language like C where an expert programmer can disassemble the code and work out what is going on. In our experience building high-performance domain-specific languages [3, 6], the last order-of-magnitude of performance comes from this process of understanding the assembly and tweaking the low-level code that produces it. Furthermore, when high-performance facilities are only optionally offered, there is a tendency for libraries to be written without them, forcing anyone looking for whole-program performance to reimplement the library at a low level.

Maintaining compartmentalized runtimes has additional advantages for low-level programming. Since the runtimes are not intertwined, it is possible to run Terra code in places where the high-level Lua runtime does not exist. We can run Terra code in another thread (we use `pthreads` directly) without worrying about interaction with Lua's runtime or garbage collector. We can save Terra code generated in one process for offline use (we provide a built-in Lua function `terralib.saveobj` to do this). This allows us to use Terra as a traditional ahead-of-time compiler while still getting the benefits of meta-programming from Lua. We can also use Terra code where Lua cannot run, such as on NVIDIA GPUs. A built-in function `terralib.cudacompile` takes Terra code and produces CUDA kernels.

Finally, compartmentalizing the runtimes means that the semantics of each runtime is very similar to existing languages (Lua and C), each of which have benefitted from a large amount of engineering effort and optimization. Since we do not modify the semantics of these runtimes drastically, we can reuse this effort. In our case, this means using LuaJIT and LLVM as libraries for Lua and Terra, respectively. Terra itself is only around 15k lines of code (compared to Lua's 20k) and substantially shorter than languages whose semantics do not closely match C/LLVM such as Rust ($>$ 110k lines). Furthermore, Terra's similarity in types and semantics to C makes backwards compatibility possible. We include a library function `terralib.includec` that can parse C files and generate Terra bindings to C code.

3.2 Interoperability without Glue

Though the runtimes are compartmentalized, we still provide ways for Lua and Terra to interact. Terra code is embedded in Lua as a first-class value. We chose a Lua-like syntax for Terra as opposed to an alterative, such as using C's syntax, because it made parsing the extension easier and gave the languages a consistent interface. This is in contrast to languages frameworks like Jeannie [7] or PyCUDA [12], which preserve the syntax of an original language.

Both languages also share the same lexical environment as well. A symbol `x` in scope in Lua code is visible in Terra code and vice versa. We saw this feature in the example code where Terra code refers to `ElemType`, a Lua variable representing a Terra type (in fact, all types such as `int` or `double` are exposed in this way). When a Terra function references another function when making a function call, the reference is resolved through Lua's native environment during typechecking.

Sharing a lexical environment might seem surprising given that the languages have compartmentalized runtimes, but one way to think of this design is that the Terra *compiler* is part of the Lua runtime. Terra code itself runs separately from its compiler, much like the design of other compiled languages. It turns out sharing this scope simplifies the process of defining Terra code. Languages like C require their own syntax for declarations and definitions whereas Terra simply tracks these features as values in the Lua state. As we saw in Section 2, powerful features such as templating simply fall out of this decision. Additionally features like namespaces are accomplished by storing Terra functions in Lua tables and conditional compilation is done through Lua control flow. This design simplifies the implementation of Terra's compiler as well, since its symbol tables are simply the already present Lua environment.

We also want to be able to use values from one language in the other. We take an approach similar to other interoperability frameworks such as those used for Scheme and C [18], or LuaJIT and C [14]. When necessary, such as across functions calls between languages, values from one language are transformed using a set of built-in rules to values in the other. Numeric types often translate one-to-one, while aggregate types have more sophisticated rules. For instance, a Lua table `{real = 4, imaginary = 5}` can be automatically converted to a Terra struct with definition `struct Complex { read : float, imaginary : float}`.

It is also useful to allow one language to directly view a value in the other, rather than perform a transformation which necessitates a copy. This interaction, however, can become complicated when the low-level language can view high-level data structures that are managed via garbage collection since it must alert the runtime that a reference remains to the object. It would also compromise the compartmentalization of the runtimes, since it encourages Terra code to use Lua objects directly, which involves making Lua runtime calls inside Terra code. We adopt Lua's "eye-of-the-needle" approach [9] to this problem, which prevents Terra code from directly referencing Lua values. Instead Terra code can manually extract primitive data values from the Lua state using Lua API functions, or it can call back into Lua code directly.

The opposite behavior, viewing manually-allocated objects from a garbage collected language, has different constraints. These objects are already being managed by hand and performance considerations when accessing them from Lua are not as critical. For this direction, we adopt the approach of LuaJIT's FFI, which allows Lua to introspect aggregate structures like Terra's structs. A Terra struct `x` of type `Complex` can be returned to Lua without conversion. Statements in Lua such as `x.real` will dynamically extract the appropriate field and apply the standard conversion rules. For simplicity, Terra objects referenced from Lua follow the same rules for memory management as if they were used in Terra directly. That is, the programmer is responsible for ensuring a Terra object referenced from Lua is not freed too early, and for freeing it appropriately. This behavior makes interaction between the two languages possible without adding major complexity to either language.

A difference between our approach and the other frameworks is that the descriptions of Terra types are already natively declared as Lua values so it is not necessary to provide separate descriptions of aggregate datatypes through a side-channel. Other approaches that integrate with C (for instance) either need to parse the header files to see the declarations, or have a separate way of manually declaring the low-level types. We reduce redundancy and the complexity of the language by making Terra type definitions a part of the Lua runtime.

3.3 Meta-programmed by Default

Finally, we use Lua as the meta-programming language for Terra. This is useful for several reasons. First, it reflects a common use of scripting languages where a shell script or Makefile is used to arrange the compilation of low-level code. Making this process more seamless enables more powerful transformations, including building auto-tuned libraries or entire domain-specific languages that compile to Terra. Secondly, relying on Lua for meta-programming allows us to keep the core of Terra simple without limiting the flexibility of the language.

Terra provides facilities for both dynamically generating code, and for dynamically generating types. For code, we use a version of multi-stage programming [24] with explicit operators. A *quotation* operator (the backtick `) creates a fragment of Terra code, analogous to creating a string literal except that it creates a first-class value representing a Terra expression rather than a string. An *escape* (e.g., `[lua_exp]`) appears inside Terra code. When a Terra quote or function definition is encountered during the execution of the Lua program, the escaped expression `lua_exp` will be evaluated (normally to another quote) and spliced into the Terra code. It is analogous to string interpolation. For instance consider the following simple program:

```
function gen_square(x)
    return `x * x
end

terra square_error(a : float, b : float)
    return [ gen_square(a) ] - [ gen_square(b) ]
end
```

When `square_error` is defined, the `gen_square` calls in the escapes will evaluate, generating quotes that square a number, resulting in a function whose body computes `a * a - b * b`.

Generation of Terra code, like that shown above, is similar to other meta-programming frameworks such as LISP macros, or traditional multi-stage programming [16, 24]. But Terra also includes the ability to generate types dynamically as well. Terra is statically typed in the sense that types are known at compilation time, but Terra functions are compiled during the execution of the Lua program, which gives us the ability to meta-program their behavior. This is a different design from statically-typed multi-stage programming where the behavior of all types (those in the first stage and later stages) are checked statically. While it gives up the advantage of type checking the program entirely ahead-of-time, it also adds a lot of flexibility. For instance, we allow objects to programmatically decide their layout and behavior.

As an example, we may want to write a program that reads a database schema and generates a type from that schema, such as this student type:

```
terra example()
  var s : Student
  s:setname("bob")
  s:setyear(4)
end
```

Rather than hard-code the type, we can construct its behavior programmatically:

```
Student = terralib.types.newstruct() -- create a new blank struct
Student.metamethods.__getentries = function()
    -- Read database schema and generate layout of Student.
    -- (e.g., { {field="name", type=rawstring},
    --          {field="year", type=int}})
   return generated_layout
end
```

When the typechecker sees a value of `Student`, it asks for its memory layout by calling the `__getentries` function defined for the type, which returns a data structure describing the layout. The layout is defined before compilation, allowing the generation of high-performance code, but the implementation of `__getentries` can adapt to runtime information. The behavior of methods is also determined programmatically, allowing the generation of "setter" methods automatically:

```
   Student.metamethods.__getmethod = function(self,methodname)
       local field = string.match(name,"set(.*)")
       if field then
         local T = Student:typeoffield(field)
5        return terra(self : &Student, v : T)
           self.[field] = v
         end
       end
       error("unknown method: "..name)
10 end)
```

When the typechecker sees that `setname` is not explicitly defined, it will call the user-provided Lua function `__getmethod` to create it. We use string matching (line 2) to find the field name, and generate a body for the method (lines 5–7).

Terra's type system is a form of meta-object protocol [11]. It is similar in some ways to F#'s type provider interface [23], but describes types at a low-level like C rather than in a runtime such as .NET's common language runtime (CLR). It combines features of meta-object protocols from dynamic languages with the generation of low-level statically-typed code, and is described in more detail in [4]. Furthermore, this mechanism is the only form of user-defined type built-in to Terra, apart from some syntax sugar for common cases such as defining a statically-dispatched method.

Using a full scripting language to meta-program code and types allows us to omit many features of typical statically-typed language, keeping Terra simple. While this approach appears different from other statically-typed languages, it is possible to think of *every* statically-typed language as a meta-program containing multiple languages. One language, at the top level, instantiates entities like types, function definitions, classes, and methods. Another language in the body of functions defines the runtime behavior. The top-level language "evaluates" when the compiler for that language is run producing a second stage containing the actual program. But this language at the top level is normally not a full programming language. Some languages may have more built-in features (e.g. inheritance, traits, etc.) but these are very specific to the language. Having too few features in this top-level language, such as the initial lack of generics in Java, can prevent a language from being sufficiently expressive and provides little ability to grow the language [21], causing people to resort to ad-hoc solutions like preprocessors or source-to-source code generators to produce generic code.

Our approach in Terra replaces this limited top-level language entirely with Lua. Doing so removes the need for many language-specific features but provides a powerful mechanism to grow the language. In Terra features like class systems or inheritance are implemented as libraries that meta-program types. This behavior is familiar in dynamically-typed languages such as LISP's CLOS [11], but we extend it to work with a low-level compiled language.

4 Alternatives

Terra and Lua represent one possible design for a two-language system. It is also worth considering alternatives. We chose Lua as our high-level language due to its simplicity and its support for embedding in other languages, but most of the design decisions could be easily adapted to other dynamically-typed languages as well such as Python, Ruby, or Javascript. Another possibility is to use a statically-typed language such as Haskell or Scala as the high-level language. Systems such as lightweight modular staging in Scala [17] or

MetaHaskell [15] are examples of multi-language systems that take this approach. Their primary advantage is the increased safety of the high-level language. For example, some systems such as MetaHaskell or traditional multi-stage programming in MetaML [24] can statically guarantee that any code that can be generated in the target language will be well typed. Other systems such as the C++ library for generating LLVM [13] do not statically guarantee the type correctness of generated code, but still benefit from being able to give static types to the target language's core concepts including its functions, types, and expressions.

However, using a statically-typed high-level language in a two-language system can introduce added complexity. It adds an additional set of typing rules to the high-level language that is different from the low-level type system, as well as an additional phase of type checking. Some meta-programming systems such as MetaML avoid this complexity by making the staging language and target language the same, but this design is not possible when we explicitly want to use different languages.

Furthermore, the addition of static types can introduce complexity in other areas of the design, such as cross language interoperability. Static type systems often include some way of modeling abstract data types and behavior associated with them such as type classes or object-oriented class systems. Support for interoperability becomes more complex if it needs to support these built-in class systems. This complexity is evident in runtimes such as .NET's CLR. Calling from CLR managed code to unmanaged C/C++ code requires the generation of wrappers by the runtime so that CLR objects can be viewed from unmanaged code and vice-versa. It has several pathways to support different use cases (P/Invoke, COM wrappers, C++ wrappers) [10]. The added features to support class system behavior in such a model make it harder to compartmentalize the two runtimes and more difficult to understand their interaction.

Our goal creating Terra was to enable interaction between two languages while keeping the entire system as simple as possible. We chose to omit static typing in the higher-level language to decrease the complexity of the system as a whole. For example, since class systems are not built into Lua as they are in the CLR, the semantics for translating values from one language to another are simpler.

5 Experiences

We have used Terra to implement many applications in areas including linear algebra libraries, image processing, physical simulation, probabilistic computing, class system design, serialization, dynamic assembly, and automatic differentiation [3, 4, 6]. Our experiences using the system suggest it allows for simpler solutions by eliminating the glue code and management of separate build systems that occur in other approaches.

We found that many design patterns that occur in statically-typed languages can be expressed concisely in Terra, with performance that is similar to other low-level code. We implemented class systems similar to Java's (250 lines) or Google's Go language (80 lines) [3]. We were also able to implement concise generic proxy classes, such as an `Array(T)` type that forwards methods to its elements in only a few lines of code [4]. It performs as well as proxies written by hand in C. Patterns requiring introspection on types, such as object serialization were particularly well suited to Terra's design. For instance, we created a simple but generic serialization library in about 200 lines that could serialize scene graphs at 7.0GB/s [4] (compared to 430MB/s for Java's Kryo [22] library). The integrated solution to meta-programming allowed us to consider more aggressive optimizations that would normally only be done in separate preprocessors. A dynamic assembler we created [4] automatically generated method implementations to assemble instructions using a simple pattern language. It could also fuse patterns together into templates, resulting in speeds 3–20 times faster than the assembler used in Google Chrome.

For software programs that rely heavily on scripting to build libraries, such as autotuners, we were able to provide much simpler solutions. Our autotuner of matrix multiply can create code that runs within 20% of the ATLAS [26] autotuner and requires only 200 lines of code (compared to thousands in ATLAS) [3]. This simplicity arises from eliminating a lot of duplicate technologies. ATLAS uses `Makefiles`, preprocessors, offline compilers, and shell scripting. Combining this functionality into a single two-language system removes most of the complexity of the build systems, allowing the programmer to focus on code optimization.

Finally, the generic meta-programming features in a low-level language make it useful in developing high-performance domain-specific languages. We have used it to implement languages for probabilistic programming [3], and image processing [6] that are competitive with solutions written using traditional software tools but are easier to architect and distribute. Currently the use of domain-specific languages is not widespread. One reason for this is that current build tools do not make it easy to generate low-level code generically. We think that using this integrated two-language design as the basis for more software systems will make it easier to design, develop, and integrate high-level domain-specific languages in practice, allowing the use of higher-level abstractions without sacrificing overall performance.

References

1 Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science and Engineering*, 13.2:31–39, 2011.

2 Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, 2012.

3 Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. In *PLDI'13*, pages 105–116, 2013.

4 Zachary DeVito, Daniel Ritchie, Matt Fisher, Alex Aiken, and Pat Hanrahan. First-class runtime generation of high-performance types using exotypes. In *PLDI'14*, pages 77–88, 2014.

5 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA'05*, pages 231–245, 2005.

6 James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. In *SIGGRAPH'14*, pages 144:1–144:11, 2014.

7 Martin Hirzel and Robert Grimm. Jeannie: Granting Java native interface developers their wishes. In *OOPSLA'07*, pages 19–38, 2007.

8 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *HOPL III*, pages 2:1–2:26, 2007.

9 Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. Passing a language through the eye of a needle. *Commun. ACM*, 54(7):38–43, July 2011.

10 Interoperability overview (C# programming guide). `https://msdn.microsoft.com/en-us/library/ms173185.aspx`.

11 Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.

12 Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.*, 38(3):157–174, 2012.

13 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, 2004.

14 The LuaJIT project. `http://luajit.org/`.

15 Geoffrey Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In *ICFP'12*, pages 311–322, 2012.

16 John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, August 1978.

17 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *GPCE'10*, pages 127–136, 2010.

18 John R. Rose and Hans Muller. Integrating the Scheme and C languages. In *LFP'92*, pages 247–259, 1992.

19 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

20 Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical report, NEC Research Institute, Inc., 1999.

21 Guy L. Steele, Jr. Growing a language. In *OOPSLA'98 Addendum*, pages 0.01–A1, 1998.

22 Nathan Sweet. Kryo. `https://code.google.com/p/kryo/`.

23 Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. F#3.0 — strongly-typed language support for internet-scale information sources. Technical report, Microsoft Research, 2012.

24 Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.

25 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *PLDI'11*, pages 132–141, 2011.

26 R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw. Pract. Exper.*, 35(2):101–121, February 2005.

27 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *POPL'10*, pages 377–388, 2010.

The Need for Language Support for Fault-Tolerant Distributed Systems

Cezara Drăgoi[1], Thomas A. Henzinger*[2], and Damien Zufferey†[3]

1 INRIA, ENS, CNRS
Paris, France
cezara.dragoi@inria.fr

2 IST Austria
Klosterneuburg, Austria
tah@ist.ac.at

3 MIT CSAIL
Cambridge, USA
zufferey@csail.mit.edu

Abstract

Fault-tolerant distributed algorithms play an important role in many critical/high-availability applications. These algorithms are notoriously difficult to implement correctly, due to asynchronous communication and the occurrence of faults, such as the network dropping messages or computers crashing. Nonetheless there is surprisingly little language and verification support to build distributed systems based on fault-tolerant algorithms. In this paper, we present some of the challenges that a designer has to overcome to implement a fault-tolerant distributed system. Then we review different models that have been proposed to reason about distributed algorithms and sketch how such a model can form the basis for a domain-specific programming language. Adopting a high-level programming model can simplify the programmer's life and make the code amenable to automated verification, while still compiling to efficiently executable code. We conclude by summarizing the current status of an ongoing language design and implementation project that is based on this idea.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases Programming language, Fault-tolerant distributed algorithms, Automated verification

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.90

1 Introduction

Replication of data across multiple data centers allows applications to be available even in the case of failures, guaranteeing clients access to the data. For instance, state machine replication achieves fault-tolerance by cloning the application on several replicas, and ensuring that the same sequence of commands is executed on each replica using message passing. Building fault-tolerant software is challenging because designers have to face (1) asynchronous concurrency and (2) faults. Asynchrony means that replicas can interleave their actions in arbitrary ways and they communicate via message passing in a network that can delay

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award).
† Damien Zufferey was supported by National Science Foundation grant 1138967.

messages for an unbounded amount of time. Hardware faults can be transient, e.g., network partition, or permanent, e.g., crashes.

In a distributed system processes only have a limited view of the global state. However, one has to enforce some global property, e.g., ensuring consistency [1] between replicas by making sure that they execute a similar sequence of commands. Ensuring such a property typically requires solving a consensus problem: roughly, each process has an initial value, and all non-faulty processes have to agree on a unique decision value picked from the initial ones, even in the presence of faults and uncertainty in the timing of events. Strong consistency can be implemented by using consensus iteratively. Noteworthy examples from industry are Google Megastore, which uses Paxos [37] to replicate primary user data across data centers on every write, the Chubby lock service [11], Apache Zookeeper [33], which embeds a consensus protocol called Zookeeper Atomic Broadcast [35], or Microsoft Autopilot [34] that manages a data center.

State machine replication has received a lot of attention in both academia and industry. A fundamental result on distributed algorithms [29] shows that it is impossible to reach consensus in asynchronous systems where at least one process might crash. Consequently, a large number of algorithms have been developed [24, 21, 13, 37, 57, 42, 35, 46, 48], each of them solving consensus under different assumptions on the type of faults, and the "degree of synchrony" of the system. Moreover, weaker versions of consensus that solvable under fewer network assumptions [18, 26] have been proposed.

Consensus algorithms are difficult both from a theoretical and practical perspective. Because of the impossibility of solving consensus in asynchronous networks in the presence of faults, the existing consensus algorithms not only make various assumptions on the network for which they are designed but have also a complex flow of data. For example, the widely use Paxos algorithm [37] was considered hard to understand. A few years after its first publication Lamport wrote "Paxos Made Simple" [50] to re-explain the algorithm in simpler terms. However, the challenge of understanding it still remained. As an example, 2014 saw the publication of the raft algorithm [48], which purpose is to be simpler to understand than Paxos.

From a practical perspective implementing fault-tolerant distributed algorithms is challenging because the programmer has to use clocks, timing constrains, and complex network programming using low-level constructs, e.g., sockets, event-handlers. High-level programming languages like Erlang, or in general library based approaches, offer limited the possibility to reason about faults. Moreover, these algorithms are rarely implemented as theoretically defined, but modified to fit the constraints of the system in which they are incorporated. A group of engineers working at Google wrote "Paxos Made Live" [12] to describe some of the problems encountered while building a system based on Paxos and possible solutions for them. They [12, Section 9] emphasizes the following:

- "There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol."
- "The fault-tolerance computing community has not developed the tools to make it easy to implement their algorithms."

[1] Both strong or weak consistency are global properties.

- "The fault-tolerance computing community has not paid enough attention to testing, a key ingredient for building fault-tolerant systems."

Despite the importance of fault-tolerant distributed algorithms, there are no automated verification techniques that can deal with the complexity of an implementation of an algorithm like Paxos. A few algorithms have been mechanically proved correct using interactive theorem provers, like Isabelle [47]. However these proofs required substantial effort and machine-checked proof of correctness for asynchronous implementations would demand even more expert time. For automated verification methods, the main difficulties are the unbounded number of replicas, the complex control structure using event-handlers, sockets, etc., and the complex data structures, e.g. unbounded buffers.

We think that the difficulty does not only come from the algorithms but from the way we think about distributed systems. Therefore, we are interested in finding an appropriate programming model for fault-tolerant distributed algorithms, that increases the confidence we have in applications based on state machine replication.

Our goal is to develop a domain-specific language for fault-tolerant distributed systems. The language must

1. be high-level to help the programmer focus on the algorithmic questions rather than spending time fiddling with low-level network and timer code;
2. compile into working, efficient systems that preserve the important properties of high-level algorithms;
3. use a programming model amenable to automated testing and verification.

Even though some of the difficulty of implementing fault-tolerant algorithms were addressed in the distributed algorithms community, there isn't a widely accepted programming language for fault-tolerant algorithms. Therefore, we think that this class of systems is relevant also for the programming language community. If we look at the past few years of programming language design and implementation, as well as software verification, i.e., POPL, PLDI, OOPSLA, and CAV, we find that only a handful of papers deal with systems where faults can occur. We believe that one should explore the advances in formal methods to increase the confidence in fault-tolerant applications. The standard formal verification approach considers the asynchronous code implemented in general programming language like C or Java, which is known for being notoriously hard. Instead we propose use alternative view over replicated systems, leading to a new programming model that eases the programmers' task and is amenable to automated verification.

In Section 2, we give an example of a fault-tolerant algorithm and explain some of the challenges in implementing and verifying it. In Section 3, we review existing work in the domain of programming languages for message-passing concurrency, models of computation for distributed systems, and the automated verification of distributed algorithms. In Section 4, we highlight several promising research directions that combine ideas across different fields to achieve the goals stated above. This section also summarizes the status of our ongoing project on this topic.

2 Example

Figure 1 shows a fault-tolerant algorithm [9] that solves the consensus. The algorithm is given in the formulation of [16], which is a round-based model: all processes execute the same code in lock-step manner. Each process knows initially a (potentially) different value for the variable `x`. The algorithm first establishes a majority of processes agreeing on the same value for `x`, then witnesses that majority, and finally decides that the value of `x` is the one

```
Repeat:
    Send:      send x to all processes
    Update:    if received more than $2n/3$ messages then
                   x := the smallest most often received value
                   if more than $2n/3$ received values are equal to x then
                       decide(x)
```

Figure 1 A round based algorithm that solves Consensus.

the majority agreed on. This particular algorithm needs four replicas to tolerate one crash, i.e., $n > 3f$, where n is the total number of replicas and f is the number of faulty processes.

Distributed algorithms are typically published in pseudo-code, or, in the best case, a formal but not executable specification [38]. For the implementation, the programmer has to make many decisions that influence the final system in a way that is hard to predict a priori. The first decision is to fix the fault model. The algorithm in Figure 1 tolerates only *benign* faults, i.e., messages can be lost but not modified. Allowing a *byzantine* attacker, which corrupts messages, requires using a modified algorithm [6]. After deciding on a benign faults, one still needs to decide whether to consider *crash-stop* or *crash-recovery*. Crash-stop means that a fault causes a replica to stop taking any steps. On the other hand, in a crash-recovery model faults are transient and a replica can rejoin system after a crash. Implementing an algorithm is a crash-recovery fault model requires a *stable memory*, i.e., writing to a permanent storage every change of the local state, or using a dedicated *recovery procedure* to be executed by a replica before rejoining the system after a crash. Next there is the question if the communication is reliable or if the network can *drop messages*. This will influence how long the processes wait to receive messages and whether a missing message is interpreted as a crash. All these choices influence the implementation: what kind of *failure-detector* to implement, how much additional information has to be added to messages, etc. A failure-detector is an application which keeps track of the processes in the system to know whether they have failed. In its simplest incarnation, a failure detector periodically sends heartbeat messages to the others processes in the system. Then, for performance reasons, these messages can be piggybacked on top of the protocol's messages to avoid the overhead of sending additional messages. Such small optimizations quickly add up, perturb the normal flow of the protocol, and lead to a complex, hard-to-maintain codebase. Finally, when the algorithm is implemented, there is the question of how to test or verify it. Asynchronous systems are notoriously known hard to test, debug, and prove correct.

When defining the semantics of a programming language, it is simple and natural to assume an asynchronous model, but most programmers and algorithm designers have a notion of time when they write code. The round structure of the algorithm in Fig. 1 can be seen as an abstract notion of time. From a more practical perspective, in the context of the programming language P [20], we had discussions about adding constructs to deal with time. P is a domain specific language for communication state machine which is used to program device drivers. A common coding pattern for drivers is to send a request, initialize a timer, and wait for either the response or the timer to fire. If the response comes first, then the timer needs to be canceled. This pattern introduces boilerplate that obscures the code. The proposal was to add a new type of transition, `Timeout(duration)`, that would automatically initialize/cancel the timer. However, in the end we decided against adding timeouts to keep the number of primitives in the language small.

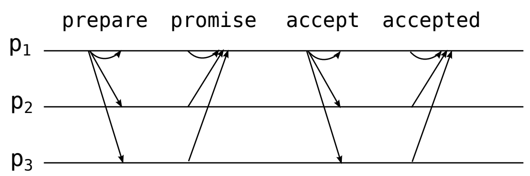


Figure 2 Paxos decomposed in four communication-closed rounds. Process p_1 has the role of proposer, acceptor, and learner. Processces p_2 and p_3 are acceptors.

3 Prior Work

Programming abstractions for message-passing systems. Formalisms such as the π-calculus [44, 45] and I/O-automata [41] have mostly been used to give a formal semantics to message-passing systems and to analyze them, but not as programming model. Some of these models have been extended with time, but they are all natively asynchronous. Furthermore, faults are not part of these models. To deal with faults one has to modeled them on top of the provided primitives, increasing dramatically the complexity of the final system.

The Actor model [31] is probably the most successful high-level abstraction for message-passing systems. CSP [32] has also seen some success with programming languages like Occam, Ada, etc., and CSP-style rendezvous concurrency is present in many other languages. Programming languages like Erlang [5] are based on the actor model and many programming languages have libraries implementing the actor model. Erlang also has dedicated library support to help the programmers write fault-tolerant applications. Finally, at the lowest level, we find the POSIX sockets. Disturbingly enough, they are probably still the most common programming abstraction for distributed systems.

In this paper, we focus on fault-tolerance though replication, considering algorithms that make uniform and non-uniform assumptions on the replicas. The programming models designed to implement replicated computation and storage [53, 58], typically take a centralized approach, based on a trusted client that interacts with all replicas. For example, the Erlang/OTP framework [53] provides fault tolerance using supervision. An application is hierarchically structured in a tree where the leaf nodes are workers and the inner nodes are supervisors. The supervisors watch over workers and other supervisors. They are responsible for starting and shutting them down in the normal course of action and, in presence of failure, responding to a fault in their children, usually by restarting the process. However, there are implementations of Paxos in Erlang designed to integrate with the OTP library and systems like Zookeeper, that combine both approaches, a uniformly replicated core that manages a large collection of workers. We are mostly interested in replicated systems that do not rely on supervision, and make uniform assumptions on the replicas, or non-uniform assumptions that are weaker than supervision.

Models of computation for distributed algorithms. Distributed algorithms and systems are mature research fields. This is by no means a complete overview of these fields, but we highlight a few ideas that have not been adopted by the language and verification communities and that we believe are good candidates to form the basis of new programming abstractions. An important idea to simplify the formulation of distributed algorithms is scoping the communication. *Communication-closed rounds* [25] encapsulate all communication within rounds. Conceptually, processes operate in lock-step: in each round they send messages, receive messages, and update their local state depending on the local state at the beginning of the round and the received messages. Many distributed algorithms can be formulated naturally in a round-based model [39, 40, 24, 16, 36, 7, 6]. Figure 2 shows how this can be done for the Paxos algorithm. Organizing the computation in rounds looks restrictive and originally was limited to synchronous systems. However, rounds can be generalized to partial synchrony [24]. Moreover, failure detectors [13] have been proposed to structure the detection and handling of failures. The combination of these ideas gives rise to round-based computational models that can *uniformly model synchronous and asynchronous systems* [30, 16, 3], e.g., Charron-Bost and Schiper [16, Table 1] showed how to do this for common fault models. In this view, an asynchronous, or faulty, system is a synchronous system with an adversarial environment that can delay, drop, or modify messages.

Verification of distributed algorithms. From the verification perspective, distributed algorithms are a very challenging class of systems because of several sources of unboundedness: messages come from unbounded domains, the number of processes is a parameter, and channels may also be unbounded. Indeed, most of the verification problems are undecidable for parameterized systems [4]. As alternative, model checking is done by instantiating the algorithms using a finite, usually small, number of processes [38, 54, 55, 56]. This approach works for finding bugs, but cannot prove an algorithm correct. Furthermore, the reduction to finite-state systems cannot model faithfully all types of faults, even for a finite number of processes. For instance, if we consider byzantine systems, a faulty process can send arbitrary messages that may not even be defined in the protocol.

The channels also make the verification problem hard. Unbounded FIFO channels causes undecidability even for two processes [8]. Making the channels lossy and fixing the number of processes makes the problem decidable [2], with a non-primitive recursive complexity [52]. Weaker channel models are usually at least EXPSPACE-hard for verification. A pervasive pattern in fault-tolerant systems is counting messages up to a threshold that depends on the number of processes in the system, e.g., $2n/3$ in Figure 1. Unfortunately, this means that these systems do not fall in the class of well-structured transition systems [27], because they do not satisfy the required monotonicity conditions. Well-structured transition systems are arguably the most general class of infinite-state systems for which verification questions are decidable [28, 1].

As a result, full formal proofs of correctness of fault-tolerant algorithms are often based on interactive proof assistants [15, 19, 17]. However, they require a substantial effort. For instance, a proof of correctness of the Paxos algorithm is about 1500 lines long [17]. Recently, we have seen the use of interactive proof assistants to program real systems [51].

4 Project Outline

We started to look at fault-tolerant distributed systems with the goal of verifying them [23]. We quickly realized that the situation is very grim if we look at them through classical, asynchronous language and verification models. Using the round-based models from the distributed algorithms community we can achieve a much higher degree of automation in the verification. After all, these models were developed as simplifications for manually reasoning about asynchronous systems, e.g., for expressing the FLP [29] proof “on the back of a napkin”[2]. Structuring the computation in communication-closed rounds is an intuitive programming model as for each round, the programmer only has to give two methods, one

[2] From the talk “Better late (40 years late!) than never: Monday morning quarterbacking the coordinated-attack problem” by Eli Gafni at Yale University on 2014-10-09.

to send messages, and the other to update the local state upon reception of messages. To cover asynchronous and faulty executions, one only needs to accept that not every message gets delivered. From the verification perspective, this model (1) removes the problem of interleavings, because computations look synchronous: all processes proceed in lock-step, and (2) allows removing all channels from the state of the system by looking at the system's invariant at the boundary between rounds.

We believe that like the verification community, the programming language community has neglected to address the question of faulty systems and has been unproductively conservative in modeling them. The opportunity for the language community is to develop better abstractions for programming and reasoning about faulty distributed systems. As in the case of verification, a good place to get inspiration from are the models of the distributed algorithms community and our goal is to adapt their ideas to the needs of programming languages. However, this is not trivial: models such as [16] have been developed to be theoretically well-behaved and to simplify reasoning about faulty systems, but little has been done to compile and execute them. Also, if we adopt such a seemingly synchronous abstraction to reason about asynchronous systems, we must tell the programmer which properties of the synchronous abstraction transfer to the actual asynchronous system. Indeed, not every property is preserved. Yet we postulate that a round-based programming abstraction is still superior to the current state of affairs, where implementations need to handle faults in the asynchronous model. In particular, control-state reachability, a classical safety property for concurrent systems, is preserved by asynchrony due to its locality. Also most properties that can be checked using well-structured transition systems are preserved. In fact, it seems that most "reasonable" properties transfer from a synchronous abstraction to asynchronous executions [19, 14], and that this can be formalized.

The heard-of model [16]. The base primitive of our language is communication-closed rounds [25] and we use the heard-of (HO) formalization. Conceptually, processes operate in lock-step, and a distributed algorithms consist of rules that determine the new state of a process depending on the state at the beginning of the round and the messages received by the process in the current round. During a round, between the send and reception of the message, an adversarial environment can choose to drop messages. The impact of the environment is captured by the heard-of sets: for a process p its heard-of set, denoted $HO(p)$, contains the processes from which p may receive messages from in a given round. To model different kinds of network assumptions, the power of the environment is restricted by constraints on the HO set, given in linear temporal logic [49]. For instance, in a synchronous system without faults the environment respects $\Box(\forall p, q.\, p \in HO(q))$, which means q hears from every p, i.e., that every message is delivered. On the other hand, an asynchronous system with arbitrary faults does not have impose any restriction on the HO set. The environment can drop an arbitrary number of messages. The HO model can express intermediate failure models between these two extremes [16, Table 1]. For instance, a partially synchronous system with eventual reliable links and at most f crashes is $\Diamond(\exists S.\, |S| > n - f \wedge \forall p.\, |HO(p)| = S)$.

The language. We are currently working on building a complete infrastructure for implementing fault-tolerant distributed systems. The front-end consists of a domain-specific language and the corresponding specification logic. The DSL is based on the "heard-of model" [16], in which algorithms are structured in communication-closed rounds and have a synchronous semantics. The specification logic is an extension of [23] which is suitable for expressing many agreement properties such as consensus. At the back-end, we have an automated verifier and compiler+runtime. The verifier checks the algorithms against the specification in the synchronous semantics and the compiler+runtime generates code for asynchronous systems such that a process running the algorithms cannot distinguish between the synchronous semantics and an asynchronous execution. In our system, the programmer writes the core of the application, e.g., the fault-tolerant distributed algorithm, in the high-level DSL. As the round abstraction is suitable for designing distributed algorithms, but not necessarily for the rest of the system, the algorithm interfaces with other software components like a service, e.g., a consensus service. For the verification, we require the programmer to provide an inductive invariant that captures the correctness idea behind the algorithm. The verifier generates Hoare-style verification conditions and tries to discharge them using an SMT solver. Finally, the compiler+runtime fully automatically generate asynchronous distributed code that is guaranteed to preserve the local fault-handling properties that were established for the algorithm.

Limitations. Round models also have their downsides. First, they impose a regular control flow. This complicates the encoding different computation paths, e.g., paths the try to speculate a potentially reliable system to reach a decision faster, or recovery paths to be taken after a crash. Protocols, with complex recovery paths should be encoded as two different algorithms: one for the normal scenario, and one for the recovery. We have started to investigating distributed systems build using only the synchronous parallel composition of the participants. Richer forms of composition could support modular composition of different algorithms, e.g., making a new algorithm by running many copies of another algorithm [22]. Finally, the runtime is dependent on the considered fault-model. A round model is a weak form of clock synchronization which requires some environment assumptions. One can guarantee the progress of an implementation of a round-based model with benign faults, by assuming partial synchrony. However, implementing a round model in the byzantine setting is roughly possible with at most $n/3$ byzantine processes, where n is the number of replicas. Therefore, using this model does not make sense to implement an algorithm whose assumptions are weaker than those imposed by the implementation of the round model.

Prototype implementation. We are currently working on a prototype implementation of DSL in the SCALA programming language. Figure 3 shows the algorithm from Figure 1 in our DSL. The language is designed as a shallow embedding. The parts that require code generation, e.g. resource and state encapsulation, are handled using macros [10] which manipulate the SCALA syntax tree, and the serialization is made transparent using the pickling library [43].

To write an algorithm in this language, a programmer needs to write the program in a sequence of `Round`. Each round has a `send` and a `update` function. Intuitively, all the processes in the system, when executing a given round execute `send` followed by `update`. A runtime is responsible for delivering messages and detecting faults. Therefore, not every message might be received. Furthermore, the runtime guarantees that messages are only visible within the round in which they are sent, even if the system is asynchronous and processes progress at different speed. Late messages are discarded and late processes will try to catch up after receiving message for a “future” rounds.

To verify the algorithm, we assume that the user gives us and the specification and inductive invariants that describe the global state of the system. For our running example, the agreement property is

$$\forall i, j.\ \mathit{decided}(i) \land \mathit{decided}(j) \Rightarrow \mathit{decision}(i) = \mathit{decision}(j)$$

```
class OTR extends Algorithm[ConsensusIO] {

  val x = new LocalVariable[Int](0)
  val decision = new LocalVariable[Int](-1)    //used by the verification
  val decided = new LocalVariable[Boolean](false) //used by the verification
  val callback = new LocalVariable[ConsensusIO](null)

  def process = new Process[ConsensusIO]{

    def init(io: ConsensusIO) {
      callback <~ io
      x <~ io.initialValue
      decided <~ false
    }

    val rounds = Array[Round](
      new Round{
        type A = Int //guarantees that the types of send and receive match

        //minimal most often received value
        def mmor(mailbox: Set[(Int, ProcessID)]): Int = ...

        def send(): Set[(Int, ProcessID)] = broadcast(x)

        def update(mailbox: Set[(Int, ProcessID)]) {
          if (mailbox.size > 2*n/3) {
            x <~ mmor(mailbox)
            if (mailbox.filter(msg => msg._1 == x).size > 2*n/3) {
              callback.decide(x)
              decided <~ true
              decision <~ x
              terminate
          } }
        }
} }  }
```

Figure 3 The on-third-rule algorithm implemented in our DSL

and this property is enforced by the following invariant:

$$\forall i.\ \neg \mathit{decided}(i) \quad \vee \quad \exists v.\ |\{i.\, x(i) = v\}| > 2n/3 \wedge \forall i.\ \mathit{decided}(i) \Rightarrow \mathit{decision}(i) = v.$$

The invariant states that either no value has been decided, or there exists a majority of replicas formed of more than $2n/3$ replicas, that agree on the value of x. Notice that the invariant is still simple due to the fact that we use a state invariant that holds only at the boundary between rounds. Therefore, the invariant does not need to refer to messages or the state of communication channels. Communication-closed rounds are key in enabling us to apply automated verification on this class of algorithms. An additional benefit of this model is that it enables us to reason about liveness. Any computation terminates, if there are two rounds in which all processes receive the messages send by a majority P of cardinality greater than $2n/3$. More precisely, in each of these two rounds the values of the HO-sets assigned by the environment satisfy the property

$$\exists P.\ \forall p.\ \mathit{HO}(p) = P \wedge |P| > 2n/3,$$

where P is a set of processed and p denotes any process in the system.

The verification procedure [23] is roughly based on computing the Venn regions for the sets occurring in the formula, e.g., $\{i.\, x(i) = v\}, \mathit{HO}(p), \mathit{mailbox}(p)$, reasoning about the cardinality constraints, e.g., $|mailbox(p)| > 2n/3$, and propagating the global conditions appearing in the definition of set comprehensions to constraints on the values of replica's local variables, e.g., $x(p) = v$ is inferred from the predicate defining the set comprehension in the invariant.

To summaries, the project aims to find a programming model that strikes a balance between program design and program analysis. More precisely, we are interested in a programming model based on high-level concepts that offer programmers freedom in reasoning about faults and asynchrony. Its formal semantics should enable the development of automated verification tools that establish the correctness of the implementations. We intend to start with one of the most studied classes of algorithms in the literature: consensus algorithms [16, 40] and their weaker forms, e.g., k-set agreement [18] or lattice agreement [26].

Acknowledgements. We thank Josef Widder for the fruitful discussions on this subject, Heather Miller and Philipp Haller for their help with the pickling library, and the anonymous reviewers for their comments.

5 Conclusion

Building correct software in the absence of faults is already a challenging task. Now, doing so when faults are integral part of the system is an even greater challenge. Addressing this will requires adopting new programming models that natively include faults and/or give ways to detect and handle them. Ideally, we should aim for failure friendly programming abstractions where faults are naturally integrate without obscuring the program's logic. More realistically, we can start by identifying the features that makes those system unnecessarily complex and find better alternatives. We briefly sketched how models developed in the distributed algorithms community also solves issues that, in the PL community, make those system hard to implement and severely limit the scope of automated verification for distributed systems.

References

1 Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General Decidability Theorems for Infinite-State Systems. In *LICS*, pages 313–321. IEEE, 1996.

2 Parosh Aziz Abdulla, Aurore Collomb-Annichini, Ahmed Bouajjani, and Bengt Jonsson. Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *FMSD*, 25(1):39–65, 2004.

3 Yehuda Afek and Eli Gafni. Asynchrony from synchrony. In Davide Frey, Michel Raynal, Saswati Sarkar, Rudrapatna K. Shyamasundar, and Prasun Sinha, editors, *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, pages 225–239, 2013.

4 Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

5 Joe L. Armstrong. The development of erlang. In Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman, editors, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, pages 196–203. ACM, 1997.

6 Martin Biely, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, André Schiper, and Josef Widder. Tolerating corrupted communication. In *PODC*, pages 244–253, 2007.

7 Fatemeh Borran, Martin Hutle, and André Schiper. Timing analysis of leader-based and decentralized Byzantine consensus algorithms. *J. Braz. Comp. Soc.*, 18(1):29–42, 2012.

8 Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

9 Francisco Brasileiro, Fabíola Greve, Achour Mostefaoui, and Michel Raynal. Consensus in one communication step. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 2127 of *Lecture Notes in Computer Science*. Springer, 2001.

10 Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

11 Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, Berkeley, CA, USA, 2006. USENIX Association.

12 Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

13 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

14 Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. A reduction theorem for the verification of round-based distributed algorithms. In *RP*, volume 5797 of *LNCS*, pages 93–106, 2009.

15 Bernadette Charron-Bost and Stephan Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.

16 Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

17 Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. A TLA+ proof system. In *LPAR Workshops*, 2008.

18 S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132 – 158, 1993.

19 Henri Debrat and Stephan Merz. Verifying fault-tolerant distributed algorithms in the heard-of model. *Archive of Formal Proofs*, 2012, 2012.

20 Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 321–332. ACM, 2013.

21 Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34:77–97, 1987. http://doi.acm.org/10.1145/7531.7533.

22 Danny Dolev and Ezra N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In Andrzej Pelc, editor, *Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings*, volume 4731 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2007.

23 Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In Kenneth L. McMillan and Xavier Rival, editors, *VMCAI*, pages 161–181. Springer, 2014.

24 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, April 1988.

25 Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.

26 Jose M. Falerio, Sriram K. Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 125–134, 2012.

27 Alain Finkel. A generalization of the procedure of karp and miller to well structured transition systems. In Thomas Ottmann, editor, *Automata, Languages and Programming, 14th International Colloquium, ICALP87, Karlsruhe, Germany, July 13-17, 1987, Proceedings*, volume 267 of *Lecture Notes in Computer Science*, pages 499–508. Springer, 1987.

28 Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

29 Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

30 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 143–152, 1998.

31 Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

32 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

33 Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIXATC*. USENIX Association, 2010.

34 Michael Isard. Autopilot: Automatic data center management. *Operating Systems Review*, 41(2):60–67, April 2007.

35 Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*, pages 245–256. IEEE, 2011.

36 Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *PODC*, pages 169–178, 2006.

37 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

38 Leslie Lamport. Distributed algorithms in TLA (abstract). In *PODC*, 2000.

39 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

40 Nancy Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

41 Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In Fred B. Schneider, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 137–151. ACM, 1987.

42 Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

43 Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. Instant pickles: generating object-oriented pickler combinators for fast and extensible serialization. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *OOPSLA*, pages 183–202, 2013.

44 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

45 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

46 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

47 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

48 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319, 2014.

49 Amir Pnueli. The temporal logic of programs. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:46–57, 1977.

50 Sergio Rajsbaum. Acm sigact news distributed computing column 5. *SIGACT News*, 32(4):34–58, December 2001.

51 Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. Shadowdb: A replicated database on a synthesized consensus core. In Michael J. Freedman and Neeraj Suri, editors, *Proceedings of the Eighth Workshop on Hot Topics in System Dependability, HotDep 2012, Hollywood, CA, USA, October 7, 2012.* USENIX Association, 2012.

52 Philippe Schnoebelen. Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets. In *MFCS*, pages 616–628, 2010.

53 Seved Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.

54 Tatsuhiro Tsuchiya and André Schiper. Model checking of consensus algorithms. In *26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, October 10-12, 2007*, pages 137–148, 2007.

55 Tatsuhiro Tsuchiya and André Schiper. Using bounded model checking to verify consensus algorithms. In *DISC*, pages 466–480, 2008.

56 Tatsuhiro Tsuchiya and André Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6):341–358, 2011.

57 Josef Widder, Martin Biely, Günther Gridling, Bettina Weiss, and Jean-Paul Blanquart. Consensus in the presence of mortal Byzantine faulty processes. *Distributed Computing*, 2012.

58 Lantian Zheng and Andrew C. Myers. A language-based approach to secure quorum replication. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014*, page 27, 2014.

Toward a Dependability Case Language and Workflow for a Radiation Therapy System

Michael D. Ernst[1], Dan Grossman[1], Jon Jacky[2], Calvin Loncaric[1], Stuart Pernsteiner[1], Zachary Tatlock[1], Emina Torlak[1], and Xi Wang[1]

1 Computer Science and Engineering, University of Washington
Seattle, Washington, USA
{mernst,djg,loncaric,spernste,ztatlock,emina,xi}@cs.washington.edu

2 Radiation Oncology, University of Washington
Seattle, Washington, USA
jon@washington.edu

Abstract

We present a near-future research agenda for bringing a suite of modern programming-languages verification tools – specifically interactive theorem proving, solver-aided languages, and formally defined domain-specific languages – to the development of a specific safety-critical system, a radiotherapy medical device. We sketch how we believe recent programming-languages research advances can merge with existing best practices for safety-critical systems to increase system assurance and developer productivity. We motivate hypotheses central to our agenda: That we should start with a single specific system and that we need to integrate a variety of complementary verification and synthesis tools into system development.

1998 ACM Subject Classification D.2.4 Software/Program Verification, J.3 Life and Medical Sciences

Keywords and phrases Synthesis, Proof Assistants, Verification, Dependability Cases, Domain Specific Languages, Radiation Therapy

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.103

1 Introduction

Safety-critical systems – containing software in which errors can lead to death, injury, and wide-scale destruction – are nothing new. Experts have built them for decades at great expense and, despite well-known failures, our trust in software continues to grow as we increasingly depend on medical devices, transportation networks, financial exchanges, and other critical infrastructure where software plays a critical role.

In recent years, only a tiny fraction of mainstream programming-languages research has *directly* targeted safety-critical systems.[1] This is surprising given the astonishing advances in automatic verification and synthesis that many of us surely *believe* could, with appropriate adaptations and focus, reduce costs and improve reliability for safety-critical systems. After all, we can now build substantial formally verified software infrastructure like the CompCert compiler [30], a reference monitor for a modern web-browser [24], a full operating

[1] Note that there are a handful significant exceptions, such as the ASTREE analyzer in avionics [3], Galois' work on highly dependable Haskell platforms [12], and Praxis' work on safety-critical systems built in Spark/Ada [1].

system kernel [27], and cryptographic protocols [2]. However, note that the majority of these systems are traditional, core computing infrastructure. It is not surprising that the community has focused on software systems closer to our deepest experience, much as there are disproportionately many plays about the theatre and novels about novelists.

But, as we discuss in this paper, there are important safety-critical systems that are architected, specified, developed, and maintained in fundamentally different ways. Successfully bringing cutting-edge programming-languages verification techniques into modern safety-critical systems will require a long-term research agenda and deep collaborations with domain experts. We are beginning such an agenda, and this paper lays out our initial plans and hypotheses. It is a call for others to pursue similar or complementary approaches. It is a preview against which we can judge progress in a few years.

We have several high-level hypotheses, each expanded upon in the rest of this paper:

- *Our methodology should proceed from the specific to the general*, working first on one particular safety-critical system, then a second and a third, and [only] then trying to abstract to general principles. We are focusing on helping develop the third-generation of the Clinical Neutron Therapy System (Section 2) at the University of Washington Medical Center, a sophisticated medical device used for about 30 years without incident on our campus.
- *The right model for building safety-critical systems is Jackson's approach of dependability cases* (see Section 3), in which heterogeneous evidence of reliability is brought together in an explicit, layered way to connect system requirements to low-level implementation details. Our goal is not to replace the human element in this process, but to enrich it with formal and automatic verification of key pieces. To this end, we plan to develop a high-level *dependability case language* (DCL) for specifying, checking, and evolving dependability cases.
- *No one verification technique is right for the entire dependability case.* In particular, we hope to use an integrated workflow of complementary technologies (Section 3) that are rarely used together on the same project today, namely:
    - A formally verified domain-specific language (DSL) for writing the safety-critical software. Indeed, the strict requirements of such systems make DSLs (or highly restricted subsets of more general languages) the standard approach already.
    - Coq [5] for proving key semantic properties of the DSL implementation, and key correctness properties of shared libraries and components. Infrastructure bugs are *contagious* in the sense that a bug in the infrastructure may cause faults in any application code running on top of it. Furthermore, infrastructure code changes rarely. It therefore makes sense to apply heavyweight verification to language implementations, as evidenced [39, 29] by the reliability of verified language platforms such as CompCert [30] and Bedrock [4].
    - Solver-aided verification and synthesis for accelerating code reviews and revisions of DSL applications. Even safety-critical applications change frequently enough to make heavyweight verification (e.g., with Coq) prohibitive. At this level, the role of tools is to accelerate the standard development process. While static analysis is routinely used in this way [14], solver-aided tools are not, despite their successful application within many DSLs (see, e.g., [37, 38]). These tools can reason about complex program properties that are often needed to establish end-to-end dependability – and that are poorly supported by static analyzers.
    - Alloy [16] for describing and checking the system architecture and design. A design-level tool must enable fast iteration and prototyping. As such, the tool must be interactive

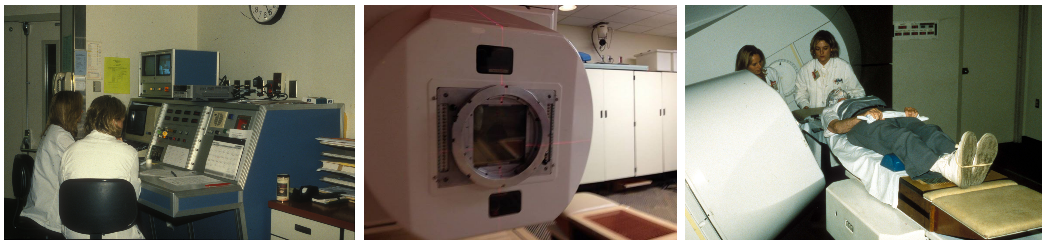


Figure 1 The CNTS console, collimator, and patient setup in gantry.

and capable of producing counterexamples. It must also support a rich logic for partial formalization of critical aspects of the system – not all parts of a design need to be subjected to formal analysis, and those that do not should be easy to abstract. Alloy makes it easy both to model and check designs, and it has a long history of discovering flaws in designs of safety- and correctness-critical systems (e.g., [10, 34, 25, 40]).

2 The Clinical Neutron Therapy System (CNTS)

The Clinical Neutron Therapy System (CNTS) at the University of Washington Medical Center (UWMC) is an advanced radiotherapy installation for treating tumors with neutron radiation. Neutron therapy is highly effective at treating inoperable tumors that are resistant to conventional electron- or photon-based radiation therapy. But the equipment for neutron therapy is also an order of magnitude more expensive than conventional radiation therapy. For this reason, CNTS is one of only three neutron therapy installations in the United States, and thus it depends extensively on custom software developed by the CNTS staff. Through deep expertise and great care, the CNTS staff has achieved a remarkable safety record, with no major incidents in over 30 years of service treating patients.

2.1 The CNTS Installation

The system consists of a cyclotron and a treatment room (Figure 1) with a computer-operated leaf collimator. The cyclotron generates a broad beam of neutrons that passes through the collimator on its way to the patient. The collimator consists of forty steel leaves and several filters, which control the shape and intensity of the beam, respectively. The collimator is mounted on a gantry that rotates 360 degrees so that the beam can enter the patient from any angle. The entry point is additionally controlled through the position of the couch (with five degrees of motion freedom) on which the patient lies during treatment.

A treatment beam prescription specifies the positions of all moving components (couch, leaves, and filters), the daily radiation dose, and the total radiation dose. Radiation therapy technologists manually ensure that all components are positioned correctly before beginning treatment. The therapy control system ensures that *the neutron beam can turn on and remain on only when all moving components are set as prescribed, and when the daily and total dose are less than prescribed.* This is the primary safety requirement for CNTS.

The therapy control system is carefully designed to enforce the CNTS safety requirement [19, 23, 20]. For example, a critical aspect of enforcing the requirement is to ensure that the system can always reach a safe state in which the beam is turned off. The design of the system therefore provides a non-software path from any state to a safe state. In particular, all basic therapy controls, such as turning the beam on and off, are implemented with hardware relays, programmable logic controllers (PLCs), or embedded microcomputers with programs

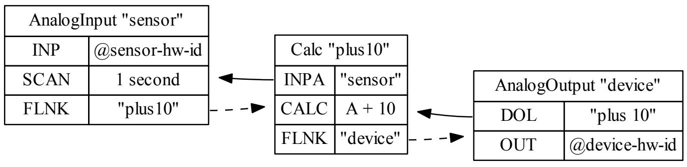


Figure 2 An example program in EPICS$_0$. Dashed lines indicate control transfers while solid lines indicate data dependencies.

in read-only memory. The software controller, which runs on general-purpose computers, is used only for advanced therapy control functions.[2] As a result, CNTS can reach a safe state (by turning off the beam through a basic control) even if its software controller crashes.

While bugs in the software controller cannot prevent the beam from being turned off, they could still cause a violation of the CNTS safety requirement, with deadly consequences. For example, the software controller is responsible for loading prescriptions from the patient database into the low-level device controllers. If this is done incorrectly, a patient could receive too much radiation. The software controller is therefore considered to be a safety-critical part of the overall system, and its development has followed rigorous practices and standards [19, 23, 20].

2.2 The CNTS Software Controller

The CNTS software controller has undergone two complete rewrites since 1984. The vendor-provided controller was originally written in FORTRAN; in 1999 it was replaced with a custom C program [19, 23]. The latter is now being phased out in favor of a new controller [20] written in a subset of a general-purpose dataflow language from the Experimental Physics and Industrial Control System (EPICS) [11]. This subset, which we call EPICS$_0$, forms a tiny embedded DSL that consists of just 19 EPICS constructs.

Figure 2 shows a simple program in EPICS$_0$ that reads sensor input once per second, adds 10 to the most recently read value, and writes the resulting value to some output device. EPICS$_0$ is like EPICS in that the programmer must explicitly specify both control transfers and data dependencies. For example, programmers need to specify both that after the "sensor" node finishes processing, it adds the "plus10" node to the processing queue (indicated by the dashed arrow starting from the "`FLNK`" field) and also that the "plus10" node will read the most recent value from the "sensor" node into its `INPA` field (indicated by the solid arrow starting from the "`INPA`" field). However, unlike EPICS, EPICS$_0$ programs guarantee that these links are always well-formed (e.g., links never point to non-existent nodes and are loop free).

As EPICS is already a popular and time-tested framework for controlling scientific instruments, the advantage of transitioning to EPICS$_0$ is that it provides numerous features that will enable a broader range of therapies at CNTS and thus increase the utility of the system. However, EPICS was not originally developed for clinical use, but rather for particle physics experiments. Adapting such research software to a safety critical presents several

[2] These include retrieving prescriptions from the patient database, loading prescribed settings into low-level device controllers, comparing the prescribed settings to those read back from the controllers, instructing the controllers in a particular sequence, and storing a record of each treatment in the database [20].

challenges. In particular, EPICS has a massive, complex code base, which is too large for the CNTS staff to fully audit. During adoption, EPICS has exhibited behavior that the CNTS staff were unable to explain, leading them to carefully avoid using certain features. In addition, the language has no formal semantics, making it difficult to reason about program's behavior. Finally, it is unclear how to port existing system requirements to this infrastructure in a way that is maintainable for future CNTS staff.

3 A Dependability Case for CNTS: the Language and the Workflow

The CNTS has operated without incident for 30 years, due to the rigorous development [23, 21] and operational [22] standards practiced by the in-house engineering and hospital staff. This level of safety has not been easy to obtain, however, relying on careful manual reasoning, documentation, and extensive testing. Our goal is to aid the CNTS engineers in maintaining the system's impressive safety record – with less effort and with higher confidence – as it transitions to the new EPICS$_0$ software controller. To that end, we propose to construct an explicit dependability case [15] for CNTS, with the help of an integrated workflow consisting of a language for expressing dependability claims and tools for supporting them with evidence.

3.1 The Dependability Case Approach to Building Reliable Systems

Safety and mission-critical systems, like CNTS or the Mars rover [14], are architected, specified, developed, and maintained according to strict best practices, by highly skilled engineers. At the system level, these practices involve detailed requirements, documentation, hazard analysis, and formalization of key parts of the system design. At the code level, they include adherence to stringent coding conventions (see, e.g., [13]), manual code reviews, use of static analysis, and extensive testing. The CNTS engineers, for example, followed [19] these practices when developing the second generation of the CNTS therapy control software – just as the NASA engineers followed them when developing the software for the Mars rover [14]. But is adherence to best practices enough to ensure that the resulting systems are indeed dependable – i.e., that they always satisfy their safety goals and requirements?

Based on a comprehensive two-year study [6] of how dependable software might be built, Jackson [15] argues that best practices alone cannot guarantee dependability. After all, a correctly implemented system may fail catastrophically if its requirements are based on an invalid assumption about its environment.[3] For this reason, Jackson proposes an approach for constructing dependable systems in which engineers produce both a system and an evidence-based argument, or a *case*, that the system satisfies its dependability goals.

In Jackson's approach, a dependability case is a collection of explicitly articulated *claims* that the system (i.e., the software, the hardware, and, if applicable, the human operators) has desired critical *properties*. Each is supported by *evidence* that may take a variety of forms, such as formal proofs, tests suites, and operating procedures. The case as a whole must be *auditable* (by third parties), *complete* (including relevant assumptions about the environment or user behavior), and *sound* (free of false claims and unwarranted assumptions). In essence, it must present a socially consumable proof [9] of the system's dependability.

We believe that the dependability case approach is the right model for developing safety-critical systems. In fact, it is the model that CNTS engineers intuitively followed when constructing the second generation of the therapy control system. As noted in Section 2, the

[3] Such an assumption was responsible for loss of life in a 1993 landing accident at the Warsaw airport [15].

design of the system was explicitly based [23] on the end-to-end safety requirement that the beam may be active only when the machine's settings match the patient's prescription. The ensuing development effort then yielded a collection of artifacts that comprise a rudimentary dependability case: a 200-page document [19] detailing the system requirements, developed in consultation with physicists and clinicians; a 2,100 line Z specification [19] of the therapy control software; a 16,000 LOC implementation of the Z specification in C; a 240-page reference manual [21]; and a 43-page therapist guide [22]. Our plan is to create an explicit case for the latest generation of the therapy control system that is powered by the EPICS$_0$ software controller [20].

3.2 A Dependability Case Language

What form should a dependability case for CNTS– or any system – take? Jackson's proposal does not mandate any specifics, noting only that the level of detail and formality for a case will vary between systems. We argue that a dependability case should be *formal* – that is, specified in a formal *dependability case language* (DCL) and subject to formal reasoning. Such a language would bring the same benefits to dependability cases that specification languages bring to software design – precision of expression and thought, automation to guard against syntactic and (some) semantic errors, and support for maintenance as the system evolves and the CNTS staff changes.

What then should a DCL look like? Existing DCLs are either logic-based languages (e.g., [17, 8]) for formalizing claims or structured notations (e.g., [26]) for relating claims to evidence. The former support mechanical analysis but have no notion of evidence. The latter include the notions of clams and evidence but, as semi-formal notations, they are not amenable to mechanical analysis. We plan to develop a new hybrid DCL that provides both mechanical reasoning and a (semi-)mechanical means of connecting claims with various forms of evidence – such as tests, Coq proofs, solver-aided reasoning, and manual reasoning by domain experts (for assumptions about the environment that cannot be otherwise discharged).

The details of our DCL are still being developed, but some necessary requirements have already become clear. First, it must be *expressive* enough to capture both system-level requirements and code-level specifications, since a dependability case spans all layers of design. Second, it must be *analyzable* – the consistency of the overall argument should be checkable in an automated fashion. Third, it must include a notion of evidence and be *flexible* enough to admit heterogeneous evidence of dependability. Finally, it must be *accessible* to domain experts who should be able to audit (but not necessarily construct) a case expressed in the language. If we succeed in striking a balance among these features, we expect the final design to be akin to SRI's Evidential Tool Bus [7] – but with a richer semantics specialized to our target domain and to our workflow of tools for generating evidence.

3.3 A Dependability Case Workflow

While a DCL helps express dependability claims and check their consistency with evidence and each other, it does not, by itself, help with the production of evidence. For that, we propose an integrated workflow of complementary technologies: Coq for infrastructure verification; solver-aided tools for accelerating the development and inspection of application code; and Alloy for describing and checking the high-level requirements and design.

As a first step, we will formalize the EPICS$_0$ DSL [20] in Coq. Such a formalization is critical, since EPICS has no formal semantics. We will then make EPICS$_0$ *solver-aided* [36, 37] – that is, equipped with automatic verification and synthesis tools based on SAT

```
[SETTING, VALUE, FIELD]

SETUP == SETTING → VALUE

BEAM ::= OFF | ON

| safe_: ℙ SETUP
| match_: SETUP ↔ SETUP
| prescription: FIELD ⇸ SETUP

┌ TherapyMachine ──────────────────
│ beam: BEAM
│ measured, prescribed: SETUP
└──────────────────────────────────

┌ SafeTreatment ───────────────────
│ TherapyMachine
├──────────────────
│ safe(measured)
│ match(measured, prescribed)
│ prescribed ∈ ran prescription
└──────────────────────────────────

∀ TherapyMachine • beam = ON ⇒ SafeTreatment
```

(a)

```
sig Setting {}
sig Value {}
sig Field {}

enum Beam {On, Off}

sig TherapyMachine {
  beam: Beam,
  measured, prescribed:  Setting -> Value
}

pred TherapyMachine.
SafeTreatment[prescription: Field -> Setting -> Value]
{
  safe[measured]
  match[measured, prescribed]
  prescribed in ran[prescription]
}

check {
  all m: TherapyMachine, p: Field->Setting->Value |
   m.beam = On => m.SafeTreatment[p]
}
```

(b)

Figure 3 An example specification in Z (a) and Alloy (b). The therapy beam can only turn on or remain on when the actual setup of the machine matches a stored prescription that the operator has selected and approved.

```
Theorem interp_ok: forall db db' es,
    interp db (inputs es) = (db', outputs es) -> star step db es db'.
```

Figure 4 An example correctness theorem in Coq.

and SMT solving. These tools are intended to accelerate the standard CNTS development workflow [20] by guiding code reviews (with lightweight verification) and revisions (with lightweight synthesis). Next, we will use Alloy to check the CNTS design for high-level dependability properties, leveraging the existing Z specification [19] for the system, which is yet to be subjected to fully automatic analysis. Finally, we will integrate all these sources of evidence – Coq proofs, automatic synthesis and verification guarantees, Alloy models, and tests – into a dependability argument in our DCL.

To illustrate the components of our workflow, consider the primary safety requirement of CNTS. Figure 3a shows a formal specification of this requirement in Z. The specification was written [19] and manually analyzed by the CNTS staff. In our workflow, the requirement would be expressed in Alloy, as shown in Figure 3b. Because Alloy was inspired by Z, the two formulations correspond closely to each other. An engineer who knows Z can switch to Alloy without much trouble and gain the benefits of its automated analysis.

In an end-to-end dependability case, the Alloy requirement would be decomposed further into assumptions about the environment and into specifications about the CNTS software controller. The former would be discharged manually by experts. The latter would be discharged (within finite bounds) by a solver-aided verifier for EPICS$_0$. Our verifier simply assumes that the EPICS$_0$ implementation is correct. Our dependability case, however, would treat this assumption as a correctness claim to be discharged with Coq.

Figure 4 shows an example theorem in Coq for the correctness of the EPICS$_0$ implementation. This theorem requires that the interpreter function `interp` behaves according to the EPICS semantics captured by the `step` relation. In particular, whenever `interp` executes an EPICS *database* `db` (which includes both application code and state) with the inputs of some event trace `es`, denoted by `(inputs es)`, and produces the resulting database `db'` with output events (`outputs es`), then the EPICS small step operational semantics allow exactly the same state transitions with the same observable input and output events. This theorem completes the dependability argument since the solver-aideded verifier (which is part of our trusted code base) uses the semantics specified by the `step` relation.

Our workflow is chosen to balance the need for high assurance with the cost of obtaining it. At the infrastructure level, the high cost of verifying rich properties manually in Coq is appropriate, since an infrastructure bug can invalidate any guarantees established for applications and because, once verified, the infrastructure changes slowly relative to application code. At the application level, however, code is continuously evolving, especially in a long-lived system such as CNTS. At this level, full automation is invaluable – the main purpose of tools is to accelerate development rather than provide total guarantees [14]. Automation is also critical at the design level, where bugs have the most serious effects [15]. We have chosen Alloy for this purpose because of its Z-like relational logic, fully automated analysis, and history of successful applications to safety-critical designs (e.g., [10, 34, 25]).

4 Related Work

There has been much prior work on building dependable systems, most of it by the software engineering community. Thanks to this work, we have effective approaches to gathering and analyzing whole-system requirements (e.g., [18]); to specifying and analyzing software designs (e.g., [35, 28, 16]); and, to turning those designs into analysis-friendly low-defect code (e.g., [14, 13, 19, 20]). The recent work on dependability cases (e.g., [6, 15, 31]) also gives us a methodology for driving the system-building process in a goal-directed fashion, so that the output of the process is both the system itself and an end-to-end argument – a social proof – that the system satisfies its critical requirements.

We aim to build on this body of work, and to produce the first integrated workflow – from languages to tools – for the development, evolution, and maintenance of a high-value radiotherapy system. In particular, we plan to leverage and extend existing work on languages for describing dependability cases [17, 8, 26] and for integrating heterogeneous tools into a workflow [7]. Unlike these prior languages, our DCL will be specialized to a narrow target domain. As such, it will be more accessible to domain experts, and more tightly integrated with the tools in our workflow – Coq [5], Alloy [16], and solver-aided verification and synthesis [36, 37].

5 Conclusion

We have outlined a research agenda to bring modern ideas in programming-language verification to the next-generation development of a safety-critical medical tool. Our approach is unusual in aiming first at one specific system before succumbing to the obvious temptation of building general tools for a large class of similar systems. Indeed, the goal of building a safe system will need to take priority over the goal of finding novel programming-languages research questions, but we believe there will be no shortage of the latter. We are optimistic that we will learn lessons that can inform research in several areas, namely interactive theorem proving, solver-aided languages, finite-system modeling tools, and – tying them together – formal languages for expressing dependability cases for safety-critical systems.

References

1 John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, April 2003.

2 Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, pages 90–101, Savannah, GA, January 2009.

3 Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, San Diego, CA, June 2003.

4 Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In PLDI'11 [32], pages 234–245.

5 Coq development team. *Coq Reference Manual, Version 8.4pl5.* INRIA, October 2014. `http://coq.inria.fr/distrib/current/refman/`.

6 National Research Council, Daniel Jackson, and Martyn Thomas. *Software for Dependable Systems: Sufficient Evidence?* National Academy Press, Washington, DC, USA, 2007.

7 Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2013.

8 Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. In *Selected Papers of the Sixth International Workshop on Software Specification and Design*, 6IWSSD, pages 3–50, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V.

9 Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

10 Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 165–174, Boston, MA, July 2004.

11 EPICS. `http://www.aps.anl.gov/epics/`.

12 Galois. `http://galois.com`.

13 Gerard J. Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–97, June 2006.

14 Gerard J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, February 2014.

15 Daniel Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, April 2009.

16 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, February 2012.

17 Daniel Jackson and Eunsuk Kang. Property-part diagrams: A dependence notation for software systems. Technical report, Massachusetts Institute of Technology, 2009. `http://hdl.handle.net/1721.1/61343`.

18 Michael Jackson. *Problem Frames: Analysing and Structuring Software Development Problems.* Addison-Wesley, 2001.

19 Jonathan Jacky. Formal safety analysis of the control program for a radiation therapy machine. In Wolfgang Schlegel and Thomas Bortfeld, editors, *The Use of Computers in Radiation Therapy*, pages 68–70. Springer Berlin Heidelberg, 2000.

20 Jonathan Jacky. EPICS-based control system for a radiation therapy machine. In *International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS)*, 2013.

21 Jonathan Jacky and Ruedi Risler. Clinical neutron therapy system reference manual. Technical Report 99-10-01, University of Washington, Department of Radiation Oncology, 2002.

22 Jonathan Jacky and Ruedi Risler. Clinical neutron therapy system therapist's guide. Technical Report 99-07-01, University of Washington, Department of Radiation Oncology, 2002.

23 Jonathan Jacky, Ruedi Risler, David Reid, Robert Emery, Jonathan Unger, and Michael Patrick. A control system for a radiation therapy machine. Technical Report 2001-05-01, University of Washington, Department of Radiation Oncology, 2001.

24 Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st Usenix Security Symposium*, pages 113–128, Bellevue, WA, August 2012.

25 Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a flash filesystem in Alloy. In *Proceedings of the 1st International Conference on Abstract State Machines, B and Z*, ABZ'08, pages 294–308, Berlin, Heidelberg, 2008. Springer-Verlag.

26 Tim Kelly and Rob Weaver. The goal structuring notation – a safety argument notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, July 2004.

27 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.

28 Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

29 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In PLDI'14 [33], pages 216–226.

30 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

31 Joseph P. Near, Aleksandar Milicevic, Eunsuk Kang, and Daniel Jackson. A lightweight code analysis and its role in evaluation of a dependability case. In *Proceedings of the 33rd International Conference of Computer Safety, Reliability and Security*, pages 31–40, Waikiki, Honolulu, HI, May 2011.

32 *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

33 *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, June 2014.

34 Tahina Ramananandro. Mondex, an electronic purse: Specification and refinement checks with the alloy model-finding method. *Form. Asp. Comput.*, 20(1):21–39, December 2007.

35 J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

36 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, Indianapolis, IN, 2013.

37 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In PLDI'14 [33], pages 530–541.

38 Richard Uhler and Nirav Dave. Smten with satisfiability-based search. In *Proceedings of the 2014 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 157–176, Portland, OR, October 2014.

39 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In PLDI'11 [32], pages 283–294.

40 Pamela Zave. Using lightweight modeling to understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012.

The Racket Manifesto*

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt

PLT Design Inc.
`{matthias,robby,mflatt,sk,eli,jay,samth}@racket-lang.org`

Abstract

The creation of a programming language calls for guiding principles that point the developers to goals. This article spells out the three basic principles behind the 20-year development of Racket. First, programming is about stating and solving problems, and this activity normally takes place in a context with its own language of discourse; good programmers ought to formulate this language as a programming language. Hence, *Racket is a programming language for creating new programming languages*. Second, by following this language-oriented approach to programming, systems become multi-lingual collections of interconnected components. Each language and component must be able to protect its specific invariants. In support, *Racket offers protection mechanisms to implement a full language spectrum*, from C-level bit manipulation to soundly typed extensions. Third, because Racket considers programming as problem solving in the correct language, *Racket also turns extra-linguistic mechanisms into linguistic constructs*, especially mechanisms for managing resources and projects. The paper explains these principles and how Racket lives up to them, presents the evaluation framework behind the design process, and concludes with a sketch of Racket's imperfections and opportunities for future improvements.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases design guidelines, language generation, full-spectrum language

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.113

1 Racket, Historically Speaking

In 1995, we set out to create an outreach project for novice programmers. After observing students in labs over the course of a year, we understood that *nobody* could teach Scheme in an hour and then focus on the essence of computing and programming – contrary to an opinion widely held by instructors due to the numerous courses based on MIT's *Structure and Interpretation of Computer Programs* [1]. What was needed, instead, was a teaching language suitable for instructing beginners. Furthermore, emacs, vi, and similar editors were overwhelming students who had never programmed before, despite their special modes for editing and interacting with Scheme. We therefore also wanted a pedagogical programming environment for our beginners, not just a re-appropriated power tool for professionals.

After we decided to implement our own teaching language and its environment, we still wanted to demonstrate that Scheme was an excellent implementation vehicle for these projects. We thought Scheme's macro system would help us experiment with language

* Over 20 years, this work was partially supported by our host institutions: Rice University, University of Utah, Brown University, University of Chicago, Northeastern University, Northwestern University, Brigham Young University, and Indiana University as well as several funding agencies and foundations: AFOSR, CORD, Darpa, the Department of Education's FIPSE program, the ExxonMobile Foundation, Microsoft, the Mozilla Foundation, NSF, the Texas Advanced Technology Program.

designs. The language also appeared to be a perfect match for constructing a simple interactive development environment (IDE); after all, many Lisp courses taught how to create a read-eval-print loop, and a lot of emacs was written in Lisp.

Using Scheme as a starting point turned out to be an acceptable choice, but we soon found we needed a lot more. We discovered that more was needed than a hygienic and pattern-oriented variant of Lisp's old macros. In the same vein, the development of a large system exposed the difference between having safe abstractions and mimicking them via `lambda`. Finally, implementing an IDE also called for executing arbitrary programs under the control of our own program – and this goal clarified that Scheme did not come with the means for managing resources and for ensuring the security of the hosting program.

Over time, we adapted Scheme to serve our needs. We built a syntax extension system atop Scheme's macros, added mechanisms for creating safe abstractions, and turned features of the surrounding operating system into linguistic constructs so that we could program resource administrators and security containers. In time, our language became a full-fledged tool for the working software engineer. By 2010, our dialect of Scheme had evolved so much that we renamed it to Racket [19] to let the world know that we had something different.

2 The Principles of Racket

While we have reported on the pedagogic aspect of the project elsewhere [7], this paper presents the design principles behind Racket and illustrates with concrete examples how they affect the reality of its implementation. It groups these principles under three slogans:

1. *Racket is about creating new programming languages quickly.*

   Programming is a form of problem solving. A proper approach uses the language of the domain to state the problem and to articulate solution processes. In support of this mode of programming, Racket helps programmers create and quickly deploy new languages. In particular, the mechanisms for creating and deploying languages must be contained within the language itself. Once Racket is installed, there must be no need to step outside to use one of its new languages. This principle is in stark contrast to the numerous external tools and command-line pre-processors that are used to create (embedded) domain-specific languages.

2. *Racket provides building blocks for strong protection mechanisms.*

   If programming is about solving problems in the correct language, systems will necessarily consist of interconnected components in several different languages. Due to the connections, values flow from one linguistic context into another. Since languages are charged with providing and preserving invariants, the creators of languages must have the power to protect the languages' invariants. By implication, Racket must come with mechanisms that enable programmers to protect individual components from their clients.

   For this reason, Racket comes with the proper building blocks to set up or construct protection mechanisms at any level, all the way from C to languages with sound, higher-order type systems, and any mixture in between.

3. *Racket turns extra-linguistic mechanisms into linguistic constructs.*

   When programmers must resort to extra-linguistic mechanisms to solve a problem, the chosen language has failed them. Even if the fix for such failures is not always obvious, programming language researchers ought to accept the general idea and work to find the proper linguistic mechanisms. For instance, Racket currently internalizes several resource-management mechanisms that are often found in an operating system. Similarly, this philosophy prohibits the idea of "projects," as found in other IDEs, because this also externalizes resource management, linking, and other aspects of program creation.

Evaluating the use of such principles must take place in a feedback loop that encompasses more than the compiler for the language. In Racket's case, the feedback loop's evaluation stage contains a range of software systems, especially DrRacket [9], the Racket IDE.

Sections 3 through 5 explain the principles in depth: language-oriented programming, protection mechanisms for full-spectrum programming, and services-as-constructs. Section 6 introduces Racket's feedback loop in some detail and how it helps us use the guidelines to turn principles into reality. Finally section 7 puts the principles in perspective, pointing out in particular where they remain goals and the research needed to reach those goals.

Listing 1 A Racket module

```
#lang racket                                                    demo.rkt

(provide
  ;; type Video = [Listof Image]
  ;; Natural -> Video
  walk-simplex)

;; -----------------------------------------------------------------------
(require "small.sim" 2htdp/image)

;; Natural -> Video
(define (walk-simplex timing)
  ... (maximizer #:x 2) ...)
```

3 Racket is a Programming-Language Programming Language

Racket is a programming language. Actually, at first glance it looks like a family of conventional languages, including a small untyped, mostly-functional by-value language (`racket/base`), a batteries-included extension (`racket`), and a typed variant (`typed/racket`).

Like all programming languages, plain Racket forces the programmer to formulate solutions to problems in terms of its built-in programming constructs. But, Racket is also a member of the Lisp family, which has always insisted on stating solutions in the most appropriate language, one suited to the problem domain. As Hudak [21] puts it, "domain-specific languages are the ultimate abstractions."

Following this reasoning, each program component is articulated in the Racket-based programming language that is best suited for the problem it solves. If the language is not available, the Racket programmer creates it, possibly even for a single module. To support this kind of system building, Racket is a programming-language programming language.

Listings 1 and 2 illustrate the principle. *(The box in the top right of a listing specifies the filename. It is not a part of the code. The astute reader will notice that this violates the principle of keeping everything in the language.)* The first module in listing 1 uses the `racket` language, which is specified in the so-called `#lang` – pronounced "hash lang" – line. The module provides a single function; the comments inside the `provide` specification informally state a type definition and a function signature in terms of this type definition. To implement this function, the module uses `(require "small.sim")` to import functionality from the module in listing 2 and then defines its own functions.

The creator of the module in listing 2 prefers a domain-specific language, because the module's purpose is to synthesize a function for a simplex, and the most natural way to specify the latter is to state a collection of linear inequalities. The comments below the `#lang` line in listing 2 state that the module exports a single function, `maximizer`. Concretely, the `#:variables` specification and the following inequalities determine the `maximizer` function. When called as `(maximizer #:x n)`, the function produces the maximal `y` value; conversely, `(maximizer #:y m)` delivers the maximal `x` value.

Listing 2 A module for describing a simplex shape

```
#lang simplex                                                    small.sim

;; provides: synthesized function maximizer:
;;     #:x Real -> Real
;;     #:y Real -> Real

#:variables x y

3 * x + 5 * y <= 10
3 * x - 5 * y <= 20
```

In support of this kind of language-oriented programming, Racket provides a syntax extension system that borrows elements from Scheme's macro system [4, 23, 24] but also improves on it in several different directions. First, the Racket syntax extension system is about defining languages [12, 25, 26], not just extending an existing language with new linguistic constructs. For example, Racket's class system [16], its first-class components [14], and its language of (loop) comprehensions are just such sub-languages, though their constructs are indistinguishable from Racket's core features. Naturally, a Racket-based language is just a module whose exports make up a new language. These exports must include certain features and may otherwise come with any syntactic constructs and run-time values deemed necessary. The module may define these exports or may import and re-export them from an existing language. Hence, a language module can easily add features to, or subtract them from, an existing language.

Second, the syntax extension system also allows a language module to redefine the meaning of existing constructs. Take function application, for example. Like Lisp, a Racket function application is just a pair of parentheses around the function and its arguments:

```
(f a ...)
```

Racket's syntax system elaborates surface syntax to kernel syntax:

```
(#%app f a ...)
```

The keyword `#%app` is Racket's internal sign post for the function application syntax – and a language can re-define its meaning. Here is a simplistic re-definition:

```
#lang racket
(provide (rename-out [call #%app]) ...)

(define-syntax-rule
  (call f a ...)
  ;; rewrites to
  (if (check-in-defines f) (#%app f a ...) (signal-error f a ...)))
```

This module defines the syntactic abbreviation `call`. A use of `call` expands to an `if` expression that checks a property of `f` and, if it holds, uses the *imported* application syntax (underlined) to create a function application; otherwise it signals an error. On export, `call` is renamed to `#%app`, meaning when another module specifies this module as its language, the compiler uses the `call` syntax to elaborate the module's function applications, e.g.,

```
(g b ...)
-- compiles to--> (#%app g b ...)
== equivalent  == (call g b ...)
-- compiles to--> (if (check-in-defines g) (#%app g b ...) (signal-error g b ...))
```

That is, the final code uses plain `racket`'s `#%app` construct to evaluate the function application – and that is regular call-by-value function application.

This example is inspired by the teaching languages [6]. In particular, the first-order functional teaching language uses it to check whether the function position is a name defined by the program or the language so that it can produce novice-friendly error messages when something else shows up. However, the pattern is used much more widely. For instance, the FrTime language uses this same mechanism to create a dataflow variant of call-by-value [2].

Third, Racket's syntax extension system grants a language-defining module access to the entire syntax tree for a guest module, not just individual nodes in the syntax tree. This access allows the collaboration between the rewriting rule for `#%app` and `define` in the above example. Indeed, this kind of communication smoothly generalizes to complex context-sensitive analysis tasks and, in particular, allows for the implementation of a rather conventional type checker [42].

Fourth, Racket comes with a library that supports the programmatic creation of lexers and parsers [34]. It is thus possible for a language implementation to transform conventional syntax into regular S-expression syntax and to subject this result to the conventional syntax extension system and its rewriting rules. See listing 2 where the implementor of a domain-specific language prefers an ASCII-mathematics notation. Importantly, the separation of parsing from the syntax extension naturally creates an interface between unrelated parts of language design – notation and meaning – and thus enables language engineers to factor the work into two independent components: design of surface notation and meaning.

Finally, Racket insists on separating the various stages of language processing, particularly enforcing a strict separation of compile-time from run-time code. For example, the rewriting rules generate pure syntax and may not embed other language values inside this syntax. Similarly, since the world of Racket languages is actually an inverted pyramid of languages atop languages, each language-processing module may have side-effects – and these side-effects must be insulated from the rest of the language-processing pipeline.

In sum, Racket's toolbox empowers programmers to create new languages quickly and thus enables language-oriented program design. The key to this achievement is to improve over Lisp and Scheme's approaches: Racket carefully stages syntax elaboration [12], eliminating Lisp's problematic `eval-when-where` approach; it enables the quick derivation of new languages from existing ones; and it enables the introduction of conventional syntax.

4 Racket Covers a Full Programming Language Spectrum

An abstraction enforces invariants. Languages are abstractions, and their creators must have the means to build the necessary enforcement mechanisms – especially when components in these languages end up in an interconnected, multi-lingual contexts. Since Racket is a language for building programming languages, it supplies the building blocks for the construction of enforcement mechanisms, too. Indeed, Racket's building blocks allow the creation of a spectrum of languages, and Racket programmers may safely compose components written in various elements of this spectrum.

To get a sense of what these enforcement requirements may mean, consider the kinds of languages a Racket programmer may build. As the literature on domain-specific languages

Listing 3 A Racket module using the foreign-function interface

```
#lang racket                                                         ffi.rkt

(provide
 ;; [Vectorof [Vectorof Real]] -> [Vectorof Real]
 simplex)

;; ---------------------------------------------------------------------
(require ffi/unsafe)

(define lib-simplex (ffi-lib "./coin-Clp/lib/libClp"))

(define (simplex M)
  ... (-simplex-set ...) ...)

(define -simplex-set
  (get-ffi-obj "simplex" lib-simplex (_fun _bytes -> _void)))
```

suggests [20], these constructions are often thin veneers over efficient C-level implementations. To support this kind of language, Racket comes with a foreign interface that allows parenthesized C-level programming. Programmers can refer to a C library, import functions and data structures, and wrap these imports in Racket values. Listing 3 shows an example of a module that imports functions from the `coin-Clp` simplex library and defines regular `racket` functions around them.

At the other end of the spectrum, a Racket programmer might wish to annotate an existing module with explicit types and expect type soundness. Doing just that is possible with `typed/racket`. Listing 4 illustrates how to transform the module from listing 1 into a typed one. Adding types moves knowledge out of comments into a statically checked sub-language, which proves the comments' validity and thus "hardens" [43] the component, because the invariants of the typed language are properly protected as its values flow into untyped components of the world.

Listing 4 A Typed Racket module

```
#lang typed/racket                                            demo-typed.rkt

(provide walk-simplex)

(: walk-simplex (-> Natural Video))

;; ---------------------------------------------------------------------
(require/typed 2htdp/image [#:opaque Image image?])
...
(define-type Video [Listof Image])

(define (walk-simplex timing)
  ... (maximizer #:x 2) ...)
```

While Racket does not automatically protect such flows of values, it comes with the tools to build invariant-enforcement mechanisms. Technically, it provides Miller's proxy mechanism [32] tailored to the needs of a Racket language builder. Racket's proxies come in two tiers: chaperones and impersonators [38]. Each monitors access to an underlying value to guarantee basic invariants. Programmers can create customized proxies that monitor access to functions, immutable values, mutable structures and objects – without ever getting in the way of other operations on the wrapped values and objects.

Listing 5 A Racket module with contracts

```
#lang racket                                                    demo-contract.rkt

(provide
  (contract-out
    [walk-simplex (-> natural-number/c (listof image?))]))

;; ------------------------------------------------------------------
(require "small.sim" 2htdp/image)

(define (walk-simplex timing)
  ... (maximizer #:x 2) ...)
```

Racket offers a comprehensive contract system implemented with the proxy mechanisms. The contracts allow components to express Eiffel-style first-order assertions [31]. The introduction of contract boundaries smoothly generalizes these first-order statements to Racket's higher-order setting. That is, with contracts a component can advertise promises and obligations on values such as closures [10], objects, classes [36], and modules [37].

Listing 6 A contract for a first-class class in Racket

```
#lang racket                                                   class-contract.rkt

(define MBTA/c
  (class/c
    [find-path
      ;; (find-path f t) finds paths from f to t
      (-> station/c station/c [listof path/c])]
    ...))
  (define (station? s) ...)
  (define path/c ...)

(provide
  ;; does the given value represent a T station?
  station?

  (contract-out
    [mbta%
     ;; represent the state of the MBTA with search functionality
     MBTA/c]

    [read-mbta-graph
     ;; an MBTA/c factory
     (-> (object/c mbta%))]))
```

Listing 5 shows how to express the comments from listing 1 into a contract. The conventional prefix syntax of the contract says that `walk-simplex` is a function, that this function accepts only natural numbers (0, 1, 2, and so on), and that it returns a list of images (checked with the `image?` predicate from the library). Racket checks this first-order contract in the expected way: if a client module applies the function to something other than a natural number, the client is blamed for a violation; if `walk-simplex` ever returns something other than a list of images, `contract.rkt` is blamed; and if there is no use, no error message is ever signaled even if `walk-simplex` were defined to return a string.

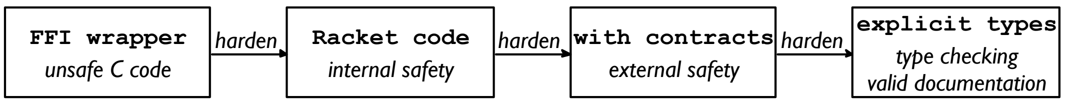
The extract of a module in listing 6 illustrates the use of higher-order contracts. Its header introduces two contracts and a flat predicate and exports the latter plus a contracted version of a class and its factory.

Currently, `typed/racket` [41] is the most important "client" of proxies and contracts. From a mechanical perspective, `typed/racket` is like `simplex`. From the teleological one, the two languages radically differ from each other; `typed/racket` is a sibling of `racket`, not just an arbitrary language implemented in the Racket world. As mentioned, its purpose is to help programmers harden untyped modules by equipping them with types.

While type checking guarantees consistency within the module and with respect to other typed modules, translating types of exported values into run-time contracts ensures a general form of type soundness, known as Tobin-Hochstadt's Blame Theorem.[1] For example, when a typed module exports a function on integers to an untyped module, the latter must not apply the function to a string; similarly, if a function on integer-valued functions flows from the typed to the untyped world, the latter must promise not to apply it to complex functions.

Listings 4 and 5 demonstrate this types-to-contracts translations in a concrete manner. The type of `walk-simplex` in the former translates to the contract shown in the latter. While this translation is straightforward for the functional core, extending this work to Racket's class-oriented fragment is the fruit of a multi-year research project [39, 40]. This extension implements both novel contract mechanisms for Racket's first-class classes as well as critical performance enhancements in the types-to-contracts translator.

Listing 7 The Racket language spectrum



With contracts and types, Racket includes a full spectrum of programming languages and, importantly, allows programmers to incrementally harden their "scripts" into programs. Initially, a Racket programmer may write a "script" in the traditional sense, that is, a thin layer around a C library. Assuming the choice of C is not performance-critical, the programmer could move the code to Racket, gaining operational and memory safety for the module in return. The third step would be the addition of contracts to the exports of this library to protect interactions between clients and the library. Finally, a programmer may equip the code with explicit, statically checked types, which creates validated documentation, improves

[1] As a library-based language, `typed/racket` is on the same footing as other libraries in the Racket ecosystem. Thus it cannot defend its invariants as thoroughly as typed languages such as Java or OCaml. Closing the remaining loopholes to enable more complete guarantees is ongoing research.

the performance[2] and maintainability of the code, and may reveal subtle mistakes. Listing 7 summarizes the hardening process in the Racket language spectrum diagrammatically.

While both contracts and types play a central role in this hardening process, the development of `typed/racket` is far more interesting from a linguistic perspective. Equipping `racket` with a type system is a challenging task. Programmers who use dynamically typed languages superimpose their own reasoning system as they design their code. It is fair to call this reasoning system a type system. Often this informal type system resembles naive set theory; at other times it incorporates elements from several different type systems.

The design challenge for `typed/racket` is to bring all of these informal type systems together in one framework – without introducing incompatibilities and contradictions. After all, when programmers harden a project, they do not want to modify their code to accommodate the type checker. Worse, any such modification might introduce a mistake or change the behavior of the program in undesirable ways. Because of the desire to allow incremental and selective hardening, it is also critical for `typed/racket` to preserve the semantics of `racket`.[3]

5 Racket Internalizes Extra-Linguistic Mechanisms

While many programming problems originate in a "real" world, program development is also a problem domain. As such, tools that support programming deserve a language of their own. Compiler writers take this idea seriously; for example, Dybvig and his group have developed a language for stating compilers as nano-scale transformations and used it for both educational [35] and commercial purposes [22]. When it comes to program development or program execution, however, IDEs resort to mechanisms from the surrounding operating system. They force programmers to develop programs in project contexts, delegate program execution to operating systems, and use *external* tools to inspect programs and their execution states. Racket's focus on languages as the key to problem solving points to the alternative solution of turning these extra-linguistic mechanisms into linguistic constructs [17].

To appreciate this domain, consider the original problem of building a pedagogic IDE for novice programmers. Clearly, the emphasis on pedagogy and novices prohibits the use of "projects;" students should be able to type in programs without any knowledge about computers and to run these programs without leaving the IDE they use. By implication, the IDE runs student programs under its control. Students make mistakes, though, and one common mistake is to launch a diverging program, that is, a program that consumes unbounded amounts of time, memory, or other resources (e.g., file ports, database handles, network connections; sometimes the access may be via instructor-provided libraries). Similarly, novices want to find mistakes in programs, meaning their instructors want to show them how to step through a program's execution. Finally, when a student submits a program to some homework server, this program must run in a security context that prohibits it from inspecting other students' solutions, attacking the server, and so on.

A close look at these requirements immediately suggests several areas of concern. Due to its design feedback loop, Racket includes the following external mechanisms as constructs at the moment: *inspectors*, which establish a hierarchy of access rights; *threads* that can be shut down from the outside; *sandboxes*, which restrict access to services; *custodians*,

[2] Performance enhancements can be realized under certain conditions; in general, types-as-contracts may reduce performance and often require performance tuning.

[3] The design of `typed/racket`'s type system is a complicated, but separate problem. A full discussion of this design would not illuminate the explanation of Racket's design principles, which is why we reserve this topic for a future paper.

which manage file handles, sockets, and database connections; *eventspaces*, which deal with GUI resources and events; and several more. The remainder of this section sketches two of these capabilities – inspectors and custodians – and how providing them inside the language provides fine-grained control over inspection and resources.

Listing 8 Inspection in Racket, part 1

```
#lang racket/base                                          inspector.rkt

(define the-inspector (current-inspector))
(define sub-inspector (make-inspector the-inspector))

(define v
  (parameterize ([current-inspector sub-inspector])
    (dynamic-require "inspected.rkt" instance-of-s)))
```

Listing 9 Inspection in Racket, part 2

```
#lang racket/base                                          inspected.rkt

(provide instance-of-s)
(struct s (fld))
(define instance-of-s (s 1))
```

Listings 8 and 9 demonstrate how Racket turns program inspection into a linguistic construct. Ordinarily, a Racket structure declaration like the one for `s` in listing 9 defines several functions: a constructor `s`, a field accessor `s-fld`, and a predicate `s?`. Unless a module exports the field accessor, instances of `s` are *opaque* to other modules in the system, i.e., other modules cannot view, access, or mutate the content of field `fld` in an instance. For example, dynamically loading module `inspected` from listing 9, retrieving `instance-of-s`, and printing it would reveal no information:

```
> (dynamic-require "inspected.rkt" 'instance-of-s)
#<s>
```

When the Racket IDE dynamically loads and evaluates a student program, however, it needs to have access to structure information for printing, stepping, and debugging.

To address these needs, Racket evaluates modules under a hierarchy of *inspectors*. If two modules run under the same inspector or incomparable inspectors in the hierarchy, they cannot view, access, or mutate each others structures unless they explicitly grant these rights via `provides` of the respective functions. In contrast, if module `A` runs under the control of inspector i and another module `B` runs under the control of an inspector j that is below i, `A` can inspect `B`'s structures – whether `B` grants these rights or not.

Consider the module in listing 8, which concretely illustrates how inspectors work. The module creates a reference to the current inspector, that is, the inspector under whose supervision it executes. It then makes another inspector; the new one is below `make-inspector`'s argument, which is the module's current inspector. The module then uses `parameterize` to set the value of the `current-inspector` to this newly created inspector for the duration of the evaluation of

```
(dynamic-require "inspected.rkt" 'instance-of-s)
```

As a result, the value of `v` is a *transparent* instance of `s`, which is defined in `inspected` but exported without access methods. Hence, when `inspector` is loaded into the read-eval-print loop of DrRacket, `v` prints as `(s 1)`.

Listing 10 Programming operating-systems patterns in Racket

```
#lang racket                                                    universe.rkt
...
(define (launch-many-worlds* . th*)
  ;; allocate resources of th ... in the currently active custodian
  (define cc (current-custodian))
  ;; allocate resources of launch-many-worlds in new custodian c*
  (define c* (make-custodian))
  (define ch (make-channel))
  (parameterize ([current-custodian c*])
    ...
    (channel-put ch
      (list i (parameterize ([current-custodian cc]) (th)))))
  ;; th ... send values to channel ch;
  ;; if any of these is an exception structure, shut down
  ...
  (when (exn? x)
    (custodian-shutdown-all c*)
    (raise x))
  ...)
```

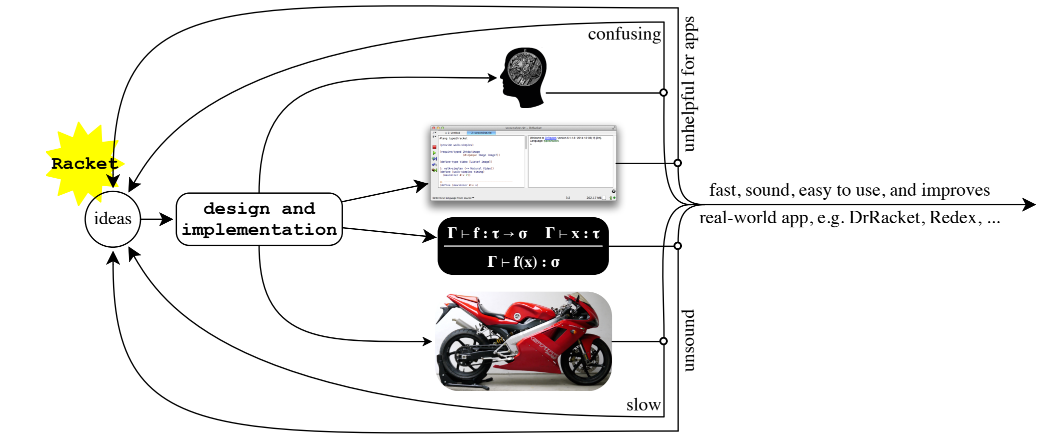
Listing 10 presents an example of resource administration, another operating-system service turned into a Racket construct. It displays the essence of the `launch-many-worlds` function, which is used to run students' distributed programs [8] in parallel. The function consumes an arbitrary number of thunks and runs them in parallel until all of them have produced a proper value or one of them has signaled an exception. Since the function itself consumes resources, it uses two *custodians*: its caller's – to manage the resources of the given thunks – and a new one – to manage its own resources, mostly threads. If any of these thunks raise an exception, the latter custodian is shut down and all of `launch-many-world`'s resources are released. For a more sophisticated pattern of killing threads safely, see Flatt and Findler's work on "kill safety" [15].

Finally, Racket also internalizes other aspects of its context. Dating back to the beginning, Racket programs can programmatically link modules [14] and classes [18]. In conventional languages, programmers must resort to extra-linguistic tools to abstract over such linguistic constructs; only ML-style languages and some scripting languages make modules and classes programmable, too.

6 The Racket Design Feedback Loop

The design of a language must take place in the context of a feedback loop. Like the feedback loop for many programming languages, Racket's feedback loop contains soundness theorems [40], performance evaluations [39], and usability studies [28]. As listing 11 shows, however, the Racket feedback loop also includes a number of software applications While the preceding sections already indicate the role that DrRacket played for developing, honing, and checking the principles, the creation of large and complex domain-specific languages such as Redex [5, 30], Scribble [13], Slideshow [11] and others have had an equally significant impact.

Listing 11 The Racket design feedback loop


Racket
ideas
design and implementation
confusing
unhelpful for apps
fast, sound, easy to use, and improves
real-world app, e.g. DrRacket, Redex, ...
$\Gamma \vdash f : \tau \to \sigma \quad \Gamma \vdash x : \tau$
$\Gamma \vdash f(x) : \sigma$
unsound
slow

Redex is probably the most sophisticated client of Racket's syntax extension system. It employs the latter at two levels: to compile the Redex language of grammar, type, and semantics definitions and as a target for the compilation. As such, Redex has stretched and expanded the syntax extension system.

Scribble is a domain-specific language for creating Racket documentation. Unlike the documentation system of conventional languages, a Scribble program can refer to, and compute with, bindings from a Racket library. As a result, programmers can easily create intensively cross-referencing manuals, language guides, and books in such a way that each occurrence of an identifier is *automatically* linked to its documentation. In fact, a Scribble file is just a Racket module, so Scribble documents come with all the benefits of other Racket code – including separate compilation, a feature absent in numerous document markup processors. Integrating Scribble with Racket exposed a gap in, and thus forced an expansion of, the phase separation model of Racket's syntax extension system [12].

Finally, Slideshow is both a domain-specific language for programming presentations and a graphical tool for displaying them. For a linguist with an awareness of the language of discourse, designing a language for the programmatic creation of presentations is a natural step. Presenters want a single point of control: they want parametrized re-use of slides, slide elements, and other concepts that are most easily expressed with a language but are difficult to obtain in a WYSIWYG tool. A programmer-as-presenter also has the natural desire to evaluate code within a presentation, possibly even the presentation itself [17]. Because of this combination, Slideshow's construction plays almost the same role in the feedback loop of design as the DrRacket IDE.

In general, all of these applications challenge the linguist in the problem-solving programmer. Each poses several different kinds of problems, best articulated and solved with problem-specific languages. In all cases, the purpose of the languages is to provide a protected and enforced abstraction. Equally important, they all need fine-grained control over resource-management mechanisms that are usually found outside of the language. Their existence and their designs both confirm the Racket principles and illustrate them, so Racketeers frequently consult these applications when they contribute new languages or new concepts to existing languages in the realm of Racket.

7 Racket, the Future: From Imperfections to Research Opportunities

Racket's design principles have produced a programming language that

- enables the rapid creation of new languages for specific problem areas and thus enables language-oriented programming;
- supports a full spectrum of general-purpose programming languages with various conventional degrees of safety; and
- internalizes mechanisms from its system context into linguistic constructs for fine-grained, programmable control.

Turning principles into reality almost always yields an incomplete, and possibly even flawed, product.[4] Racket is no exception, but we consider these imperfections as opportunities for future research. The remainder of this paper sketches some of them.

Racket's key advantage is its syntax extension system. It makes experienced programmers extremely productive, but it comes with an extraordinarily steep learning curve. Its syntax elaboration algorithm is hard to understand; its toolbox is large and complex; and it has some brittle, unexplored corners that occasionally trip up even experienced programmers. The situation calls for simplifications of the syntax system and for the creation of a smooth ramp for the toolbox (in terms of both tools and documentation).

In addition, the syntax extension system does not allow for a separation of concerns, and programs suffer from this. For example, many programming languages allow programmers to separate specifications from implementations. In conventional Racket, contracts play the role of specifications, functions implement them, and programmers may choose to separate the two concerns in a module. No such separation exists for the syntax system. While Culpepper's dissertation [3] research has made some progress in this direction, a lot more work on separating syntax specifications from syntax implementations is needed. Realizing both will greatly improve Racket's support for the principle of language-oriented programming.

Besides a language, modern programmers need an ecosystem. Indeed, many programmers equate languages with their ecosystems. For Racket, this equation means that the creation of a new language ought to include the derivation of an IDE from DrRacket. To some extent, DrRacket can already support new languages automatically, e.g., with on-line syntax checking and simple refactoring actions. For other tools, such as a syntax-directed stepper, this process would need a significant amount of work and comes without guidance or automation.[5]

The currently available enforcement mechanisms give rise to a full spectrum of conventional programming languages: Typed Racket, Racket with contracts, Racket, and `ffi/unsafe` Racket. Although this spectrum is expressive, it lacks power at both ends. To achieve full control over its context, Racket probably needs access to assembly languages on all possible platforms (from hardware to the web's JavaScript). How to integrate this power in a portable manner is unclear. To realize the full power of types, Racket will have to be equipped with dependent types. Tobin-Hochstadt and his Typed Racket group are currently working on first steps in this direction, focusing on numeric constraints in `typed/racket`. When a Racket program uses vectors, its corresponding typed variant type-checks what goes into these vectors and what comes out, but like ML or Haskell, indexing is left to a (contractual) check in the run-time system. Integrating Xi and Pfenning's form of programming with numeric constraints [44] into `typed/racket` is a natural step beyond plain types.

[4] The same caveat applies to the design *process*, not only its result, but covering the positives and negatives of the design process is beyond the scope of this paper.

[5] Spoofax [27] comes with domain-specific languages for the generation of IDE tools, but also relies on extra-linguistic mechanisms.

More generally, Racket's spectrum of languages creates a multi-lingual world for programmers, though with far more structure than currently found in practice. Even though Matthews and Findler [29] have studied the basics of multi-lingual programs, their theory covers only a small part of this world. Our work on Racket clearly calls for extensions of this result in several directions, including the sound interaction between by-value and by-name (or lazy) variants of Racket, typed and dependently typed variants, and so on. We expect that studying how to protect verified code as it is co-mingled with other kinds of code will yield new insights into Racket's safety mechanisms.

Racket must also broaden its horizon and consider security concerns, both as an enforcement action but also as an application of the third principle. While sandboxes address some of the security concerns of running a student program in a homework submission server, properly addressing this problem calls for articulating security policies and enforcing them in a system. Moore, et al. [33] recently presented Shill, a secure scripting language implemented atop Racket. Their work exposed serious gaps between Racket's principle of language-oriented programming and its implementation as well as in Racket's approach to enforcing security. Once again, we consider these weaknesses an opportunity to improve Racket and expect to study these problems in the near future.

Acknowledgments. We wish to thank Claire Alvis, Ryan Culpepper, John Clements, Stephen Chang, Richard Cobbe, Greg Cooper, Christos Dimoulas, Bruce Duba, Carl Eastlund, Burke Fetscher, Cormac Flanagan, Kathi Fisler, Tony Garnock-Jones, Paul Graunke, Dan Grossman, Kathy Gray, Casey Klein, Jacob Matthews, Scott Owens, Greg Pettyjohn, Jon Rafkind, Vincent St-Amour, Paul Steckler, Stevie Strickland, James Swaine, Asumu Takikawa, Kevin Tew, Neil Toronto and Adam Wick for helping us hone the guidelines.

In addition to the reviewers, Ben Greenman, Vincent St-Amour, Asumu Takikawa made several suggestions on improving the presentation.

References

1 Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

2 Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308, 2006.

3 Ryan Culpepper. Fortifying macros. *J. Functional Programming*, 22(4–5):439–476, 2012.

4 R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

5 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* The MIT Press, 2009.

6 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs.* MIT Press, 2001.

7 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *J. Functional Programming*, 14(4):365–378, 2004.

8 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A Functional I/O System. In *International Conference on Functional Programming*, pages 47–58, 2009.

9 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *J. Functional Programming*, 12(2):159–182, 2002.

10 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pages 48–59, 2002.

11 Robert Bruce Findler and Matthew Flatt. Slideshow: Functional presentations. *J. Functional Programming*, 16(4–5):583–619, 2006.

12 Matthew Flatt. Composable and compilable macros: You want it *when?* In *International Conference on Functional Programming*, pages 72–83, 2002.

13 Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. In *International Conference on Functional Programming*, pages 109–120, 2009.

14 Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Programming Language Design and Implementation*, pages 236–248, 1998.

15 Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *Programming Language Design and Implementation*, pages 47–58, 2005.

16 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems*, pages 270–289. Springer, 2006.

17 Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (*or* revenge of the son of the Lisp machine). In *International Conference on Functional Programming*, pages 138–147, 1999.

18 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Principles of Programming Languages*, pages 171–183, 1998.

19 Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

20 Martin Fowler. *Domain-specific Languages*. Addison-Wesley, 2010.

21 Paul Hudak. Domain specific languages. In Peter H. Salas, editor, *Handbook of Programming Languages*, volume 3, pages 39–60. MacMillan, Indianapolis, 1998.

22 Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *International Conference on Functional Programming*, pages 343–350, 2013.

23 Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *Lisp and Functional Programming*, pages 151–161, 1986.

24 Eugene E. Kohlbecker and Mitchell Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *Principles of Programming Languages*, pages 77–84, 1987.

25 Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, 2001.

26 Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. In *International Symposium on Generative and Component-Based Software Engineering*, pages 105–120, September 1999.

27 Kats C.L. Lennart and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *Object-Oriented Programming Systems, Languages & Applications*, pages 444–463, 2010.

28 Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Technical Symposium on Computer Science Education*, pages 499–504, 2011.

29 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31(3):1–44, 2009.

30 Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Rewriting Techniques and Applications*, pages 2–16, 2004.

31 Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

32 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

33 Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. Shill: A secure shell scripting language. In *Operating Systems Design and Implementation*, pages 183–199, 2014.

34 Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin Mcmullan. Lexer and parser generators in Scheme. In *Scheme and Functional Programming*, pages 41–52, 2004.

35 Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass framework for compiler education. *J. Functional Programming*, 15(5):653–667, 2005.

36 T. Stephen Strickland, Christos Dimoulas, Asumu Takikawa, and Matthias Felleisen. Contracts for first-class classes. *ACM Trans. Program. Lang. Syst.*, 35(3):11:1–11:58, 2013.

37 T. Stephen Strickland and Matthias Felleisen. Contracts for first-class modules. In *Dynamic Language Symposium*, pages 27–38, 2009.

38 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *Object-Oriented Programming Systems, Languages & Applications*, pages 943–962, 2012.

39 Asumu Takikawa, Daniel Feltey, Sam Tobin-Hochstadt, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. Towards practical gradual typing. In *European Conference on Object-Oriented Programming*, 2015. To appear.

40 Asumu Takikawa, Stevie Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Object-Oriented Programming Systems, Languages & Applications*, pages 793–810, 2012.

41 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Principles of Programming Languages*, pages 395–406, 2008.

42 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Programming Language Design and Implementation*, pages 132–141, 2011.

43 Tobias Wrigstad, Patrick Eugster, John Field, Nate Nystrom, and Jan Vitek. Software hardening: A research agenda. In *Script to Program Evolution*, pages 58–70, 2009.

44 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Programming Language Design and Implementation*, pages 249–257, 1998.

Image References

Head Logo image in listing 11 is remixed from Nuri Aydoğdu's under AY SA 3.0.
Motorcycle image in listing 11 is remixed from Klaus Nahr's under AT SA 2.0.

A Theory AB Toolbox

Marco Gaboardi[1] and Justin Hsu[2]

1 **University of Dundee, UK, and Harvard University, US**
`m.gaboardi@dundee.ac.uk`

2 **University of Pennsylvania, US**
`justhsu@cis.upenn.edu`

Abstract

Randomized algorithms are a staple of the theoretical computer science literature. By careful use of randomness, algorithms can achieve properties that are simply not possible with deterministic algorithms. Today, these properties are proved on paper, by theoretical computer scientists; we investigate formally verifying these proofs.

The main challenges are two: proofs about algorithms can be quite complex, using various facts from probability theory; and proofs are highly customized – two proofs of the same property for two algorithms can be completely different. To overcome these challenges, we propose taking inspiration from paper proofs, by building common tools – abstractions, reasoning principles, perhaps even notations – into a formal verification toolbox. To give an idea of our approach, we consider three common patterns in paper proofs: the union bound, concentration bounds, and martingale arguments.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Verification, randomized algorithms

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.129

1 Introduction

One of the most unexpected discoveries in the study of algorithms is the *power of randomness*. For many tasks, random algorithms perform better than deterministic algorithms, both in theory and in practice. Therefore, it's not surprising that random algorithms have been widely adopted in practice; examples span a wide range of computer science, from machine learning to database technology to graphics.

A natural question is: How correct are these algorithms? Today, proofs of these requirements are accomplished by enlisting highly skilled humans to produce "*paper proofs*" by hand. However, even the most skilled humans are not perfect, and even if an algorithm is proved correct, who is to say that it is *implemented* correctly? We propose to use tried-and-true techniques from program verification – like type systems and more general static analyses – to check, and perhaps automatically derive, *formal* versions of the requisite *paper proofs*.

At first glance, the paper proofs are extremely diverse, conjuring up insights that seem totally mystifying. Faced with this problem, a tempting approach is to use general-purpose theorem provers. These are extremely expressive, but there is a real cost: The proofs work at a low level of abstraction that bears little resemblance to the informal proof. As a result, the formal proofs are verbose and require substantial manual effort to construct. An expert who knows the formal proof perfectly clearly often labors to convince a theorem prover of this fact, and may be unable to make any sense of a formal proof.

To avoid these drawbacks, we envision building a toolbox of verification techniques following an overarching principle: *take inspiration from human reasoning*. While some parts of a given paper proof may be genuinely novel, experts often rely on simple yet powerful abstractions for intuitive reasoning about particular properties; accordingly, paper proofs remain simplest, most concise, and most lightweight. As much as possible, we want to take advantage of these patterns used when humans reason about randomized algorithms.

Compared to carrying out proofs with general purpose theorem provers, we expect that narrowing the gap between informal and formal reasoning will lead to simpler, more intuitive proofs; humans familiar with the paper proof of properties should find it possible to use our verification framework with minor training. By restricting the type of reasoning, we also expect a higher degree of automation in our tools, reducing the verification burden even further.

As we hope to build everything around proof patterns, we must accept that there are facts about randomized algorithms that we will not be able to verify. Some theorems may use "ad hoc" tools – or at least, uncommon tools that are not worth the effort in formalizing. The focus of our toolbox – at least at first – is to provide a tool for composing mathematical facts rather than verifying everything from the ground up. Based on our survey of proofs we might hope to verify, we believe a small collection of primitive axioms and operations along with powerful composition patterns will already suffice to verify many interesting proofs.

2 What is correctness for randomized algorithms?

Randomized algorithms serve many functions, some mundane – movie recommendations, elevator scheduling; some important – driving directions, credit approval ratings; and some life-critical – robotic surgery, air traffic control, safety systems in cars.

These applications involve randomization in a variety of ways. For instance, an algorithm may only have access to "noisy" inputs, and the goal is to control the effect of the noise. Or the algorithm may introduce randomness to a deterministic problem; maybe the added noise helps smooth out worst-case inputs, leading to faster execution or smaller space usage. Or maybe the randomness is inherently part of the problem of interest.

With so many settings, correctness can mean many different things. For instance, we can consider a program correct if it returns an *optimal*, or alternatively we can consider it correct if it returns a output *quickly* that may not be optimal, but is close enough. There are probabilistic versions of typical metrics for deterministic algorithms, like expected accuracy or expected convergence rate, but there are also performance metrics specific to randomized algorithms, like generalization error in machine learning algorithms. Further complicating the picture, properties often model how a program scales as the input size is changed. For instance, an algorithm that sorts a list of n elements with n^2 operations on average is qualitatively worse than an algorithm that uses $n \log n$ operations on average. This scaling behavior is critical when evaluating algorithms, which may be run on very large sets of data.

A common thread in proving accuracy of randomized algorithms is reasoning about *probabilities*, whether explicitly or implicitly. For example, an interesting guarantee might be "With at least 99% probability, the algorithm computes an answer within 0.01 of the true answer". A different accuracy and convergence rate guarantee could be "The expected value of the complexity of the randomized algorithm to reach an accuracy bound α is quadratic in $1/\alpha$". These probabilities may also depend on input parameters, like the size of the input or the desired level of confidence.

Though there have been quite successful formalizations of probability theory (for just one example, [2]), they typically model probabilities from a single point of view – say, as explicit distributions. In contrast, typical paper proofs treat probabilities in a multi-faceted way. The proof may work with random variables, when the reasoning looks a bit like symbolically manipulating algebraic formulas; say, two samples Z_1 and Z_2 from a normal distribution may be added together as $Z_1 + Z_2$. In other cases, proofs may work with probabilities of certain events directly; say, proving that $\Pr[X > 10] = 0.1$. Proofs may even involve looking at probabilities geometrically.

In our view, the main challenge of handling proofs about randomized algorithms is organizing and structuring the essential argument so that the numerous theorems can be brought to bear in a usable and lightweight way.

3 The current state-of-the-art

We are certainly not the first to argue the fact that programming language research should develop more techniques for randomized algorithms.

In a closely related area, there is a large amount of work in analyzing and testing approximate algorithms, for instance recent work by Sampson et al. [18]. In most of these works the goal is to provide guarantees on the performance of randomized programs by carrying out first some dynamic analysis inferring a synthetic representation of the random process, and then some static analysis to simplify and test the results against a specification. While interesting, this approach does not provide actual proofs of the rigorous properties.

There are also a number of works on the verification of randomized algorithms, either by using full-blown formalization in a theorem prover [2], or ad-hoc verification techniques based on program analysis [7]. There has been recent success in proving correctness (for various definitions of correctness) of randomized algorithms in specific domains like cryptography [3, 5], differential privacy [13, 6], and more.

However, looking at these and other approaches in the literature, it seems that there are few tools that produce proofs remotely similar to paper proofs! Indeed, while attending the Approx 2014 [1] workshop co-located with PLDI 2014 – devoted to the topic of verifying randomized computation – we were struck by the lack of attention to *proofs*.

We believe that part of the problem here is that all these approaches have been studied by programming languages researchers with programming languages tools for the programming languages community. Our goal is instead to provide a toolbox that is interesting and useful for people from the algorithm and machine learning community but based on techniques from the programming language community. In some sense, we aim for a toolbox combining interesting aspects of both theory A and theory B.

4 Our proposal: Take abstractions from paper proofs

As we have discussed, we envision tools built around common structures humans use when proving correctness properties. These patterns take a variety of shapes and forms, but fundamentally, they are *abstractions*: they are used to give structure to the key parts of an argument, in a flexible and concise form, while concealing the boilerplate, "boring" parts of the proofs. From this point of view, what makes these patterns interesting is that while there are the typical abstractions we know and love – modularity, inductive arguments, etc. – there are also patterns that are harder to recognize. For instance, they may involve reasoning about code in a non-modular way, or grouping random sampling in particular, non-local ways. We believe that understanding and capturing patterns from paper proofs – empirically, the best abstractions – is a key step to designing usable and powerful verification tools.

As a first step in designing our toolbox we will focus on accuracy. Here, we briefly discuss three such proof patterns and how they might be profitably incorporated into a verification framework. For concreteness we will focus mostly on language-based verification, though we expect these principles to be valuable in a wide variety of verification approaches.

4.1 Case study: The union bound

The union bound states that if event A happens except with probability p_A, and event B happens except with probability p_B, then both A and B happen except with probability at most $p_A + p_B$. In the context of randomized algorithms, A and B are often taken to be the event that some random noise is small. The probabilities p_A and p_B are often called *failure probabilities* – they are the probabilities that event A or B fails to happen.

Even though randomized algorithms naturally mix probabilistic and deterministic operations, the accuracy proof can be simplified by separating out the analysis of the random operations. A common first step when proving accuracy is to apply the union bound to argue that, except with probability p, all the random noise added is small. Assuming this fact, the rest of the analysis can then treat each sampling operation as returning small noise, and prove accuracy *without* explicitly reasoning about randomness.

In general, the reasoning naturally divides into two parts: (1) keep track of the total probability that the noise is too large, using the union bound to aggregate the failure probability of various sampling operations, and (2) carry out the accuracy analysis by viewing the original program as a *deterministic* program, where each sampling operation is assumed to return some small value.

For example, consider an algorithm that adds noise to each element of a list:[1]

Listing 1 Applying a union bound

```
noiselist (in : list R) :[b] { out : list R | dist(in, out) < T } =
  match in with
  | Nil -> return Nil
  | Cons x xs ->
    sample noisy <- addnoise T x in
    sample rest <- noiselist xs in
    return (Cons noisy rest)
```

The noise function `addnoise` comes with an accuracy specification encoding the probability of adding noise more than T. The annotation in the first line state that the output list `out` has each element at most T away from the corresponding element in input list `in`. The annotation `[b]` indicates the *failure probability*: the accuracy assertion holds with probability at least $1 - b$.

The union bound is carried out by the use of `sample`. In more detail, suppose we have shown that program e satisfies accuracy assertion A with failure probability p_A, and program e' satisfies accuracy assertion B with failure probability p_B, *assuming that A holds.* Then, our language concludes that the program `sample x <- e in e'` (which samples x from e and runs e') satisfies assertion B, but now with the failure probability given by the union bound: $p_A + p_B$.

Looking more closely, we can view the union bound as a kind of composition principle: to analyze the failure probability of a big algorithm, it's enough to analyze the failure probability of each of its component programs. This is a powerful principle that simplifies the reasoning on paper, and we should take advantage of this principle in verification as well.

[1] We give pseudocode in a functional language to illustrate our ideas, but we believe these ideas can be used broadly in a variety of verification settings, see Section 5 for further discussion.

4.2 Case study: Reasoning about independence

While a wide range of randomized algorithms can be proved accurate with just the union bound, some algorithms require a finer analysis based on *independence of random variables*. For instance, some accuracy results rely on *concentration around the mean* (sometimes called *concentration bounds*): the principle that, if we take a series of independent random draws from a distribution μ, the average of the draws is with high probability not too far from the mean of μ. For example, the accuracy proof of a random counter [8, 10] from the privacy literature uses a concentration bound for sums of independent draws from the Laplace distribution.

Arguments in terms of independence are problematic for program verification. They are often global, involving reasoning about collections of random draws that may be far apart in the actual code of the program. A priori, it may not be clear that the random draws are even independent. To provide some structure, we can imagine a two-stage approach.

First, we can separate random sampling from the main program and annotate it with where and how to apply the independence bounds. This makes it easier to verify independence for collections of random draws.

For example, suppose we want to take three independent samples from some distribution μ. Furthermore, say that there is a concentration bound stating that the sum of up to three independent draws from μ satisfies some property ϕ. In a first stage analysis we might write the following:

Listing 2 Applying independence bounds

```
sample n1 <- mu in
sample n2 <- mu in
sample n3 <- mu in

sum1 = boundsum n1 :[b] phi1 in
sum2 = boundsum (n1, n2) :[b] phi2 in
sum3 = boundsum (n1, n2, n3) :[b] phi3 in
(sum1, sum2, sum3)
```

Each `boundsum` applies the concentration bound to a list of random samples. The returned values `[sum1, sum2, sum3]` represent the sums n_1, $n_1 + n_2$, and $n_1 + n_2 + n_3$ respectively, each tagged with a probabilistic assertion ϕ_1, ϕ_2, or ϕ_3 and failure probability b given by the concentration bound. For instance, the assertions might look like

$$\phi_1(x) := |x - c| < T_1$$
$$\phi_2(x) := |x - 2c| < T_2$$
$$\phi_3(x) := |x - 3c| < T_3,$$

where c is the mean of distribution μ.

The guarantees from concentration bounds – ϕ_i holds with probability at least $1 - \beta$ – are probabilistic assertions, which can then be combined with the union bound. Indeed, the two principles are often used together in paper proofs, first applying concentration bounds relying on independence to arrive at some preliminary facts, then applying union bounds to combine the facts.

In the second stage, the samples, along with facts derived from applying the independence bounds, are fed into a program written in our union bound language. There, the samples can be used and the facts assumed as needed, tracking the failure probability. In the simple example above, after applying the independence bounds, we may want to reason about adding up the sums. The corresponding second stage program might look as follows:

Listing 3 Second stage program

```
sumup (s1 :[b] { x : R | phi1 } )
      (s2 :[b] { x : R | phi2 } )
      (s3 :[b] { x : R | phi3 } )
      :[3 * b] { out : R | dist(out, 6 * c) <  T1 + T2 + T3 } =
sample x <- s1 in
sample y <- s2 in
sample z <- s3 in
return (x + y + z)
```

Note that the inputs to this program carry the assertions derived in the first-stage program above. Also note that the `sample` operation combines the failure probability (b for each assertion) to give the final failure probability ($3b$).

4.3 Case study: Martingale reasoning

Our third and final pattern involves *martingales*. Formally, a martingale is a sequence of random variables $X_1, X_2, \ldots$. We allow X_i to depend on all previous random variables $X_1, \ldots, X_{i-1}$. The martingale property requires that this sequence is somehow "memory-less": the information at time step i is captured by X_{i-1}, rather than the whole sequence $X_1, \ldots, X_{i=1}$. More formally, for any concrete values $x_1, \ldots, x_{i-1}$, we require

$$\mathbb{E}[X_i \mid X_1 = x_1, \ldots X_{i-1} = x_i] = x_i$$

A canonical example is a *random walk*. A person starts at a point on a line (call it 0), and at each time step, flips a fair coin. If heads, she goes to the left one unit; if tails, she goes to the right unit. If we let X_i be her position at time i, then $X_1, \ldots, X_i$ is a martingale sequence.

A particularly powerful fact about martingales is the following theorem. We will present it informally, since the details are somewhat technical to spell out.

▶ **Theorem 1** (Optional Stopping Theorem, Informal). *Let $X_0, X_1, X_2, \ldots$ be a martingale sequence, and let $\tau \in \mathbb{N}$ be a* stopping time *– a random variable that depends only on the martingale values before τ. For instance, we should be able to decide if $\tau = 3$ given just X_0, X_1, X_2, X_3. Then,*

$$\mathbb{E}[X_\tau] = \mathbb{E}[X_0].$$

Note that on the left, the time τ may be random as well, since it depends on the random sequence.

We think the optional stopping theorem can be a powerful tool for verifying probabilistic programs with loops. For instance, consider the following loop:

Listing 4 A loop

```
X = Y = 0;
while (|X| < 10) do
  Y = X;
  sample f <- flip(-1, 1);
  X = X + f;
end
```

The optional stopping theorem provides a powerful and flexible way to reason about this loop, which involves probabilistic sampling. Specifically, we can think of the iteration when this loop terminates as a stopping time – this iteration depends only on the previous iterations. If we know that X evolves as a martingale, then we can conclude that

$$\mathbb{E}[X_{fin}] = \mathbb{E}[X_0] = 0.$$

Since we know that X_{fin} is 10 or -10, this immediately lets us conclude that $\Pr[X_{fin} = 10] = \Pr[X_{fin} = -10]$, something that is awkward to prove via other means.

We have outlined a simple example, just to give an idea of what verification via martingales may look like. There is a wide variety of proofs that can be carried out with similar martingale arguments, just by choosing the correct martingale and applying the optional stopping theorem. Picking the proper martingale to deduce a particular fact is something of a black art, and not something we would expect could be handled automatically. In many respects, finding the right martingale is similar to specifying a loop invariant – some human insight seems to be required.

That being said, martingales are a powerful technique for proving facts about probabilistic programs. For instance, it's know that many independence bounds (as described in Section 4.2) are actually consequences of optional stopping. Rather than building independence bounds into our toolbox (i.e., assuming them), martingales may allow us to actually *derive* these principles directly.

5 Building the toolbox

So far, we've seen several powerful reasoning principles that could be integrated into a formal verification framework. However, this still leaves us quite a ways away from an actual, usable tool. As we see it, there are at least three promising routes to that goal.

5.1 Working within an existing theorem prover

The last decades have seen an explosive growth and maturation in theorem prover technology. A celebrated recent result is the machine-checked proof of the Feit-Thompson theorem from group theory [14], evidence that contemporary theorem provers are starting to be able to prove complex theorems from the mathematical literature. Since proofs about randomized algorithms share many similarities with mathematical proofs – ad hoc arguments, high-level reasoning, custom notation – this bodes well for verifying randomized algorithms.

Accordingly, we could imagine developing Coq theorems and libraries to implement the common reasoning principles we have sketched. The main challenges in this approach, in our mind, are two-fold: 1) developing a flexible and reusable theory of random variables, which may also require a large investment in formalizing probability theory, and 2) connecting the reasoning to the actual program.

5.2 Developing a custom type system

Since rich type systems have been quite successful [4, 13, 16] for verifying differential privacy, a notion of privacy for randomized algorithms, the idea of a custom type system for verifying accuracy is quite appealing. However, there appear to be serious challenges – not insurmountable by any means, but serious nonetheless.

In our view, the success of type systems for verifying privacy was enabled by particular compositional features of differential privacy that allow an extraordinarily modular and clean analysis. The target proofs of privacy are roughly of the same form and structure, barring a few important exceptions.

In contrast, proofs of accuracy are far more diverse. There are a variety of reasoning principles, and they can be combined almost every which way. Furthermore, many proofs use non-local reasoning, like arguments involving independence; for instance, we may need a way of working with groups of random samples that are spread wide throughout the program, reasoning about when the groups are disjoint, when they are not, etc. It's possible that technologies like separation logic could be brought to bear on this kind of analysis, but it's far from clear.

5.3 Developing a custom Hoare logic

The final direction is to work with imperative programs, and develop a Hoare logic. The language itself wouldn't need to be very complex; say, a simple While language with commands for sampling from built-in distributions, and maybe an expectation operator.

However, a key challenge is to design an expressive logic for assertions. Randomized algorithms often work with a number of variables that depends on the input parameters. The assertion logic must be concise and flexible enough to express complex invariants over all of these variables and samples. A natural starting point would be work by Gonthier et al. [14] on developing formal versions of mathematical notation.

This is the direction we believe is most promising. Along with our collaborators, we are currently looking into building a Hoare logic for accuracy in EasyCrypt [5], an interactive verification system for proofs about cryptographic properties.

6 Looking ahead

The three case studies above are meant to give just a taste of the kind of toolbox we have in mind. Of course, there are many possible extensions; we briefly sketch a few directions below.

6.1 More tools in the toolbox

The most natural extension is simply adding more advanced tools.

Refining the union bound

The union bound in Section 4.1 is used extremely frequently, but it is a somewhat loose bound. There are several known refinements that give sharper bounds on the failure probability under certain assumptions. One promising candidate is the Lovász Local Lemma [12], which gives a better bound when each event is independent of most of the other events.

Advanced concentration bounds

While the concentration bounds in Section 4.2 require independence, there are more general bounds that relax this requirement. For instance, the Azuma-Hoeffding bound [9] gives concentration for martingale sequences $X_0, X_1, \ldots$. This is a weaker requirement than full independence, but can be trickier to prove.

6.2 Combining the tools

So far, we have proposed tools that focus on individual reasoning principles in a somewhat independent fashion. However, there is no reason that these techniques must be used in isolation. Obviously, there are some rules governing how, say, independence bounds can be mixed with martingale reasoning, but in paper proofs, different principles can be flexibly mixed and matched. In our view, understanding how to formalize and flexibly combine different tools is an important direction for handling more complex proofs.

In Section 4.2, we have a proposed a very small step in this direction: mixing union bounds and independence bounds. However, our proposal leaves much to be desired. For instance, it requires that independence bounds must be applied first, then union bounds. For many algorithms, this may require coding the algorithm in an unnatural way to conform to this proof structure. An ideal solution would avoid this unnecessary burden.

6.3 Beyond accuracy

While accuracy is certainly an important property of randomized algorithms, there are a host of other properties that could be handled by formal verification.

Incentive and game-theoretic properties

In game theory settings, the inputs to an (often randomized) algorithm are controlled by rational agents that may seek to manipulate the output of the algorithm but changing their input. *Incentive properties* guarantee that agents can't gain much by this kind of manipulation.

From a program verification point of view, incentive properties are *relational program properties*. They reason about the relation between two runs of the same program: one where agents report honestly, and one where agents manipulate their inputs. As we observed in recent work [4], basic incentive properties can be fruitfully handled with relational program verification techniques.

However, the current state of verification for incentive properties is stuck on the same obstacle: More complex incentive properties depend on accuracy guarantees, which the tools we have are ill-equipped to handle. There are several examples for algorithms which compute equilibria – in a nutshell, the goal is to coordinate players to actions where no player has incentive to deviate (a kind of *stability* condition), but players may have incentive to manipulate which equilibrium is chosen. For instance, Rogers and Roth [17] propose an algorithm for computing Nash equilibrium that depends on the accuracy of randomized counters; Kearns, et al. [15] propose an algorithm for computing so-called coarse correlated equilibrium, where the incentive properties depend on the accuracy of a sophisticated learning algorithm.

Randomized computational complexity

The computational complexity of randomized algorithms is a deep area of study. At the most basic level, we might want to verify the complexity of a program that has probabilistic running time. The typical notion here is *expected running time*, the average time the program takes. The problem is most interesting (and most difficult) when the program runs a loop with a probabilistic guard. In such cases, the program may not even always terminate!

By now, computational complexity is a very developed field with an extensive collection of complexity proofs. However, there are few (if any) domain specific tools to actually build complexity proofs today. A general question is whether there is a reasonably small class of patterns that we can target to verify a large class of examples.

Alternative notions of randomness

Since truly random bits are expensive to generate in practice, there is a large body of literature looking at more restricted sources of randomness. For instance, we may use a pseudorandom generator to turn a few truly random bits into a much longer string of psuedorandom bits; maybe instead of working with independent random variables, we work with weaker guarantees that are easier to achieve, like pairwise independent random variables. These algorithms are ingenious, and often have complex proofs of correctness.

The probabilistic method

Finally, our toolbox can be viewed as capturing reasoning about probabilistic quantities, not just quantities related to algorithms. For general mathematical proofs, a powerful tool is the *probabilistic method*, invented by Erdős [11]. This principle shows the (non-constructive) existence of an object satisfying a specific property P by first constructing a *random* instance of the property, then showing the probability P holds is strictly positive. The tools of probability theory are powerful and broadly applicable to general mathematics; to the extent that our toolbox can capture probabilistic reasoning, we should be able to model general mathematical proofs.

7 Conclusion

As our field of programming languages (and more generally, formal verification) matures, it is crucial to export our tools and techniques to other fields, so that all kinds of researchers can hear what formal verification has to say, as it were. By interacting with other fields, our field stands to be greatly enriched by exchanging ideas and considering new, interesting properties to verify. Our proposal to verify properties of randomized algorithms is a step in this program, but just a step. We hope there are many more steps to come.

References

1 First SIGPLAN Workshop on Probabilistic and Approximate Computing (APPROX'14). Co-located with PLDI 2014, Edinburgh, Scotland., 2014.

2 Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in coq. *Science of Computer Programming*, 74(8):568–589, 2009.

3 Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California*, pages 193–206, 2014.

4 Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, 2015.

5 Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st annual conference on Advances in Cryptology (CRYPTO)*, pages 71–90, 2011.

6 Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Philadelphia, Pennsylvania*, pages 97–110, 2012.

7 Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Beijing, China*, pages 169–180, 2012.

8 T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. *ACM Transactions on Information and System Security*, 14(3):26, 2011.

9 Devdatt P Dubhashi and Alessandro Panconesi. *Concentration of measure for the analysis of randomized algorithms.* Cambridge University Press, 2009.

10 Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. Differential privacy under continual observation. In *ACM SIGACT Symposium on Theory of Computing (STOC), Cambridge, Massachusetts*, pages 715–724, 2010.

11 Paul Erdős. Graph theory and probability. *Canadian Journal of Mathematics*, 11:34–38, 1959.

12 Paul Erdos and László Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. *Infinite and finite sets*, 10:609–627, 1975.

13 Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. Linear dependent types for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 357–370, 2013.

14 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving*, pages 163–179. Springer, 2013.

15 Michael Kearns, Mallesh Pai, Aaron Roth, and Jonathan Ullman. Mechanism design in large games: Incentives and privacy. In *ACM SIGACT Innovations in Theoretical Computer Science (ITCS), Princeton, New Jersey*, pages 403–410, 2014.

16 Jason Reed and Benjamin C Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, 2010.

17 Ryan M Rogers and Aaron Roth. Asymptotically truthful equilibrium selection in large congestion games. In *ACM SIGecom Conference on Economics and Computation (EC), Palo Alto, California*, pages 771–782, 2014.

18 Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, Scotland*, pages 112–122, 2014.

Tracking the Flow of Ideas through the Programming Languages Literature

Michael Greenberg*[1], Kathleen Fisher[2], and David Walker[3]

1 **Pomona College, US**
2 **Tufts University, US**
3 **Princeton University, US**

Abstract

How have conferences like ICFP, OOPSLA, PLDI, and POPL evolved over the last 20 years? Did generalizing the Call for Papers for OOPSLA in 2007 or changing the name of the umbrella conference to SPLASH in 2010 have any effect on the kinds of papers published there? How do POPL and PLDI papers compare, topic-wise? Is there related work that I am missing? Have the ideas in O'Hearn's classic paper on separation logic shifted the kinds of papers that appear in POPL? Does a proposed program committee cover the range of submissions expected for the conference? If we had better tools for analyzing the programming language literature, we might be able to answer these questions and others like them in a data-driven way. In this paper, we explore how *topic modeling*, a branch of machine learning, might help the programming language community better understand our literature.

1998 ACM Subject Classification I.2.7 Natural Language Processing – Text analysis

Keywords and phrases programming languages literature, topic models, irony

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.140

1 Introduction

Over the past half-century, the programming language community has developed a foundational new science, the science of software, that now governs much of our world. One of the legacies of this great work is a vast literature – a literature so large no one person can know more than the tiniest fraction of it. Unfortunately, our normal interaction with this literature is tragically limited. At best, we read a small selection of papers in our area, look at their related work sections and follow the citations back a level or two. Occasionally, a friend will alert us to a neat paper that she has read. That is about it.

If we had better tools for analyzing the programming language literature, there are so many more interesting questions we could ask and answer:

- How have the topics at PL conferences evolved over the last 20 years?
- Did generalizing the OOPSLA call for papers broaden the conference?
- How has O'Hearn's classic paper on separation logic influenced POPL?
- Where did the themes touched by my paper appear in the past?

In addition to answering questions like these for the sake of curiosity, tools better able to analyze the programming language literature might be of help to the chairs of programming language conferences and the editors of programming language journals. Program chairs

* This work was done while Michael was still a postdoc at Princeton University.

would like to know that their PCs cover the appropriate topics effectively. Editors would like reviewers with sufficient expertise, but, perhaps, different perspectives.

Many of these questions are deep semantic questions about the true flow of ideas through the programming language literature and beyond. We cannot answer these questions with perfect accuracy or certainty, but we may be able to approximate them. To investigate this possibility, we have begun to explore an analysis technique developed in the machine learning community, known as *topic models* [4]. Generally speaking, topic models are probabilistic descriptions of the words that appear in a document or a collection of documents. Such models provide insights into the topics that pervade a corpus and the relative weights of those topics. We can use these models to compare individual papers or groups of papers, and we can track the ebb and flow of topics over time. Such models have recently been used to analyze articles from the journal *Science*, the *Yale Law Review*, and other archives of documents [4]. They have also been used to explore the links betweeen academic literature and funding [15].

In this paper, we describe our initial efforts to apply these models and techniques to the programming language literature and to ask questions of interest to the programming language community. For data, we assembled a corpus of 4,355 abstracts from four of the top programming languages conferences (ICFP, OOPSLA, PLDI, and POPL) and the full text of all 2,257 POPL and PLDI papers that have appeared as of February 2015. We have generated models of this corpus as a whole and for the conferences individually and compared them. We have also analyzed individual documents and compared papers based on their models. We have built a web-based tool (available from `tmpl.weaselhat.com`) that allows other researchers to browse the programming language literature by topic, find related papers, and visualize the evoluation of the topics studied in PL over time. Source code for our tools is available on github.[1] Although our results to date are intriguing, this paper marks only the beginning of the project; many interesting questions remain.

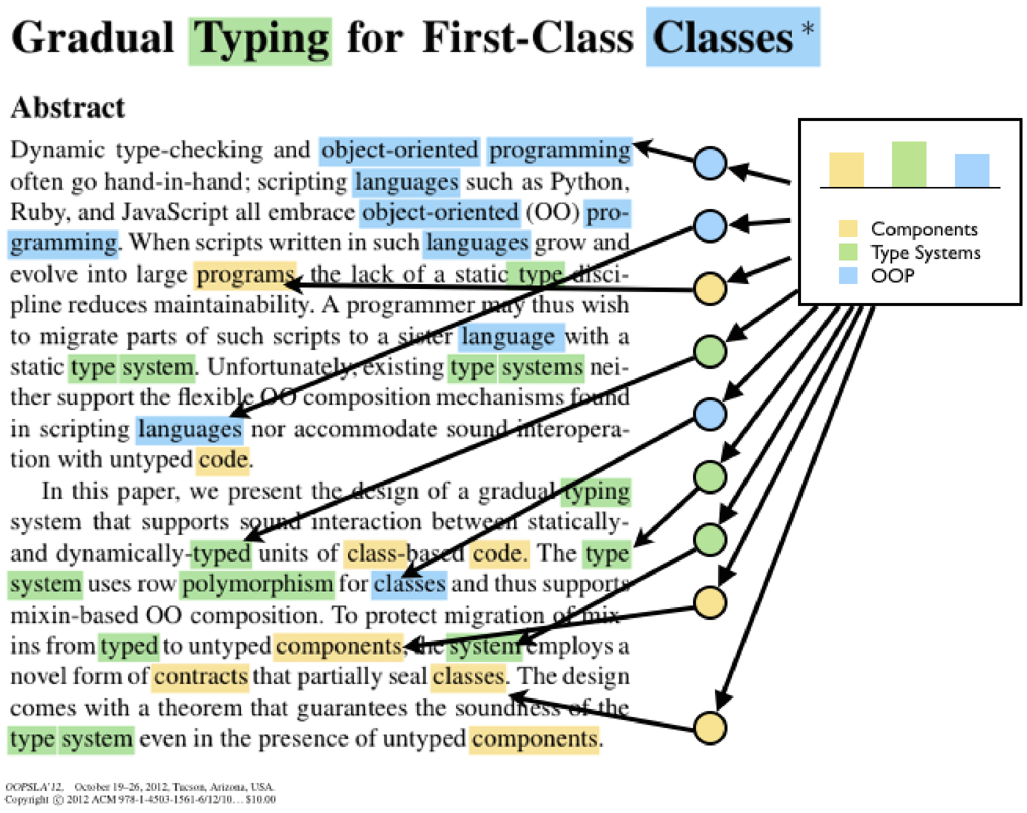
The remainder of this paper is organized as follows. We first describe our methods (Section 2). Specifically, we give a tutorial introduction to *Latent Dirichlet Allocation* topic models and we describe how we curated our corpus. We next present our preliminary results (Section 3) and discuss future work (Section 4). We end with a discussion of related work (Section 5) and concluding thoughts (Section 6).

2 Methodology

2.1 Topic Models

To explain the ideas behind the simplest topic model, the Latent Dirichlet Allocation (LDA), we adapt the explanation given by David Blei [4] to a programming-languages setting. In this model, each document is considered to be a mixture of some number of topics. For example, consider the paper "Gradual Typing for First-Class Classses" [17] shown in Figure 1, which describes a type system for gradually converting dynamically-typed object-oriented programs into statically typed programs in a companion language. We manually highlighted various words occuring in the abstract: yellow for words related to components, green for words related type systems, and blue for words related to Object-Oriented Programming (OOP). If we highlighted the entire paper in this way, it would be apparent that the document mixes concepts from these three topics: components, type systems, and OOP.

[1] See `https://github.com/mgree/sigplan/`.


Gradual Typing for First-Class Classes*

Abstract

Dynamic type-checking and object-oriented programming often go hand-in-hand; scripting languages such as Python, Ruby, and JavaScript all embrace object-oriented (OO) programming. When scripts written in such languages grow and evolve into large programs, the lack of a static type discipline reduces maintainability. A programmer may thus wish to migrate parts of such scripts to a sister language with a static type system. Unfortunately, existing type systems neither support the flexible OO composition mechanisms found in scripting languages nor accommodate sound interoperation with untyped code.

In this paper, we present the design of a gradual typing system that supports sound interaction between statically- and dynamically-typed units of class-based code. The type system uses row polymorphism for classes and thus supports mixin-based OO composition. To protect migration of mixins from typed to untyped components, the system employs a novel form of contracts that partially seal classes. The design comes with a theorem that guarantees the soundness of the type system even in the presence of untyped components.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10... $10.00

Figure 1 LDA topic modeling is based on the generative process illustrated here. It works over a fixed collection of topics assumed to be shared by all documents in the corpus (top right). First, we select a probabilty distribution over the topics (histogram). Then, to produce each word in the document, we first choose a topic from the distribution (the colored circles) and then we choose the actual word from the word distribution associated with the topic (the highlighted words).

LDA models this intuition that a document is a mixture of different topics. In the LDA framework, each "topic" is represented as a probability distribution over a fixed collection of words. The probability associated with each word indicates how strongly the word belongs to the topic. For example, words like 'type' and 'polymorphism' have high probabilities in the "Type System" topic, while 'component' and 'contract' have high probabilities in the "Components" topic. In contrast, words like 'cpu' and 'register' have low probabilities in both topics. A document is, in turn, modelled as a distribution over such topics.

In the paper in Figure 1, the probability distribution over topics would assign weights to the topics "Components," "Type Systems," and "OOP." The chart at the right of the figure shows this distribution, with "Type Systems" carrying slightly greater weight than either "Components" or "OOP." The words in the paper are drawn from the probability distributions associated with these topics. To put it another way, if we wanted to generate this document, or one similar to it, we would pick each word for the document independently according to the following *generative process*: (1) pick a topic (according to the topic distribution) and then (2) pick a word from that topic (according to its distribution over words). Other papers in the SIGPLAN corpus would exhibit a different mix of topics. A key point is that all the documents in an LDA framework share the same collection of topics, just mixed in different proportions (including nearly zero).

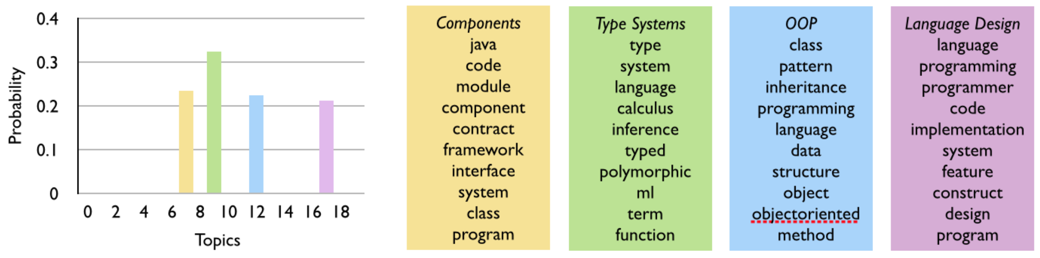


Figure 2 LDA inference for 20 topics run on a corpus of 4,355 titles and abstracts from SIGPLAN conferences and applied to the example paper "Gradual Typing." The probability distribution at the left shows the inferred topic mixture for the paper. The boxes at the right show the 10 most probable words for the four topics present in the paper.

Of course the purpose of topic modeling is not to generate random documents but rather to automatically discover the topics present in a corpus. We *observe* the documents, but the topic structure is *hidden*, including the set of topics, the topic distribution for each document, and the per-document, per-word topic assignment. The challenge of this area of research is to reconstruct the hidden topic structure from an observed corpus, essentially running the generative process backward, attempting to answer the question: What fully-defined topic model most likely explains the given corpus?

We ran David Blei's LDA-C[2] over a corpus of 4,355 titles and abstracts from ICFP, OOPSLA, PLDI, and POPL papers that are available in the ACM Digital Library, having set the number of topics at 20.[3] This process produced a topic model M, judged to be the most likely 20-topic model to explain the SIGPLAN corpus given a set of 20 "seed" papers, randomly selected by LDA. We then applied M to the "Gradual Typing" paper to impute the topic distribution that best explains the content of that paper. The results appear in Figure 2. This histogram on the left shows that although M could have assigned any probability distribution to the twenty topics, it selected only four. The right side of the figure shows the most likely words associated with those four topics. Looking at these words, we can see that three of the topics correspond to the ones we higlighted in Figure 1: Components, Type Systems, and OOP. LDA identified a fourth topic, whose words suggest it might be called "Language Design." Note that LDA does not produce the titles for the topics; we manually came up with names for them based on the most likely words (see Section 2.4). Automatically inferring such titles is an open research problem.

2.2 The corpora

We used the SIGPLAN research papers stored in the ACM Digital Library (DL)[4] to build two distinct corpora for this project. Our *abstract corpus* contains one "document" for every paper from every available year of ICFP (1996–2014), OOPSLA (1986–89,1991–2014), PLDI (1988–2014), and POPL (1973,1975–2015). Each such document is comprised of the concantenation of the paper title, its author list, and its abstract. The scraping process we used to build this corpus is imperfect – it doesn't find every paper listed in the DL (see

[2] `http://www.cs.princeton.edu/~blei/lda-c/`
[3] One of the limitations of LDA is that it will not help decide how many topics appear in a corpus. The user must decide in advance and seed the inference mechanism with the chosen number.
[4] `http://dl.acm.org/`; SIGPLAN conferences are available from `http://dl.acm.org/sig.cfm?id=SP946`

Section 2.6). Our *full-text corpus* contains one document for each of the 2,257 POPL and PLDI papers that have appeared as of February 2015. We used `pdftotext` from the Xpdf tool suite[5] to convert the `pdf` files from the ACM DL to raw text. For each document in both corpora, we separately store metadata, recording its author list along with when and where it was published.

Once we had compiled the raw data for the two corpora, we converted it into a form that LDA-C can read: a document file that represents each document as a "bag of words." Parsing forces us to decide which words are interesting and how to count those words. For example, the word 'we' appears many times in academic writing, but it isn't a meaningful topic word in the way that 'type' or 'flow' or 'register' are. So that we may focus on meaningful words, we drop all words on a *stopword* list from the document. We started with a standard stopword list for English,[6] amending it with two lists of words. First, we added words which appeared in more than 25% of the documents in our corpus. We also added words that felt irrelevant, such as 'easily', 'sophisticated', 'interesting', and 'via'.

Once we winnowed out the stopwords, we had to turn each text into a bag of words: a list pairing words with the number of times they occur. This process raises more questions requiring judgment: Are 'type' and 'types' different words? What about 'typing' and 'typical'? To answer such questions, we counted words as the same when they reduced to the same uninflected word via NLTK's[7] WordNet [19] lemmatizer. In this case, 'type' and 'types' both reduce to the uninflected word 'type', but 'typing' and 'typical' are left alone. Finally, we coalesced hyphenated words into single words, e.g., 'object-oriented' became 'objectoriented' and 'call-by-value' became 'callbyvalue'.

2.3 Our topic models

Topic models are parameterized by a number k, which represents the desired number of topics; at the moment, no machine learning techniques exist to automatically determine the appropriate number of topics. To explore this space, we used LDA-C to build topic models for both corpora for k equal to 20, 50, 100, and 200.

On the abstract corpus, a typical run of LDA-C with 20 topics takes about 2 minutes on a four-core 2.8Ghz Intel i7 with 16GB of RAM and a solid-state hard drive; simultaneous runs with 200, 300, and 400 topics take just shy of two hours. On the full-text corpus, generating the topic model for k=50 requires under an hour. It takes roughly 8 hours to generate the topic model for k=200.

Once LDA-C has processed our corpus, it produces files indicating how much each word belongs to a topic and how much each document belongs to each topic. We translate LDA-C's output to three files:

- a human-readable listing of the top 10 words and top 10 papers for each topic;
- a spreadsheet collating the metadata for each document (title, authors, year, and conference) with that document's topic weights; and
- a spreadsheet aggregating the total topic weights for each year of each conference.

The human-readable listing of top words and papers for each topic was a vital check on the sanity of LDA-C's output as we debugged our process; we also used it to assign names to the topics when we made graphs.

[5] `http://www.foolabs.com/xpdf/download.html`
[6] `https://pypi.python.org/pypi/stop-words/2014.5.26`
[7] `http://www.nltk.org/`

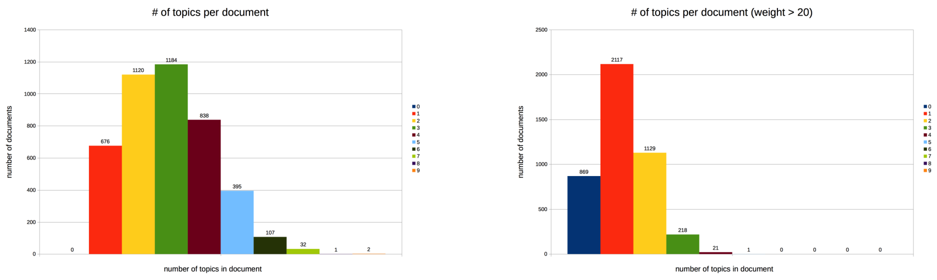


Figure 3 Number of topics per paper (left: normalized per topic, right: normalized per topic, minimum topic weight of 20).

Register Allocation	Program Transformation	Concurrency	Parsing
register (-3.6)	program (-4.0)	thread (-3.5)	grammar (-3.9)
instruction (-4.4)	loop (-4.5)	lock (-4.4)	attribute (-4.0)
code (-4.5)	variable (-4.6)	operation (-4.5)	string (-4.6)
variable (-4.7)	statement (-4.8)	memory (-4.6)	tree (-4.6)
graph (-4.9)	assertion (-4.9)	read (-4.6)	set (-4.6)
figure (-4.9)	value (-5.0)	execution (-4.7)	symbol (-4.7)
value (-5.0)	expression (-5.1)	program (-4.9)	language (-4.7)
loop (-5.0)	procedure (-5.2)	access (-5.0)	production (-4.7)
node (-5.0)	true (-5.3)	data (-5.0)	pattern (-4.8)
range (-5.0)	condition (-5.3)	concurrent (-5.0)	parser (-4.8)

Figure 4 Consensus topic names and top ten keywords (with log-likelihood for that word in that topic) for selected topics from the k=20 topic model on the full-text PLDI-POPL corpus.

2.4 Validating the topic models

Our topic models are sane: every paper is assigned *some* topic weights (Figure 3). On the left, we see that every document has non-minimal (0.03189) weight in *some* topic; on the right, we see that nearly one in five papers (869 out of 4,355) doesn't have a weight higher than 20 in any topic. Running with more topics reduces the number of indeterminate, topic-less documents. However, more topics can also lead to over-fitting and difficulty in distinguishing the underlying concepts. For comparison, Blei's analysis of the journal Science used 100 topics; Analysis of the Yale Law Review used 20 topics. We have experimented with up to 400 topics, but elected to present 20 for simplicity here.

Topic models do not provide names for the inferred topics, just lists of key words and the strength of the association between each paper and the topic. As another validity check, each of us independently named each topic for k=20 by looking at the keywords and highest ranking papers. Each of the lists were very similar; for most topics, it was a straightforward process to produce a consensus name for each topic. Figure 4 shows these consensus names and the associated top ten keywords for the k=20 topic model of the PLDI-POPL full-text corpus. We found topic 12 (given the title Program Transformation) from this list puzzling. At first glance, the keywords seem like a plausible if generic grouping, but the papers seem unrelated. A closer inspection of the top-ranked papers reveals that they all involve program transformations, particularly transformations related to numeric programs. This topic seems to be an example of a topic type that we had hypothesized but not previously seen: one where the papers are grouped by approach rather than the problem they are trying to solve.

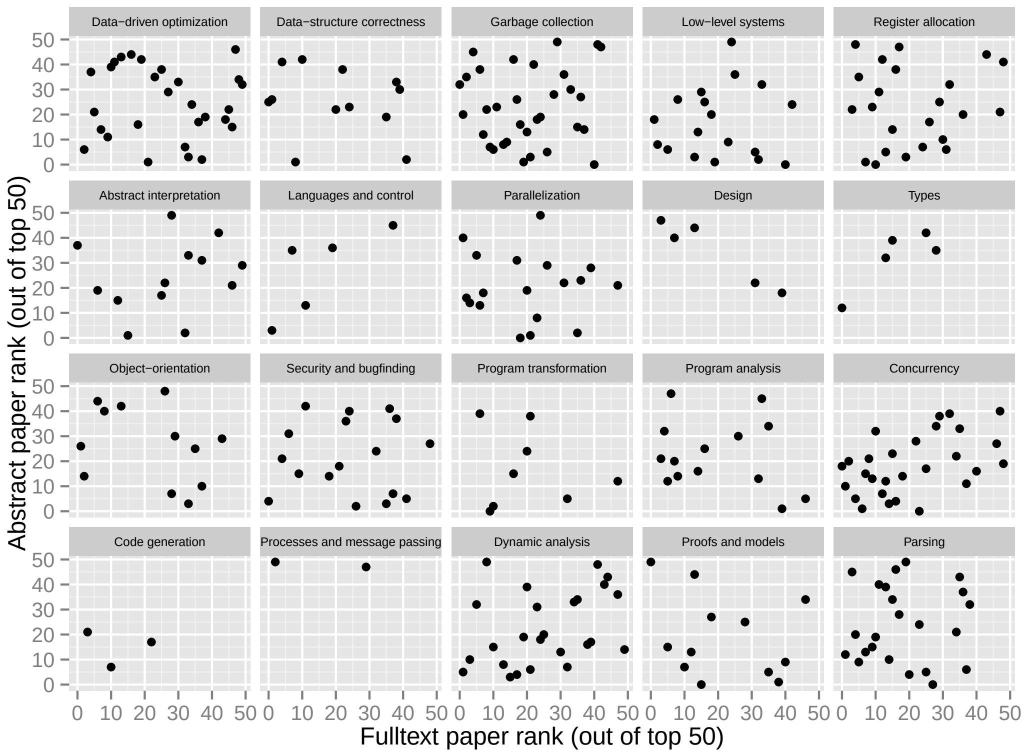


Figure 5 Common papers in the top 50 papers of LDA models built using abstracts and full text, by topic.

2.5 Abstract vs full-text models

What documents should we feed to LDA? We have two corpora at our disposal: the abstracts from ICFP, OOPSLA, PLDI and POPL (with some omissions – see Section 2.6); and the full text of PLDI and POPL. Once a topic model has been built, it can be used to classify documents of any size. So how should we build our models?

We ran LDA with k=20 on two corpora drawn from PLDI and POPL: one used abstracts, one used full text. In general, both models produced similar paper rankings in each topic (Figure 5). Each dot in Figure 5 represents a paper that appears in the top 50 papers in both models for a given topic. The x-axis is the paper's rank in the full-text model; the y-axis is the paper's rank in the abstract model. Prefect agreement would be a diagonal line of dots. Most topics show a fair to high agree of general agreement, though some show very little. Hearteningly, every topic shows *some* agreement.

We haven't yet answered the methodological question – perhaps existing research in machine learning can guide us. We use both models to explore the data in this paper.

2.6 Limitations and impediments

One limitation of our current work is that our corpora include only data from four programming language conferences. There are many other high-quality venues that publish relevant work, including ASPLOS, CC, CONCUR, ECOOP, ESOP, SAS, *etc.* We have omitted them only because adding more data sources is labor intensive. We felt that the data from the four conferences we have chosen was sufficient for an initial exploration.

Problems with data quality begin at the ACM. The ACM DL is missing data. The abstracts for ICFP, OOPSLA, PLDI, or POPL are missing for 1991. POPL is missing abstracts from 1996 through 1999. The ACM DL listings don't obviously differentiate hour-long invited keynote talks and the shorter talks in the sessions. We chose to include abstract-less keynotes as "documents" . . . but is that the right decision?

Scraping the DL for abstracts is not completely straightforward. Some papers are clearly in the HTML but our scraper doesn't detect them. Ideally, we would have access to the database underlying the DL, with programmatic access to metadata and the full text of each paper. Even then, extracting full text from PDFs isn't foolproof: math can show up as garbage characters that confuse LDA, or whole sections can fail to appear.

3 Results

While topic models can be used to analyze the programming languages literature in several different ways, we have focused primarily on two broad questions. The first question concerns how the topics present in the various programming languages conferences have evolved over time. The second involves the use of topic models to search for related work.

3.1 Characterizing the conferences

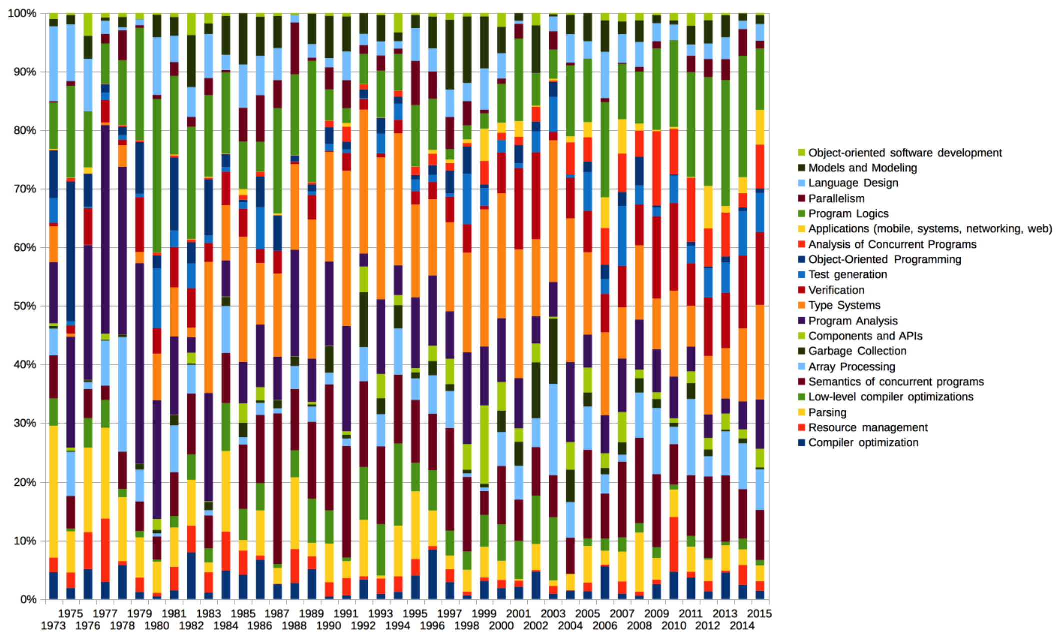
The prevailing wisdom is that each of the four major SIGPLAN conferences has its own character. Topic models highlight some of the distinctions between conferences as well as some of the similarities. They also provide insight into some of the major changes in research focus that have occurred over the last 40 years.

The four conferences, holistically

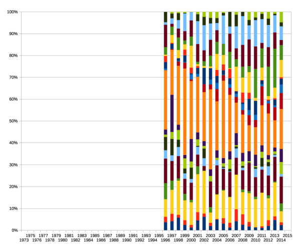
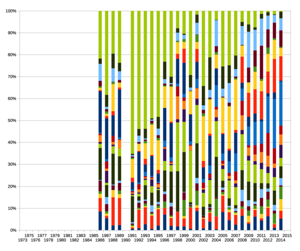
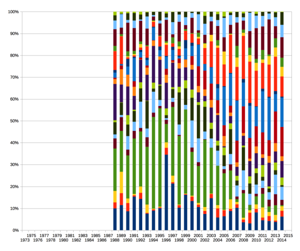
In our first experiment, we separated papers into collections according to conference (ICFP, OOPSLA, PLDI and POPL) and year (1973-2015). In each conference, in each year, we can apply a topic model to determine the topics and weight at which they appear in that year.

Figure 6 presents the results for our 20-topic model generated from the abstracts drawn from the 4 conferences. The top chart represents research presented at POPL between 1973 and 2015. Each column represents a year; the length of a colored line represents the weight of the corresponding topic in the given year. The three smaller charts present data for ICFP, OOPSLA and PLDI respectively. When viewing these charts online, we recommend zooming in to inspect the details.

Analysis of these charts confirms many broad intuitions about the topics that dominate certain conferences – the longer the line of a particular color, the more content from the corresponding topic appears in the conference. For instance, ICFP and POPL are both home to many papers on type systems, the bright orange topic that runs through the middle of the diagrams. OOPSLA and PLDI contain less content on these topics. PLDI is best known for its focus on classic compiler optimizations (the dark blue at the bottom of each chart) or low-level compiler optimization (the dark green 4th from the bottom). None of the other conferences contain such an emphasis on those topics. And as we will see in more detail later, there has been a dramatic reduction in the number of publications in the area, even at PLDI. OOPSLA, unsurprisingly, is best known for its papers on object-oriented software development, the light green dominating the top of the OOPSLA chart (and displayed in minute quantities at the other conferences). Interestingly, even at OOPSLA, we see a remarkable decrease in the prevalence of this topic beginning in the early 2000s through to 2015.



(POPL)

(ICFP) (OOPSLA) (PLDI)

Figure 6 Holistic, year-by-year topic model plots for four programming languages conferences. Topic model uses abstracts.

The four conferences, by topic

The previous charts present the big picture at each major conference and provide a means to compare the weights of different topics against one another. However, we also found it interesting to isolate the topics and to compare the conferences to each other on a topic-by-topic basis. Figure 7 presents a series of 20 graphs, one for each topic that we learned from the abstracts of the four conferences. The title we chose for each topic is presented at the top of each graph. The x-axis represents the year (ranging from 1973-2015) and the y-axis represents the weight of the topic. Each line represents a different conference: ICFP (red), OOPSLA (green), PLDI (turquoise) and POPL (pink). The grey shadow surrounding each line provides an estimate of the error involved in drawing the curve (the tighter the band, the better the estimate).[8]

[8] Each curve is generated using Loess smoothing over the set of points generated from that topic for each paper in the conference in question. We used the smoothing package from R's graphical library (stat_smooth(method=loess)). See `http://rgm3.lab.nig.ac.jp/RGM/R_rdfile?f=ggplot2/man/stat_smooth.Rd&d=R_CC` for details.

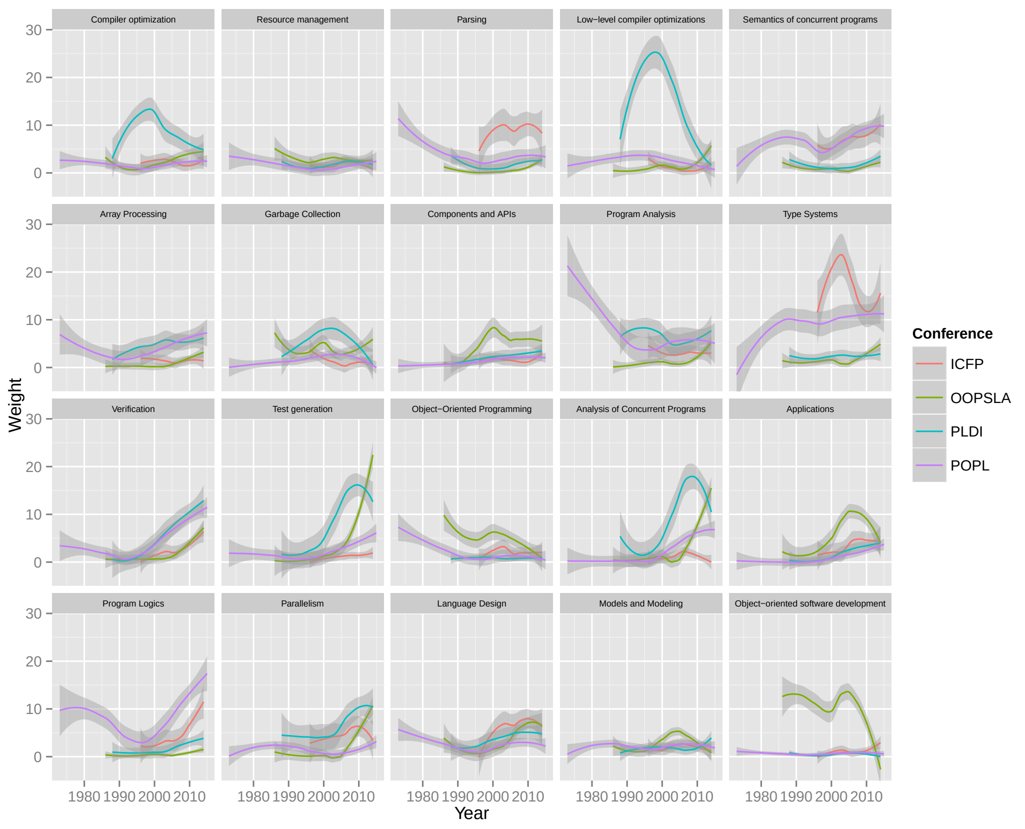


Figure 7 Topic-by-topic plots comparing four programming languages conferences. Topic model uses k=20 and abstracts.

Before beginning the experiment, one question we had was whether the change in the Call For Papers for OOPSLA in 2007 and the subsequent rebranding in 2010 of the umbrella event as SPLASH correlated with any changes in OOPSLA content. One of the goals of the Call for Papers changes was to broaden OOPSLA and deemphasive object-oriented programming. To consider this question, two topics, object-oriented programming (3rd row, 3rd column in Figure 7) and object-oriented software development (4th row, 5th column) are particularly relevant. Object-oriented programming pertains to analysis of object-oriented programming language features and the use of those features. It is characterized by keywords (in order of relevance) class (significantly more relevant than the others), pattern, inheritance, programming, language, data, structure, object, object-oriented, and method. Object-oriented software development pertains to software engineering methods and program design. It is characterized by the words design, software, object-oriented, development, programming, object, system, process, tool and project. The two most relevant documents are both panels: *The OO software development process*, a panel from OOPSLA 1992 [6] and *Agile management – an oxymoron?: Who needs managers anyway?*, a panel from OOPSLA 2003 [1]. Both panels have abstracts that were included in the OOPSLA proceedings in their respective years. When it comes to object-oriented programming, the charts show there was a steady decline in content at OOPSLA beginning in around the year 2000, when it was at a local maximum at the end of the dot-com boom. This decline persisted through the changes to OOPSLA's rebranding with little apparent dip. When it comes to object-oriented software development, there was a peak in the mid-2000s near the time of the rebranding, and a rather preciptious drop-off thereafter. One other topic that shows a similar trend is the topic on applications, which includes mobile, systems, web and network programming.

Since investigation of elements of object-oriented programming and software development have been on the decline at OOPSLA over the past decade or so, a natural question to ask is what topics have been taking their place? The answer includes topics such as test generation, analysis of concurrent programming, parallelism, and verification. Each of those topics show a significant spike at OOPSLA since the mid-2000s. It may well be that OOPSLA's rebranding encouraged more submissions along these lines.

Another trend that emerges from these graphs involves topics surrounding program optimizaton. The compiler optimization topic (row 1, column 1) and the low-level compiler optimization topic (row 1, column 4) both show spikes through the 90s and strong downward trend through the 2000s. OOPSLA has seen a slight uptick in low-level compiler optimization since roughly 2010, but across all four programming languages conferences, research in optimization appears down significantly.

PLDI vs POPL, by topic

PLDI and POPL are broadly perceived as the two most prestigious conferences in programming languages. At any given time, the topics they support heavily are bound to be at the center of programming languages research.

Figure 8 presents a series of charts similar to those we just studied in Figure 7. However, here we focus strictly on POPL and PLDI, and we generated a model using the full text of each paper in each conference.

To begin, there are many interesting similarities between the two conferences. For instance, on topics such as abstract interpretation, design, object orientation, program transformation, program analysis, code generation, and parsing, PLDI and POPL have followed very similar trends. Some of those trends are unsurprising: POPL had a substantial presence in parsing in the 70s, which has continuously decreased. PLDI followed suit shortly after its inauguration. Both PLDI and POPL had a larger presence in code generation that has dropped off. On the other hand, algorithmic and symbolic techniques for static program analysis (titled abstract interpretation) have steadily increased in both conferences since the mid-nineties. Here, one might speculate that the advent of practical SAT and SMT tools may have helped push this topic forward.

On the other hand, there are some substantial differences between the conferences. When it comes to types, PLDI has been relatively stable over the course of its existence. POPL's weight in this topic has steadily increased over the years. Another interesting trend involves language definition and control structures (row 2, column 2), which involves analysis of topics such as continuations, partial evaluation and lazy execution. At POPL, there was a huge spike during the 90s, which has since waned. At PLDI, these topics never really made much of an appearance. POPL also sees a lot of activity in highly theoretical topics involving proofs and semantic models for programs. On the other hand, at PLDI, topics such as parallelization and dynamic analysis are persistently higher than at POPL.

3.2 Finding your friends

Topic models offer a similarity measure, allowing us to compare documents or collections of documents to one another. When two documents (or sets of documents) are assigned topic vectors that are close in a geometric sense, they contain similar topic distributions. To study

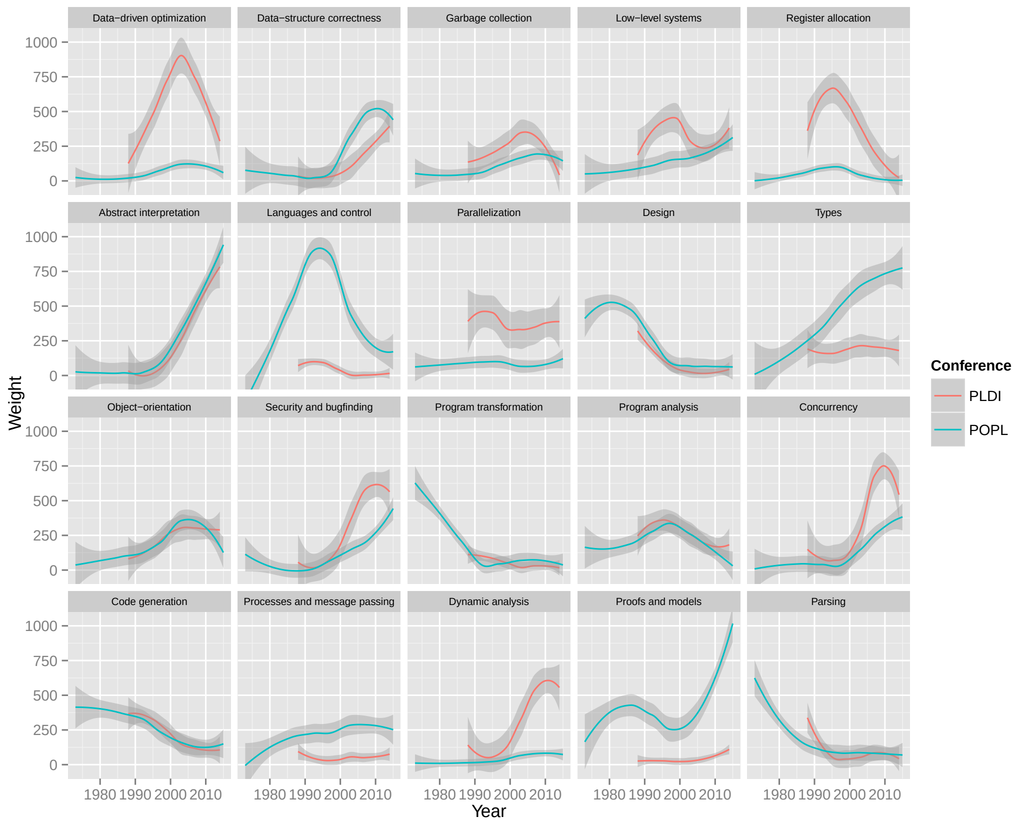


Figure 8 Topic-by-topic plots comparing PLDI and POPL. Topic model uses k=20 and full text.

the similarity measure generated by the topic models, we compared work cited by a paper to a random selection of POPL and PLDI papers. Our hypothesis was that models for related work would usually be closer to the model for a given paper than randomly selected papers would be.

To test our hypothesis, we selected four papers (one involving each of the authors, and one more) – *Concurrent Data Representation Synthesis* by Hawkins et al. [10] (CDRS), *Proof-Carrying Code* by George Necula [12] (PCC); *From System F to Typed Assembly Language* by Morrisett et al. [11] (TAL); and *Space-Efficient Manifest Contracts* by Greenberg [9] (SEMC). For each paper, we inferred a topic vector using LDA with k=20 and a corpus built on abstracts. We then inferred topic vectors for each paper's citations.[9] Figure 9 displays the results of our analysis. More specifically, for each paper (CDRS, PCC, SEMC, TAL), the left-most boxplot (in red) shows the distribution of Euclidean distance between each paper and its citations. In such boxplots, the third quartile, the top of the boxplot contains 75% of cited papers below it, while the bottom of the boxplot demarcates the 25% boundary. The height of the box is the interquartile range (IQR) and the vertical lines stretch 1.5*IQR

[9] We excluded one citation from Greenberg – his thesis. Since it's *much* longer than a conventional conference paper, the thesis is 'far' from the paper only because all of its topic weights are larger in an absolute sense, even though the topic vector is largely in a similar direction. From the other papers, we also excluded several cited books, theses or other documents that were either vastly longer than a conference paper or unobtainable.

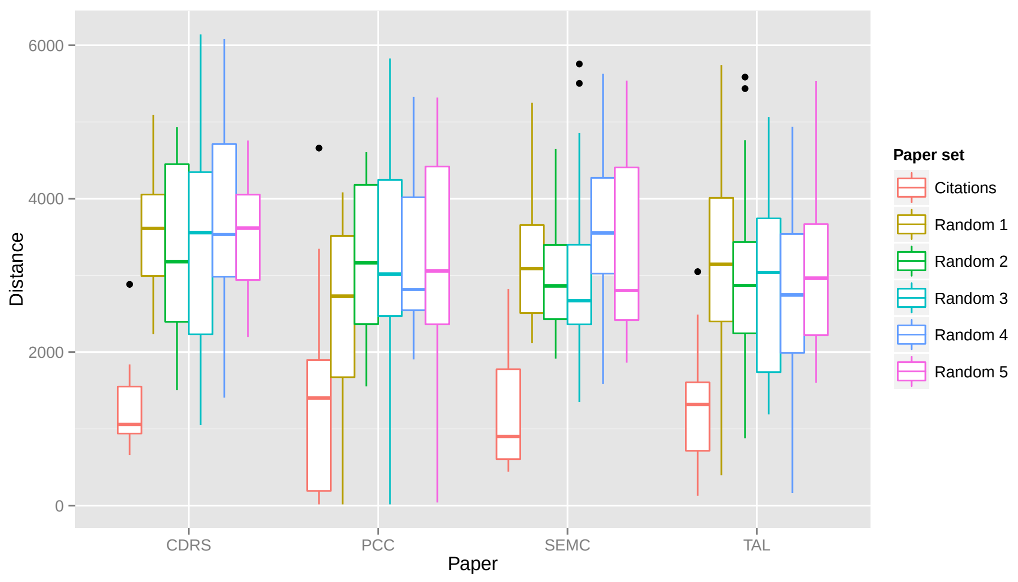


Figure 9 Are citations closer in topic space than random papers? Topic model uses abstracts.

from the top or the bottom of the box. Dots beyond those lines are outliers. The papers CDRS, SEMC, TAL have low medians and third quartiles, indicating that the bulk of their related is relatively close to the paper. In all these cases, 75% of cited papers are closer to the original paper than 75% of the random papers.

We included the PCC paper to illustrate that not all papers in our data set have particularly useful models. The PCC paper is one such paper. According to our model, it is relatively “topic-less” – its topic vector is quite short and close to the origin – so it’s hard to use its topic vector to classify related work. Indeed, its related work overlaps a little more with the randomly generated papers.

Having established that papers are relatively close to their related work in topic space, we can flip this idea on its head: given a paper, we can find papers in our corpus that are close to it – these may be related work! We’ve implemented a prototype tool for finding related work at `http://tmpl.weaselhat.com`. You can upload a PDF and we will identify the twenty most related papers from PLDI and POPL.

At present, our related work tool is simple: it runs a single model on a fixed corpus. With more data, we could offer more potential suggestions. With more computational power and engineering, we could use a suite of models (with different document seeds, number of topics, or data sources) to “boost” our related work search.

Even in this most primitive form, our tool has produced useful and surprising results. On his first test of the system, David Walker put in some of his collaborators’ recent work – Osera and Zdancewic’s *Type-and-Example-Directed Program Synthesis* [14] – and found a related paper by Nix entitled *Editing by example* (POPL 1984) that Osera and Zdancewic were unaware of [13]. Older papers like this one may be difficult to find via conventional search.[10] Michael Greenberg put in a recent networking workshop paper [8] and found some

[10] Google keyword searches including “program synthesis from examples” and “synthesizing string transformations from examples” do not display this result in the first three pages of links returned. More recent research floats to the top.

interesting work that solves a related problem in a different area [7]. We find these early anecdotal results encouraging. We hope that topic modeling can provide help with automated related work search driven by *documents* rather than *keywords*.

4 Future Work

One goal of this work is to build a website offering programming language researchers access to the tools and/or analysis results we find are most interesting or effective. Our prototype is up at `http://tmpl.weaselat.com/`. The source is freely available at `https://github.com/mgree/sigplan/`, allowing other communities to reuse our work.

Understanding the conferences

There are a number of ways we can paint a more detailed picture of our programming languages conferences. First, we can simply try running LDA with more topics. Second, we can investigate richer topic models. For instance, dynamic LDA [3] models the shift of topic content over time, and has been used for *lead/lag* analysis: When new topics appear, do they appear in one conference and spread to others? One very interesting study looked at the relationship between the topics funded and the reearch that appeared at conferences [15]. We could obtain the summaries of funded NSF proposals (and possibly other institutions) to determine if we can identify relationships between funded topics and conference content.

Understanding the researchers

We can also use topic models to learn topics for researchers, not just documents. There are many ways to do this – from collating all of a researcher's oeuvre to using time-aware techniques, like dynamic LDA [3]. Given a topic model for each researcher, it is possible to construct several additional tools. For instance, to help ensure a program committee has interests distributed similarly to the expected distribution of papers for a conference, we could check that the aggregated topic weights of the PC's publications are similar to the aggregated topic weights of the papers from the past year's submissions. To chose which papers to review, PC or ERC members can find papers with topics similar to their own – an idea that has already been explored in the Toronto Paper Matching System [5].

We may also be able to use models of PL researchers to answer questions about how researchers change over time. Do they stay in one topic, or work in many topics? Are there researchers who lead their conferences? Can we apply social network models to co-authorship graphs to understand how people share interest in topics? Can this help in assembling workshops or Dagstuhl-like seminars?

5 Related work

There is much work in the machine learning community aimed at understanding large collections of documents. David Blei's web page [2] on topic modeling contains pointers to several survey papers, talks and software packages. We were also inspired by the work of Charlin *et* al. [5], the architects of the Toronto Paper Matching System, which aims to match reviewers to papers using topic models. This prior work is all quite general; we aim to use topic-modeling software to study the flow of ideas across the programming languages literature specifically, something which has not be done before using similar methods.

There are various other ways one might analyze the programming languages literature. For instance, in a recent article for CACM [16], Singh *et. al.* studied research in hardware architecture between 1997 and 2011 on the basis of occurrences of keywords. Measuring keyword occurrences has the benefit of simplicity, but it does not identify collections of words – themes, as Blei would call them – that find themselves colocated in documents. These themes may help us characterize papers and researchers succinctly and may suggest interesting measures to compare them. Moreover, the LDA algorithm identifies these themes for us without anyone having to specify them.

TMVE [18] is a topic model visualization engine that offers a baseline expectation of how users might interact with a topic model. It produces a static website for a single topic model, while we envision more dynamic interactions on our website, like our prototype of related-work search.

6 Conclusion

Topic models offer the potential to give us a new lens on the programming languages literature. We already have some interesting results analyzing the content of our major conferences through this lens and we believe there are many more interesting questions to attempt to answer with these techniques.

Acknowledgments. This work is supported in part by the NSF under grant CNS-1111520. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1 Lougie Anderson, Glen B. Alleman, Kent Beck, Joe Blotner, Ward Cunningham, Mary Poppendieck, and Rebecca Wirfs-Brock. Agile management - an oxymoron?: Who needs managers anyway? In *OOPSLA*, pages 275–277, 2003.

2 David Blei. Topic modelling. `http://www.cs.princeton.edu/~blei/topicmodeling.html`, 2015.

3 David Blei and John Lafferty. Dynamic topic models. In *International Conference on Machine Learning*, 2006.

4 David M Blei. Probabilistic topic models. *Communications of the ACM*, 55(4):77–84, 2012.

5 Laurent Charlin and Richard Zemel. The Toronto Paper Matching System: An automated paper-reviewer assignment system. In *ICML Workshop on Peer Reviewing and Publishing Models*, June 2013. See also `http://papermatching.cs.toronto.edu/webapp/profileBrowser/about_us/`.

6 Dennis de Champeaux, Robert Balzer, Dave Bulman, Kathleen Culver-Lozo, Ivar Jacobson, and Stephen J. Mellor. The OO software development process (panel). In *OOPSLA*, pages 484–489, 1992.

7 Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. In *Programming Language Design and Implementation (PLDI)*, 2013.

8 Marco Gaboardi, Michael Greenberg, and David Walker. Type systems for SDN controllers, 2015. PLVNET.

9 Michael Greenberg. Space-efficient manifest contracts. In *Principles of Programming Languages (POPL)*, 2015.

10 Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *Programming Language Design and Implementation (PLDI)*, 2012.

11 Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Principles of Programming Languages (POPL)*, 1998.

12 George C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*, 1997.

13 Robert Nix. Editing by example. In *Principles of Programming Languages (POPL)*, 1984.

14 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Programming Language Design and Implementation (PLDI)*, 2015.

15 Xiaolin Shi, Ramesh Nallapati, Jure Leskovec, Dan McFarland, and Dan Jurafsky. Who leads whom: Topical lead-lag analysis across corpora. In *NIPS Workshop*, 2010.

16 Virender Singh, Alicia Perdigones, and Fernando R. Mazarrón José Luis Garcia, Ignacio Cañas-Guerroro. Analyzing worldwide research in hardware architecture, 1997-2011. *CACM*, 58(1):76–85, 2015.

17 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, pages 793–810, New York, NY, USA, 2012. ACM.

18 TMVE: Topic model visualization engine. `https://code.google.com/p/tmve/`, 2015.

19 Princeton University. Wordnet. `http://wordnet.princeton.edu`, 2010.

InterPoll: Crowd-Sourced Internet Polls

Benjamin Livshits and Todd Mytkowicz

Microsoft Research, US

Abstract

Crowd-sourcing is increasingly being used to provide answers to online polls and surveys. However, existing systems, while taking care of the mechanics of attracting crowd workers, poll building, and payment, provide little to help the survey-maker or pollster in obtaining statistically significant results devoid of even the obvious selection biases.

This paper proposes InterPoll, a platform for programming of crowd-sourced polls. Pollsters express polls as embedded LINQ queries and the runtime correctly reasons about uncertainty in those polls, only polling as many people as required to meet statistical guarantees. To optimize the cost of polls, InterPoll performs query optimization, as well as bias correction and power analysis. The goal of InterPoll is to provide a system that can be reliably used for research into marketing, social and political science questions.

This paper highlights some of the existing challenges and how InterPoll is designed to address most of them. In this paper we summarize some of the work we have already done and give an outline for future work.

1998 ACM Subject Classification D.2.6 Programming Environments

Keywords and phrases CrowdSourcing, Polling, LINQ

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.156

1 Introduction

Online surveys have emerged as a powerful force for assessing properties of the general population, ranging from marketing studies, to product development, to political polls, to customer satisfaction surveys, to medical questionnaires. Online polls are widely recognized as an affordable alternative to in-person surveys, telephone polls, or face-to-face interviews. Psychologists have argued that online surveys are far superior to the traditional approach of finding subjects which involves recruiting college students, leading to the famous quip about psychology being the study of the college sophomore [16].

Online surveys allow one to reach wider audience groups and to get people to answer questions that they may not be comfortable responding to in a face-to-face setting. While online survey tools such as Instant.ly, SurveyMonkey, Qualtrics, and Google Customer Surveys take care of the mechanics of online polling and make it easy to get *started*, the results they produce often create more questions than they provide answers [18, 25, 21, 39, 106, 48].

Surveys, both online and offline, suffer from *selection biases*, as well as non-response, and coverage issues. These biases are not trivial to correct for, yet without doing so, the data obtained from surveys may be less than representative, which complicates generalizing to a larger population. InterPoll allows the developer to both *estimate* and *correct* for the biases and errors inherent in the data they are collecting.

It is also not so obvious how many people to poll. Indeed, polling too few yields results that are not statistically significant; polling too many is a waste of money. None of the current survey platforms help the survey-maker with deciding on the appropriate number of samples. Today's online survey situation can perhaps be likened to playing slot machines

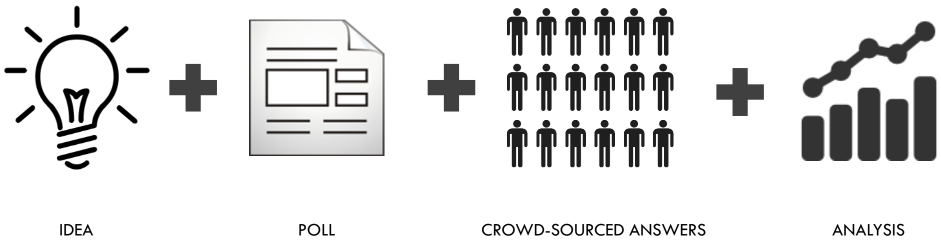
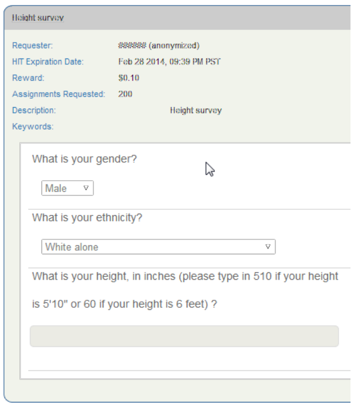


Figure 1 Conceptual architecture of InterPoll highlighting the major steps.

with today's survey sites playing the role of a casino; it is clearly in the interest of these survey sites to encourage more polls being completed.

In addition to the issue of data quality and representativeness, *cost* of the polls is an important consideration for poll makers, especially given that thousands of participants may be required. Even if answering a single question can costs cents, often getting a high level of assurance for targeted population segment involves hundreds of survey takers at significant cost. In fact, deciding on how to properly target the survey is a non-trivial task: if general audience surveys cost $.10 per question and *targeted* ones cost $.50 per question, is it better to ask five times as many questions of the general audience and then post-process the results or is it better to ask fewer questions of the targeted audience? Given that demographic targeting can often involve dozens of categories (males, 20–30, employed full-time, females, 50–60, employed part-time, females, 20–30, students, etc.) how does one properly balance the need for targeted answers and the cost of reaching these audiences?

We see these challenges as interesting *optimization problems*. To address some of these issues, InterPoll has an optimization engine whose goals is to determine (a sequence of) questions to ask and targeting restrictions to use. The primary goal of the optimization is to get a certain level of certainty in a developer-provided question (i.e. do men aged between 30–50 prefer *Purina Dog Chow* to *Precise Naturals Grain Free*), while minimizing the cost involved in running the poll on a large scale.


Height survey

Requester: ###### (anonymized)
HIT Expiration Date: Feb 28 2014, 09:39 PM PST
Reward: $0.10
Assignments Requested: 200
Description: Height survey
Keywords:

What is your gender?
Male

What is your ethnicity?
White alone

What is your height, in inches (please type in 510 if your height is 5'10" or 60 if your height is 6 feet) ?

Figure 2 Sample form produced by InterPoll for the Mechanical Turk back-end.

This Paper: This paper presents a high-level summary of InterPoll, a platform for in-application scripting of crowd-sourced polls, giving developers streamlined access to crowd-sourced poll data. The processing pipeline of InterPoll is shown in Figure 1. In this paper we briefly summarize the research on InterPoll we have already done and outline some of the avenues for future work.

InterPoll is an attempt to balance human and machine computation. One of the goals is to allow for easy integration with existing programs. As a result, we opted to use LINQ queries [65] already widely used by developers, instead of proposing a DSL. InterPoll is implemented as a library that can be linked into an existing application.

Note that due to LINQ's runtime introspection features, this allows us to effectively build an optimizing runtime for surveys within a library. InterPoll performs query optimization [59], as well as bias correction and power analysis [60], among other features, to enable a system that can be reliably used for research in marketing, social, and political sciences.

Motivating Examples

As mentioned above, one of the goal of InterPoll is to make running crowd-sourced polls easy for the developer. We accomplish this by using LINQ [65], language-integrated queries. LINQ is natively supported by .NET, with Java providing similar facilities with JQL.

▶ **Example 1** (Basic filtering). A simple poll may be performed the following way.

```
1 var people = new MTurkQueryable<Person>(true, 5, 100, 2);
2 var liberalArtsPairs = from person in people
3         where person.Employment == Employment.STUDENT
4         select new {
5                 Person = person,
6                 Value = person.PoseQuestion<bool>(
7                         "Are you a liberal arts major?")
8                 };
```

The first line gets a handle to a population of users, in this case obtained from Mechanical Turk, although other back-ends are also possible. Populations on which we operate have associated demographic information; for example, note that the `where` clause on line 3 ensures that we only query (college) students.

This poll will ask (college) students if they study liberal arts, producing an iterator of ⟨`Student`, `bool`⟩ pairs represented in .NET as `IEnumerable`.

▶ **Example 2** (Counting). Given `liberalArtsPairs`,

```
1 var libralArtMajors = from pair in liberalArtsPairs where pair.Value == true select person;
2 double percentage = 100.0 * libralArtMajors.Count() / liberalArtsPairs.Count();
```

it is possible to do a subsequent operation on the result, such as printing out all pairs or using, the `Count` operation to count the liberal arts majors. The last line computes the percentage of liberal art majors within the previously collected population.

▶ **Example 3** (Uncertainty). InterPoll explicitly supports computing with uncertain data, using a style of programming proposed in Bornholt *et al.* [10].

```
1 var liberalArtWomen = from person in people where person.Gender == Gender.FEMALE
2   where person.Employment == Employment.STUDENT select person.PoseQuestion<bool>("Are you a liberal arts major?");
3
4 var liberalArtMen = from person in people where person.Gender == Gender.MALE
5   where person.Employment == Employment.STUDENT select person.PoseQuestion<bool>("Are you a liberal arts major?");
6
7 var femaleVar = liberalArtWomen.ToRandomVariable(); var maleVar = liberalArtMen.ToRandomVariable();
8 if (femaleVar > maleVar) Console.WriteLine("More female liberal arts majors.");
9 else Console.WriteLine("More male liberal arts majors.");
```

Here, we convert the Boolean output of the posted question to a random variable (line 7). Then we proceed to compare these on line 8. Comparing two random variables (`femaleVar` and `maleVar`) results in a Bernoulli which the C# type system then implicitly casts (i.e., in the `>` comparison on line 8) into a boolean by running a t-test on the resulting Bernoulli[10].

▶ **Example 4** (Explicit t-tests). Here we explicitly perform the t-test at a specified confidence interval.

```
1 var test = liberalArtMen.ToRandomVariable() > liberalArtWomen.ToRandomVariable();
2 if (test.AtConfidence(.95)) { ... }
```

The test and the confidence interval determine the outcome of a power analysis that InterPoll will perform to decide how many (male and female) subjects to poll.

▶ **Example 5** (Optimizations). Suppose we are conducting a marketing study of dog owners' preference for Purina Puppy Chow. Specifically, we are trying decide if married women's attitude toward this product is more positive than that of married men.

```
var puppyChowWomen = from person in people where person.PoseQuestion<bool>("Are you a dog owner?")
          == true where person.Gender == Gender.FEMALE where person.Relationship == Relationship.MARRIED
   select person.PoseQuestion<bool>("Would you consider using Purina Puppy Chow?");
```

Similarly, for men:

```
var puppyChowMen = from person in people where person.PoseQuestion<bool>("Are you a dog owner?"))
          == true where person.Gender == Gender.MEN where person.Relationship == Relationship.MARRIED
   select person.PoseQuestion<bool>("Would you consider using Purina Puppy Chow?");
```

To compare these two, the following comparison may be used:

```
if (puppyChowWomen > puppyChowMen) Console.WriteLine("Women like puppy chow more");
```

In this case it is not so obvious how to sample from the population: a naïve strategy is to sample women first, then sample men. However, another strategy may be to sample everyone (who is `MARRIED`) and to separate them into two streams: one for women, the other for men. Lastly, sampling from the same population is likely to yield a disproportional number of samples in either population. For example, 64% of users of the uSamp platform are women [101] as opposed to 51%, as reported by the US 2012 Census. InterPoll's LINQ abstractions let a polster specify what to query and leaves the particular strategy for implementing a poll to InterPoll's optimizations.

Challenges

The examples described above raise a number of non-trivial challenges.

Query optimization: How should these queries be executed? How can the queries be optimized to avoid unnecessary work? Should doing so take the surrounding .NET code into which the queries are embedded into account? Should they be run independently or should there be a degree of reuse (or result caching) between the execution plans for the men and women? While a great deal of work on database optimizations exist, both for regular and crowd-sourced databases, much is not directly applicable to the InterPoll setting [14, 38, 69, 83, 9, 68], in that the primary goal of InterPoll optimizations is reducing the amount of money spent on a query.

Query planning: How should we run a given query on the crowd back-end? For instance, should pre-filter crowd workers or should we do post-filtering ourselves? Which option is cheaper? Which crowd back-end should we use, if they have different pricing policies? Should the filtering (by gender and relationship status) take place as part of population filtering done by the crowd provider?

Bias correction: Given that men and women do not participate in crowd-sourcing at the same rate (on some crowd-sourcing sites, one finds about 70% women and 30% men [78, 42, 71]), how do we correct for the inherent population bias to match the more equal gender distribution consistent with the US Census? Similarly, studies of CrowdFlower samples show a disproportionately high number of democrats vs. republicans [27]. Mobile crowd-sourcing attracts a higher percentage of younger participants [32].

Ignorable sample design: Ignorable designs assume that sample elements are missing from the sample when the mechanism that creates the missing data occurs at random, often referred to as missing at random or completely missing at random [77]. An example of non-ignorable design is asking what percentage of people know how to use a keyboard:

in a crowd sample that need a keyboard to fill out the survey, the answer is likely to be nearly 100%; in the population as a whole it is likely to be lower.

Power analysis: Today, the users of crowd-sourcing are forced to decide how many participants or workers to use for every task, yet there is often no solid basis for such a decision: too few workers will produce results of no statistical significance; too many will result in over-payment. How many samples (or workers) are required to achieve the desired level of statistical significance?

Crowd back-end selection: Given that different crowd back-ends may present different cost trade-offs (samples stratified by age or income may be quite costly, for example) and demographic characteristics, how do we pick an optimal crowd for running a given set of queries [67]? How do we compare query costs across the back-ends to make a globally optimal decision?

Quality control: What if the users are selecting answers at random? This is especially an issue if we ask about properties that eschew independent verification without direct contact with the workers, such as their height. A possible strategy is to insert attention-checking questions also called "catch trials" and the like [70, 46, 96].

Privacy of human subjects: Beyond the considerations of ethics review boards (IRBs) and HIPAA rules for health-related polls, there is a persistent concern about being able to de-anonymize users based on their partial demographic and other information.

Domains

InterPoll lowers the effort and expertise required for non-expert programmers to specify polls which benefits many non-experts in some of the following domains.

Social sciences: social sciences typically rely on data obtained via studies and producing such data is often difficult, time-consuming, and costly [25]. While not a panacea, online polls provide a number of distinct advantages [53].

Political polls: these are costly and require a fairly large sample size to be considered reliable. By their very nature, subjects from different geographic locales are often needed, which means that either interviewers need to cast a wide net (exit polling in a large number of districts) [88, 8] or they need to conduct a large remote survey (such as telephone surveys) [107, 30, 90].

Marketing polls: While much has been written about the upsides and downsides of online surveys [25, 77, 2, 22], the ability to get results cheaply, combined with the ease of targeting different population segments (i.e., married, high income, dog owners) makes the web a fertile ground for marketing polls. Indeed, these are among the primary uses of sites such as Instant.ly [101], Survey Monkey [41], and Google Customer Surveys [66].

Health surveys: A lot of researchers have explored the use of online surveys for collecting health data [93, 76, 26, 94, 5].

In all of the cases above, in addition to population biases, so-called *mode effects*, i.e. differences in results caused by asking questions online vs. on the telephone vs. in person are possible [12, 87, 109, 23, 32, 82, 80, 107, 30, 90, 47, 6].

2 Three Themes

In this section, we cover three themes we have explored in our research focusing on InterPoll so far. Section 2.1 talks about optimizing InterPoll queries. Section 2.2 discusses power analysis. Lastly, Section 2.3 gives an overview of unbiasing.

```
from structure in from person in employees where person.Wage > 4000 && person.Region == "WA"
     select new { Name = person.Name, Boss = person.GetBoss(), Sales = person.GetSales(42) }
where structure.Sales.Count > 5 select structure.Boss;
```

```
from person in employees where (person.Wage > 4000 && person.Region == "WA") && (person.GetSales(42).Count > 5)
 select person.GetBoss();
```

Figure 3 Query flattening.

2.1 Optimizations

We have applied two kinds of optimizations to InterPoll queries: static and runtime optimizations [59]. The former can be used to address poor programming practices used by developers or users who program InterPoll queries. Additionally, static optimizations are used to *normalize* InterPoll queries before runtime optimizations can be applied to them. In optimizing queries in InterPoll, we try to satisfy the following goals:

- Reduce **overall cost** for running a query; clearly for many people, reducing the cost of running survey is the most important "selling feature" when it comes to optimizations. Not only does it allow people with a low budget to start running crowd-sourced queries, it also allows survey makers to 1) request more samples and 2) run their surveys more frequently. Consider someone who may previously have been able to run surveys *weekly* now able to do so *daily*.
- Reduce the **end-to-end time** for running a query; we have observed that in many cases, making surveys requires *iterating* on how the survey is formulated. Clearly, reducing the running times allows the survey maker to iterate over their surveys to refine the questions much faster. Consider someone who needs to wait for week only to discover that they need to reformulate their questions and run them again.
- Reduce the **error rate** (or confidence interval) for the query results; while we support unbiasing the results in InterPoll, one of the drawbacks that is often cited as a downside of unbiasing is that the *error rate* goes up. This is only natural: if we have an unrepresentative sample which we are using for extrapolating the behavior for the overall population, the high error rate will capture the paucity of data we are basing our inference on.

LINQ provides an accessible mechanism to access the internal parts of a LINQ query via LINQ Expressions. Each LINQ query is translated into an expression AST, which can then be traversed and rewritten by LINQ Providers. InterPoll provides an appropriate set of visitors that rewrite LINQ query trees so as to both optimize them and also connect those query trees to the actual Mechanical Turk crowd. This latter "plumbing" is responsible for obtaining Mechanical Turk data in XML format and then at runtime parsing and validating it, and embedding the data it into type-safe runtime data structures.

2.1.1 Static Optimizations

Figure 3 gives an example of *flattening* LINQ expression trees and eliminating intermediate structures that may otherwise be created. In this example, `structure` is "inlined" into the query so that all operations are on the `person` value.

The other two optimizations we have implemented are *query splitting* which separates general and demographic questions that filter based on demographic characteristics into a where `clause` and *common subexpressions elimination* which identifies and merges common subexpressions in LINQ queries.

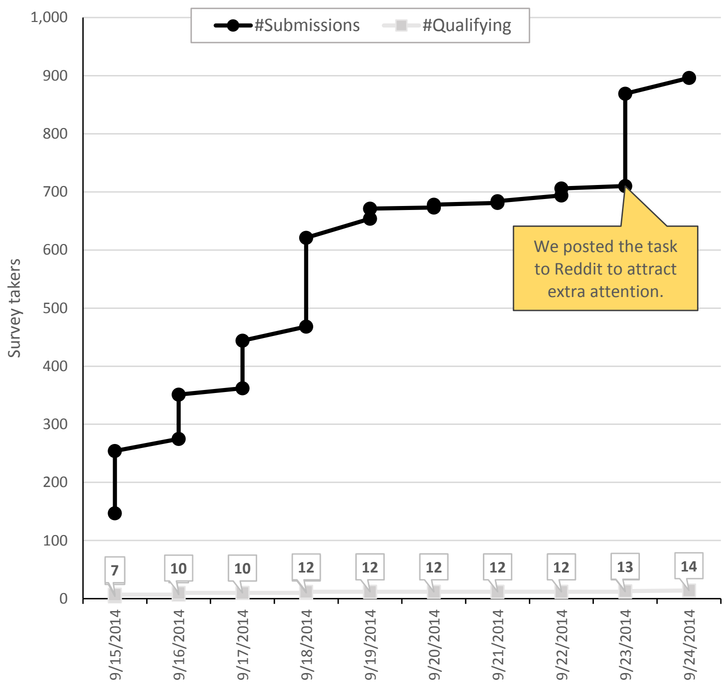


Figure 4 Completions for qualifications and passing for the budget query in Figure 5.

2.1.2 Dynamic Optimizations

However, runtime optimizations are generally more fruitful and, as mentioned above, can be implemented as a LINQ rewriting step that at runtime can change the evaluation strategy in response to statistics that are gathered at runtime.

Yield: The *yield optimization* addresses the problem of low-yield: this is when `where` clauses return only a very small percentage of the sampled participants, yet we have to pay for all of them. We have explored several strategies for evaluating such queries and evaluated the trade-offs between the cost of obtaining a certain number of responses and the completion time. Note that in practice, it is not uncommon to wait a week (or more) for certain uncommon demographics.

Figure 4 shows a long-running query where we ask for people's attitudes toward the US Federal budge deficit, with the `where` clause being:

```
person.Income==INCOME_35_000_TO_49_999 &&
person.Ethnicity==BLACK_OR_AFRICAN_AMERICAN
```

The query is shown in Figure 5.

The graph shows that despite about 900 people taking this poll over a full week only 14 meet the demographic constraints. By using *qualification tests*, an MECHANICAL TURK mechanism which requires a user successfully answer "qualification questions" (at low cost) before being able to complete the full survey (at full cost), INTERPOLL's optimized evaluation strategy can reduce the overall cost of this query by 8×.

Rebalancing: INTERPOLL supports answering decision questions of the form r_1 `boolOp` r_1, where both r_1 and r_2 are random variables obtained from segments of the population. To

```
var query = from person in people select new {
    Attitude = person.PoseQuestion(
        "How do you think the US Federal Government's yearly budget deficit has changed since January 2013?",
        "Increased a lot ", "Increased a little ", "Stayed about the same", "Decreased a little ", "Decreased a lot "),
    Gender = person.Gender, Income = person.Income, Ethnicity = person.Ethnicity ,
};
query = from person in query where person.Income == Income.INCOME_35_000_TO_49_999
                && person.Ethnicity == Ethnicity.BLACK_OR_AFRICAN_AMERICAN select person;
```

Figure 5 Budget deficit attitudes query.

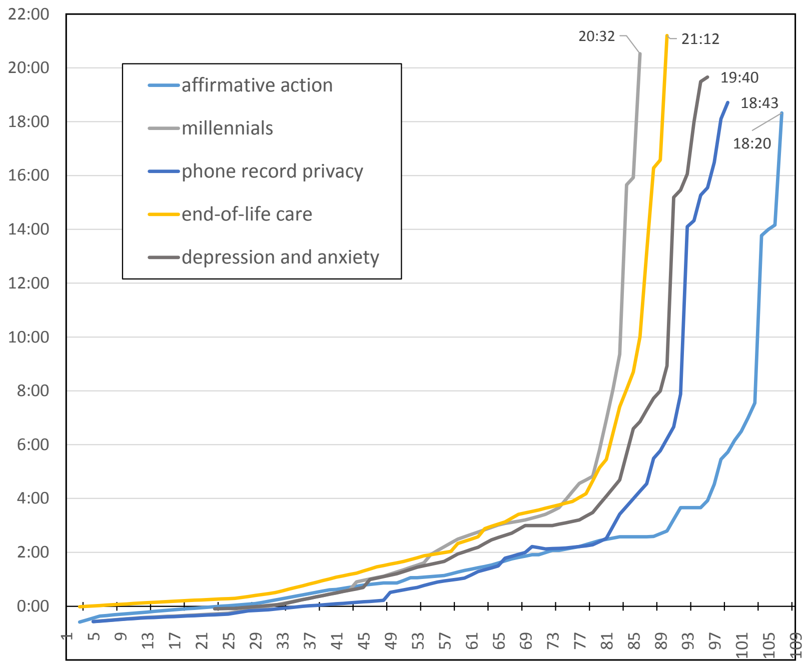


Figure 6 Time savings attained through the rebalancing optimization.

answer such *decision queries*, InterPoll repeatedly considers *pairs* of people from the categories on the left and right hand sides and then performs a sequential probability ratio test [60] to decide how many samples to request. A common problem, however, is that the two branches of the comparison are *unbalanced*: we are likely to have an unequal number of males and females in our samples, or people who are rich or poor, or people who own and do not own dogs.

The main mechanism for *rebalancing optimizations* is to reward the sub-population that is scarce more and the one that is plentiful less by adjusting the payoff of the Mechanical Turk task. Figure 6 shows the effect of increasing the reward by measuring the time difference to get to x completes between the default strategy and the fast branch of our rebalancing strategy. While the number of completes (people) is shown on the x axis, the times are measured in hours, shown on the y axis. Overall, the time savings achieved through rebalancing are significant: the fast branch gets to 70 completes over 2 hours faster and, for all strategies, to get to 90 completes, the it takes up to 21 more hours.

Panel building: The last optimization is motivated by the desire to avoid unbiasing by constructing *representative* population samples. Unbiasing based on unrepresentative samples frequently widens the confidence interval and is, therefore, less than desirable. Just like in real world polling, in InterPoll we have created ways to pre-build representative panels. An illustrative example is unbiasing height values (in cm) based on *ethnicity.* We limit our analysis to 50 respondents. Out of those, 35 were male and 15 female. Considering the female sub-population only, the mean height is *virtually the same* as the Wikipedia value of height in the US. Unbiasing the female heights with respect to ethnicity, however, produces 166.15 ± 5.32 cm. This is significantly larger than the true value and the difference emerges when we discover that our sample is reweighed using a weight of 50 for African Americans (12% in the population but only 1 in our sample). This taller woman (188 cm) has a large effect on the unbiased mean.

2.2 Power Analysis

We have implemented support for power analysis using an approach referred to as sequential acceptance sampling (SPRT) [102]. This approach to hypothesis testing requires drawing (pairs of) samples from Mechanical Turk until a convergence condition is met. The convergence condition is calculated based on the number of positive and negative responses. While the details of this process are discussed in a prior paper [60], below we show some queries and the number of samples required to resolve each. These queries are taken from ten real-life debates conducted via the Intelligence Squared site `http://intelligencesquaredus.org`.

Figure 7 shows a summary of our results for each of the debate polls. Alongside the outcome of each poll, we show the power analysis-computed power. We also show the dollar cost required to obtain the requisite number of samples from the crowd. `ObesityIsGovernmentBusiness` was the most costly debate of them all, requiring 265 workers, of which 120 (45%) were yes votes, whereas 145 (55%) said no.

Task	Outcome	Power	Cost
`MilennialsDontStandAChance`	No	37	$3.70
`MinimumWage`	No	43	$4.30
`RichAreTaxedEnough`	No	51	$5.10
`EndOfLife`	No	53	$5.30
`BreakUpTheBigBanks`	Yes	73	$7.30
`StrongDollar`	No	85	$8.50
`MarginalPower`	No	89	$8.90
`GeneticallyEngineeredBabies`	Yes	135	$13.50
`AffirmativeActionOnCampus`	Yes	243	$24.30
`ObesityIsGovernmentBusiness`	No	265	$26.50

Figure 7 Ten debates: outcomes, power analysis, costs.

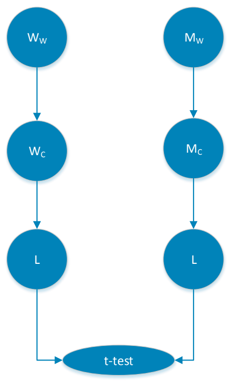
2.3 Unbiasing

During the 1936 U.S. presidential campaign, the popular magazine Literary Digest conducted a mail-in election poll that attracted over two million responses, a huge sample even by today's standards. Literary Digest notoriously and erroneously predicted a landslide victory for Republican candidate Alf Landon. In reality, the incumbent Franklin D. Roosevelt decisively won the election, with a whopping 98.5% of the electoral vote, carrying every state except for Maine and Vermont. So what went wrong? As has since been pointed out, the magazine's forecast was based on a highly non-representative sample of the electorate – mostly car and telephone owners, as well as the magazine's own subscribers – which underrepresented Roosevelt's core constituencies. By contrast, pioneering pollsters, including George Gallup, Archibald Crossley, and Elmo Roper, used considerably smaller but representative samples to predict the election outcome with reasonable accuracy. This triumph of brains over brawn effectively marked the end of convenience sampling, and ushered in the age of modern election polling.

It is broadly acknowledged that while crowd-sourcing platforms present a number of exciting new benefits, conclusions that may result from crowd-sourced experiments need to be treated with care [7, 6]. External validity is an assessment of whether the causal estimates deduced from experimental research would persist in other settings and with other samples. For instance, concerns about the external validity of research conducted using student samples (the so-called college sophomore problem) have been debated extensively [16].

The composition of population samples found on crowd-sourcing sites such as MECHANICAL TURK generally differs markedly from the overall population, leading some researchers to question the overall value of online surveys [6, 35, 25, 53, 78, 42, 71, 43, 11, 3, 52, 81, 10]. Wauthier *et al.* [103] advocate a bias correction approach for crowd-sourced data that we generally follow.

▶ **Example 6** (Unbiasing)**.** Consider deciding if there are more female liberal art majors than than the there are male ones. The ultimate comparison will be performed via a t-test. However, the first task is to determine the expected value of female and male liberal art majors given that we drew S samples from the crowd. These values can be computed as shown below: $E[L_W|C] = Pr[L|W_C] \times Pr[W_C|W_W] \times S$ and $E[L_M|C] = Pr[L|M_C] \times Pr[M_C|M_W] \times S$ where L_W and L_M are the number of female and male liberal art major, respectively, W_C and M_C stand for a woman/man being in the crowd, and W_W and M_W stand for a woman/man being the world as reflected by a broad population survey such as the US census; the latter two probabilities may be related at 51 : 49, for example.



Note that our goal is to discern the expected value of liberal art majors per gender *in the world*. We can unbias our data by using the probability of observing a woman in the crowd given there is a woman in the world: $E[W_L|W] = E[W_L|C] \times P(W_C|W_W)$ and similarly for men $E[M_L|M] = E[M_L|C] \times P(M_C|M_W)$.
While $E[W_L|C]$ and $E[M_L|C]$ can be approximated by observing the crowd-sourced results for the female and male sub-segments of the population, coefficients such as $P(W_C|W_W)$ can be computed from our knowledge of crowd population vs. that in the world in general. For example, if women to men are at 50%:50% in the world and at 30%:70% in the crowd, $P(W_C|W_W) = .7$ and $P(M_C|M_W) = .3$.

Note that the above example presents a simple model that does not, for example, explicitly represent the factor of ignorability [33], pg. 202 of our experimental design. Also note that unbiasing generally may need to be done before we perform a t-test to reshape the underlying distributions.

3 Related Work

There are several bodies of related work from fields that are usually not considered to be particularly related, as outlined below.

3.1 Crowd-Sourcing Systems

There has been a great deal of interest in recent years in building new systems for automating crowd-sourcing tasks.

Toolkits: TurKit [58] is one of the first attempts to automate programming crowd-sourced systems. Much of the focus of TurkIt is the iterative paradigm, where solutions to crowd-sourced tasks are refined and improved by multiple workers sequentially. The developer can write TurkIt scripts using JavaScript. AutoMan [4] is a programmability approach to combining crowd-based and regular programming tasks, a goal shared with Truong *et al.* [98]. The focus of AutoMan is on computation reliability, consistency and accuracy of obtained results, as well as task scheduling. Turkomatic [57, 56] is a system for expression crowd-sourced tasks and designing workflows. CrowdForge is a general purpose framework for accomplishing complex and interdependent tasks using micro-task markets [54]. Some of the tasks involve article writing, decision making, and science journalism, which demonstrates the benefits and limitations of the chosen approach. More recently, oDesk has emerged as a popular marketplace for skilled labor. CrowdWeaver is a system to visually manage complex crowd work [51]. The system supports the creation and reuse of crowd-sourcing and computational tasks into integrated task flows, manages the flow of data between tasks, etc.

Wiki surveys [79] is a novel approach of combining surveys and free-form interviews to come up to answers to tough questions. These answers emerge as a result of pair-wise comparisons of individual ideas volunteered by participants. As an example, participants in the wiki survey were presented with a pair of ideas (e.g., “Open schoolyards across the city as public playgrounds” and “Increase targeted tree plantings in neighborhoods with high asthma rates”), and asked to choose between them, with subsequent data analysis employed to estimate “public opinion” based on a large number of pair-wise outcomes.

We do not aim to adequately survey the vast quantity of crowd-sourcing-related research out there; the interested reader may consult [108]. Notably, a great deal of work has focused on matching users with tasks, quality control, decreasing the task latency, etc.

Moreover, we should note that our focus is on *opinion polls* which distinguishes InterPoll work from the majority of crowd-sourcing research which generally requires giving solutions to a particular task, such as deciphering a license plate number in a picture, translating sentences, etc. In InterPoll, we are primarily interested in self-reported opinions of users about themselves, their preferences, and the world at large.

Some important verticals: Some crowd-sourcing systems choose to focus on specific verticals. The majority of literature focuses on the following four verticals:

- social sciences [27, 5, 3, 53, 13, 11, 16, 35, 74];
- political science and election polls [90, 6, 7, 87, 5, 47, 107];
- marketing [41, 101, 25]; and
- health and well-being [93, 94, 26, 76, 106, 5, 7, 2, 81, 19].

3.2 Optimizing Crowd Queries

CrowdDB [29] uses human input via crowd-sourcing to process queries that regular database systems cannot adequately answer. For example, when information for `IBM` is missing in the underlying database, crowd workers can quickly look it up and return as part of query results, as requested. CrowdDB uses SQL both as a language for posing complex queries and as a way to model data. While CrowdDB leverages many aspects of traditional database systems, there are also important differences. CrowdDB extends a traditional query engine with a small number of operators that solicit human input by generating and submitting work requests to a microtask crowd-sourcing platform. It allows any column and any table to be marked with the `CROWD` keyword. From an implementation perspective, human-oriented query operators are needed to solicit, integrate and cleanse crowd-sourced data. Supported crowd operators include `probe`, `join`, and `compare`.

Marcus *et al.* [61, 62, 63] have published a series of papers outlining a vision for Qurk, a crowd-based query system for managing crowd workflows. Some of the motivating examples [61] include identifying people in photographs, data discovery and cleansing (who is the CEO of a particular company?), sentiment identification in Twitter messages, etc.

Qurk implements a number of optimizations [63], including task batching, replacing pairwise comparisons with numerical ratings, and pre-filtering tables before joining them, which dramatically reduces the overall cost of sorts and joins on the crowd. End-to-end experiments show cost reductions of 14.5x on tasks that involve matching up photographs and ordering geometric pictures. These optimization gains in part inspire our focus on cost-oriented optimizations in InterPoll.

Marcus *et al.* [62] study how to estimate the *selectivity* of a predicate with help from the crowd, such as filters photos of people to those of males with red hair. Crowd workers are shown pictures of people and provide either the gender or hair color they see. Suppose we could estimate that red hair is prevalent in only 2% of the photos, and that males constitute 50% of the photos. We could order the tasks to ask about red hair first and perform fewer HITs overall. Whereas traditional selectivity estimation saves database users time, optimizing operator ordering can save users money by reducing the number of HITs. We consider these estimation techniques very much applicable to the setting of InterPoll, especially when it comes to free-form `PoseQuestion`, where we have no priors informing us of the selectivity factor of such a filter. We also envision of a more dynamic way to unfold questions in an order optimized for cost reduction.

Kittur *et al.* [51] present a system called CrowdWeaver, designed for visually creating crowd workflows. CrowdWeaver system supports the creation and reuse of crowd-sourcing and computational tasks into integrated task flows, manages the flow of data between tasks, and allows tracking and notification of task progress. While our focus in InterPoll is on embedding polls into general-purpose programming languages such as C#, InterPoll could definitely benefit from a visual task builder approach, so we consider CrowdWeaver complimentary.

Somewhat further afield, Gordon *et al.* [34] describe a language for probabilistic programming and give an overview of related work. Nilesh *et al.* [20] talk about *probabilistic databases* designed to work with imprecise data such as measured GPS coordinates, and the like.

3.3 Database and LINQ Optimizations

While language-integrated queries are wonderful for bringing the power of data access to ordinary developers, LINQ queries frequently do not result in most efficient executions. There has also been interest in both formalizing the semantics of [14] and optimizing LINQ queries.

Grust *et al.* propose a technique for alternative efficient LINQ-to-SQL:1999 compilation [38]. Steno [68] proposes a strategy for removing some of the inefficiency in built-in LINQ compilation and eliminates it by fusing queries and iterators together and directly compiling LINQ queries to .NET code.

Nerella *et al.* [69] relies on programmer-provided annotations to devise better queries plans for language-integrated queries in JQL, Java Query Language. Annotations can provide information about shapes of distribution for continuous data, for example. Schueller *et al.* [83] focus on bringing the idea of *update propagation* to LINQ queries and combining it with reactive programming. Tawalare *et al.* [95] explore another compile-time optimization approach for JQL.

Bleja *et al.* [9] propose a new static optimization method for object-oriented queries dealing with a special class of sub-queries of a given query called "weakly dependent sub-queries." The dependency is considered in the context of SBQL non-algebraic query operators like *selection* and *projection*. This research follows the stack-based approach to query languages.

3.4 Web-Based Polls and Surveys

Since the time the web has become commonplace for large segments of the population, we have seen an explosion of interest in using it as a means for conducting surveys. Below we highlight several papers in the growing literature on this subject [2, 5, 17, 18, 21, 22, 25, 28, 44, 31, 35, 36, 37, 39, 45, 49, 50, 52, 53, 64, 81, 84, 89, 97, 106, 2, 5, 17, 18, 21, 22, 25, 28, 44, 31, 35, 36, 37, 39, 45, 49, 50, 52, 53, 64, 81, 84, 89, 97, 106].

Online Demographics: Recent studies reveal much about the demographics of crowd-sourcing sites such as Amazon's Mechanical Turk [6, 35, 25, 53, 104, 23, 78, 42, 71, 43, 11, 3, 52, 81, 74]. Berinsky *et al.* [6] investigate the characteristics of samples drawn from the Mechanical Turk population and show that respondents recruited in this manner are often *more* representative of the U.S. population than in-person convenience samples – the modal sample in published experimental political science – but *less* representative than subjects in Internet-based panels or national probability samples. They succeeded in replicating three experiments, the first one of which focuses on welfare spendings or assistance to the poor. They compared Mechanical Turk results with those obtained via the General Social Surveys (GSS), a nationally-representative face-to-face interview sample. While subtle differences exist, the overall results were quite similar between the GSS and Mechanical Turk (37% vs 38%). The second experiment involves replicating the so-called *Asian disease* experiment, which involves asking respondents to choose between two policy options. The results were comparable to those obtained in the original experiment by Tversky and Kahneman [99] on a student sample. The last experiment is described in Kam *et al.* [85] and involves measuring the preference for a risky policy option over a certain policy option. Additionally, Berinsky *et al.* discuss the internal and external validity threats. These three experiments provide a diverse set of studies to reproduce using InterPoll.

Ipeirotis [43, 42] focuses his analysis on the *demographics* of the Mechanical Turk marketplace. Overall, they find that approximately 50% of the workers come from the United States and 40% come from India. Significantly more workers from India participate on Mechanical Turk because the online marketplace is a primary source of income, while in the US most workers consider Mechanical Turk a secondary source of income. While money is a primary motivating reason for workers to participate in the marketplace, workers also cite a variety of other motivating reasons, including entertainment and education. Along with other studies, Ipeirotis provides demographic comparisons for common categories such as gender, age, education level, household income, and marital status for both countries. Ipeirotis [42] digs deeper into worker motivation, cost vs. the number of workers interested, time of completion vs. reward, etc. We believe that this data can be useful to give more fine-grained cost predictions for InterPoll queries and producing more sophisticated query plans involving tasks priced at various levels, for example. Additionally, while our initial focus is on query cost, we should be able to model completion rates fairly precisely as well. Of course, demographic data is also important for unbiasing query results.

Paolacci *et al.* [71] compare different recruiting methods (lab, traditional web study, web study with a specialized web site, Mechanical Turk) and discuss the various threats to validity. They also present comparisons of Mechanical Turk samples with those found through subject recruitment at a Midwestern university and through several online discussion boards that host online experiments in psychology, revealing drastic differences in terms of the gender breakdown, average age, and subjective numeracy. The percentage of failed catch trials varied as well, but not drastically; Mechanical Turk workers were quite motivated to complete the surveys, compared to those found though online discussion boards. While data quality does not seem to be adversely affected by the task payoff, researcher reputation might suffer as a result of poor worker perception and careless researchers "black-listed" on sites such as `http://turkopticon.differenceengines.com`.

Ross *et al.* [78] describe how the worker population has changed over time, shifting from a primarily moderate-income, U.S.-based workforce toward an increasingly international group, with a significant population of young, well-educated Indian workers. This change in population points to how workers may treat Turking as a full-time job, which they rely on to make ends meet. The paper contains comparisons across nationality, gender, age, and income, pinpointing a trend toward a growing number of young, male, Indian Turkers. Interesting opportunities exist for cost optimizations in InterPoll if we determine that different worker markets can provide comparable results (for a given query), yet are priced differently.

Buhrmester *et al.* [11] report that demographic characteristics suggest that Mechanical Turk participants are at least as diverse and more representative of non-college populations than those of typical Internet and traditional samples. Most importantly, they found that the quality of data provided by Mechanical Turk met or exceeded the psychometric standards associated with published research.

Andreson *et al.* [1] report that Craigslist can be useful in recruiting women and low-income and young populations, which are often underrepresented in surveys, and in recruiting a racially representative sample. This may be of particular interest in addressing recruitment issues in health research and for recruiting non-WEIRD (Western, Educated, Industrialized, Rich, Democrat) research subjects [40].

Online vs. Offline: Several researchers have studied the advantages and disadvantages of web-based vs. telephone or other traditional survey methodologies [2, 23, 30, 86, 90, 105, 107], with Dillman [21] providing a book-length overview. Sinclair *et al.* [86] focus on epidemiological research, which frequently requires collecting data from a representative sample of the community, or recruiting members of specific groups through broad community approaches. They look at response rates for mail and telephone surveys, but web surveys they consider involve direct mailing of postcards and inviting recipients to fill out an online survey and as such do not provide compelling incentives compared to crowd-sourced studies. Fricker [30] compare telephone and Web versions of a questionnaire that assessed attitudes toward science and knowledge of basic scientific facts. However, again, the setting differs significantly from that of InterPoll, in that crowd workers have a direct incentive to participate and complete the surveys.

Duffy [23] give a comparison of online and face-to-face surveys. Issues studies include interviewer effect and social desirability bias in face-to-face methodologies; the mode effects of online and face-to-face survey methodologies, including how response scales are used; and differences in the profile of online panelists, both demographic and attitudinal. Interestingly, Duffy *et al.* report questions pertaining to technology use should not be asked online, as they result in much higher use numbers (i.e., *PC use at home* is 91% in the online sample vs. 53 in the face-to-face sample). Surprisingly, these differences pertain even to technologies such as DVD players and digital TV. They also conclude that online participants are frequently better informed about issues such as cholesterol, and are likely to quickly search for an answer, which compromises the ability to ask knowledge-based questions, especially in a crowd setting. Another conclusion is that for online populations, propensity score weighting has a significant effect, especially for politically-oriented questions.

Stephenson *et al.* [90] study the validity of using online surveys vs. telephone polls by examining the differences and similarities between parallel Internet and telephone surveys conducted in Quebec after the provincial election in 2007. Both samples have demographic characteristics differing slightly, even after re-weighting, from that of the overall population. Their results indicate that the responses obtained in each mode differ somewhat, but that few inferential differences would occur depending on which dataset were used, highlighting the attractiveness of online surveys, given their generally lower cost.

Biases: Biases in online crowds, compared to online populations, general populations as well as population samples obtained via different recruitment techniques have attracted a lot of attention [3, 47, 73, 74, 76, 75, 80], but most conclusions have been positive. In particular, crowds often provide more diversity of participants, on top of higher completion rates and frequently quality of work.

Antin *et al.* [3] study the *social desirability bias* on MECHANICAL TURK. They use a survey technique called *the list experiment* which helps to mitigate the effect of social desirability on survey self-reports. Social desirability bias refers to "the tendency of people to deny socially undesirable traits or qualities and to admit to socially desirable ones" [15]. Among US Turkers, they conclude that social desirability encourages over-reporting of each of four motivating factors examined; the over-reporting was particularly large in the case of money as a motivator. In contrast, among Turkers in India we find a more complex pattern of social desirability effects, with workers under-reporting "killing time" and "fun" as motivations, and drastically over-reporting "sense of purpose."

Survey sites: In the last several years, we have seen surveys sites that are crowd-backed. The key distinction between these sites and INTERPOLL is our focus on optimizations and statistically significant results at the lowest cost. In contrast, survey sites generally are incentivized to encourage the survey-maker to solicit as many participants as possible . At the same time, we draw inspiration from many useful features that the sites described below provide.

Most survey cites give easy access to non-probability samples of the Internet population, generally without attempting to correct for the inherent population bias. Moreover, while Internet use in the United States is approaching 85% of adults, users tend to be younger, more educated, and have higher incomes [72]. Unlike other tools we have found, Google Customer Surveys support re-weighting the survey results to match the demographics of the Current Population Survey (CPS) [100].

SurveyMonkey claims to be the most popular survey building platform [41]. In recent years, they have added support for data analytics as well as an on-demand crowd. Market research seems to be the niche they are trying to target [92]. SurveyMonkey performs ongoing monitoring of audience quality through comparing the answers they get from their audience to that obtained via daily Gullop telephone polls [91]. They conclude that the SurveyMonkey Audience 3-day adjusted average, for 5 consecutive days is within a 5% error margin of Gallup's 14-day trailing average. In other words, when corrected for a higher average income of SurveyMonkey respondents in comparison to the US census data, SurveyMonkey is able to produce effectively the same results as Gallup, with only 3-days of data instead of 14 for Gallup.

Instant.ly and uSamp [101] focus primarily on marketing studies and boast an on-demand crowd with very fast turn-around times: some survey are completed in minutes. In addition to rich demographic data, uSamp collects information on the industry in which respondents are employed, their mobile phone type, job title, etc., also allowing one to filter and aggregate using these demographic characteristics.

Unlike other sites, Google Surveys results have been studied in academic literature. McDonald *et al.* [66] compares the responses of a probability based Internet panel, a non-probability based Internet panel, and Google Consumer Surveys against several media consumption and health benchmarks, leading the authors to conclude that despite differences in survey methodology, Consumer Surveys can be used in place of more traditional Internet-based panels without sacrificing accuracy.

Keeter *et al.* [48] present a comparison of results performed at Pew to those obtained via Google Customer Surveys. Note that demographic characteristics for survey-takes appear to be taken from DoubleClick cookies and are generally inferred and not verified (an approach taken by Instant.ly). A clear advantage of this approach is asking fewer questions; however, there are obvious disadvantages.

Apparently, for about 40% of survey-takes, reliable demographic information cannot be determined. The Google Consumer Survey method samples Internet users by selecting visitors to publisher websites that have agreed to allow Google to administer one or two questions to their users. As of 2012, there are about 80 sites in the *Google Surveys publisher network* (and 33 more currently in testing). The selection of surveys for eligible visitors of these sites appears random. Details on the Google Surveys "survey wall" appear scarce [24].

The Pew study attempted to validate the inferred demographic characteristic and concluded that for 75% of respondents, the inferred gender matched their survey response. For age inference, the results were mixed, with about 44% confirming the automatically inferred age range. Given that the demographic characteristics are used to create a stratified sample, and to re-weight the survey results, these differences may lead to significant errors; for instance, fewer older people using Google Consumer Surveys approved of Obama's job performance than in the Pew Research survey. The approach taken in InterPoll is to *ask* the user to provide their demographic characteristics; we would immensely benefit from additional support on the back-end level to obtain or verify the user-provided data. Google Customer Surveys have been used for information political surveys [55].

The Pew report concludes that, demographically, the Google Consumer Surveys sample appears to conform closely to the demographic composition of the overall internet population. From May to October 2012, the Pew Research Center compared results for 48 questions asked in dual frame telephone surveys to those obtained using Google Consumer Surveys. Questions across a variety of subject areas were tested, including: demographic characteristics, technology use, political attitudes and behavior, domestic and foreign policy and civic engagement. Across these various types of questions, the median difference between results obtained from Pew Research surveys and using Google Consumer Surveys was 3 percentage points. The mean difference was 6 points, which was a result of several sizable differences that ranged from 10–21 points and served to increase the mean difference. It appears, however, that Google Survey takers are no more likely to be technology-savvy than an average Internet user, largely eliminating that bias. A key limitation for large-scale survey appears to be the inability to ask more than a few questions at a time, which is a limitation of their format [24], and the inability to administer questions to the same responder over time. The focus in InterPoll is on supporting as many questions as the developer wants to include.

4 Conclusions

This paper presents a vision for InterPoll, a language integrated approach to programming crowd-sourced polls. While much needs to be done to achieve the goals outlined in Section 1, we envision InterPoll as a powerful system, useful in a range of domains, including social sciences, political and marketing polls, and health surveys.

References

1 Sarah Anderson, Sarah Wandersee, Ariana Arcenas, and Lynn Baumgartner. Craigslist samples of convenience: recruiting hard-to-reach populations. Unpublished.

2 D Andrews, B Nonnecke, and J Preece. Electronic survey methodology: A case study in reaching hard-to-involve Internet users. *International Journal of . . .*, 2003.

3 J Antin and A Shaw. Social desirability bias and self-reports of motivation: a study of Amazon Mechanical Turk in the US and India. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012.

4 Daniel Barowy, Charlie Curtsinger, Emery Berger, and Andrew McGregor. AutoMan: A platform for integrating human-based and digital computation. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications – OOPSLA'12*, page 639, January 2012.

5 T S Behrend, D J Sharek, and A W Meade. The viability of crowdsourcing for survey research. *Behavior research methods*, January 2011.

6 A Berinsky, G Huber, and G Lenz. Evaluating Online Labor Markets for Experimental Research: Amazon.com's Mechanical Turk. *Political Analysis*, 20(3):351–368, July 2012.

7 Adam J AJ Berinsky, Gregory A GA Huber, and Gabriel S Lenz. Using mechanical Turk as a subject recruitment tool for experimental research. *Typescript, Yale*, pages 1–26, 2010.

8 Samuel J. Best and Brian S. Krueger. *Exit Polls: Surveying the American Electorate, 1972–2010*. CQ Press, 2012.

9 M Bleja, T Kowalski, and K Subieta. Optimization of object-oriented queries through rewriting compound weakly dependent subqueries. *Database and Expert Systems*, pages 1–8, January 2010.

10 James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: A First-order Type for Uncertain Data. *SIGARCH Comput. Archit. News*, 42(1):51–66, 2014.

11 M Buhrmester and T Kwang. Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data? *on Psychological Science*, January 2011.

12 Trent D Buskirk, D Ph, and Charles Andrus. Online Surveys Aren't Just for Computers Anymore! Exploring Potential Mode Effects between Smartphone and Computer-Based Online Surveys. *American Statistical Association (ASA), events and resources for statisticians, educators, students*, pages 5678–5691, 2010.

13 J Chandler, P Mueller, and G Paolacci. Methodological concerns and advanced uses of crowdsourcing in psychological research. *Behavioral Research*, 2013.

14 James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming – ICFP'13*, page 403, January 2013.

15 D. L. Clancy and K. Phillips J. Some effects of "Social desirability" in survey studies. *The American Journal of Sociology*, 77(5):921–940, 1972.

16 Christopher Cooper, David M McCord, and Alan Socha. Evaluating the college sophomore problem: the case of personality and politics. *Journal of Psychology*, 145(1):23–37, 2011.

17 M Couper. Designing effective web surveys, 2008.

18 M P Couper. Review: Web surveys: A review of issues and approaches. *The Public Opinion Quarterly*, pages 1–31, January 2000.

19 Franco Curmi and Maria Angela Ferrario. Online sharing of live biometric data for crowd-support: Ethical issues from system design. Unpublished, 2013.

20 Nilesh Dalvi, Christopher Ré, and Dan Suciu. Probabilistic Databases: Diamonds in the Dirt. *Communications of the ACM*, 2009.

21 D Dillman, R Tortora, and D Bowker. Principles for constructing Web surveys. Unpublished, 1998.

22 M Duda and J Nobile. The fallacy of online surveys: No data are better than bad data. *Human Dimensions of Wildlife*, 2010.

23 B Duffy, K Smith, and G Terhanian. Comparing data from online and face-to-face surveys. *International Journal of*, January 2005.

24 Justin Ellis. How Google is quietly experimenting in new ways for readers to access publishers' content, 2011.

25 Joel Evans, New Hempstead, and Anil Mathur. The value of online surveys. *Internet Research*, 15(2):195–219, January 2005.

26 Jeremy Eysenbach, Gunther Eysenbach, and Jeremy Wyatt. Using the Internet for Surveys and Health Research. *Journal of Medical Internet Research*, 4(2):e13, January 2002.

27 Emma Ferneyhough. Crowdsourcing Anxiety and Attention Research, 2012.

28 K Fort, G Adda, and K B Cohen. Amazon Mechanical Turk: Gold mine or coal mine? *Computational Linguistics*, pages 1–8, January 2011.

29 Michael Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. CrowdDB: answering queries with crowdsourcing. *SIGMOD'11: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1–12, June 2011.

30 S Fricker, M Galesic, R Tourangeau, and T Yan. An experimental comparison of web and telephone surveys. *Public Opinion Quarterly*, 2005.

31 M Fuchs. Mobile Web Survey: A preliminary discussion of methodological implications. *Envisioning the survey interview of the future*, January 2008.

32 Marek Fuchs and Britta Busse. The Coverage Bias of Mobile Web Surveys Across European Countries. *International Journal of Internet Science*, 4(1):21–33, 2009.

33 Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis*. CRC Press, 3rd edition, 2014.

34 Andrew D Gordon, Johannes Borgstr, Nicolas Rolland, and John Guiver. Tabular: A Schema-Driven Probabilistic Programming Language. Technical report, Microsoft Research, 2013.

35 Samuel Gosling, Simine Vazire, Sanjay Srivastava, and Oliver John. Should we trust web-based studies? A comparative analysis of six preconceptions about Internet questionnaires. *American Psychologist*, 59(2):93–104, January 2004.

36 R.M. Groves. *Survey Errors and Survey Costs*. Wiley Series in Probability and Statistics. Wiley, 1989.

37 Robert M. Groves, Floyd J. Fowler Jr., Mick P. Couper, James M. Lepkowski, Eleanor Singer, and Roger Tourangeau. *Survey Methodology*. Wiley, 2009.

38 Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-safe LINQ compilation. *Proceedings of the VLDB Endowment*, 3(1-2):162–172, September 2010.

39 H Gunn. Web-based surveys: Changing the survey process. *First Monday*, 2002.

40 Joseph Henrich, Steven J Heine, and Ara Norenzayan. The weirdest people in the world? *The Behavioral and brain sciences*, 33(2-3):61–83; discussion 83–135, June 2010.

41 HubSpot and SurveyMonkey. Using online surveys in your marketing. Unpublished.

42 P G Ipeirotis. Analyzing the Amazon Mechanical Turk marketplace. *XRDS: Crossroads*, January 2010.

43 P G Ipeirotis. Demographics of Mechanical Turk. *2010*, January 2010.

44 Floyd J. Fowler Jr. *Survey Research Methods (4th ed.)*. SAGE Publications, Inc., 4 edition, 2009.

45 R Jurca and B Faltings. Incentives for expressing opinions in online polls. *Proceedings of the ACM Conference on Electronic Commerce*, 2008.

46 Adam Kapelner and Dana Chandler. Preventing Satisficing in Online Surveys : A "Kapcha" to Ensure Higher Quality Data. *CrowdConf*, 2010.

47 S Keeter. The impact of cell phone noncoverage bias on polling in the 2004 presidential election. *Public Opinion Quarterly*, 2006.

48 Scott Keeter, Leah Christian, and Senior Researcher. A Comparison of Results from Surveys by the Pew Research Center and Google Consumer Surveys. http://www.people-press.org/files/legacy-pdf/11-7-12 Google Methodology paper.pdf, 2012.

49 P Kellner. Can online polls produce accurate findings? *International Journal of Market Research*, 2004.

50 A Kittur, E H Chi, and B Suh. Crowdsourcing user studies with Mechanical Turk. *Proceedings of the SIGCHI conference on*, January 2008.

51 Aniket Kittur, Susheel Khamkar, Paul André, and Robert Kraut. CrowdWeaver: Visually Managing Complex Crowd Work. *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work – CSCW'12*, page 1033, January 2012.

52 R Kosara and C Ziemkiewicz. Do Mechanical Turks dream of square pie charts? *Proceedings Beyond time and errors: novel evaLuation methods for Information Visualization*, 2010.

53 Robert Kraut, Judith Olson, Mahzarin Banaji, Amy Bruckman, Jeffrey Cohen, and Mick Couper. Psychological Research Online: Report of Board of Scientific Affairs' Advisory Group on the Conduct of Research on the Internet. *American Psychologist*, 59(2):105–117, January 2004.

54 Robert E Kraut. CrowdForge : Crowdsourcing Complex Work. *UIST*, pages 43–52, 2011.

55 Paul Krugman. What People (Don't) Know About The Deficit, April 2013.

56 A Kulkarni, M Can, and B Hartmann. Collaboratively crowdsourcing workflows with turkomatic. *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, January 2012.

57 A P Kulkarni, M Can, and B Hartmann. Turkomatic: automatic recursive task and workflow design for mechanical turk. *CHI'11 Extended Abstracts on Human*, January 2011.

58 G Little, L B Chilton, M Goldman, and R C Miller. TurKit: tools for iterative tasks on Mechanical Turk. *Proceedings of UIST*, pages 1–2, January 2009.

59 Benjamin Livshits and George Kastrinis. Optimizing human computation to save time and money. Technical Report MSR-TR-2014-145, Microsoft Research, November 2014.

60 Benjamin Livshits and Todd Mytkowicz. Saving money while polling with interpoll using power analysis. In *In Proceedings of the Conference on Human Computation and Crowdsourcing (HCOMP 2014)*, November 2014.

61 A Marcus, E Wu, Karger, S R Madden, and R C Miller. Crowdsourced databases: Query processing with people. *2011*, January 2011.

62 Adam Marcus, David Karger, Samuel Madden, Robert Miller, and Sewoong Oh. Counting with the crowd. *Proceedings of the VLDB Endowment* ,, 6(2), December 2012.

63 Adam Marcus, Eugene Wu, David Karger, Samuel Madden, and Robert Miller. Human-powered sorts and joins. *Proceedings of the VLDB Endowment* ,, 5(1), September 2011.

64 W Mason and S Suri. Conducting behavioral research on Amazon's Mechanical Turk. *Behavior research methods*, January 2012.

65 Joe Mayo. *LINQ Programming*. McGraw-Hill Osborne Media, 1 edition, 2008.

66 Paul Mcdonald, Matt Mohebbi, and Brett Slatkin. Comparing Google Consumer Surveys to Existing Probability and Non-Probability Based Internet Surveys. `http://www.google.com/insights/consumersurveys/static/consumer_surveys_whitepaper.pdf`.

67 Patrick Minder, Sven Seuken, Abraham Bernstein, and Mengia Zollinger. CrowdManager – Combinatorial Allocation and Pricing of Crowdsourcing Tasks with Time Constraints. *Workshop on Social Computing and User Generated Content in conjunction with ACM Conference on Electronic Commerce (ACM-EC 2012)*, 2012.

68 Derek Murray, Michael Isard, and Yuan Yu. Steno: automatic optimization of declarative queries. *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–11, June 2011.

69 Venkata Nerella, Sanjay Madria, and Thomas Weigert. An Approach for Optimization of Object Queries on Collections Using Annotations. *2013 17th European Conference on Software Maintenance and Reengineering*, pages 273–282, March 2013.

70 Daniel M. Oppenheimer, Tom Meyvis, and Nicolas Davidenko. Instructional manipulation checks: Detecting satisficing to increase statistical power. *Journal of Experimental Social Psychology*, 45(4):867–872, July 2009.

71 G Paolacci, J Chandler, and P G Ipeirotis. Running experiments on Amazon Mechanical Turk. *Judgment and Decision*, January 2010.

72 Pew Research Center. Demographics of Internet users, 2013.

73 Steven J Phillips, Miroslav Dudík, Jane Elith, Catherine H Graham, Anthony Lehmann, John Leathwick, and Simon Ferrier. Sample selection bias and presence-only distribution models: implications for background and pseudo-absence data. *Ecological applications : a publication of the Ecological Society of America*, 19(1):181–97, January 2009.

74 P Podsakoff, S MacKenzie, and J Lee. Common method biases in behavioral research: a critical review of the literature and recommended remedies. *Journal of Applied Psychology*, 88(5):879–903, 2003.

75 Ramo and S M Hall. Reaching young adult smokers through the Internet: Comparison of three recruitment mechanisms. *Nicotine & Tobacco*, January 2010.

76 D Ramo, S Hall, and J Prochaska. Reliability and validity of self-reported smoking in an anonymous online survey with young adults. *Health Psychology*, 2011.

77 Allan Roshwalb, Neal El-Dash, and Clifford Young. Toward the use of Bayesian credibility intervals in online survey results. `http://www.ipsos-na.com/knowledge-ideas/public-affairs/points-of-view/?q=bayesian-credibility-interval`, 2012.

78 J Ross, A Zaldivar, L Irani, B Tomlinson, and M Silberman. Who are the crowdworkers?: shifting demographics in Mechanical Turk. *CHI'10 Extended*, January 2009.

79 Matthew Salganik and Karen Levy. Wiki surveys: Open and quantifiable social data collection. http://arxiv.org/abs/1202.0500, February 2012.

80 L Sax, S Gilmartin, and A Bryant. Assessing response rates and nonresponse bias in web and paper surveys. *Research in higher education*, 2003.

81 L Schmidt. Crowdsourcing for human subjects research. *Proceedings of CrowdConf*, 2010.

82 M Schonlau, A Soest, A Kapteyn, and M Couper. Selection bias in Web surveys and the use of propensity scores. *Sociological Methods & Research*, 37(3):291–318, February 2009.

83 G Schueller and A Behrend. Stream Fusion using Reactive Programming, LINQ and Magic Updates. *Proceedings of the International Conference on Information Fusion*, pages 1–8, January 2013.

84 S Sills and C Song. Innovations in survey research an application of web-based surveys. *Social science computer review*, 2002.

85 Cindy D. Simasa2 and Elizabeth N. Kama. Risk Orientations and Policy Frames. *The Journal of Politics*, 72(2), 2010.

86 Martha Sinclair, Joanne O'Toole, Manori Malawaraarachchi, and Karin Leder. Comparison of response rates and cost-effectiveness for a community-based survey: postal, internet and telephone modes with generic or personalised recruitment approaches. *BMC medical research methodology*, 12(1):132, January 2012.

87 Nick Sparrow. Developing Reliable Online Polls. *International Journal of Market Research*, 48(6), 2006.

88 Robin Sprou. Exit Polls: Better or Worse Since the 2000 Election? Joan Shorestein Center on the Press, Politics and Public Policy, 2008.

89 J Sprouse. A validation of Amazon Mechanical Turk for the collection of acceptability judgments in linguistic theory. *Behavior research methods*, January 2011.

90 L B Stephenson and J Crête. Studying political behavior: A comparison of Internet and telephone surveys. *International Journal of Public Opinion Research*, January 2011.

91 SurveyMonkey. Data Quality: Measuring the Quality of Online Data Sources. http://www.slideshare.net/SurveyMonkeyAudience/surveymonkey-audience-data-quality-whitepaper-september-2012, 2012.

92 SurveyMonkey. Market Research Survey; Get to know your customer, grow your business, 2013.

93 M Swan. Crowdsourced health research studies: an important emerging complement to clinical trials in the public health research ecosystem. *Journal of Medical Internet Research*, January 2012.

94 Melanie Swan. Scaling crowdsourced health studies : the emergence of a new form of contract research organization. *Personalized Medicine*, 9:223–234, 2012.

95 Swati Tawalare and S Dhande. Query Optimization to Improve Performance of the Code Execution. *Computer Engineering and Intelligent Systems*, 3(1):44–52, January 2012.

96 Emma Tosch and Emery D. Berger. Surveyman: Programming and automatically debugging surveys. In *Proceedings of Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'14, 2014.

97 Roger Tourangeau, Frederick G. Conrad, and Mick P. Couper. *The Science of Web Surveys.* Oxford University Press, 2013.

98 H L Truong, S Dustdar, and K Bhattacharya. Programming hybrid services in the cloud. *Service-Oriented Computing*, pages 1–15, January 2012.

99 Amos Tversky and Daniel Kahneman. The Framing of Decisions and the Psychology of Choice The Framing of Decisions and the Psychology of Choice. *Science*, 211(4481):453–458, 1981.

100 US Census. Current population survey, October 2010, school enrollment and Internet use supplement file, 2010.

101 USamp. Panel Book 2013. 2013.

102 A. Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 06 1945.

103 Fabian L Wauthier and Michael I Jordan. Bayesian Bias Mitigation for Crowdsourcing. *Neural Information Processing Systems Conference*, pages 1–9, 2011.

104 R W White. Beliefs and Biases in Web Search. *2013*, January 2013.

105 K Wright. Researching Internet-Based Populations: Advantages and Disadvantages of Online Survey Research, Online Questionnaire Authoring Software Packages, and Web Survey Services. *Journal of Computer-Mediated Communication*, 2005.

106 J Wyatt. When to use web-based surveys. *Journal of the American Medical Informatics Association*, 2000.

107 D Yeager, J Krosnick, L Chang, and H Javitz. Comparing the accuracy of RDD telephone surveys and internet surveys conducted with probability and non-probability samples. *Public Opinion Quarterly*, 2011.

108 X Yin, W Liu, Y Wang, C Yang, and L Lu. What? How? Where? A Survey of Crowdsourcing. *Frontier and Future Development of*, January 2014.

109 Clifford Young, John Vidmar, Julia Clark, and Neale El-Dash. Our brave new world: blended online samples and performance of no probability approaches. Ipsos Public Affairs.

The Silently Shifting Semicolon

Daniel Marino[1], Todd Millstein[2], Madanlal Musuvathi[3], Satish Narayanasamy[4], and Abhayendra Singh[5]

1 **Symantec Research, US**
`daniel_marino@symantec.com`
2 **University of California, Los Angeles, US**
todd@cs.ucla.edu
3 **Microsoft Research, US**
`madanm@microsoft.com`
4 **University of Michigan, Ann Arbor, US**
`nsatish@umich.edu`
5 **University of Michigan, Ann Arbor, US**
`ansingh@umich.edu`

Abstract

Memory consistency models for modern concurrent languages have largely been designed from a system-centric point of view that protects, at all costs, optimizations that were originally designed for sequential programs. The result is a situation that, when viewed from a programmer's standpoint, borders on absurd. We illustrate this unfortunate situation with a brief fable and then examine the opportunities to right our path.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases memory consistency models, sequential consistency, safe programming languages, data races

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.177

1 A fable

Off to a good start

Alice is a bright and enthusiastic freshman in your programming languages (PL) class, who has also signed up for a class on French cuisine – computer science and cooking are her passions. In your first class you proudly proclaim that PL research has simplified the lives of programmers and increased their productivity. Type safety and automatic memory management, concepts that were once esoteric, have achieved mainstream success today. You summarize: "Modern programming languages enforce simple yet powerful abstractions that hide the complexities of the machine and make programming easier and safer."

You then explain and formalize *sequential composition*, a fundamental concept that strings instructions together. "In the program

$$A\ ;\ B$$

the semicolon instructs the machine to perform A and *then* B." This makes immediate sense to Alice. From the time she was a toddler (A: wash your hands, and B: you can eat a cookie) through today when she follows a cooking recipe (A: heat oil in a skillet, and B: add in chopped vegetables), Alice has implicitly used sequential composition without thinking about it. She realizes the importance of formalizing such intuitive notions. Alice is thrilled by the mathematical rigor of your class.

Soon after, Alice learns a related concept, *parallel composition*, but this time in her cooking class. Some recipes require executing two or more sequences of instructions in parallel, thereby maximizing the usage of kitchen resources while minimizing the time taken for the recipe. While the meaning of such recipes is specified informally, Alice uses her PL skills that she so admirably learned from you to formalize parallel composition. "The recipe

$$A\ ;\ B \quad \| \quad C$$

instructs me to do A before B but gives me the freedom to start C at any time – the presence of C is not a justification to do B before A." Alice smiles to herself and is excited by her formalization as she bikes to your class.

A chink in the armor

Unfortunately, Alice's excitement is short-lived as she writes her first parallel program in Java:

```
A: Pasta p = new Pasta();     ||  C: if (cooked)
B: cooked = true;                        p.drain();
```

This program mysteriously throws a null-pointer exception every so often and sometimes has even stranger behavior. Alice, for the life of her, cannot understand what's happening, since she is pretty sure her program is correct. Imagine her surprise when you respond "Oh, there is a bug in your program. You are assuming that A is always executed before B."

"A bug? You taught me that $A; B$ means 'A followed by B,' and we even formalized it! If I ask a chef to boil some pasta in a pot and set a timer for two minutes, but the pot is empty when the timer fires, I will blame the chef for not executing the recipe properly. I will *never* blame the recipe."

Fortunately you have a simple explanation. "You see Alice, a compiler performs some optimizations by reordering instructions in your program. At runtime, the hardware can execute instructions out of order, depending on whether the instructions hit or miss in the cache, and for various other reasons. So sometimes B is executed before A."

Alice is dumbstruck. "Optimizations? Caches? I am hearing these terms for the first time. Do I need to understand all of them to understand what $A; B$ means? Programming languages are supposed to hide all these details from me."

"They are supposed to, and they do. The problem is that you have a *data race* in your program."

Alice is puzzled. "And . . . what is a data race?"

"Ah, let me explain. In a multithreaded program, there exists a partial order among instructions called the *happens-before* relation. A program is said to have a data race if two instructions access the same memory location, one of them is a write, and they are not ordered by the happens-before relation."

"This all sounds confusing." Alice thinks for a bit . . .

Java is an unsafe language

In a short moment, Alice's eyes gleam. "I think I got it. I just need to ensure that a happens-before ordering exists between two writes to the same variable, and between a write and a read. Then I will not have data races. Which means I can operate under the illusion that semicolons represent sequential composition: $A; B$ will mean 'A and then B.' "

"Exactly Alice! You nailed it."

But as Alice thinks harder she begins to get worried again. "But . . . this illusion breaks the moment I introduce a data race in my program."

"Yes."

"Hmm, doesn't this mean that Java is unsafe, since it does not protect the abstraction of the semicolon? If I inadvertently introduce a data race then my program goes haywire in unpredictable ways. This is similar to the illusion of types in C/C++. A bug, like an out-of-bounds array access, can break this abstraction."

You are taken aback. "I guess you are right. This *is* a safety issue." You think for a moment and then respond, "But . . . protecting the semantics of semicolons would make programs run too slowly."

"How slowly?"

"We don't know exactly. The overheads obviously depend on the program, the programming language, the compiler, and the hardware. Experiments suggest an average overhead of 26% for a set of Java benchmarks [34]. With appropriate hardware changes, we can reduce this overhead significantly [16, 29, 5, 22, 33, 11]."

Alice is dumbfounded. "That seems entirely acceptable and is well within the cost I am already paying by moving from C++ to Java in order to obtain memory safety guarantees."

Understanding data races

You are undeterred. "Yes, but PL design is a delicate balance between programmability and performance. We have to ensure that the utility of an abstraction justifies its cost. For instance, taking the effort to protect semicolons will not eliminate the possibility of concurrency errors. You need to avoid data races anyway."

Alice takes another pause to understand what you just said. And her eyes gleam once again. "Ah, I think I am finally getting it. Data races are concurrency errors. So fixing my data races both provides sequential composition and ensures that there are no concurrency errors in my program."

You: "Er, not really. Data races are neither necessary nor sufficient for concurrency errors."

Alice is now visibly disappointed. "So I can write a data-race-free program that still has a concurrency error?"

You nod.

Alice shakes her head. "On the other hand, I can write a program that contains a data race but is nonetheless correct?"

You: "Yes, like the program you wrote earlier – it is correct in a language where semicolons means sequential composition. Fortunately there is a simple fix. Java will provide the desired semantics as long as you make all of your data races explicit. You can fix the bug in your program just by adding a `volatile` annotation on the `cooked` variable."

The horror, the horror

Alice takes another pause. "`Vola`-what? So, you want me to mark every data race in my program with this `volatile` annotation." Alice sighs. "I guess this is all fine as long as there are tools to help novice programmers like me to detect and remove data races. Maybe like the red squigglies I get when I make a syntax error?"

You wish you had good news. "This happens to be an active field of study. Static data-race detectors currently suffer from too many false positives [28, 25]. Dynamic data-race detectors [14] have performance overheads that are orders-of-magnitude more than the gain from compiler and hardware optimizations that are causing these issues in the first place. Finally, there are research proposals [8, 6, 20] for specialized programming models that guarantee data-race freedom, but they are not ready for the mainstream yet."

Alice takes a deep breath. "Ok, I guess with no tool support, I have to always assume that my programs contain data races. So I will just have to get used to this weaker semantics for semicolons. By the way, what is that semantics and what does it guarantee?"

You: "Unfortunately, this also turns out to be a hard problem. We have made several attempts to formalize the precise semantics and to ensure some basic sanity properties, but doing so while safely allowing the optimizations we desire is still an open problem."

Alice: "Ouch! How in the world are you programming with these semicolons then?"

You: "Luckily, it doesn't seem to matter in practice. Programs just seem to run ok even though there is the potential for unpredictable behavior."

Alice just cannot believe what her professor just said. "I quit! You taught me that modern programming languages enforce simple and powerful abstractions, but apparently performance trumps everything else. From now on I will stick to cooking – at least my kitchen never reorders my instructions behind my back!"

2 The moral of the story

"Safe" languages like Java have improved the world by bringing type and memory safety to the mainstream. We all know the benefits and preach them to our students and colleagues: providing strong guarantees to programmers for all programs that are allowed to execute, obviating large classes of subtle and dangerous errors, and cleanly separating a language's interface to programmers from its implementation details.

Yet the PL research community is largely silent as this trend is being inadvertently reversed in the standardization of memory consistency models for concurrent programming languages [23, 7]. Just as achieving memory safety in unsafe languages requires unrealistic programmer discipline, achieving sane semantics in Java requires programmers to meticulously avoid data races with little language support for doing so. In this world, data races become the new dangling pointers, double frees, and buffer overflows, exposing programs to hardware and compiler implementation details that can affect program behavior in unpredictable and unsafe ways. Worse, contrary to popular belief, most data races are not concurrency errors. Our attempt with Alice's story is to bring these issues to the forefront of PL researchers' minds and agendas, hopefully in a humorous and thought-provoking way.

We propose two tenets for the semantics of multithreading in safe languages:

▶ Tenet 1. Following Pierce's definition [27], safe programming languages should protect their own abstractions – no program, including a buggy one, should be able to break the language abstractions.

▶ Tenet 2. Sequential composition is a fundamental abstraction in programming languages.

We believe that these tenets are uncontroversial and widely accepted in the PL community. The second tenet may not be explicitly considered often, but a bit of thought shows it to be self-evident. If sequential reasoning is no longer valid, how do we understand pre- and post-conditions of functions? What does it mean for a library to have an interface specification? In short, how do we modularly build and reason about programs?

The necessary conclusion from these two tenets is that *safe programming languages should preserve sequential composition for all programs.* That is, safe languages should guarantee sequential consistency (SC) [21][1] – it should not be possible for programs to expose the effects of compiler and hardware reorderings.

Put another way, recent work on standardizing memory models is *system-centric*: it assumes that essentially all sequentially valid hardware and compiler optimizations are sacrosanct and instead debates the semantics one should assign to semicolons. Somehow the *programmer-centric* view, which argues for treating the semantics of semicolons as sacrosanct and designing the software/hardware stack to efficiently provide this semantics, has gotten lost.

The memory models in Java as well as C and C++ stem from the classic work on data-race-free-0 (DRF0) memory consistency models for hardware [2]. DRF0 argues that consistency models weaker than SC are hard for programmers to understand and therefore insists that platforms provide mechanisms for programmers to regain SC. Specifically, DRF0 guarantees that *data-race-free* (DRF) programs will have SC semantics. DRF0 therefore argues for Tenet 2, but by providing weak or no guarantees when programmers fail to follow the DRF discipline, it violates Tenet 1. This situation is particularly problematic because, despite decades of research, enforcing data-race-freedom is hard statically [28, 25], dynamically [14], or through language mechanisms [8, 6, 20]. Therefore, it is safe to assume that most programs will have data races and thus will be exposed to counter-intuitive non-SC behaviors.

Given the arguments above, it is perhaps surprising that no mainstream language design committee considers SC to be a viable programming interface. Even languages like Python, which hugely favor productivity over performance, are unwilling to guarantee SC. Why are PL designers so willing to trample on the fundamental abstraction of a semicolon? In our experience, we have seen three arguments made against SC:

1. "Why encourage 'racy' programming?" – SC is unnecessary as programs should be data-race free anyway.
2. "Didn't the Java memory model (JMM) solve this problem?" – SC is unnecessary since Java provides safety guarantees for non-SC programs.
3. "It's simply too expensive." – SC is not practical for modern languages and hardware.

The next three sections rebut these arguments.

3 Data races as red herrings

One argument against SC is that data races are concurrency errors, and thus the effort to provide SC over DRF0 only benefits buggy programs that should be fixed anyway. The need for safety immediately nullifies this argument – programmers, as humans, will violate any unchecked discipline, and safe languages should protect their abstractions despite such violations. But it is still important to ask if data races should be treated as concurrency errors.

A *memory race* (or simply a race) occurs when two threads concurrently access the same memory location and at least one of them is a write. Memory races are abundant in shared-memory programming and are used to implement efficient inter-thread communication and synchronization, such as locks. *A* data race *is simply a memory race that is not syntactically identified in the program.* Such explicit identification occurs through the use of standard language synchronization mechanisms, like locks, as well as through programmer annotations.

[1] SC preserves a global shared-memory abstraction in addition to preserving sequential composition.

Therefore it is easy to show that data races are neither necessary nor sufficient for a concurrency error [30, 19]. By a concurrency error, we mean a violation of an intended program invariant that is due to the interactions among multiple threads.

Here is a simple example of a DRF program in Java that has a concurrency error:

```
synchronized(lock){ temp = balance; }
temp++;                                  ||  synchronized(lock){ balance++; }
synchronized(lock){ balance = temp; }
```

In this code, the update to `balance` from the right thread can be lost if the threads interleave in the wrong way. But this program is DRF as the only memory race occurs on the `lock` variable via the locking mechanism that is provided by the language. In this case, ensuring that no updates are lost requires each update to be *atomic*, for example by modifying the left thread to hold the lock for its entire duration.

Conversely, programs can contain data races but yet behave as intended (assuming SC semantics), like the one Alice wrote above. That program properly preserves the invariant that the pasta is not drained until it has been cooked, but it has a data race because the `cooked` variable has not been explicitly annotated as `volatile`. Research on data-race detection amply shows [31, 37, 26, 12, 19] that most data races (e.g., [12] reports 90%) are not concurrency errors. They are either intentional races that the programmer failed to mark as such or unintentional races that are nonetheless correct under SC (e.g., writing the same value to a variable from multiple threads).

Thus, from a program correctness perspective, the primary reason to remove data races from programs today is to protect against compiler and hardware optimizations. DRF0's requirement to exhaustively annotate all memory races is therefore largely a distraction from what should be the main goal: to help programmers identify and avoid concurrency errors. To this end, languages should provide mechanisms that allow programmers to specify and statically check desired *concurrency disciplines*. For example, `guarded-by` annotations [13] allow programmers to specify which locks are intended to protect which data structures, and atomicity annotations [15] allow programmers to specify which blocks of code are intended to execute atomically. In stark contrast to these kinds of annotations, `volatile` annotations do not convey the intended concurrency discipline and do not help the compiler to identify concurrency errors.

Nevertheless, data-race-free programming provides some benefits from a program understanding perspective. First, despite the `volatile` annotation's limited expressiveness as described above, it allows programmers to explicitly document inter-thread communication. However, note that the DRF discipline does not require all inter-thread communication to be documented. For example, a reference to a thread-safe queue does not require a `volatile` annotation even if threads use the queue operations to communicate – the DRF discipline only pertains to the individual, low-level memory races in the queue's implementation.

Second, a DRF program has the nice property that any synchronization-free region of code is guaranteed to be atomic [1]. This allows reasoning about program behavior as interleavings of these coarse-grained atomic regions. Programmers have to conservatively assume that dynamically dispatched method calls as well as library calls could introduce synchronization and accordingly break atomicity. Even so, the resulting granularity is still coarser in practice than what is possible in a non-DRF program, where only regions of code with no thread-shared accesses can be treated as atomic.

However, these benefits of the DRF discipline are *not* a justification to choose DRF0 over SC. DRF0 is a strictly weaker memory model than SC. This means that, from a programmer's standpoint, DRF0 shares all the deficiencies of SC, and SC shares all the strengths of DRF0. For instance, neither under SC nor under DRF0 can programmers rely on the benefits of the DRF discipline by default, due to the lack of language or tool support for enforcing that discipline. Instead, programmers must conservatively assume the presence of data races. If a programmer can somehow ensure that a program obeys the DRF discipline, however, then all the benefits of that discipline are guaranteed, both under DRF0 and under SC.

Hence, SC and DRF0 are equivalent in terms of their benefits relative to the DRF discipline. But SC additionally provides safety by protecting all programs from the counterintuitive effects of hardware and compiler optimizations, while DRF0 does not.

4 SC and the Java Memory Model

Another common objection we hear is the claim that Java's memory model has solved the safety issues with the DRF0 memory model, leaving no further incentive for SC. Indeed, the Java memory model is intended to provide a semantics to programs containing data races that still ensures important safety properties, including type safety, memory safety, and preventing "out of thin air" reads, whereby the read of a variable returns a value that was never written to that variable [23].

In the decade since its introduction, the current JMM has been the subject of much scrutiny in the academic literature. It is now known to prevent compiler optimizations that it intended to allow [9]. In fact, certain optimizations that would always be valid under SC (such as redundant read elimination) can lead to behavior that violates the JMM. As a result, commonly used compilers for Java, while likely DRF0-compliant, generate code that violates the JMM for some racy programs [36]. Yet we know of no replacement that is under consideration. In other words, despite years of research, the community has not found a viable middle ground between SC's safety guarantees and DRF0's performance.

Further, even if these issues are resolved and the JMM can achieve its goals, SC is still necessary for two reasons. First, the JMM is complex and difficult to understand. Any future compiler optimization should be carefully analyzed to ensure its JMM compliance. Given the experience to date, it is likely that this will lead to subtle compiler bugs that can subvert the JMM's guarantees in unpredictable ways. Second, we argue that the JMM's safety guarantees are much weaker than what most programmers would expect and rely upon. While the JMM has focused on obviating out-of-thin-air reads for security reasons, it is easy for other non-SC behaviors that *are* JMM-compliant to nonetheless cause serious security vulnerabilities. For example, the following variant of Alice's program can exhibit an injection attack when the right thread unknowingly accesses the unsanitized version of `d`:

```
Data d = getInput();        if (sanitized)
d.sanitize();          ||          d.use();
sanitized = true;
```

In other words, any behavior that violates the programmer's SC-based reasoning can potentially lead to unauthorized data access, and hence imply serious consequences for privacy and security. This concern is not merely hypothetical: we are aware of an instance where a C++ compiler optimization introduced a time-of-check-to-time-of-use security vulnerability in a commonly used operating-system kernel. Perhaps surprisingly, the vulnerability was fixed not by changing the code, but by disabling the problematic compiler optimization.

5 The path forward

In our fable, the thoroughly frustrated Alice accused computer scientists of prioritizing performance above safety. While we believe that achieving safety is worth sacrificing some speed, pragmatism insists that we still retain reasonable performance. We would be wrong to claim that providing SC is extremely efficient on existing hardware platforms. But equally wrong is the blanket claim that "SC is impractical as it disables most compiler and hardware optimizations." The truth is somewhere in between and we will attempt to quantify the cost of providing SC below. Then we will describe a feasible path toward achieving SC in modern languages.

In many ways, the current situation mirrors the situation for garbage-collected languages a few decades ago. The PL community in the large understood the importance of type safety and automated memory management (well before the security implications for C/C++ programs became known) and collectively worked to improve the efficiency of type-safe languages. As a result type and memory safety has become the default in modern, mainstream programming languages. In the same vein, it is up to us as a community to determine if the abstraction of sequential composition is worth protecting.

Let us quantify the cost of SC over DRF0 languages. Using DRF0 terminology, memory accesses in a program can be classified into *synchronization* accesses and *data* accesses. Synchronization accesses are identified through the use of standard synchronization constructs and through annotations such as `volatile`. DRF0 languages must guarantee SC semantics for synchronization accesses, and they do so by both restricting compiler optimizations and emitting appropriate hardware fences that restrict hardware optimizations. On the other hand, the compiler and the hardware can freely optimize data accesses within synchronization-free regions of code.

Now consider a simple SC compiler which follows a "volatile-by-default" approach. It must ensure SC semantics for all accesses, whether synchronization or data. However, if the compiler can statically prove some accesses to be DRF, it is safe to aggressively optimize them. This includes accesses to local variables, compiler-generated temporaries, objects that do not escape a particular thread, and member objects that are consistently protected by the monitor of a class. The table below compares these two approaches:

	Optimized?	
Access Kind	DRF0	SC
synchronization	N	N
data, DRF provable	Y	Y
data, DRF not provable	Y	N

As we can see, the only difference between DRF0 and SC lies in their treatment of data accesses that the compiler cannot prove to be DRF. DRF0 simply assumes that such accesses are DRF, optimizing them at the expense of safety, while SC conservatively assumes that such accesses might not be DRF, obtaining safety at the expense of performance. This simple approach to ensuring SC semantics is, even today, much less expensive than is commonly thought. Research has shown ([34] for Java, [24] for C/C++) that the performance loss due to forgone compiler optimizations is negligible, since many optimizations are already compatible with SC and the rest can still be safely applied to thread-local variables. The hardware cost of additionally issued fences can be more significant, but even then, Alglave *et al.* [3] show the runtime overhead of this approach for `memcached` to be only 17.5% on x86 and 6.8% on ARM.

This overhead will continue to decrease as hardware fences become more efficient. Hardware vendors are already under pressure to optimize their fences to efficiently support synchronization accesses in DRF0 languages, and recent research suggests that significant improvements have been made [10]. Hill argued years ago [18] that there is little performance incentive to relax memory models in hardware since techniques abound to efficiently mask hardware optimizations [16, 29, 17, 5, 22, 33, 11]. Hardware platforms that are unwilling to commit to a strong memory model interface can still use these techniques to optimize fences, thereby enabling the compiler to efficiently guarantee SC to the programmer.

Orthogonally, the PL community should strive to increase the number of optimizations that the compiler can perform, and reduce the number of hardware fences it must emit, while still guaranteeing SC behavior. One approach is to prove more memory accesses as DRF, which allows them to be optimized within synchronization-free regions. But note that data-race freedom is just one sufficient condition to enable some SC-preserving optimizations, and many other opportunities exist. For instance, delay set analysis [32, 34, 4] can identify accesses that can be reordered without violating SC, in spite of the fact that they may race with accesses in other threads. Alglave *et al.* [3] report that this technique reduces the overhead of SC on `memcached` to 1.1% on x86 and 3% on ARM. Scaling such analyses to large programs, including techniques for sound modular analysis, is an important avenue for future research.

Further, an SC compiler can leverage the kinds of concurrency annotations that we mentioned in Sec. 3 to soundly and modularly identify SC-preserving compiler optimizations. For instance, the `guarded-by` annotation [13] communicates a particular locking discipline, allowing an SC compiler to optimize accesses to the protected location for the entire duration that the guarding lock is held. This includes optimizations that reorder operations across unrelated synchronization accesses – an opportunity that even a DRF0 compiler would not find since it lacks information about which synchronization accesses protect which memory locations. Therefore programmers have two good reasons to provide these annotations: avoiding concurrency errors and speeding up their code. We believe that continued research into new annotations and language constructs that establish statically checkable concurrency disciplines will uncover additional opportunities for compilers to produce fast SC code.

6 SC, DRF0, and Unsafe Languages

The safety argument we have made so far obviously does not apply to unsafe languages like C/C++. An explicit design goal in these languages is to avoid language abstractions that are not efficiently implementable on existing hardware. Given the current cost of hardware fences, it is unlikely that the overheads of SC are acceptable for these "bare metal" languages. Thus, at first blush, DRF0 seems to be the only feasible choice to ensure SC reasoning. Given that unsafe languages already require programmer discipline to retain important abstractions like memory safety, it seems acceptable to additionally require that programmers exhaustively annotate all memory races.

However, the experience with C/C++ standardization shows that even DRF0 is not efficiently implementable on current hardware platforms [7]. As a result, C/C++ has settled for a memory model *weaker* than DRF0, which we call Weak DRF0 (WDRF0). DRF programs are *not* guaranteed SC semantics in WDRF0. To get SC, programmers have to additionally avoid the use of the so-called *low-level* atomic primitives. The weak semantics of DRF programs in C++ is similar in complexity to the semantics of non-DRF programs in Java (and with similar problems in precisely pinning down this semantics [35]).

The C++ standard discourages the use of low-level atomic primitives by giving the vanilla `atomic` annotation (which is the analogue of Java's `volatile`) SC semantics. The expectation is that low-level atomics will only be used by a handful of experts to implement performance-critical libraries. Nevertheless, it is unclear how to effectively hide the weakness within these libraries such that "well behaved" programs that use them can enjoy SC reasoning. Additionally, given the predominance of benign data races in legacy software and the continued need for programmers to use ad-hoc synchronization idioms (from double-checked locking to redundant initialization of variables to write-only debug variables), it is likely that low-level atomic primitives will be present in application-level code as well.

Thus, we are currently in the unfortunate situation in which C/C++ programmers are not guaranteed SC, even if they take care to properly annotate all their races. A primary description of the C++ memory model [7] acknowledges this deficiency and states that DRF0 "is the model we should strive for," which will require hardware vendors to reduce the cost of fences. But as argued above, any improvement in the performance of hardware fences also improves the performance of SC. In effect, the DRF0 memory model is sandwiched between the currently feasible but unacceptably weak memory model WDRF0 on the one side and SC on the other side. Any improvement in the performance gap between WDRF0 and DRF0 will reduce the performance gap between DRF0 and SC even more. We predict that when DRF0 becomes acceptably efficient for C/C++, the performance of SC over DRF0 will also be acceptably small. In such a setting, it is unclear if the programmer effort to meticulously annotate all memory races and the potential for unsafe behaviors makes DRF0 worthwhile even for C/C++ programs.

7 Conclusion

The semantics of the semicolon is fundamental in mainstream programming languages. This paper makes a case that semicolons should mean what everyone already thinks they mean, namely sequential composition: $A; B$ should *always* mean A and then B. In such a world, Alice can write programs that behave according to her intuition, expert programmers writing low-level libraries don't have to carefully figure out what fences to insert where, and researchers don't have to debate what a four-line concurrent program should mean.

Language runtimes relax the meaning of semicolons today simply due to an *accident of history*. When multiprocessors were first designed, hardware architects did not know effective ways to hide single-processor optimizations from multithreaded programs, leading to relaxed-memory-model interfaces and associated expensive fences. As Hill elegantly argued more than 15 years ago [18], this is no longer necessary – architects now have a variety of techniques [16, 29, 17, 5, 22, 33, 11] to efficiently hide hardware optimizations from software (or equivalently, to drastically reduce the cost of fences), several of which are already implemented in commercial processors such as x86. Many of the design decisions that went into the Java and C/C++ memory models are centered around the need to minimize the number of hardware fences, thereby perpetuating the historical accident at the cost of program safety and programmer sanity.

We believe that it is high time to rectify this situation. We hope that Alice's story and our associated arguments will convince language designers to bring SC into the mainstream, just as Java did for type and memory safety a few decades ago. As we have argued, safe languages must preserve sequential composition, and the overhead of SC is much less than commonly thought and will be significantly reduced over time. We believe the safety benefits of SC are well worth the overhead, especially in this day and age when programmers routinely give up integer factors of performance, for example by using scripting languages, in exchange for increased productivity.

Current and future programmers need our help. If we don't save the semicolons, who will?

Acknowledgements. We would like to thank Hans Boehm, Mark Hill, Doug Lea, Kathryn McKinley, Ken McMillan, Todd Mytkowicz, Shaz Qadeer, Michael Scott, Karin Strauss, Ben Zorn, and the anonymous reviewers for their insightful feedback on this paper.

References

1 Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, August 2010.

2 Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA'90, pages 2–14, New York, NY, USA, 1990. ACM.

3 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *CoRR*, abs/1312.1411, 2013.

4 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *Computer Aided Verification - 26th International Conference, CAV 2014*, pages 508–524, 2014.

5 Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA'09, pages 233–244, New York, NY, USA, 2009. ACM.

6 Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'09, pages 97–116. ACM, 2009.

7 Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'08, pages 68–78, New York, NY, USA, 2008. ACM.

8 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'02, pages 211–230, New York, NY, USA, 2002. ACM.

9 Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proceedings of the 16th European Conference on Programming*, ESOP'07, pages 331–346, Berlin, Heidelberg, 2007. Springer-Verlag.

10 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP'13, pages 33–48. ACM, 2013.

11 Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. Weefence: Toward making fences free in TSO. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 213–224. ACM, 2013.

12 John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

13 Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI'00, pages 219–232, New York, NY, USA, 2000. ACM.

14 Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'09, pages 121–133, New York, NY, USA, 2009. ACM.

15 Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, August 2008.

16 K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, ICPP'91, pages 355–364, 1991.

17 Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA'99, pages 162–171, Washington, DC, USA, 1999. IEEE Computer Society.

18 Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31:28–34, 1998.

19 Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012. ACM.

20 Lindsey Kuper and Ryan R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC'13, pages 71–84. ACM, 2013.

21 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 100(28):690–691, 1979.

22 Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient sequential consistency via conflict ordering. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 273–286, New York, NY, USA, 2012. ACM.

23 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'05, pages 378–391, New York, NY, USA, 2005. ACM.

24 Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an SC-preserving compiler. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, pages 199–210, New York, NY, USA, 2011. ACM.

25 M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI'06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, 2006.

26 Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 22–31, New York, NY, USA, 2007. ACM.

27 Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

28 Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'06, pages 320–331, New York, NY, USA, 2006. ACM.

29 Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'97, pages 199–210, New York, NY, USA, 1997. ACM.

30 John Regehr. Race condition vs. data race. `http://blog.regehr.org/archives/490`.

31 S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

32 Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

33 Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. *SIGARCH Comput. Archit. News*, 40(3):524–535, June 2012.

34 Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent Java programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'05, pages 2–13, New York, NY, USA, 2005. ACM.

35 Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'15, pages 209–220, New York, NY, USA, 2015. ACM.

36 Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP'08, pages 27–51, Berlin, Heidelberg, 2008. Springer-Verlag.

37 Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP'05, pages 221–234, New York, NY, USA, 2005. ACM.

Everything You Want to Know About Pointer-Based Checking

Santosh Nagarakatte[1], Milo M. K. Martin*[2], and Steve Zdancewic[3]

1 **Department of Computer Science, Rutgers University, US**
`santosh.nagarakatte@cs.rutgers.edu`
2 **Computer and Information Sciences, University of Pennsylvania, US**
`milom@cis.upenn.edu`
3 **Computer and Information Sciences, University of Pennsylvania, US**
`stevez@cis.upenn.edu`

Abstract

Lack of memory safety in C/C++ has resulted in numerous security vulnerabilities and serious bugs in large software systems. This paper highlights the challenges in enforcing memory safety for C/C++ programs and progress made as part of the SoftBoundCETS project. We have been exploring memory safety enforcement at various levels – in hardware, in the compiler, and as a hardware-compiler hybrid – in this project. Our research has identified that maintaining metadata with pointers in a disjoint metadata space and performing bounds and use-after-free checking can provide comprehensive memory safety. We describe the rationale behind the design decisions and its ramifications on various dimensions, our experience with the various variants that we explored in this project, and the lessons learned in the process. We also describe and analyze the forthcoming Intel Memory Protection Extensions (MPX) that provides hardware acceleration for disjoint metadata and pointer checking in mainstream hardware, which is expected to be available later this year.

1998 ACM Subject Classification D.3.4 Processors, D.2.5 Testing and Debugging

Keywords and phrases Memory safety, Buffer overflows, Dangling pointers, Pointer-based checking, SoftBoundCETS

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.190

1 Introduction

Languages like C and its variants are the gold standard for implementing a wide range of software systems from low-level system/infrastructure software to performance-critical software of all kinds. Features such as low-level control over memory layout, explicit manual memory management, and proximity to the hardware layout have made C the dominant language for many domains. However, C language implementations with their focus on performance do not ensure that programmers use C's low level features correctly and safely. Further, weak typing in C necessitates dynamic checking to enforce C language abstractions, which can cause additional performance overheads.

Memory safety ensures that all memory accesses are well-defined according to the language specification. Memory safety violations in C arise when accesses are to memory locations (1) that are beyond the allocated region for an object or an array (known as *spatial memory safety violations* or bounds errors) and/or (2) that have been deallocated while managing

* On leave at Google.

memory through manual memory management (known as *temporal memory safety violations*, use-after-free errors or dangling pointer errors).

Without memory safety, seemingly benign program bugs anywhere in the code base can cause silent memory corruption, difficult-to-diagnose crashes, and incorrect results. Worse yet, lack of memory safety is the root cause of multitude of security vulnerabilities, which result by exploiting a memory safety error with a suitably crafted input. The buffer overflow vulnerabilities, use-after-free vulnerabilities, and other low-level vulnerabilities resulting from memory safety violations have compromised the security of the computing ecosystem as a whole [42, 50, 8, 43, 3].

1.1 Memory Safety for C versus Java/C#

Languages such as Java and C# enforce memory safety with a combination of strong typing, runtime checks, and automatic memory management. The type casts are either disallowed or restricted. The runtime checks are performed to ensure that the array accesses are within bounds and type casts are between objects in the same object hierarchy. Automatic memory management (*e.g.*, using garbage collection) ensures that any reachable object is not freed.

Unfortunately, C's weak typing allows arbitrary type casts and conflates arrays and pointers. Preventing memory safety violations therefore requires checks on every memory access, because it is difficult to distinguish between pointers that point to a single element from pointers that point to an array of elements. Information about allocation/object sizes is lost in the presence of unsafe type casts. Hence, Java/C# style checking is infeasible to enforce memory safety for C. Further, avoiding temporal safety violations with automatic memory management (*e.g.*, using a garbage collector) does not allow the low-level control of memory allocations and deallocations to which C/C++ programmers are accustomed for implementing systems/infrastructure software.

As a consequence of C's weak typing and other low-level features, a pointer (*e.g.*, `void *p`) in C code can be (1) a pointer to a memory location allowed by the language specification (*e.g.*, arrays, structures, single element of a particular data type, and sub-fields in a structure), (2) an out-of-bounds pointer, (3) a dangling pointer pointing to deallocated memory locations, (4) a `NULL` pointer, (5) an uninitialized pointer, and (6) a pointer manufactured from an integer. To check whether a memory access (pointer dereference) is valid, the challenging task is not necessarily checking memory accesses; the primary challenge is maintaining and propagating sufficient information (metadata with each pointer) to perform such checking in the presence of C's weak typing and other low-level features.

1.2 State-of-the-Art in Enforcing Memory Safety for C

Given the importance of the problem, comprehensively detecting and protecting against memory safety violations is a well researched topic with numerous proposals over the years (see Szekeres *et al.* [48] for a survey). Lack of memory safety was originally regarded as a software quality problem. Thus, majority of techniques were debugging tools rather than *always-on* deployment of such solutions. Subsequently, when lack of memory safety resulted in numerous security vulnerabilities, many proposed solutions addressed the symptoms of these security vulnerabilities rather than the root cause of these errors (*i.e.*, lack of memory safety). The solutions that enforce memory safety can be broadly classified into three categories: tripwire approaches, object-based approaches, and pointer-based approaches.

Tripwire approaches place a guard block of invalid memory between memory objects. The guard block prevents contiguous overflows caused by walking past an array boundary with a small stride. The tripwire approaches are generally implemented by tracking a few bits of state for each byte in memory; the additional bits indicate whether the location is currently valid [18, 40, 44, 49, 53]. When the memory is allocated, these bytes are marked as valid. Every load or store is instrumented to check the validity of the location. AddressSanitizer [46], a compiler implementation of the tripwire approach, is widely used to detect memory errors. Tripwire approaches can detect a class of buffer overflows (small strides) and use-after-free security vulnerabilities (when memory is not reused).

The object-based approaches [9, 12, 14, 22, 45] are based on the principle that all pointers are properly derived pointers to their intended referent (the object they point to). Hence these approaches check pointer manipulations to ensure that the resultant pointer points to a valid object. The distinguishing characteristic of this approach is that metadata is tracked per object and associated with the location of the object in memory, *not* with each pointer to the object. Every pointer to the object therefore shares the *same* metadata. Object-based approaches keep the memory layout unchanged which increases the source compatibility with existing C code. Object-based approaches are generally incomplete in the presence of type casts between pointers and pointers to subfields of an aggregate data type. Further, they require mechanisms to handle out-of-bound pointers as creating out-of-bound pointers is allowed by the C standard. SAFECode [9, 12] and Baggy Bounds [1] are efficient implementations of the object-based approach using whole program analysis and allocation bounds, respectively.

The third approach is the pointer-based approach, which tracks metadata with each pointer. The metadata provides each pointer a view of memory that it can access according to the language specification. The pointer-based approach is typically implemented using a *fat pointer* representation that replaces some or all pointers with a multi-word pointer/metadata. Two distinct pointers can point to the same object and have different base and bound associated with them, so this approach overcomes the sub-object problem [5, 35, 47]. When a pointer is involved in arithmetic, the actual pointer portion of the fat pointer is incremented/decremented. On a dereference, the actual pointer is checked with its metadata. Proposals such as SafeC [2], CCured [37, 7], Cyclone [21], MSCC [52], and others [11, 38, 41] use this pointer-based approach to provide varying degree of memory safety guarantees. Our SoftBoundCETS project was inspired by such prior pointer-based checking projects.

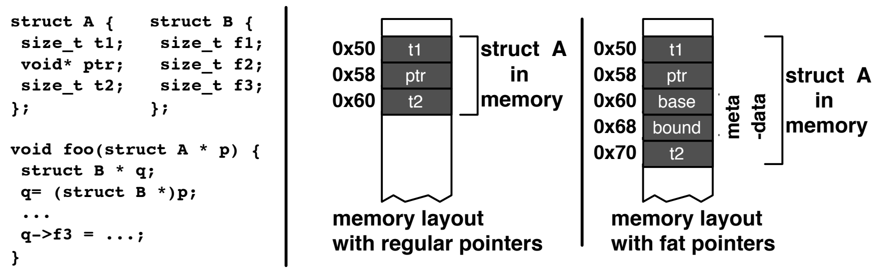
1.3 Goals of the Project

We started the SoftBoundCETS project[1] with the following goals:

- Comprehensive safety. Detect all memory safety errors in a fail stop fashion to completely prevent an entire class of memory safety related security vulnerabilities.
- Low performance overhead. Memory safety enforcement has to be carried out on deployed programs to prevent security vulnerabilities. Hence, any performance overhead should be indiscernible to the end user to be widely adopted.
- Source compatibility. As significant C code base exists, we wanted to enforce memory safety without requiring changes to existing code (*i.e.*, maintain source compatibility). However, recompilation of the code is expected.

The prior solutions that inspired our project failed to attain all of the above goals. Cyclone [21] was comprehensive with low overhead, but it required significant code modifica-

[1] The name SoftBoundCETS is a concatenation of the names of its components: SoftBound [35] and CETS [36].



```
struct A {     struct B {
 size_t t1;     size_t f1;
 void* ptr;     size_t f2;
 size_t t2;     size_t f3;
};             };

void foo(struct A * p) {
 struct B * q;
 q= (struct B *)p;
 ...
 q->f3 = ...;
}
```

Figure 1 Memory layout changes in a program with fat pointers. The code snippet also shows how a pointer involved in arbitrary type casts can overwrite the pointer metadata.

tions. CCured attained the goals of reasonable performance overhead and comprehensiveness for providing spatial safety. CCured maintained metadata with pointers by changing the representation of a pointer into a fat pointer (ptr, base, bound). It relied on a garbage collector to provide temporal safety.

However, there were two major drawbacks with the CCured approach: interfacing with libraries and type casts. With the use of a fat-pointer, CCured required marshalling/de-marshalling of pointers, which resulted in deep copies of data structures while interfacing with external libraries through wrappers. In the presence of unsafe type casts, the pointer metadata could be potentially overwritten (see Figure 1). CCured used a whole program inference to detect such pointers involved in casts (*e.g.*, WILD pointers), which prevented separate compilation and WILD pointers had higher performance overhead. To mitigate these issues, the programs had to be changed/rewritten to avoid such type casts or use run-time type information (RTTI) extensions.

We started our project to address compatibility issues with CCured in an effort to make it easily usable with large code bases while avoiding garbage collection. Our goal was to provide checked manual memory management in contrast to automatic memory management. The SoftBoundCETS project enforces memory safety by injecting code to maintain per-pointer metadata and checking the metadata before dereferencing a pointer. To provide compatibility, the per-pointer metadata is maintained in a disjoint metadata space leaving the memory layout of the program unchanged. We have implemented pointer-based checking in various ways (see Table 1): within the compiler, in hardware, and with hardware instructions for compiler instrumentation. We have used the compiler instrumentation prototype of SoftBoundCETS to compile more than a million lines of C code. The latest compiler prototype is available at `https://www.cs.rutgers.edu/~santosh.nagarakatte/softbound/`. Intel has recently announced Memory Protection Extensions [20], which provides hardware acceleration for a similar pointer-based compiler instrumentation for enforcing spatial safety.

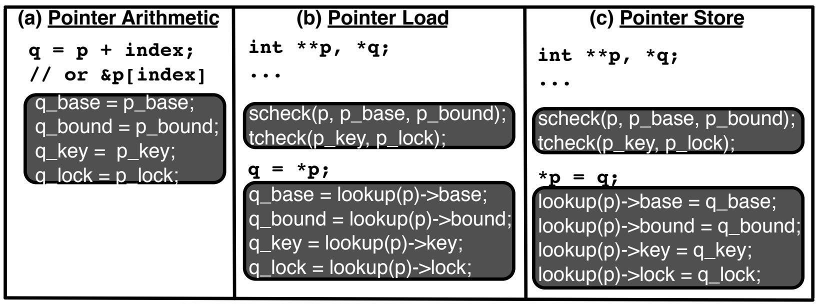
In the next section, we describe our approach, design decisions, various implementations and their trade offs. In Section 3, we describe the Intel Memory Protection Extensions (MPX) [20], similarities/differences between SoftBoundCETS and MPX while highlighting the pros and cons of the design decisions. We reflect on the lessons learned in Section 4.

2 Pointer-Based Checking with Disjoint Metadata

The SoftBoundCETS project uses a pointer-based approach, which maintains metadata with each pointer. The metadata provides the pointer a view of the memory that it can safely access. Although pointer-based checking has been proposed and investigated earlier [37, 2, 21, 52], the key difference is that our approach maintains the metadata disjointly in a shadow memory

Table 1 Various implementations of pointer-based checking developed as part of the SoftBound-CETS project, distinguished based on instrumentation method (Instr.), support for spatial safety, temporal safety, instrumentation of integer operations, support for check optimizations, performance overhead, and data structures used for disjoint metadata.

	Instr.	Spatial safety	Temporal safety	Instr. integer ops	Check Opts	Slow -down	Disjoint meta-data
HardBound [11]	Hardware	Yes	No	Yes	No	5-20%	Shadow
SoftBound [35]	Compiler	Yes	No	No	Yes	50-60%	Hash/ Shadow
CETS [36]	Compiler	No	Yes	No	Yes	30-40%	Trie
SoftBound-CETS [31]	Compiler	Yes	Yes	No	Yes	70-80%	Trie
Watchdog [32, 33]	Hardware	Yes/No	Yes	No	No	10–25%	Shadow
WatchdogLite [34]	Hybrid	Yes	Yes	No	Yes	10–20%	Shadow



```
q = p + index;
// or &p[index]
q_base = p_base;
q_bound = p_bound;
q_key =  p_key;
q_lock = p_lock;
```

```
int **p, *q;
...
scheck(p, p_base, p_bound);
tcheck(p_key, p_lock);
q = *p;
q_base = lookup(p)->base;
q_bound = lookup(p)->bound;
q_key = lookup(p)->key;
q_lock = lookup(p)->lock;
```

```
int **p, *q;
...
scheck(p, p_base, p_bound);
tcheck(p_key, p_lock);
*p = q;
lookup(p)->base = q_base;
lookup(p)->bound = q_bound;
lookup(p)->key = q_key;
lookup(p)->lock = q_lock;
```

Figure 2 (a) Pointer metadata propagation with pointer arithmetic, (b) metadata propagation through memory with metadata lookups on loads, and (c) metadata lookups with pointer stores.

region, which provides an opportunity to revisit pointer-based checking (generally considered invasive) for retrofitting C with practical memory safety satisfying the above three goals.

We describe the main design choices with our approach: (1) metadata for spatial safety, (2) metadata for temporal safety, (3) propagation of metadata, and (4) organization of the metadata shadow space. We also describe how it enforces comprehensive detection in the presence of type casts and provides compatibility with existing C code while supporting external libraries. Figure 2 and Figure 3 illustrate the pointer-based metadata, propagation and checking abstractly using pseudo C code notation. We use the term SoftBoundCETS interchangeably to refer to both our approach and the various prototypes built in this project shown in Table 1.

2.1 Spatial Safety Metadata

To enforce spatial safety, the base and bound of the region of memory accessible via the pointer is associated with the pointer when it is created. The base and bound are each typically 64-bit values (on a 64-bit machine) to encode arbitrary byte-granularity bounds information. These per-pointer base and bounds metadata fields are sufficient to perform a bounds check prior to a memory access (Figure 3d). This representation permits the creation of out-of-bounds pointers and pointers to the internal elements of objects/structs and arrays (both of which are allowed in C/C++).

2.2 Temporal Safety Metadata

To enforce temporal safety, a unique identifier is associated with each memory allocation (Figure 3a). Each allocation is given a unique 64-bit identifier and these identifiers are never reused. To ensure that this unique identifier persists even after the object's memory has been deallocated, the identifier is associated with all pointers to the object. On a pointer dereference, the system checks that the unique allocation identifier associated with the pointer is still valid.

Performing a validity check on each memory access using a hash table or a splay tree can be expensive [2, 22], so an alternative is to pair each pointer with two pieces of metadata: an allocation identifier – the *key* – and a *lock* that points to a location in memory called *lock location* [41, 52, 36, 5, 32]. The key and value at the lock location will match if and only if the underlying memory for the object is still valid (*i.e.*, it has not been deallocated). Rather than a hash table lookup, a dereference check then becomes a direct lookup operation – a simple load from the lock location and a comparison with the key (Figure 3c). Freeing an allocated region changes the value at the lock location, thereby invalidating any other (now-dangling) pointers to the region (Figure 3b). Because the keys are unique, a lock location itself can be reused after the space it guards is deallocated.

(a) Memory Allocations

```
p = malloc(size);
```

```
p_key = next_key++;
p_lock = allocate_lock();
*(p_lock) = p_key;
p_ base = p;
p_bound = p != 0 ? p+size: 0;
```

(b) Memory Deallocations

```
free(p);
```

```
*(p_lock) = INVALID;
deallocate_lock(p_lock);
```

(c) Temporal Check

```
tcheck(p_key, p_lock) {
  if (p_key != *(p_lock))
    raise exception();
}
```

(d) Spatial Check

```
scheck(p, p_base, p_bound, size) {
  if (p < p_base ||
      p + size >= p_bound)
    raise exception();
}
```

Figure 3 (a) Pointer metadata creation on memory allocations, (b) identifier metadata being invalidated on memory deallocations, (c) lock and key checking using identifier metadata, and (d) spatial check performed using bounds metadata.

2.3 Metadata Propagation

The metadata – base, bound, lock, and key – are associated with a pointer whenever a pointer is created. These metadata are propagated on pointer manipulation operations such as copying a pointer or pointer arithmetic (Figure 2a). The metadata for pointers in memory is maintained in a disjoint metadata space [11, 35, 32, 36, 17]. Figure 4 illustrates the metadata maintained in the disjoint metadata space. The disjoint metadata space protects the metadata from malicious corruption and leaves the memory layout of the program intact, retaining compatibility with existing code.

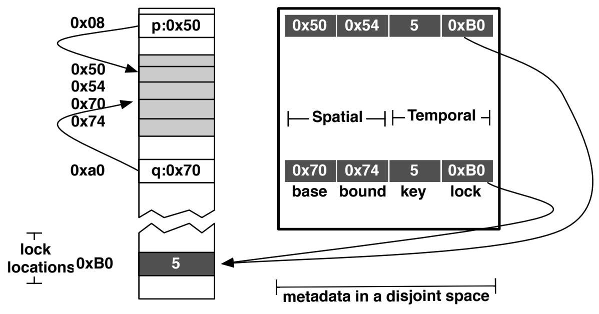


Figure 4 Metadata maintained with each pointer in memory with SoftBoundCETS. There are two pointers **p** and **q** which point to different sub-fields in the same allocation. Hence they have different bounds metadata but the same lock and key metadata.

The metadata is propagated with pointer arguments across function calls. If the pointer arguments are passed and returned on the stack, metadata for these pointer arguments are available through accesses to the disjoint metadata space. However, in practice propagating metadata with function calls is not straightforward. Arguments are often passed in registers according to the function calling conventions of most ISAs, and C allows variable argument functions. Furthermore, function calls (indirect calls) can be made through function pointers. Function pointers can be created through unsafe type casts to functions with incompatible types. Calling a function through such a function pointer should not be allowed to manufacture metadata, which may result in memory accesses to arbitrary memory locations.

We have explored two approaches to propagate metadata for pointer arguments: adding metadata as extra arguments [35, 36] and using a shadow stack for propagating metadata [31]. We pass metadata as extra arguments for functions that are not involved in type casts and used with indirect calls. We use a shadow stack for all other function calls including variable argument functions. The shadow stack provides a mechanism for dynamic typing between the arguments pushed at the call site and the arguments retrieved by the callee. The shadow stack is slower but it ensures that the callee never successfully dereferences a non-pointer value pushed by the caller in the call stack by treating it as a pointer value. An exception is triggered only when such pointers are dereferenced but not when they are created in accordance with the C specification.

2.4 Implications of Disjoint Metadata Design

To use disjoint metadata, we had to address the following questions: (1) How are memory locations mapped to their disjoint metadata? (2) Is metadata maintained for every memory location or only for memory locations with pointers?, and (3) Is metadata updated on every memory access?

Different implementations of the disjoint metadata space have different memory overhead and performance trade-offs. We have explored three implementations: shadow space (a linear region of memory) [11, 35, 32], a hash table [35], and a trie data structure [39, 36, 16]. Shadow space is beneficial when the size of the metadata is significantly smaller than the granularity of the memory region (*e.g.*, 1-bit of metadata for every 16 bytes). Hash tables can experience conflicts, and resizing the hash table causes overheads. In th end, we settled upon a two level trie data structure that maps the entire 64-bit virtual address space. Although mapping exists for the entire virtual address space, the entries in the trie are allocated only when the metadata is used.

Disjoint metadata accesses are expensive compared to fat pointers because additional instructions are required to translate a memory address to the corresponding metadata address. To reduce performance overheads, SoftBoundCETS maintains metadata in the disjoint metadata space only for pointers in memory and performs metadata loads (stores) only when the loaded (stored) value from (into) a memory location is a pointer as shown in Figure 2b and Figure 2c, respectively. The metadata for pointers in temporaries (registers) are maintained in temporaries (registers). Hence, the accesses to the disjoint metadata space typically occur with pointer-chasing code or linked data structures. Most programs have fewer pointers in memory compared to data and metadata is maintained only with pointers. Hence, the memory overhead is significantly lower than the worst-case 4$\times$ overhead (about 50% on average for SPEC benchmarks).

SoftBoundCETS initializes metadata to an invalid value when a pointer is created from an integer (in registers), which causes any subsequent check on such pointer dereferences to raise an exception. SoftBoundCETS does not access the disjoint metadata space when non-pointer values are written (read) to (from) memory. However, a side effect of instrumenting only pointer operations is that a pointer can be manufactured from a integer through memory. However, SoftBoundCETS allows such a dereference through a manufactured pointer only when the resulting pointer belongs to the same allocation and is within bounds of the object pointed by the pointer before the cast. We illustrate how SoftBoundCETS still provides comprehensive protection against memory errors with such type casts below.

2.5 Comprehensive Protection in the Presence of Type Casts

Our approach enforces comprehensive memory safety because (1) metadata is manipulated/accessed only through the extra instrumentation added, (2) metadata is not corrupted and accurately depicts the region of memory that a pointer can legally access, and (3) all memory accesses are conceptually checked before a dereference.

Unlike fat pointers, a store operation using a pointer involved in an unsafe type cast can only overwrite pointer values but not the metadata in the disjoint metadata space. When pointers involved in arbitrary casts are subsequently dereferenced, the pointer is checked with respect to its metadata. As the correctness checking uses the metadata to ascertain the validity of the memory access and the metadata is never corrupted, our approach ensures comprehensive detection of memory safety errors.

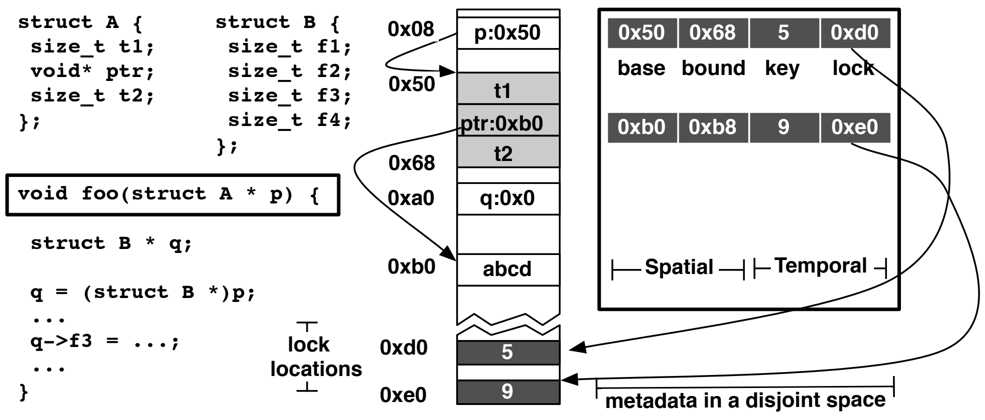
Figure 5 pictorially represents the disjoint metadata space as it is updated in the presence of arbitrary type casts. We will refer to this example later to illustrate a subtle interplay between type casts and comprehensive protection with Intel MPX. Figure 5(a) shows a program with two structure types `struct A` and `struct B` and a function `foo`, which has pointer `p` of type `struct A` as an argument. The location pointed by pointer `p` is allocated and resident in memory as shown in Figure 5(a). The sub-field `ptr` in the allocated memory region pointed by pointer `p` points to some valid memory location. Pointers `p` and `ptr` are resident in memory and have metadata in the disjoint metadata space as shown in Figure 5(a). Figure 5(b) depicts the execution of the program where it creates pointer `q` from pointer `p` through an arbitrary type cast. The metadata for pointer `q`, which is assumed to be resident in memory, is copied from pointer `p`.

The program writes integers to memory locations using pointer `q` as shown in Figure 5(c). As a result, `ptr` sub-field is overwritten with arbitrary non-pointer values. Our approach does not access the disjoint metadata space on integer operations. Hence, the metadata is not corrupted. When the program later tries to dereference `ptr` in memory, the dereference would be checked with respect to its metadata and memory safety errors would be detected.

2.6 Compatibility with Existing Code and Libraries

To enforce memory safety with legacy programs, SoftBoundCETS handles programs with type casts, supports separate compilation, and provides wrappers for commonly used libraries. The use of disjoint metadata enables comprehensive protection with type casts. Rewriting C source code to avoid type casts is not necessary.

SoftBoundCETS supports separate compilation because the instrumentation is local and does not require whole program analysis in contrast to CCured [37]. Separate compilation also allows creation of memory-safe libraries. In the absence of library sources, code compiled with SoftBoundCETS can interface with library code through wrappers for the exported library functions. In the absence of such wrappers, code instrumented with SoftBoundCETS will not experience false violations as long as the external libraries do not return pointers or update pointers in memory.



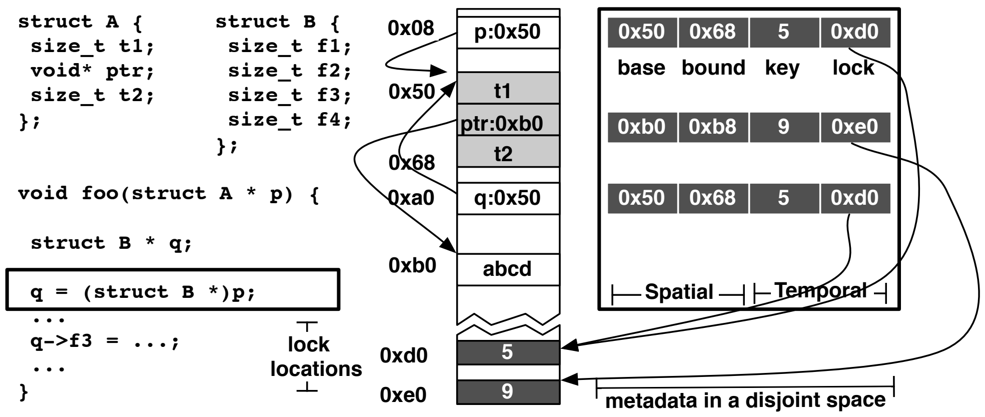
```
struct A {        struct B {
 size_t t1;        size_t f1;
 void* ptr;        size_t f2;
 size_t t2;        size_t f3;
};                 size_t f4;
                  };

void foo(struct A * p) {

 struct B * q;

 q = (struct B *)p;
 ...
 q->f3 = ...;
 ...
}
```

(a) Pointer p points to structure A in memory. The subfield of A points to another location in memory.



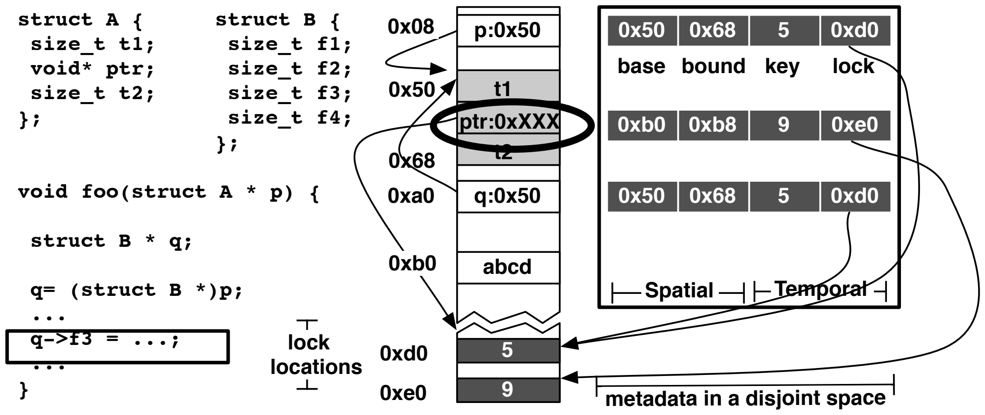
```
struct A {        struct B {
 size_t t1;        size_t f1;
 void* ptr;        size_t f2;
 size_t t2;        size_t f3;
};                 size_t f4;
                  };

void foo(struct A * p) {

 struct B * q;

 q = (struct B *)p;
 ...
 q->f3 = ...;
 ...
}
```

(b) Pointer q points to the structure pointed by p considering it to be of type struct B.



```
struct A {        struct B {
 size_t t1;        size_t f1;
 void* ptr;        size_t f2;
 size_t t2;        size_t f3;
};                 size_t f4;
                  };

void foo(struct A * p) {

 struct B * q;

 q= (struct B *)p;
 ...
 q->f3 = ...;
 ...
}
```

(c) Pointer q writes a junk value into the ptr field. However the metadata is untouched and still consistent.

Figure 5 This figure illustrates how disjoint metadata protects the metadata. Writes to memory locations involved in arbitrary type casts can only modify pointer values (*ptr* field in the *struct A*) but not the metadata. When the pointer *ptr* is dereferenced, the dereference will not be allowed and the memory safety violation would be caught.

When an external library returns or updates pointers in memory, wrappers provide the glue code between the instrumented code and the external library. Writing wrappers is easier with SoftBoundCETS compared to CCured [37] because it is not required to perform deep copies of data structures when memory-safe code interfaces with an external library. However, the disjoint metadata space should be updated in the wrapper whenever the external library updates pointers in memory. Although our approach makes it easier to write wrappers, it can be tedious and error prone for some libraries. We provide wrappers for the commonly used libraries (*e.g.*, Linux utilities, libc, and networking utilities) with our publicly available compiler prototype. Intel MPX further makes it even easier to incrementally deploy spatial safety checking by storing the pointer value redundantly in the metadata space to be permissive when non-instrumented code modifies the pointer and does not properly update the bounds metadata, which presents a compatibility/safety trade off as described in Section 3.

2.7 Efficient Instrumentation and Challenges

One of the key requirements for low performance overhead is efficient instrumentation and implementation of the pointer-based checking approach described above. Table 1 lists the numerous designs that we have explored in the hardware-software tool chain with different types of instrumentation, safety guarantees, and performance trade-offs [35, 36, 32, 33, 31, 10, 34]. Binary instrumentation and source-to-source translation have been popular with various proposals for enforcing partial memory safety. Source-to-source translation is widely used because pointer information is readily available in the source code. Binary-based techniques have been popular partly due to the availability of the dynamic binary translation tools like PIN [29] and Valgrind [40].

Our project benefited, albeit serendipitously, from the LLVM compiler that maintains pointer information from the source code in the intermediate representation (IR) [26]. Instrumenting within the compiler provides three main benefits: (1) checking can be performed on optimized code after executing an entire suite of conventional compiler optimizations, (2) pointers and memory allocations/deallocations can be identified precisely by leveraging the information available to the compiler, and (3) a large number of checks can be eliminated statically using check elimination optimizations.

Figure 6 presents the percentage runtime overhead of the latest compiler-based SoftBoundCETS prototype within the LLVM compiler (LLVM-3.5.0) over an un-instrumented baseline. SoftBoundCETS enforces comprehensive memory safety at 76% overhead on average with SPEC benchmarks. The overhead is significantly lower with I/O intensive benchmarks and various utilities such as OpenSSL, Coreutils, and networking utilities (less than 30%). A variant of pointer-based checking described above that propagates metadata with all pointer loads and stores but checks only store operations can enforce memory safety at 23% overhead on average with SPEC benchmarks. Store-only checking, which allows read operations on out-of-bound locations and with dangling pointers, is sufficient to prevent all memory corruption based security vulnerabilities.

The remaining performance overhead with the compiler instrumentation is attributed to the following sources: (1) spatial checks (5 x86 instructions), (2) temporal checks (3 x86 instructions), (3) metadata loads (approximately 14 x86 instructions), (4) metadata stores (approximately 16 x86 instructions), (5) shadow stack accesses, and (6) additional spills and restores due to register pressure.

To further reduce overheads with compiler-based approaches, we have explored memory safety instrumentation within hardware with varying amounts of hardware support [11, 32,

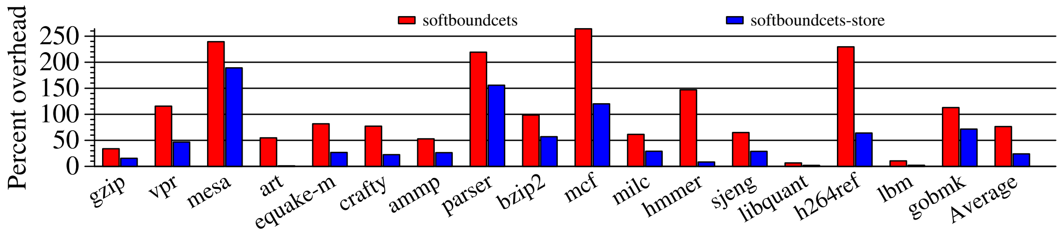


Figure 6 Runtime execution time overheads of the SoftBoundCETS compiler prototype with comprehensive memory safety checking (left bar of each stack) and while checking only stores (right bar of each stack) on a Intel Haswell machine. Smaller bars are better as they represent lower runtime overheads.

33]. HardBound [11] and Watchdog [32, 33] implicitly check every memory access within hardware by receiving information about pointer allocations from the runtime with additional instructions. The benefits of implicit checking within hardware are streamlined execution of checks and metadata accesses and the reduction in the number of register spills/restores by leveraging hardware registers for metadata.

In the final design point we proposed, hardware provides acceleration for spatial checks, temporal checks, metadata loads and metadata stores with ISA extensions and the compiler performs metadata propagation, performs check elimination, introduces check instructions and metadata access instructions into the binary [34]. This division of labor between the software stack and the hardware reduces the invasiveness of the hardware changes. With compiler-inserted checking instructions, we have demonstrated that comprehensive memory safety can be enforced at approximately 20% performance overhead on average with SPEC benchmarks. Further, these hardware instructions prevent all memory corruption vulnerabilities with store-only checking with approximately 10% performance overhead on average for SPEC benchmarks. Intel MPX has adopted a similar approach with compiler-based instrumentation with hardware instructions for enforcing spatial safety.

3 Intel Memory Protection Extensions

Intel has announced the specification of Memory Protection Extensions (MPX) [20] for providing hardware acceleration for compiler-based pointer-based checking with disjoint metadata slated to appear in the "Skylake" processors later in 2015. Like SoftBoundCETS with hardware instructions [34], MPX (1) provides hardware acceleration for compiler-based pointer-based checking, (2) uses disjoint metadata for the pointers in memory, and (3) provides ISA support for efficient bounds checking. Significantly, MPX does not address use-after-free errors. MPX uses the same basic approach pioneered by the SoftBoundCETS project, and this section describes some of the differences.

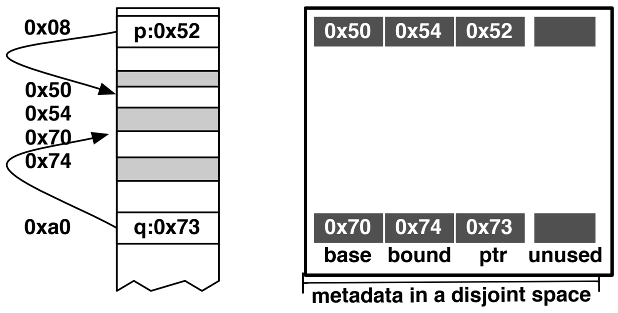


Figure 7 MPX maintains four pieces of metadata with each pointer: base, bound, the pointer redundantly in the metadata space and the fourth unused space.

3.1 MPX Instructions and Operation

The Intel MPX extension provides new instructions that a compiler can insert to accelerate disjoint per-pointer metadata accesses and bounds checking. MPX aims for seamless integration with legacy and MPX code with minimal changes to the source code. One design goal of MPX is to allow binaries instrumented with MPX to execute correctly (but without performing bound checks) on pre-MPX hardware. MPX achieves this goal by selecting opcodes that are NOOPs (no operation) on existing x86 hardware for the new instructions. Hence, a MPX binary can run on MPX-enabled machines and non-MPX machines.

MPX introduces four new 128-bit bound registers (B0-B3). MPX provides the following instructions: (1) `BNDCL` – check pointer with its lower bound, (2) `BNDCU` – check pointer to the upper bound, (3) `BNDSTX` – store metadata to the disjoint metadata space, (4) `BNDLDX` – load metadata from the disjoint metadata space, and (5) `BNDMK` – to create bounds metadata for a pointer. MPX also extends function calling conventions to include these bound registers.

A key innovation of MPX beyond our work is the support for incremental deployment of bounds checking. MPX redundantly stores the pointer value in the metadata space along with the base and bound metadata as shown in Figure 7. When a pointer is loaded from memory, the corresponding metadata is loaded from the disjoint metadata space and the loaded pointer is compared with the pointer redundantly stored in the metadata space. If the pointer in the metadata space and the pointer loaded do not match (typically occurs when the non-instrumented code modifies the pointer and does not properly update the bounds metadata), then MPX allows the pointer to access any memory location by un-bounding it. MPX design adopts the compatible-but-unsafe model, allowing best-effort checking of instrumented code until all code has been recompiled for MPX.

3.2 Type Casts and Comprehensiveness with Intel MPX

The Intel MPX's support for incremental deployment of bounds checking results in the loss of comprehensiveness in the presence of insidious type casts from integers to pointers either directly or indirectly through memory. Particularly, the arbitrary pointer manufactured through type casts in Figure 5(c) will be allowed by MPX to access any location in memory because (1) the pointer in the metadata space is not updated during an integer store, (2) the pointer loaded and the pointer in the metadata space would mismatch on a metadata load, and (3) the result is an un-bounded pointer. The compiler can identify the occurrence of such type casts either implicitly or explicitly and warn the programmer about them. Nevertheless, MPX is a step in the practical deployment of memory safety checking in production. Although Intel MPX may appear permissive, the spatial safety protection provided by Intel MPX is similar or stronger to the protection provided by fat-pointer approaches while easing the problem of interfacing with external libraries. Although the current ISA specification does not disable this behavior with un-bounding, Intel could easily add a stricter mode that changes this behavior to be safer by default.

4 Reflections on Memory Safety Enforcement

We briefly describe our experience in enforcing memory safety with large code bases and reflect on the performance overheads, various aspects of the language, implementation, and the hardware-software stack. We describe changes to the language and the hardware-software stack, which can potentially make memory safety enforcement inexpensive while being expressive for C's domain.

4.1 Performance Overheads

Low performance overhead is one of the key requirements for adoption of memory safety enforcement techniques. Our experiments indicate that compiler instrumentation with hardware acceleration in the form of new instructions can provide comprehensive memory safety with approximately 10–20% performance overhead for computation-based SPEC benchmarks and less than 10% I/O intensive utilities and applications. These overheads are within the reach of the acceptable threshold for a large class of applications. For example, a large fraction of C code is compiled at the `O2` optimization level because the code is unstable with higher optimization levels (likely due aggressive optimizations by the compiler in the presence of undefined behavior and implementation-dependent behavior [51]). For many "fast enough" applications, an additional 5–10% overhead for enforcing memory safety will not be perceivable to the user. The store-only instrumentation within the compiler with hardware acceleration can prevent all memory corruption based security vulnerabilities is an attractive option for reducing overheads further. Store-only checking provides much better safety than control-flow integrity with similar performance overheads [24, 15]. An application's observed performance overhead depends on the memory footprint, amount of pointer-chasing code, and the locality in the accesses. Hence, it remains to be seen what performance cost users will tolerate in exchange for the security and safety of the computing ecosystem.

We have explored only simple check optimizations with our prototype. A wide range of check optimizations based on loop invariant code motion and loop peeling can reduce overhead. For example, compiler optimizations based on weakest preconditions have reduced the performance overhead of spatial safety enforcement by 37% for the SPEC benchmarks [13]. Moreover, simple code transformations can significantly reduce performance overhead. For example, a small change to the data structure used by the SPEC benchmark `equake` not only improved the baseline execution time performance by 60% but also reduced overhead of memory safety with SoftBoundCETS from 3× to 30%.

4.2 Implementation Specific C Dialects

Other than performance, one of the biggest impediments in the adoption of memory safety enforcement techniques is the pervasiveness of implementation-specific C dialects. In contrast to the C standard, a fraction of the C code base depends on the behaviors provided by contemporary C implementations [4]. Are deviations from the C standard acceptable? There is no unanimous answer but such reliance on implementation specific behaviors result in non-portable code and can be exploited aggressively by an optimizing compiler (see classification of undefined behaviors in LLVM in [28]). We highlight examples that are arguably not in conformance with the C standard, which can cause false violations with our approach.

4.2.1 Narrowing of Bounds for Sub-Objects

When the program creates a pointer to a sub-field of an aggregate type, should the bounds of the resultant pointer be just the sub-field or the entire aggregate object? Our approach provides the ability to easily narrow the bounds of pointers, which in turn allows us to prevent internal object overflows. When instructed to check for overflows within an object, we shrink the bounds of pointer when creating a pointer to a field of a *struct* (*e.g.*, when passing a pointer to an element of a *struct* to a function). In such cases, we narrow the pointer's bounds to include only the individual field rather than the entire object.

Although most programmers expect shrinking of bounds, it can result in false violations for particularly pathological C idioms. For example, a program that attempts to operate on three consecutive fields of the same type (*e.g.*, x, y, and z coordinates of a point) as a three-element array of coordinates by taking the address of x will cause a false violation. Another example of an idiom that can cause false violations comes from the Linux kernel's implementation of generic containers such as linked lists. Linux uses the ANSI C *offsetof()* macro to create a *container_of()* macro, which is used when creating a pointer to an enclosing container *struct* based only on a pointer to an internal *struct* [23]. Casts do not narrow bounds, so one idiom that will not cause false violations is casting a pointer to a *struct* to a *char** or *void**.

Another case in which we do not narrow bounds is when when creating a pointer to an element of an array. Although tightening the bounds is such cases may often match the programmer's intent, C programs occasionally use array element pointers to denote a sub-interval of an array. For example, a program might use *memset* to zero only a portion of an array using *memset(&arr[4], 0, size)* or use the *sort* function to sort a sub-array using *sort(&arr[4], &arr[10])*. We have found these assumptions and heuristics match the source code we have experimented with, but programmer can control the exact behavior with compiler command line flags.

4.2.2 Integer Arithmetic with Integer to Pointer Casts

Masking pointer values is a common idiom in low-level systems code to use the lower order bits of the pointer representation to store additional information with aligned pointers. The last three bits are always zero in the pointer representation on a 64-bit machine with an aligned pointer. These bits can be used to store tags, which can maintain additional information about the pointer. Such operations would involve casting pointers to integers, masking the values, and casting integers back to pointers. Such casts may also be performed after arbitrary integer arithmetic. Occasionally, pointers can be created from integers due to (old) interfaces to utilities. For example, the interface to the system functions in Microsoft Windows on 32-bit systems (*Kernel32.dll*) used integers for pointers, which resulted in pointer to integer and integer to pointer casts. Our approach will set the metadata of the pointer cast from an integer to be invalid and raise an exception on dereferencing such pointers. Avoiding such exceptions require explicit setting of bounds by the programmer. If such cast operations occur through memory, our approach allows dereference of such pointers as along as the pointer type cast from an integer belongs to the same allocation and is within bounds of the object pointed by the pointer before the cast.

4.3 Interoperability and Engineering

Pointer-based checking is more invasive compared to other approaches as each pointer operation needs to be tracked, propagated with metadata, and checked. Hence, significant engineering is required to make it practical. AddressSanitizer [46], which uses a tripwire approach, is less invasive but still required significant engineering for use with code bases of the Chromium browser and other utilities. Our experience indicates that significant engineering and enabling memory safety checking using a simple compile time flag is crucial for adoption. Maintaining interoperability with un-instrumented code is necessary to encourage incremental deployment. Intel MPX with its support for incremental deployment will likely increase the adoption of memory safety enforcement with pointer-based checking.

4.4 Language Design Considerations

We provide some suggestions for the future standards of the C programming language to ease memory safety enforcement. One way to ease memory safety enforcement is by restricting type casts. Enforcing spatial safety with strong typing is easier compared to weak typing (similar to observations in CCured [37]). More restrictive type cast rules imply easier spatial safety enforcement because only array bounds need to be checked. Unfortunately, implementing large software systems (with some form of polymorphism/reusable code) requires type casts in C. Object oriented variants of C such as C++ have reasonable subsets that can enforce spatial safety while being suitable for building infrastructure/systems code [10].

Even when type casts are necessary, preventing the creation of pointers from non-pointers will enable easier memory safety enforcement for a large class of applications compared to the state-of-the-art now. When creation of pointers from non-pointers is essential, explicitly creating a separate data type other than integers will ease the enforcement of memory safety.

The C language conflates arrays and pointers, which requires bounds checking on every memory access. Annotations that identify non-array pointers, which point to a single element in contrast to an array of elements, can avoid such checking. In the case of linked data structures, such annotations will significantly reduce spatial safety overheads by reducing not only the overheads due to checks but also disjoint metadata accesses. The overhead with Intel MPX for spatial safety can be reduced significantly by revisiting annotations similar to Cyclone [21] and Deputy [6].

4.5 Implementation Considerations

We highlight the implementation considerations for efficient pointer-based checking with compiler assisted instrumentation and hardware acceleration similar to MPX.

Type information in the compiler is known to benefit numerous domain-specific instrumentation [30]. Maintaining information about types in the program within the compiler tool chain will ease the enforcement of memory safety especially with Intel MPX. If maintaining full type information is infeasible, the compiler should at least maintain information about pointer and non-pointer types. This information must be preserved and maintained with compiler optimizations, which would enable memory safety instrumentation on optimized code. The LLVM compiler maintains best effort type information, which helped the SoftBoundCETS project to perform pointer-based checking within the compiler with low performance overhead.

Automatic memory management through garbage collection is an attractive alternative to checked manual memory management. Automatic memory management can increase programmer productivity as they do not have to manually manage memory. However, garbage collection's effectiveness decreases in the presence of weak typing with C and can cause memory leaks. The pointer-based metadata can be used to perform precise garbage collection [10, 4]. In contrast to identifier metadata for enforcing temporal safety, spatial safety metadata can be used enforcing temporal safety by setting the bounds of the deallocated pointer and all its aliased pointers in the metadata space to a invalid value [47]. Any subsequent dereference of such a deallocated pointer will raise an exception as the bounds metadata is invalid. Similar dangling pointer nullification has been proposed to detect use-after-free errors [27]

4.6 Multithreading

A pointer-based approach should ensure atomicity of the checks, metadata accesses, and the memory access in multithreaded programs, which can be ensured by relying on the compiler. If a pointer operation, temporal safety check and the metadata accesses occur non-atomically, interleaved execution and race conditions (either unintentional races or intentional races) can result in false violations and miss true violations. To avoid them, the compiler instrumentation must ensure that: (1) a pointer load/store's data and metadata access execute atomically, (2) checks execute atomically with the load/store operation, and (3) allocation of metadata is thread safe. The compiler can satisfy requirement #3 by using thread-local storage for the identifiers. The compiler can ensure requirements #1 and #2 for data-race free programs by inserting metadata access and check instructions within the same synchronization region as the pointer operation. For programs with data races, if the compiler can perform the metadata access as an atomic wide load/store and perform the temporal check after the memory operation, then the approach can detect all memory safety violations (but can experience false violations). Alternatively, the compiler can either introduce additional locking or use best-effort bounded transactional memory support in latest processors [19, 25](*e.g.*, Intel's TM support in Haswell) to avoid false violations.

5 Conclusion

Memory safety enforcement is the job of the language implementation. A language implementation that is compatible with the C standard can enforce comprehensive memory safety at low overheads with pointer-based checking. Restrictions on the creation of pointers from non-pointers, annotations distinguishing pointers to arrays from pointers to non-array objects, and preserving the pointer information within the compiler can ease the job of enforcing memory safety.

We conclude that it is possible to enforce comprehensive memory safety with low performance overheads using pointer-based checking with disjoint metadata and an efficient streamlined implementation. From our experience in building pointer-based checking in various parts of the tool chain, we anticipate that hardware acceleration (like Intel MPX) will reduce the performance overheads significantly while reducing the hardware changes. Intel MPX with support for incremental deployment of memory safety checking and its store-only instrumentation will likely make always-on deployment of spatial safety enforcement a reality.

Acknowledgments. We thank Vikram Adve, Emery Berger, John Criswell, Chrisitan Delozier, Joe Devietti, Michael Hicks, David Keaton, Peter-Michael Osera, Stelios Sidiroglou-Douskos, Nikhil Swamy, and Adarsh Yoga for their inputs during the course of this project. This research was funded in part by Intel Corporation, and the U.S. Government by DARPA contract HR0011-10-9-0008, ONR award N000141110596 and NSF grants CNS-1116682, CCF-1065166, CCF-0810947, and CNS-1441724. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

1 Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

2 Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

3 Stephen Bradshaw. Heap spray exploit tutorial: Internet explorer use after free aurora vulnerability. `http://www.thegreycorner.com/2010/01/heap-spray-exploit-tutorial-internet.html`.

4 David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

5 Weihaw Chuang, Satish Narayanasamy, and Brad Calder. Accelerating meta data checks for software correctness and security. *Journal of Instruction Level Parallelism*, 9, June 2007.

6 Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, 2007.

7 Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

8 Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the Foundations of Intrusion Tolerant Systems*, pages 227–237, 2003.

9 John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

10 Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M.K. Martin, and Steve Zdancewic. IroncladC++: A Library-augmented Type-safe Subset of C++. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'13, 2013.

11 Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

12 Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 162–171, 2006.

13 Yulei Sui Ding Ye, Yu Su and Jingling Xue. Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *Proceedings of the 25th IEEE Symposium on Software Reliability Engineering*, 2014.

14 Frank Ch. Eigler. Mudflap: Pointer Use Checking for C/C++. In *GCC Developer's Summit*, 2003.

15 Isaac Evans, Samuel Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, 2015.

16 Kittur Ganesh. *Pointer Checker: Easily Catch Out-of-Bounds Memory Accesses.* Intel Corporation, 2012. `http://software.intel.com/sites/products/parallelmag/singlearticles/issue11/7080_2_IN_ParallelMag_Issue11_Pointer_Checker.pdf`.

17 Saugata Ghose, Latoya Gilgeous, Polina Dudnik, Aneesh Aggarwal, and Corey Waxman. Architectural support for low overhead detection of memory viloations. In *Proceedings of the Design, Automation and Test in Europe*, 2009.

18 Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter Usenix Conference*, 1992.

19 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

20 Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, 319433-022 edition, October 2014. `https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf`.

21 Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

22 R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1997.

23 Greg Kroah-Hartman. The linux kernel driver model: The benefits of working together. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, Inc., June 2007.

24 Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

25 James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

26 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, page 75, 2004.

27 Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Internet Society Symposium on Network and Distributed Systems Security*, 2015.

28 Nuno Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

29 Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.

30 Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995.

31 Santosh Nagarakatte. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*. PhD thesis, University of Pennsylvania, 2012.

32 Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

33 Santosh Nagarakatte, Milo M K Martin, and Steve Zdancewic. Hardware-enforced comprehensive memory safety. In *IEEE MICRO 33(3)*, May/June 2013.

34 Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO'14*, page 175, 2014.

35 Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

36 Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, 2010.

37 George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

38 Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2004.

39 Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 65–74, 2007.

40 Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

41 Harish Patil and Charles N. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. In *Software Practice and Experience 27(1)*, pages 87–110, 1997.

42 J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. In *IEEE Security and Privacy 2(4)*, pages 20–27, 2004.

43 Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical report, SRI International, February 2009.

44 Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, pages 291–302, 2005.

45 Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 159–169, February 2004.

46 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

47 Matthew S. Simpson and Rajeev Barua. Memsafe: Ensuring the spatial and temporal memory safety of c at runtime. In *Software Practice and Experience, 43(1)*, 2013.

48 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

49 Guru Venkataramani, Brandyn Roemer, Milos Prvulovic, and Yan Solihin. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, pages 273–284, 2007.

50 David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2000.

51 Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating System Principles*, 2013.

52 Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126, 2004.

53 Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 307–316, 2003.

New Directions for Network Verification

Aurojit Panda[1], Katerina Argyraki[2], Mooly Sagiv[3], Michael Schapira[4], and Scott Shenker[5]

1 UC Berkeley, US
2 EPFL, CH
3 Tel Aviv University, IL
4 Hebrew University of Jerusalem, IL
5 UC Berkeley and ICSI, US

Abstract

Network verification has recently gained popularity in the programming languages and verification community. Much of the recent work in this area has focused on verifying the behavior of simple networks, whose actions are dictated by static, immutable rules configured ahead of time. However, in reality, modern networks contain a variety of middleboxes, whose behavior is affected both by their configuration and by mutable state updated in response to packets received by them. In this position paper we critically review recent progress on network verification, propose some next steps towards a more complete form of network verification, dispel some myths about networks, provide a more formal description of our approach, and end with a discussion of the formal questions posed to this community by the network verification agenda.

1998 ACM Subject Classification C.2.6 Internetworking, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Middleboxes, Network Verification, Mutable Dataplane

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.209

1 Introduction

Verification – by which we mean the general practice of checking the correctness of a computer-based system before it is put into use – was first developed to check the correctness of hardware, and is now increasingly used in the software development process. While networks have been around for many decades and are now an essential piece of our computational infrastructure, only recently has verification been applied to ensure their correctness.[1] As a result, there is now a growing literature on systems that can verify that the current or proposed network configuration (as represented by router forwarding tables) obey various important invariants (such as no routing loops or dead-ends). These systems – which allow network operators to verify that their networks will operate correctly, in terms of some well-defined invariants – represent a valuable, and long overdue, step forward for networking, which for too long was satisfied with not only *best-effort* service but also *best-guess* configuration. In this position paper we critically review this recent progress, propose some next steps towards a more complete form of network verification, dispel some myths about networks, provide a more formal description of our approach, and end with a discussion of the formal questions posed to this community by the network verification agenda.

[1] There has been much work on verifying network protocols and their implementations, but until recently almost none on verifying a given network configuration.

In the rest of this section, we provide some necessary background on networks and the current verification techniques.

The application of verification to the forwarding behavior of networks coincided with the rise of Software-Defined Networking (SDN)[2]. While not essential for the use of verification in networks, SDN provides a useful platform on which to deploy these tools so we discuss verification within the SDN context. Networks are comprised of two planes: the *data plane* which decides how packets are handled locally by each router (based on the local forwarding state and other information, such as state generated by previous packets), and the *control plane* which is a global process that computes and updates the local forwarding state in each router. In legacy networks, both planes are implemented in routers (with the data plane being the forwarding code or datapath, and the control plane being the global routing algorithm), but in SDN there is a clean separation between the two planes. The SDN control plane is logically centralized, and implemented in a few servers (called controllers) that compute and then install the necessary forwarding state. SDN-controlled routers only implement the data plane, executing a very simple datapath (OpenFlow [14]) in which the routing state is a set of $\langle match, action\rangle$ flow entries: all packets with headers matching the *match* entry are subject to the specified *action* which is often either to forward out a specific port (perhaps with a slightly modified header) or to drop the packet. Each router in the network is configured with a table containing flow entries (henceforth referred to as the router's *configuration*), and the network configuration is the set of flow tables for all switches and routers in the network.

The first wave of verification tools [12, 11, 10, 9] analyzed the global behavior of a network made up of switches obeying this simple forwarding model. As a packet travels through the network, its next-hop is dictated by the routing state in the current router; thus, this network-wide behavior can be thought of as the composition of the routing state in each router. These early verification tools would take a snapshot of network state (either that which is already in the network, or that which the control is poised to insert into the network) and then verify whether some basic invariants held. These invariants (which are specified by the network operator) are typically quite simple and few in number: reachability (*e.g.,* packets from host A can reach host B), isolation (*e.g.,* packets from host A cannot reach host B), loop-freedom (no packet enters into an infinite loop), and no dead-ends (no packet arrives at a router which cannot forward it to another router or to the end-destination). Subsequent network verification tools (*e.g.,* [5, 2, 17, 18]) make the same assumptions about the datapath, but generalize along various other dimensions.

All of these tools leverage the fact that the simple forwarding model renders the datapath *immutable*; by this we mean that the forwarding behavior does not change until the control plane explicitly alters the routing state (which happens on relatively slow time scales). Thus, one can verify the invariants before each control-plane-initiated change and know that the network will always enforce the operator-specified invariants.

While the notion of an immutable datapath supported by an assemblage of routers makes verification tractable, it does not reflect reality. Modern enterprise networks are comprised of roughly $2/3$ routers[3] and $1/3$ *middleboxes* [23]. Middleboxes – such as firewalls, WAN optimizers, transcoders, proxies, load balancers, intrusion detection systems (IDS) and the like – are the most common way to insert new functionality in the network datapath, and

[2] Previous work including FANG [13], SPAN [6], Margrave [19], etc. has looked at verifying firewall policies for enterprise networks, but made no attempt to verify the actual forwarding behavior.

[3] In this paper we do not distinguish between routers and switches, since they obey similar forwarding models.

are commonly used to improve network performance and security.[4]

Just as the configuration of a router is the state it uses to make forwarding decisions, the configuration of a middlebox is its set of policies (*e.g.,* drop all Skype packets). Different middleboxes are capable of implementing widely different policies (*e.g.,* an application firewall can drop Skype packets, a cache on the other hand cannot) and hence middlebox configuration cannot easily be fit into a flow-table like abstraction shared by all middleboxes. The configuration of a network is the configurations of all of its routers, all of its middleboxes, and the topology of the physical network connecting these elements. The goal of verification is to ensure that a given network configuration supports a given invariant.

While useful, middleboxes are a common source of errors in the network [21], with middleboxes being responsible for over 40% of all major incidents in networks. Thus, one cannot ignore middleboxes when verifying network configurations.

However, middleboxes do not adhere to the simple forwarding model in routers. Many middleboxes have a *mutable* datapath, in which the handling of a packet depends not just on immutable forwarding state, but also on the sequence of previously encountered packets (*e.g.,* a firewall allows packets from a flow into a network only if it has previously seen an outgoing packet from that flow). This dependence on previously seen packets renders the datapath quite mutable (with state changing at packet timescales). This prevents the use of current verification techniques, because the control over packet behaviors is no longer centralized in the control plane, but can depend on the history of packets seen by each middlebox.

Thus, we must find a way to verify network behavior in the presence of middleboxes. The more complex forwarding model supported by middleboxes renders the network Turing-complete. Furthermore, since the forwarding behavior can depend arbitrarily on packet history, the verification technique must also cope with reasoning about a potentially unbounded state space. Thus, verification techniques that cope with middleboxes look much more like general program verification than the current generation of specialized network verification techniques. The next generation of network verification tools must address two main technical challenges:

- How do we model these mutable datapaths so that verification is tractable?
- How can we feasibly analyze a network made up of these mutable datapaths?

We address these two challenges in the following sections, and then discuss how to formalize this approach and end by describing a set of open questions.

2 How to Model Middleboxes?

The natural approach for verifying mutable datapaths would be to apply standard program verification techniques to the code in each middlebox (and then extend this to the network as a whole, which is the problem we address in the next section). The practical problem with this approach is that middlebox code is typically proprietary, and any approach that relies on middlebox vendors releasing their code is doomed to fail. Moreover, there is a deeper conceptual problem with this approach. The invariants specified by network operators often use abstractions, such as user identity, host identity, application-type (of the traffic), and whether or not the traffic is “suspicious” (*e.g.,* after deep packet inspection). In fact, recent efforts to build policy languages are built around a similar set of abstractions [20].

[4] We should note that the network function virtualization (NFV) movement is moving middleboxes out of separate physical machines and into VMs that can be hosted on a cluster of servers; however, nothing in the move from physical to virtual middleboxes changes our story.

The correctness of these abstractions often *cannot* be fundamentally verified (*e.g.,* a middlebox in the middle of the network cannot always know for sure which host the packet came from given the various forms of spoofing or relaying available) or even precisely defined (*e.g.,* what is "suspicious" traffic?). Yet these abstractions are quite useful (and already widely used) in practice, and operators are willing to live with their approximations (*e.g.,* various techniques can be used to limit spoofing so that in some contexts host identification can rely on IP or MAC addresses without great risk).

To allow reasoning in terms of high-level abstractions without worrying about the various approximations that go into their definition, we model middleboxes in two parts: a reasonably simple abstract model that captures the action of a middlebox in terms of high-level primitives and an *oracle* that is described by the set of abstractions it supports. The oracle maps packets to one or more abstract classes (*e.g.,* this packet is from a Skype flow from host A and user X to host B and user Y), while the abstract model describes how the middlebox forwards packets belonging to different abstract classes (*e.g.,* a middlebox might be configured to drop all suspicious packets, or only allow packets from host A to reach host B but no other hosts). For instance, for an IDS that identifies suspicious packets and forwards them to a scrubbing box, the oracle part of the model determines which packets are suspicious and the abstract model is what dictates that such packets are forwarded to a scrubbing box.

The oracles in different middleboxes may use very different techniques to implement these abstractions. While operators care about the quality of this mapping, the goal of our network verification approach is to check that a network configuration correctly enforces invariants *assuming that the oracles are correct.* Also, different oracles may support different sets of abstractions (*e.g.,* some firewalls may be able to identify Skype traffic, and others not), and this would be described as part of the middlebox model.

In contrast, the abstract models are fairly generic in the sense that the abstract model of a firewall applies to most firewalls. The degree of detail in these abstract models depends on the invariants one wants to check. The basic network invariants of reachability and isolation only require that the abstract model describe the forwarding behavior (*e.g.,* if and where each packet is forwarded). Our initial target is verifying these basic invariants, as these properties are by far the most important safety property provided by networks. If one wants to support performance-oriented invariants, then the abstract model must include timing information (*e.g.,* what packet delays might occur), and other extensions are needed to consider invariants that address simultaneity (two properties always hold at the same time). For simplicity, we do not consider such extension here.

Separating middlebox models into an abstract model and an oracle has several advantages.

- It captures the fact that there are a limited number of middlebox "types", with many implementations of each. The abstract model applies to all of these implementations (and is fairly simple in nature), and implementations mainly differ in the abstractions and features offered by the oracle. Thus, our verification approach – which asks whether invariants are enforced assuming the oracles are right – can be applied independent of the implementations.
- It differentiates between improvements in the oracle (*e.g.,* adding new abstractions to recognize application types), which is what consumes the bulk of the development effort, and verifying correctness of the network configuration.
- It could change the vendor ecosystem by allowing (or requiring) vendors to provide the abstract model (and a description of which abstractions their oracle supports) along with their middlebox. Network operators could then perform verification, while vendors could keep their implementations private.

- While it would be useful for vendors to *verify* that their code obeys the abstract model (using standard code verification methods), what makes this approach particularly appealing is that vendors and operators alike can *enforce* that the middleboxes obey the abstract model. The abstract models (and we have built several of them) are so simple that they execute much faster than the actual middlebox implementation, so one can run these abstract models in parallel and ensure that the middleboxes take no action that does not obey the abstract model.

3 Network-Wide Verification

Modeling individual middleboxes is only the first step; our ultimate goal is to verify network-wide invariants in a network containing middleboxes and routers. There are numerous technical challenges (such as how to deal with loops), but in this short position paper we focus on the most challenging one: how can we feasibly perform verification in very large networks (containing on the order of thousands of middleboxes and routers)? To see why this is hard, consider the case of an isolation invariant (packets from host A cannot reach host B) in a network with many middleboxes. Since middlebox behavior depends not just on their configuration but on the packet history they've seen (since their datapath is mutable), verifying that this invariant holds, even if we ignore the possibility of packet loops, involves checking that there is no sequence of packets – involving packets sent from anywhere in the network at any time – that includes a packet from host A reaching host B.

Without additional assumptions, this is not feasible as the forwarding of packets from host A to host B can arbitrarily depend on other hosts (*e.g.,* on whether some other host C previously sent packets to another host D). However, the most common classes of middleboxes have an important property: the handling of a packet from host A to host B depends only on the sequence of packets (seen by the middlebox) between hosts A and B. That is, many mutable datapaths exhibit a useful kind of locality. For instance, in IP firewalls, the middlebox tracks established connections, and allows packets to pass if they are either explicitly permitted by policy or belong to an established connection. A connection between host A and B can only be established by host A and B. We therefore do not need to consider the actions of any other hosts in the network. We call middleboxes whose behavior for a pair of hosts depends only on the traffic sent between these hosts "RONO (Rest-Of-Network Oblivious) middleboxes". See Section 5 for a formal definition of RONO. RONO is not a rare property; in fact, most middleboxes (including firewalls, WAN optimizers, load balancers, and others) are RONO. Moreover, one can verify whether a middlebox is RONO by statically analyzing its abstract model.

Note that in many practical cases, the composition of RONO middleboxes is also RONO. As a result, in a network containing only RONO middleboxes we can verify reachability properties on a small subset of the network (the path between two hosts) and these properties would equivalently hold in the context of the wider network. We have leveraged this fact to verify correctness of a network containing 30,000 middleboxes in under two minutes.

4 Common Myths about Networks and SDN Verification

We next highlight some common myths about SDN networks and contrast them with the reality of today's networks.

Myth #1: *SDN networks only have controllers and OpenFlow routers, with all complicated (particularly mutable) packet processing done at the controller.* The early SDN literature [4, 16] showed examples where anything expressible using OpenFlow rules was pushed down to routers, while anything more complex was implemented in the controller. Complicated functionality included both complex processing that could not be performed at routers and simple tasks that required mutability (*e.g.,* learning switches and stateful firewalls). However, doing this processing at the controller does not scale and, in reality, middleboxes are used to provide most of the processing functions not implementable on routers, and most routers provide some mutable behavior (e.g., learning switch).

Myth #2: *Centralization, as provided by SDN, is what makes current network verification efforts possible.* Centralization is neither necessary nor sufficient for network verification. Not necessary: Verification in a network with immutable datapaths only requires being able to access router forwarding state, and current commercial network verification efforts can do this in legacy networks by using commonly available commands to read this forwarding state. Not sufficient: Regardless of SDN, current network verification efforts cannot verify networks that have middleboxes with mutable datapaths (which describes almost all real networks).

Myth #3: *Middleboxes are an aberration that will be eliminated by the rise of SDN.* Quite the opposite is true. Not only are middleboxes here to stay, but SDN itself has been evolving to incorporate middleboxes [22]. Furthermore, recently there has been an effort to move middleboxes from dedicated hardware (which is time-consuming to deploy) into virtual machines that can be deployed on quicker timescales, on existing hardware, and at lower cost. This effort is generally described as NFV (network function virtualization), and has gained significant traction commercially (comparable to or exceeding that of SDN), and recent efforts at defining a common configuration language, *e.g.,* Congress‘[20], treat middleboxes (virtualized or not) as first-class network citizens.

Myth #4: *We should write all network code and configuration in declarative languages, because their use makes verification easy.* In general, reasoning about declarative languages is undecidable [7]. It is true that verification is easy for declarative programs that do not use recursive rules (e.g., Congress [20] or NLOG programs), even in the presence of mutable states. But then, verification is equally easy for imperative programs (e.g., Python, Java, or C programs) that honor certain restrictions, e.g., do not use loops. So, in the end, it is unclear that declarative languages can make a practical difference in verification. Some argue that declarative programs are easier to read and debug, once a programmer gets used to them. On the other hand, their readability becomes questionable in the presence of side-effects.

Once one discards these myths, it becomes clear that network-verification efforts must directly confront the presence of mutable datapaths. While the approach described here may not be optimal, it is currently the only one that confronts the reality of today's and tomorrow's networks. It is time to take the next step in network verification.

5 Formalizing the Mutable Data Plane

The previous sections argued why a new approach to network verification is needed and briefly outlined what it might look like. In this section, we sketch a concrete way to formalize and prove interesting properties of networks of middleboxes.

Listing 1 Model for an IDS

```
1  oracle suspicious? (packet: Packet) : Boolean;
2
3  model ids (p: Packet) = {
4    when suspicious?(p) =>
5        forward {}
6    default =>
7        forward {p}
8  }
```

DeMillo et al. [3] previously argued that specifying the desired behavior of a program (or network) is hard. Indeed, the lack of a precise specification is a major problem for program and network verification.

The primary function of networks is to allow hosts to communicate with each other. Reachabality, the property that a certain class of packets sent from host A can reach host B, and its converse, isolation, are fundamental to networks: all useful networks must satisfy some set of reachability properties and their verification is thus universally important. In the rest of this section we limit our discussion to Reachability invariants.

We formally state these invariants using temporal logic where we assume fairness, *i.e.,* we assume any continuously enabled transition will eventually occur. First, we define two relations: $\mathit{Send}(n, p)$ indicating some network entity (node) n sent a packet p (at some time), and $\mathit{Recv}(n, p)$ indicating node n received packet p. Given these relations any reachability property can be expressed in LTL (Linear Temporal Logic) as

$$\forall p \in \text{Packet} : \Box(\mathit{Send}(\mathit{src}, p) \wedge \mathit{Predicate}(p) \implies \Diamond(\mathit{Recv}(\mathit{dest}, p)))$$

This temporal logic statement says that a packet p sent by src which satisfies $Predicate$ is eventually received by $dest$. Similarly, isolation can be formally expressed by requiring that a packet sent by src, satisfying $Predicate$ is never received by $dest$:

$$\forall p \in \text{Packet} : \Box(\mathit{Send}(\mathit{src}, p) \wedge \mathit{Predicate}(p) \implies \Box(\neg\mathit{Recv}(\mathit{dest}, p)))$$

Predicate in the definitions above is specified using the same abstractions used to specify network policies, *i.e.,* either in terms of packet header fields (source, destination, etc.) or in terms of the abstraction provided by a middlebox oracle (§2). For example, a property saying no SSH traffic can reach a server d can be expressed as

$$\forall s \in \text{Node}, p \in \text{Packet} : \Box(\mathit{Send}(s, p) \wedge \mathit{ssh}(p) \implies \Box(\neg\mathit{Recv}(d, p))) \tag{1}$$

where $\mathit{ssh}(p)$ returns true if an Oracle classifies the packet as belonging to an SSH connection. We reason about reachability and isolation properties assuming that the Oracles are correct. Verifying the isolation property in Equation (1) therefore requires answering the question: "assuming SSH traffic is correctly identified, can a packet belonging to an SSH connection reach d?"

As stated earlier, we model a middlebox as an oracle and a simple abstract model. The Oracle provides abstractions that are used to specify the properties being checked. We expect models for middleboxes (which include both an oracle and a generic model) to be specified using a constrained programming language. Listing 1 shows an example of such a specification for an intrusion detection system (IDS). The IDS oracle provides one abstraction, `suspicious?`, defined in the first line. The abstract model is defined in lines 4–9 and uses this abstraction. First we check to see if the packet is suspicious (the Oracle's decision here

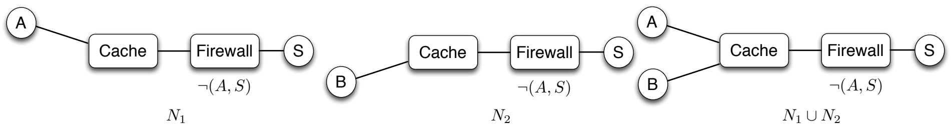


Figure 1 Example where networks are not composable with respect to reachability properties.

might be based on what packets it has seen previously), and drop the packet (line 6) or forward the packet (line 9) depending on the value returned by the Oracle.

Next, we develop abstract semantics for middleboxes. We use these to reason about general properties (such as composition) that apply to all middleboxes. Our semantic model is defined over a potentially infinite set of packets, P. We augment packets to include information about their location (*i.e.,* the middlebox or switch port). We also define the operator $\doteq$, such that $p_1 \doteq p_2$ implies that packets p_1 and p_2 are identical except for their location. Finally, we use P^* to represent the set of all (potentially unbounded) sequence of packets. The abstract model middlebox m is a function $m \colon P \times P^* \to 2^P$ which takes a packet ($p \in P$) and a history ($h \in P^*$) of all the packets that have previously been processed by m and produces a (possibly empty) set of packets $m(p,h)$. Given this model a switch is a simple function for which $m(p,h) \doteq \{p\}$. Similarly a simple firewall, f (whose decision process is represented by *allowed*) can be expressed as

$$f(p,h) = \begin{cases} \{p'\} \;\; p' \doteq p & \text{if } \mathit{allowed}(p,h) \\ \{\} & \mathit{otherwise}\,. \end{cases}$$

Ideally, we would like to be able to reason about the network *compositionally*, *i.e.,* the correctness of the network should follow from the correctness of smaller, simpler components. Compositional reasoning can reduce the cost of verification and enable incremental verification of changes in the network. Compositionality also allows us to potentially verify invariants in much larger networks, both by allowing us to parallelize verification and by reducing the size of the problem that needs to be provided to the SMT solver. Compositionality has been important for making verification tractable in other domains, for instance the use of rely-guarantees [15, 8], was important for enabling verification of concurrent programs.

We start by defining what it means to be able to compositionally verify a network. Let us define the union of two networks N_1 and N_2 in the natural way, *i.e.,* $N_1 \cup N_2$ contains the union of all nodes and links in each of N_1 and N_2. Consider two networks: N_1, where property P_1 holds (represented as $N_1 \models P_1$); and N_2, where property P_2 holds. We can compose the proofs for properties P_1 and P_2 if and only if both P_1 and P_2 hold for $N_1 \cup N_2$. For example, consider verifying the invariant "A and B cannot receive data from S" in network N in Figure 1. If compositional verification is possible for N, then the property holds in N if and only if A cannot receive data from S in N_1, and B cannot receive data from S in N_2. More formally, we can verify properties P_1 and P_2 compositionally for network N if for any $N_1 \subset N$ and $N_2 \subset N$

$$\frac{N_1 \models P_1, N_2 \models P_2}{(N_1 \cup N_2) \models P_1 \wedge P_2}\,.$$

Generally, one cannot perform compositional verification of reachability properties. For instance, consider the example in Figure 1. The cache in this example records all requests to and the corresponding responses from S. On receiving a new request, the cache checks to see if it has previously recorded a response for this request, in which case it returns the saved

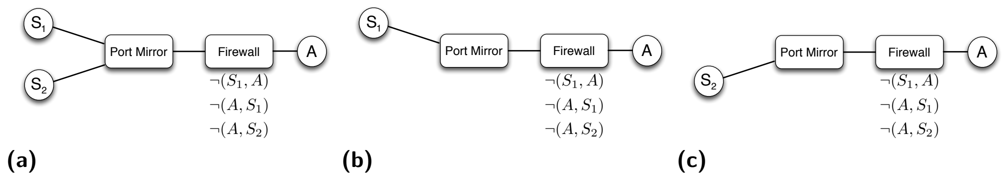


Figure 2 Example where the composition of two RONO middleboxes is not RONO.

response; otherwise the cached forwards the request, unmodified, to the firewall. The firewall drops all requests sent from A to S, but otherwise forwards all other requests and responses unmodified. In network N_1, A can never receive a response from S (thus is isolated). However in the composed network $N_1 \cup N_2$, if B sends a request r and receives response r' from S, then A can also request r and receive r'.

One key insight is that despite being impossible in general, there exists an important subset of networks where compositional reasoning can be used to verify reachability properties. We have found that networks which contain only a special class of middleboxes, Rest-of-Network Oblivious (RONO) middleboxes (§3) are often amenable to compositional verification. A RONO middlebox is one whose forwarding behavior (which is all we care about for reachability and isolation) for a pair of hosts depends only on the traffic sent between these hosts. More formally, define the restriction $h|_{(A,B)}$ of a packet history $h \in P^*$ to be the largest subsequence of h containing only those packets that were sent between host A and B; we then define a middlebox m to be RONO if and only if

$$\begin{aligned} &\forall p : p.src = A \land p.dest = B \quad f(p, h) = f(p, h|_{(A,B)}) \quad \text{and} \\ &\forall p : p.src = B \land p.dest = A \quad f(p, h) = f(p, h|_{(A,B)})\,. \end{aligned} \tag{2}$$

Similarly, we define an entire network as RONO, if its semantics can be described using a function that meets the condition stated in Equation 2.

Surprisingly, we find that not all networks that contain only RONO middleboxes are RONO, *i.e.,* RONO is not closed under composition. For example, consider the network in Figure 2. The firewall in this example is stateful and is configured to allow *no communication* between A and S_1. Furthermore, the firewall configuration allow A and S_2 to communicate, provided A establishes the connection, *i.e.,* sends the first packet. The port mirror is configured to duplicate and send all packets received from the firewall to both S_1 and S_2. Beyond these policies, both the firewall and port mirror forward traffic as expected (*i.e.,* they forward packets towards their intended destination). In this example, the port mirror is stateless, and hence trivially RONO according to Equation 2. The behavior of the stateful firewall is also RONO. However, the composition of these two middleboxes is not RONO. When we consider just S_1 and A in isolation (Figure 2b), *i.e.,* a case where S_2 neither sends nor receives any packets, A and S_1 cannot communicate, since all packets between them are dropped at the firewall. However, when we remove this restriction, *i.e.,* allow S_2 to send or receive packets, we find that A can in fact communicate with S_1: S_2 needs to first send a packet establishing a connection with A, any subsequent responses sent by A are allowed through the firewall, and duplicated so they are received by both S_1 and S_2. Therefore, it is simple to see that the abstract semantics of the network are different depending on whether we consider the entire packet history or a restriction, and the network is therefore not RONO, despite containing only RONO middleboxes. This shows that RONO is not closed under composition, *i.e.,* the composition of two RONO middleboxes might not be RONO.

However, despite RONO not being closed under composition in general, we have found that many existing networks are in fact RONO, and are hence amenable to compositional verification. Further, given that operators frequently add new middleboxes to their network, and RONO networks are precisely those where such addition cannot disrupt unaffected parts of the network, we think that many existing network might, in fact, be RONO by design.

6 Some Open Problems in Network Verification

Finally, we present some open problems that we have encountered while looking at how to verify mutable dataplanes. This list is not exhaustive, but is rather an attempt to list the first set of hurdles that need to be crossed given this new network verification agenda.

Decidability of Verification. When processing a packet, a middlebox might access potentially unbounded state. This prevents the use of finite-state model checking, and other verification techniques are undecidable for general programs in this class. We are currently working on a limited programming language that is rich enough to specify many existing middleboxes and to enable verification of some interesting network properties, including reachability properties. What other network properties can be verified in a decidable manner remains an important open problem.

Specification. While we have provided some tools that allow us to specify and check reachability properties; extending this to other invariants, for example performance-based invariants is challenging. How middleboxes and properties are specified also has a huge impact on verification time and decidability. Therefore, it is crucial to pick specifications that are rich enough to permit operators to express interesting and useful properties, yet narrow enough to permit automated reasoning.

Conditions for Compositional Verification. We have found a set of sufficient conditions that allow compositional verification of networks. However, finding a set of necessary conditions remains an open problem. Necessary conditions allowing compositional verification are useful not just for the formal verification community, but might also provide important insights about how networks should be designed and configured.

Correctness-Preserving Transformations. It might be possible to extend some of our results on compositional reasoning to show that the addition of certain types of middleboxes can never affect some class of invariants. We know this is true for some middleboxes in reality, e.g., the addition of a stateless firewall can never affect an isolation invariant (though it might invalidate some reachability invariants). Developing a theory for when this holds might be useful in developing techniques to help simplify network changes.

Verifying Parametric Topologies. Some network topologies are parametric. For example, one can generate a fat-tree topology [1] for a given datacenter size. It is possible that we can leverage compositional verification techniques to verify properties independent of the parameter. This would both speed-up verification and perhaps provide insights into the kinds of networks that are easily evolvable.

Acknowledgments. We thank Ori Lahav for help with the RONO proof. We also thank Cole Schlesinger, Aditya Akella, Shriram Krishnamurthi and Nate Foster for help debugging parts of the RONO proof. We are grateful to Justine Sherry, Colin Scott and the anonymous reviewers for their comments and suggestion on this paper. This research is supported in part by NSF grants 1420064, 1343947, 1040838; ISF grant 420/12; Israel Ministry of Science Grant 3-9772; a Marie Curie Career Integration Grant; the Israeli Center for Research Excellence in Algorithms; and by funding from Intel and AT&T.

References

1 Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

2 Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *POPL*, 2014.

3 Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. In *POPL*, 1977.

4 Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *ACM SIGCOMM CCR*, 38(3):105–110, 2008.

5 Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *PLDI*, 2013.

6 Swati Gupta, Kristen LeFevre, and Atul Prakash. Span: a unified framework and toolkit for querying heterogeneous access policies. In *HotSec*, 2009.

7 Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Static analysis in datalog extensions. *J. ACM*, 48(5):971–1012, September 2001.

8 Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.

9 Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

10 Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

11 Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

12 Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.

13 Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. In *Security and Privacy*, 2000.

14 Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. In *SIGCOMM CCR*, 2008.

15 Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

16 Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *NSDI*, 2013.

17 Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

18 Tim Nelson, Arjun Guha, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. A balance of power: Expressive, analyzable controller programming. In *HotSDN*, 2013.

19 Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

20 OpenStack. Congress. `https://wiki.openstack.org/wiki/Congress`; retrieved 01/08/2015.

21 Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *IMC*, 2013.

22 Scott Shenker. SDN: Looking Back, Moving Forward. Talk at Interent2 Technology Exchange 2014.

23 Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM*, 2012.

A Few Lessons from the Mezzo Project

François Pottier[1] **and Jonathan Protzenko**[2]

1 **INRIA, FR**
`francois.pottier@inria.fr`
2 **Microsoft Research Redmond, US***
`jonathan.protzenko@ens-lyon.org`

Abstract

With *Mezzo*, we set out to design a new, better programming language. In this modest document, we recount our adventure: what worked, and what did not; the decisions that appear in hindsight to have been good, and the design mistakes that cost us; the things that we are happy with in the end, and the frustrating aspects we wish we had handled better.

1998 ACM Subject Classification D.3.2 Programming Languages

Keywords and phrases static type systems, side effects, aliasing, ownership

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.221

1 A word about *Mezzo*

Mezzo is a programming language in the tradition of ML, whose type system incorporates a notion of ownership. The type system is close to separation logic: at each program point, a set of *permissions* describe what fragment of the heap "we" (the current procedure on the current thread) have access to and in what ways we can affect this fragment without violating what "others" (the callers of the current procedure on the current thread, as well as concurrently executing threads) assume about it. The type system of *Mezzo* has been mechanically proved sound [4, 1]: well-typed *Mezzo* programs are *memory-safe* and *data-race free*. A comprehensive paper about *Mezzo* is in submission [2]; bold and daring readers may also wish to read the second author's thesis [15].

Mezzo enables new programming patterns: thanks to its powerful ownership discipline, *type-changing updates* are permitted. This yields great flexibility: for instance, typestate, which is usually expressed using extra predicates that refine types [6], is now seen as a particular case of type-changing updates. Furthermore, a combination of singleton types and structural types allows the type system to track local aliasing relations. To keep the complexity of the type system under control, we deliberately do not support certain advanced ownership disciplines (such as static regions with multi-focusing; fractional permissions; etc.). In order to partially make up for the lack of these advanced features, we encourage the programmer to rely on dynamic tests when dealing with complex aliasing of mutable data. To that effect, *Mezzo* provides a novel adoption/abandon mechanism that allows a dynamic test to yield a static permission. *Mezzo* also provides locks, which are another dynamic mechanism for (temporarily) obtaining a static permission. In practice, adoption/abandon and locks are typically used in concert.

Mezzo does not just live on paper: one very explicit goal was to design a language *for programmers*, not just a stack of Greek symbols in a conference paper. To that effect, we split

* The work covered in this paper was carried out while at INRIA.

the language in several layers (one for the programmer, a lower-level one for the type-checker, and a kernel one for the proof of soundness). Furthermore, we designed type inference algorithms and wrote an implementation, which can be tried out online [16]. Finally, the *Mezzo* repository [14] contains several thousand lines of examples, made up mostly of data structures and algorithms.

2 Some fundamental design decisions

The *Mezzo* project started with the PhD of the second author, whose title was "towards better static control of side-effects". Our first decision was to design a new language, rather than build upon an existing one. A related decision was to focus exclusively on the surface language and on its type system. After type erasure, *Mezzo* is essentially a subset of OCaml, so a (well-typed) *Mezzo* program can be compiled and executed by translation to untyped OCaml (i.e., OCaml with unsafe type casts).

Starting afresh offers many benefits. The feature set of a fresh language is minimal. We did not have to adapt our type discipline to deal with objects, modules, or GADTs, to cite only a few of OCaml's features. The implementation of a fresh language is simple: we did not have to implement *Mezzo*'s type-checker as an extension of OCaml's type-checker, which would have been technically very difficult. The design of a fresh language can be radical: we did not have to retain compatibility with an existing code base. In fact, we were at liberty to deeply alter the way programmers think about types. Instead of viewing types as descriptive (and unalterable), we incorporate ideas from separation logic, we think of types as descriptive and prescriptive at the same time (and allow them to change as the program runs). Thus, we must abandon Hindley-Milner type inference, and replace it with a type-checking and type inference algorithm which in many ways resembles forward symbolic execution [5] and abstract interpretation.

Focusing on a functional core influenced the design of the language. In *Mezzo*, algebraic data types are fundamental. They can express state change, temporarily broken invariants, refined predicates, aliasing relationships... Had we chosen to construct an object-oriented language, the core building blocks of the language would have been objects instead.

Starting anew also has several drawbacks. Perhaps the biggest one is the absence of a conversion path from OCaml to *Mezzo*. Even though untyped *Mezzo* is (a subset of) untyped OCaml, there is no gradual way of porting a program from OCaml to *Mezzo*: the entire program must be altered so as to abide by *Mezzo*'s type discipline. Furthermore, we do not have a good way of allowing interoperability between OCaml and *Mezzo*. Although the function type and the `ref` type in OCaml and in *Mezzo* look superficially similar, they have different meaning: it is unsafe to naively let a mutable object (or a function that accesses a mutable object) cross the boundary. Perhaps one could restore safety by inserting dynamic checks at the boundary; this seems nontrivial and was left as future work.

For a research project that lacks the resources to go mainstream, the lack of interoperability is perhaps tolerable. We do feel, however, that providing a conversion path would have helped us write more examples and receive more user feedback. New programming languages that aim at seizing a market share do deal with interoperability: TypeScript has "definitions" for adding a typed interface on top of existing JavaScript code; Rust can be made ABI-compatible with C libraries, as long as suitable Rust types are provided for C functions.

3 Growing the language

One of our starting points was theoretical work by Charguéraud and by the first author [7]. We originally focused on the type system of a core language, leaving the design of a surface language to a later stage. This approach turned out to be flawed. Lacking any impetus from sample programs, we had a hard time figuring out where to push the design of the core language. Lacking any connection with the surface, we had an even harder time figuring out how inference could possibly work. We ended up entangled in technicalities and unable to form a grander vision of where we should go.

Things improved when we decided to drop this approach and instead work on the surface language first. One of the best steps we took at this point was to write several sample programs, even before the language existed. Designing a new language is a broad, ill-defined task. Writing fictitious programs, finding out exactly how we would like them to look and feel, and understanding what guarantees we would like the type system to provide, proved highly beneficial. This activity allowed us to understand early on that some of the features that we had in mind (such as static regions) were not practical; it also allowed us to imagine several novel features, such as adoption/abandon, a more flexible replacement for static regions.

3.1 Syntax, syntax, syntax

We spent a surprising amount of time working out the details of the syntax. This may seem futile; yet, a concise and natural syntax is a requisite for wide adoption, and a sign that the underlying concepts are simple.

Consider the type of the list `length` function. This type must express that the caller loses ownership of the list while the `length` function operates over it and, once the function returns, the caller regains ownership. Here is what this type looks like in the internal language of the type-checker:

$$\forall(a : \mathsf{type}), \forall(l : \mathsf{value}), (=l \mid l \,@\, \mathsf{list}\; a) \rightarrow (\mathsf{int} \mid l \,@\, \mathsf{list}\; a)$$

Thanks to our syntactic sugar, the user simply writes:

```
val length: [a] list a -> int
```

This simplification is obtained in two steps. First, in the surface syntax, we adopt the convention that a permission that appears in the domain of a function type also appears (implicitly) in its codomain. This means that $l \,@\, \mathsf{list}\; a$ can be omitted in the right-hand side of the arrow. Second, because the right-hand side no longer refers to the name l, it is no longer necessary to introduce this name. The universal quantification over l disappears, and the left-hand side of the arrow becomes just $\mathsf{list}\; a$.

In contrast, the type of list concatenation must express the fact that this function consumes the ownership of its two arguments. In the surface syntax, this can be done in any one of the following ways:

```
val append: [a] (consumes (list a, list a)) -> list a
val append: [a] (consumes list a, consumes list a) -> list a
val append: [a] (consumes xs: list a, consumes ys: list a) -> zs: list a
```

To allow the user to express fine-grained control over which arguments are lost and which are preserved, the **`consumes`** annotation can be nested arbitrarily deep within structural (tuple and data) types. The carefully-chosen convention that "permissions are preserved unless otherwise specified" has proven, in our experience, to save a lot of annotations.

Our last example is the `iter` function on lists. Its type differs from its classic OCaml counterpart in that the client function `f` may impose a type-changing update on the list elements. This type illustrates the flexibility of the **`consumes`** keyword: here, the permission for `xs` is consumed, while the permission for `f` is preserved. This type also illustrates a situation where the function's codomain `(| xs @ list b)` refers to `xs`, a name for the first component of the function's argument. This exploits the "name introduction" construct `x: a`, which again can be nested at arbitrary depth within a structural type.

```
val iter : [a, b] (
  consumes xs: list a,
  f: (consumes x: a) -> (| x @ b)
) -> (| xs @ list b)
```

The type of `f` says: "Give me exclusive control over my argument `x` at type `a`. I will return to you a unit value, along with exclusive control of `x`, now at type `b`". The type of `iter` has a similar reading: it requires unique ownership of `xs` at type `list a` and returns unique ownership of `xs` at type `list b`.

We did not come up with this all at once, naturally. Our functions initially took multiple arguments. While trying to work out the translation of our conventions for function types into more atomic constructs, we realized that we could replace multiple-argument functions with single-argument functions and simplify the whole translation process. The functions `append` and `iter` above expect just one argument, which is a pair. This view is made possible by the fact that the `consumes` annotation and the name introduction construct `xs: ...` can appear inside a structural type. This design simplified the translation from surface syntax to internal syntax, made the surface syntax more regular, and augmented the expressive power of our function and structural types.

Incidentally, the tuple type `(x: t, y: u)` is not a primitive dependent tuple. It is an ordinary tuple whose components contain name introduction constructs. It is desugared using existential quantification and singleton types. The desugaring is symmetric, so both `t` and `u` may refer to both `x` and `y`. There is no left-to-right bias.

3.2 Algebraic data types are central

One thing that we believe is a strength of our type system is the unified vision of structure and ownership that it provides. For instance, we do not need to distinguish a type environment and a state environment that would store information such as "this list cell is uninitialized" or "this cell is initialized and will no longer be mutated". Instead, everything is expressed in the type. A typical way to encode state information about a memory block is to exploit algebraic data types, as follows.

```
data mutable cell =
  Cell { head: (); tail:      () }
data list a =
  Cons { head:  a; tail: list a } | Nil

val () =
  let x = Cell { head = (); tail = () } in
  (* x is uninitialized and has type: Cell { head:  (); tail:       () } *)
```

```
x.head <- 0;
x.tail <- Nil;
(* x has been mutated and has type: Cell { head: int; tail:      Nil } *)
tag of x <- Cons;
(* x has been frozen and has type:  Cons { head: int; tail:      Nil } *)
(* which may be folded back to:     Cons { head: int; tail: list int } *)
(* which may be folded back to:     list int                           *)
```

Structural types provide a natural, integrated mechanism for tracking the state of a memory block, via its *tag*, or data constructor. By referring to the data type definitions, the type-checker knows that a memory block whose type is `Cell { ... }` is mutable (and uniquely owned), while a memory block whose type is `Cons { ... }` is immutable (and shared). In a structural type, the types of the fields are arbitrary: they need not match the types that appear in the data type definition. However, a structural type can be folded back to a nominal type, as in the last line above, only if the types of the fields do match the definition.

The above example shows a variety of features that we believe are novel. (1) We conflate products and sums in data type definitions. This comes at no runtime cost (in the OCaml heap, every block carries a tag anyway) and in our experience improves readability. (In contrast, OCaml does not allow naming the arguments of a data constructor.) (2) The tag update instruction adds a little bit of low-level expressive power (at the surface level, OCaml does not allow mutating the tag of a memory block). (3) Furthermore, as shown above, tag update can express freezing: by definition, the tag `Cell` implies mutability, while the tag `Cons` implies immutability. (4) Structural types allow keeping precise track of field updates: the type of a field can change at every update. This is sound because mutable memory blocks are uniquely owned. (5) A `match` construct refines a nominal type to a structural type, henceforth allowing field access. For instance, if `xs` has type `list t`, then a *Mezzo* programmer may choose to write:

```
match xs with Cons -> f xs.tail | Nil -> ... end
```

In contrast, an OCaml programmer would have to bind the tail of `xs` to a fresh variable as part of the pattern: `match xs with Cons (_, tail) -> f tail | Nil -> ... end`. In *Mezzo*, both styles are permitted.

Separation logic [17] and alias types [18] have shown that it is important to keep track of must-alias relationships in order to get an accurate description of the heap. Consider a variable `x` which has type `Cons { head: a; tail: list a }`. If one reads the `head` field by writing `let y = x.head in ...`, who "owns" the first list element at type `a`? Is it still owned by `x`, so to speak, or is it now owned by `y`? Rather than introduce a concept of "data type with a hole in it", or a concept of "borrowing", we choose to rely on *singleton types*. A singleton type `=x` has only one inhabitant, namely `x` itself. In the above situation, the *Mezzo* type-checker automatically introduces two auxiliary names `h` and `t` and makes the following statements:

```
x @ Cons { head: =h; tail: =t }
h @ a
t @ list a
```

Then, upon finding the definition of `y`, the type-checker adds the statement that `y` has type `=h`, which one may write either `y @ =h` or in the form of an equation:

```
y = h
```

The above four statements, or *permissions*, imply that `x.head` and `y` are aliases and can be used interchangeably. Furthermore, permissions are ownership assertions. The permission `h @ a` represents the ownership of a heap fragment whose root is `h` and whose extent is described by the type `a`. Similarly, `t @ list a` represents the ownership of a heap fragment rooted at `t`. Because a singleton type implies no ownership, the equation `y = h` claims no ownership. For the same reason, the permission `x @ Cons { head = h; tail = t }` represents the ownership of just one block of memory, at address `x`, and describes its contents in an exact way. In short, types and permissions have both layout and ownership readings.

All of this shows that structural types and singleton types are central concepts in *Mezzo*: together they convey not only shape information ("`x` is the address of a `Cons` cell"), but also must-alias information ("its head is `h`") and ownership information ("we own the memory block at address `x`"). Naturally, these ideas are inspired by Alias Types [18] and by Separation Logic [17]. We mesh them with algebraic data types, so that they integrate well with the language and allow expressing a variety of key programming patterns in a natural way.

For this approach to work, the system must have rules to decompose and recompose types, so as to switch back and forth between a coarser view where "`x` has type `list a`" and a lower-level, finer-grained description in terms of singleton types. Furthermore, the type-checker must be able to transparently apply these rules whenever necessary. During the implementation of the type-checker, we found this a challenge.

3.3 Ownership is central

Mezzo must support type-changing updates, not only because we are interested in typestate-checking as a high-level goal, but also because (as shown above) the manner in which we exploit singleton types essentially leads *every* update to be a type-changing update. However, a strong update is sound only if the modified object is uniquely owned. Therefore, an ownership discipline is required.

Turning this argument around, one may accept ownership as the primary concept, and view mutability as a consequence of it. *Because* we have exclusive access to a memory area, it is safe to change its contents in an arbitrary way. This cannot break what others assume about this area, because in fact they cannot assume anything about it; and this cannot cause a race condition, because they cannot access this area. It seems like this point of view is shared within the Rust project [11].

One way of understanding ownership is in terms of permissions. "We" are granted the right to perform certain operations ("we own this data structure exclusively, hence we may read and write it") whereas "others" are denied the right to perform certain operations ("we own this block exclusively, hence others cannot read or write it"). Another way of understanding ownership is in terms of knowledge. Exclusive ownership of a data structure means "we" know that this data structure exists, whereas "others" don't; shared ownership means "we" know about this data structure and "others" may know about it too. Naturally, the two views are dual, and either of them defines the other. If one thinks primarily in terms of permissions, then our "knowledge" is whatever assertion is stable in the face of permitted interference by others. Conversely, if one thinks primarily in terms of knowledge, then we have "permission" to perform whatever action preserves the knowledge of others. This dual understanding of ownership is reminiscent of rely-guarantee [10]. The classic type disciplines of C, Java, OCaml, etc., are a special case where every object is considered potentially shared and (as a consequence) every update must be type-preserving.

Anecdotal evidence suggests that ownership is a good concept for *users*, as it helps them have a mental model of what's legal and what's not, of what may happen and what may not happen, but also for *language designers*, as it helps figure out at an intuitive level whether a proposed typing rule makes sense in terms of ownership. One such mechanism is adoption and abandon.

3.4 Flexible ownership via adoption and abandon

Adoption/abandon works, in essence, as follows. If one owns an object `x` at type `t`, one may relinquish ownership of `x` and **`give`** it to an adopter `y`. One can think of `y` as maintaining a runtime list of its adoptees, which it owns. (Our implementation uses a more efficient scheme, where an adoptee points to its adopter). Adoption consumes the unique permission `x @ t`: the type system no longer keeps individual track of `x`. (It also consumes `x @ unadopted`: this guarantees that every object has at most one adopter, a condition that our implementation scheme requires.) Instead, the aggregate permission `y @ adopts t` represents the ownership of all adoptees of `y` as a group. While `x` is adopted, it no longer has type `t`. Still, it has type `adoptable`, which means it is a valid address. The address `x` can be copied without restriction: this allows mutable data to become aliased, and still remain usable, as described now. When one wishes to regain ownership of `x`, one may attempt to **`take`** `x` from `y`. This operation checks, at runtime, that `x` currently appears in the list of adoptees of `y`, and takes it out of this list. This yields the permissions `x @ t` and `x @ unadopted` again. The point of the runtime check is to prevent the duplication of these permissions.

`x` **has type**	**We know that. . .**	**Others know that. . .**	**mode**
`adoptable`	`x` is a valid heap address	`x` is a valid heap address	duplicable
`unadopted`	`x` is a valid heap address `x` is not currently adopted	`x` is a valid heap address	affine
`adopts t`	every adoptee of `x` has type `t`	–	affine

The ownership hierarchy was, initially, a purely static concept that was expressed in the types. Adoption and abandon allow part of this hierarchy to exist (and to be queried and modified) at runtime. The **`give`** and **`take`** operations allow move between the static and dynamic regimes of ownership.

This mechanism is perhaps one of the main contributions of *Mezzo*: it provides an *escape hatch* out of the purely static discipline. This allowed us to keep the type system relatively simple. If we had tried to statically keep track of complex ownership patterns, we would have exceeded our complexity budget.

Throughout the course of the *Mezzo* project, we had to navigate between two conflicting goals. On the one hand, we wanted a language that was more than a research project. For instance, a traditional affine (or linear) type system is not expressive enough for real-world programming: we had to allow, one way or another, a certain degree of aliasing. On the other hand, we wanted the system to remain usable by (skilled) programmers: we had to reject proposed features of the type system that we deemed too technical. Abandon and adoption was judged a good compromise.

3.5 Beyond duplicable versus affine?

The various *modes* of ownership were the topic of much discussion. For simplicity, *Mezzo* as it stands distinguishes only two modes, or two kinds of permissions, namely duplicable (i.e., shared) permissions and affine (i.e., non-duplicable, or exclusive) permissions. A duplicable permission can be viewed as affine; the converse is not permitted. Every permission is in fact affine.

One may think of classifying types using kinds, where a kind is "duplicable" or "affine". However, in order to type-check list `length`, one would then need kind polymorphism:

$$\mathsf{val\ length}\colon \forall\kappa, \forall(\alpha : \kappa), \forall(l : \mathsf{value}), (=l \mid l \mathbin{@} \mathsf{list}\ a) \to \ldots$$

This additional layer of quantification seems too much of a burden. We prefer to view modes as predicates over types, much like type classes. Thus, the user can explicitly request that a type satisfy a certain mode:

```
val f: [a] duplicable a => (x: a) -> ...
val g: [a] affine a => (x: a) -> ...
```

Because every type is affine, the constraint `affine a` can in fact be omitted. This gives rise to concise types in many situations.

In theory, one could add other modes, such as "linear". Viewing an affine permission as linear would be permitted; the converse would be forbidden. In principle, this could help programmers ensure that resources (e.g., file descriptors) are properly freed. In practice, though, this would make the system more verbose: the constraint `affine a` would no longer be a tautology and might have to appear in many places. Furthermore, in order to prove a "complete collection" theorem (i.e., a linear object is never lost), one would have to impose a restriction: a resource guarded by a lock must be affine. (This is necessary because locks are never de-allocated.) This makes the idea less attractive: for instance, a lock cannot guard a file descriptor. Furthermore, even if "complete collection" holds, in practice, it seems that there are still (admittedly contrived) ways of getting rid of a linear object without properly freeing it; e.g., by converting it to the existential type `{a} (a | linear a)` and "giving" it to a "black hole" adopter that one threads through the entire program. For all these reasons, we left the exploration of "linear" (and other modes) to future work.

4 Looking at some code

Some programs can be expressed in a palatable manner using *Mezzo*. This is less true of some other programs. While there is no theorem classifying easy-to-write and hard-to-write programs, we have a few examples that are representative of the typical experience of writing *Mezzo* code.

4.1 Example #1: one-shot functions

A programming pattern that pops up quite often is that of "one-shot functions". Also known as linear arrows, these functions may only be called once. In *Mezzo*, by default, a function can only capture duplicable permissions, and is itself duplicable. Therefore, every *Mezzo* function may be called as many times as one wishes, provided of course that the caller is able to supply the permissions that this function requires. Nevertheless, one can simulate a one-shot function in *Mezzo*, as follows.

```
alias osf a b = {p: perm} (((consumes (a | p)) -> b) | p)

val promote [a, b, p: perm] (f: (consumes (a | p)) -> b | consumes p) :
  (| f @ osf a b) = ()
```

A one-shot function of `a` to `b` (where `a` and `b` are types) is a package of a function of `a | p` to `b` along with a single copy of `p`, where `p` is an opaque permission. That is, in order to invoke this function, one must supply not only an actual argument of type `a`, but also the permission `p`. Because `p` is existentially quantified (as indicated by the curly brackets), it is regarded as affine: that is, it cannot be duplicated. Because the first invocation of the function consumes `p`, any further invocation is forbidden. Thus, a function of type `osf a b` may be called at most once.

Creating a one-shot function should incur no run-time penalty. One can either use the `promote` function above, whose net effect is to perform an existential packing, and assume that the compiler will get rid of what is actually a no-op; or, one can write a `pack` instruction that bypasses the function call and performs the existential packing directly.

This encoding of one-shot functions may seem heavy, and the reader may wonder whether one could instead view affine functions as a primitive notion. We note that our approach is more general, as it allows encoding other related concepts, such an affine choice between two functions, e.g., "you may invoke either the success continuation or the failure continuation, but only one of them, and at most once".

The version of the one-shot function above takes and returns a run-time argument. We can define a restricted version of one-shot functions that only operates over permissions. The type below implements the "magic wand" of separation logic, written $p \multimap q$.

```
alias wand (pre: perm) (post: perm) = osf (| pre) (| post)
```

In this restricted version, the function takes and returns at run-time the unit value, but statically transforms the permission `pre` passed along with the argument into `post`.

These programming patterns would qualify as "easy to express" in *Mezzo*. They do sometimes require annotations, which can be made cumbersome by the fact that we need a `let flex` construct to refer to anonymous, existentially-quantified variables. That being said, we've been happy that these can be defined in the language, and don't have to be built-in features.

4.2 Example #2: strong updates

Code that performs strong updates is, in general, well-suited to the static discipline of *Mezzo*. A representative example is our implementation of "futures" (also known as "promises", or suspended computations).

A suspended computation works as follows: we allocate a mutable reference in the heap; we evaluate the computation and write its result into the reference; we *freeze* the reference (make it immutable) to express the fact that no further mutations shall occur.

In its initial state, a suspension is thus a standard *Mezzo* reference, defined as follows.

```
data mutable ref a =
  Ref { contents: a }
```

By performing a tag-update from `Ref` to `R` we can freeze a suspension into its final state, that is, into a `result a`.

```
data result a =
  R { result: a }
```

(From a more general perspective, tag mutations in *Mezzo* allow checking more invariants, as our type system is able to statically track this "freezing" pattern, where a data structure starts mutable, then is made immutable.)

There is a catch, however: we initially allocate a *mutable* reference, which is *affine*, hence not shareable. We do want to share it, however, as multiple clients will want to wait for the computation to finish in the background. We thus define the type `future a`. It is duplicable; this is the type the client manipulates, and that we export as `abstract future a` in the module's interface.

```
alias future a = (s: unknown, lock (s @ result a))
```

The type definition above is relatively concise: thanks to our binding rules, the name `s` may appear anywhere in the tuple. Quite surprisingly, the lock protects a duplicable permission, which may seem counter-intuitive. One way to intuitively understand the type `future a` is to think of it as a promise that, by the time the client acquires the lock, the reference will be frozen with the result of the computation.

The permission `s @ result a` is, naturally, not available initially; actually, we use the lock as a semaphore, as the code sample below demonstrates.

```
(* Creating a future from a one-shot function [k]. *)
val new [a] duplicable a => (consumes k: osf () a) : future a =
  (* The suspension starts out as a reference,
   * whose contents does not matter. *)
  let s = newref () in
  (* The lock is created in the [locked] state; the permission
   * [s @ result a] is *not* available yet! *)
  let l : (l: lock (s @ result a) | l @ locked)  = new_locked () in
  (* The computation that is spawned in the background. *)
  let compute (| consumes (k @ osf () a * s @ ref () * l @ locked)) : () =
    s := k();
    (* Turn [s @ Ref { contents: () }] into [s @ R { result: a }], that is,
     * into [s @ result a]. *)
    tag of s <- R;
    (* We can now release the lock for the first time. *)
    release l
  in
  (* Concurrently compute and return the future: *)
  spawn compute; (s, l)
```

Reading the result of the computation is then a mere matter of acquiring the lock; the first read will occur after the transition of the "semaphore" from a reference to a `result a`.

This flavor of programming, where a uniquely-owned data structure is mutated, then frozen, tends to work well in Mezzo. We have some flagship examples on lists: we can define tail-recursive versions of `map` and `concat`, where a new list cell is temporarily mutable, and is frozen (i.e., becomes immutable) once it has been fully initialized.

We also have the example of lazy thunks, where a combination of subtyping witnesses and existential quantification allows implementing thunks *in* Mezzo while ensuring that the type `lazy a` is covariant[1].

We agree that the suspensions example is somewhat technical; it seems to us, though, that it is still a strength of Mezzo that we can explain these precise mutations and ownership

[1] Unlike suspensions, the details are truly technical; the curious reader can read the `stdlib/lazy.mz` file in the source repository; it is heavily commented.

transfers within the type system. Doing the same in ML would require unsafe casts or heavy run-time checks.

4.3 Example #3: mutable trees

Another kind of code that we have found to work well in *Mezzo* is imperative code without aliasing. Our implementation of mutable balanced binary search trees is representative of that class of programs, which in general includes all list-like and tree-like mutable data structures.

Our module for mutable trees represents about a thousand lines of *Mezzo* code. It adapts OCaml's tree library so as to allow in-place mutation. It does not use adoption/abandon.

```
data tree k a =
  | Empty
  | mutable Node {
      left: tree k a; key: k; value: a; right: tree k a; height: int
    }
```

We never mutate the `Empty` constructor; therefore, only the `Node` constructor needs to be mutable. Allowing `Empty` to remain immutable allows one to allocate a single value and use it subsequently in all places where an `Empty` constructor is required, thus acting like a null pointer.

The tree module features several internal functions; an interesting one is the function that re-balances a tree.

```
val bal: [k, a] (
  consumes t: Node {
    left: tree k a; key: k; value: a; right: tree k a; height: unknown
  }
) -> tree k a
```

This function type is interesting in several ways. First, it demands that the argument `t` be not just a `tree k a`, but specifically a `Node`. This kind of pre-condition would be expressed in OCaml as a comment; *Mezzo* enforces it via the static type-checking discipline. This pattern is one that we've come to use frequently.

Second, because the function writes the `height` field but does not read it, it does not need any hypothesis about the type of this field. We therefore document in the function type that the function will not read the `height` field.

Most of the functions in this module require a comparison function. Rather than requiring the user to provide a comparison function for each call to, say, `find`, we defined a new dependent type that bolts a specific comparison function onto the tree type.

```
data mutable treeMap k (cmp : value) a =
  TreeMap { tree: tree k a; cmp | cmp @ (k, k) -> int }
```

This is not a radically new theoretical feature, but rather another programming pattern that we have found convenient in many situations. In OCaml, one would either risk mixing a tree with an unrelated comparison function, or one would have to use functors. *Mezzo* offers a more light-weight mechanism for that.

4.4 Example #4: borrowing

Borrowing is a term that covers multiple issues. Let us tackle a specific one: assigning a type to the `find` function for lists. This raises a difficulty in terms of ownership: because `find` returns a pointer to a list element, it duplicates this element. (That is, the element is now accessible both via the list and as the result of `find`.) Hence, `find` must be restricted to lists of duplicable elements. How can one work around this restriction?

A higher-order version of `find` can be easily written, where the caller passes a function that describes what to do once the element is found.

```
val find: [a] (
  xs: list a,
  pred: a -> bool,
  f: (x: a) -> ()
) -> ()
```

This is not satisfactory, however, because it requires the user to change the control-flow. What we would like is for the caller to get a pointer into the list, and make sure the list is invalidated as long as the caller holds a pointer to the element. Thus, the element would be temporarily "borrowed" from the list.

The good news is, this ownership pattern *can* be expressed in *Mezzo*. The bad news is, it requires crafting some "ninja" *Mezzo* code that is outside the reach of a casual user. The signature of such a `find` function is as follows, where we reuse the `wand` type we saw earlier:

```
alias focused a (post: perm) =
  (x: a, w: wand (x @ a) post)

val rec find: [a] (consumes xs: list a, pred: a -> bool)
-> either
     (| xs @ list a)
     (focused a (xs @ list a))
```

The *focused* type describes the pair of an element, along with a wand that *consumes* the element, in exchange for a certain permission `post`.

The `find` function consumes ownership of the list, and either returns it immediately, meaning that the element was not found, or returns a *focused* element that, once "surrendered", grants ownership of the original list again. This signature hence expresses faithfully the protocol for ownership transfer that we outlined earlier.

Quite regrettably, the implementation of `find` is fairly difficult to comprehend, and proper explanations are outside the scope of this paper (they can be found in our journal paper [3]). Let us just mention briefly some ingredients:

- the identity function serves as the implementation of the magic-wand – in other words, we abuse a function that does nothing at run-time to encode invariants in the type system, for lack of ghost code;
- the inference engine of *Mezzo* cannot figure out how to synthesize the "magic wand" – therefore, the user must perform a manual `pack` instruction and provide the existential witness;
- in order do so, the user must be able to name the type variables that are automatically introduced by the type-checker when it "auto-unpacks" an existentially quantified type or permission. Currently, this is done by using a `let flex` construct which allows naming a

type variable after has been introduced. Using this construct requires a rather low-level understanding of the type-checker.

Similar, if only greater difficulties are the heart of our work on iterators [9].

This latter example certainly falls in the category of programs that are harder to express in *Mezzo* than they ought to be. In the current state of things, we do not know how to make these programs with complex ownership protocols easier to write in *Mezzo*. There is, of course, the option of restricting `find` to duplicable elements. This may sound like a very special use-case, but in *Mezzo*, one can always convert affine elements to duplicable ones, using adoption/abandon:

- define a global adopter object,
- initially `give` all elements to the adopter,
- manipulate elements of type `dynamic` (duplicable),
- wrap every access to an element with a pair of `take` and `give`.

Therefore, a container of non-duplicable elements can always be viewed as a pair of a container of duplicable elements (adoptees) and an adopter. This is not satisfactory, though: the use of adoption/abandon is never desirable as it has a runtime cost and introduces a possibility of failures.

Taking a step back, other programming patterns are certainly hard to express in *Mezzo*, and if we were to extend the type system with more ad-hoc rules, we would certainly consider extra candidates. For instance, taking an immutable pointer to a mutable block (C++'s `const` modifier) is currently not supported by *Mezzo*; the right way to support this would certainly be fractional permissions.

5 Difficulties revealed by the implementation

Implementing *Mezzo* proved remarkably beneficial. This task allowed us to confirm and strengthen our basic intuitions about the type discipline. At the same time, it revealed many difficult problems that we initially did not clearly anticipate.

One particularly thorny issue is the "merge problem". Consider a `match` expression where `x` has type Nil at the end of one branch, and has type `Cons { head = h; tail: Nil }` at the end of the other branch, where `h` has type `a`. What is the type of `x` after the match expression? `list a` seems a natural answer. A closer look, though, shows that `list (=h)` is also a valid answer, although most certainly not the one the user had in mind! The problem becomes even more difficult in the presence of exclusively-owned data, and admits no principal solution in the general case.

Another thorny issue is inference of polymorphic instantiations. Suppose `x` has type `Cons { head = h; tail: Nil }` and `h` has type `a`. If one calls `identity x`, then the type-checker tries to infer a polymorphic instantiation. Several alternatives arise: `list a`, of course, but also `Cons { head = h; tail: Nil }`, `Cons { head: a; tail: Nil }`, and other variations, including the singleton type `=x` and the "top" type `{a} a`. Here, the final decision is innocuous: the identity function, after all, returns the ownership to the caller, so regardless of the instantiation choice, all is well. One can easily imagine non-trivial cases where the decision has dramatic consequences on the rest of the type-checking process.

As we saw, *Mezzo* can express "one-shot functions". This is nice, and can be extended to describe more complex situations, such as a one-off choice between several functions, or a function whose type changes after every call. Unfortunately, after a one-shot function has been called, its type lingers. The function still has type `(a | consumes p) -> b`, for some opaque permission p, and the type-checker cannot see that this makes the function useless. More generally, a function may have multiple types. *Mezzo* can express intersection types: one can easily state that x has type t *and* type u. Intersection types can sometimes, but not always, be simplified. In particular, at a call site, if the function happens to have several types, it is not easy for the type-checker to determine which one applies; in fact, several of them could apply, and the choice could have consequences down the road!

Our work on iterators [9] stumbles upon all of the above difficulties. To summarize our current position: the type system is in theory remarkably powerful and regular, and can (at least in some cases) express complex protocols of ownership transfer. In practice, however, using the system requires a great deal of expertise and a non-trivial amount of annotations.

After all, this is not surprising: the *Mezzo* type-checker needs to somehow perform type inference in System F (a subset of *Mezzo*!), decide entailment in separation logic, and solve the frame inference problem (at function call sites). These problems are hard or undecidable. The problem is possibly made harder by the presence of (contravariant) function types, which are absent in first-order separation logics.

5.1 The state of the implementation

Our current implementation relies on heuristics and limited backtracking. The type-checker does not always terminate. The situations where it diverges are rare, but it is not completely clear how to rule them out. Most of the time, our heuristics work well. Nevertheless, they are unsatisfactory in principle, and difficult to maintain. Furthermore, should inference fail, the user is puzzled, and type errors are difficult to explain.

Perhaps a few examples may illustrate the kind of difficulties that we encounter in the type-checker. Many difficulties are related to the comparison of function types. For instance, the type-checker cannot prove that the following type is a subtype of itself.

```
val f: (| p * q) -> ()
```

The reason is, the algorithm for deciding subtyping compares functions; it then compares domains. In the process, the algorithm is faced, on the one-hand, with rigid variables p * q and, on the other hand, with flexible (unification) variables ?p * ?q. Figuring out that ?p ought to instantiate onto p and ?q onto q amounts to a associativity/commutativity search. The algorithm does not perform this kind of search. More generally, because of our translation of external syntax into an internal one, a function may have several, equivalent types (with or without singleton types, for instance); the type-checking algorithm then needs to deal with a variety of equivalent representations seamlessly.

Other situations involve finding existential witnesses: a rule of our system allows instantiating a unification variable a into the conjunction of b (a type) and p (a permission), written b | p.

```
alias t1 = [a] () -> a
alias t2 = [p: perm] () -> (() | p)
```

The type t1 is a subtype of t2: it suffices to instantiate a into b | p, then to instantiate b onto the unit type (). The type-checking algorithm is unable to figure that out, though: exploring the application of this typing rule would render the search space too big.

The picture depicted above is bleaker than it ought to be. In practice, a syntactic comparison makes sure that a type is always a subtype of itself (and provides, based on experimental measurements, a significant performance boost). Furthermore, the algorithm for comparing function types has been extensively fine-tuned and works most of the time. By "most of the time", we mean that over the 7,000 lines of "real" *Mezzo* that we have written (this includes blank lines and comments, but excludes files that are just unit test-cases), only 97 type applications appear, meaning roughly one annotation every 70 lines of code. We believe this to be an acceptable penalty.

Furthermore, after an internal presentation in the C++ group at Microsoft, it turned out that programmers there were absolutely unfazed at the prospect of a compiler that may fail, unpredictably, for some unknown reason, and may require them to provide more annotations. It seems that this is the common lot of a C++ programmer who uses templates, and the audience was unanimously happy to pay the unpredictability for more expressive power.

It thus seems that the situation is not as bad: failures of the type-checker are, after all, rare, and the user can always annotate herself out of a tricky type-checking situation.

6 The proof of Mezzo

The machine-checked definition of *Mezzo* is organized as a kernel, on top of which sit three (almost) independent extensions. The kernel calculus can be described as a concurrent call-by-value λ-calculus, equipped with an affine, polymorphic, value-dependent type-and-permission system. The extensions are: (a) strong (i.e., affine, uniquely-owned) mutable references; (b) dynamically-allocated, shareable locks; (c) adoption and abandon. We prove type soundness ("well-typed programs do not crash") and data-race freedom ("well-typed programs are data-race free").

The definitions and proofs add up to about 14K (non-blank, non-comment) lines of code. Out of this, a library for working with de Bruijn indices and a library for reasoning about separation, both of which are reusable, occupy about 2Kloc each. The remaining 10Kloc are split between the kernel (roughly 4Kloc) and its three extensions (roughly 6Kloc).

The current state of the formalization, with which we are fairly pleased, is the result of an iterative process. There were three main iterations.

The initial iteration was Pottier's proof of type soundness for a (sequential) affine type system with mutable state and hidden state [12].

The second iteration was a first version of the definition and proof of *Mezzo*. Compared to the previous iteration, a few important technical simplifications are worth mentioning. First, whereas the previous type system had (singleton and group) regions and a type for region inhabitants (e.g., "x has type $[\rho]$" means that x is an inhabitant of the region ρ), *Mezzo* has value-dependent singleton types (thus, "x has type $=x$"). Second, whereas the previous system required equi-recursive types (which play a role in the meta-theory of the anti-frame rule), *Mezzo* gets away with iso-recursive (i.e., algebraic) data types, whose theory of equality is significantly simpler. Third, whereas the previous proof relied on two calculi, connected by a rather tricky proof that "erasure is a simulation", we were able to work with just one calculus, where it is clear that permissions do not exist at runtime. A snapshot of the formalization of *Mezzo* at this point in time is given by an unpublished paper [13].

In the third and last iteration, we placed greater emphasis on the modularity of the formalization. In the previous iteration, the treatment of mutable memory blocks and that of adoption and abandon were intermingled. The type constructor for memory blocks served also as a type for adopters and as a type for adoptees; this led to a monster typing rule (rule Block [13, Figure 5]). In this iteration, instead, we introduced two new type constructors, `adopts` T and `unadopted`, which deal with these aspects, independently of the structure of memory blocks. This made the system more expressive (e.g., `give` and `take` can now be viewed as ordinary polymorphic functions, whereas their types previously could not be expressed).

This also made the proof more modular: mutable references and adoption and abandon are now almost independent of one another. (They cannot be entirely independent, as adoption and abandon requires embedding an adopter pointer within every memory block.) Finally, we added concurrency and locks, which were not covered by the previous two iterations.

The current proof can be found online [1] and is described by a paper in submission [2]. One unsatisfactory aspect is that, although we would like to think of *Mezzo* as a kernel calculus plus three "almost" independent extensions, in reality the current Coq formalization is monolithic. Achieving true modularity in the syntax of the calculus, in its semantics, in its typing rules, and in the proofs, is difficult and still a research topic [8].

7 What now?

There remain a lot of open questions about *Mezzo* itself.

- Can we design, even if just on paper, a type-checking and type inference algorithm that is sound and complete for a subset of *Mezzo*?
- Can we work out a good mechanism for interacting with existing OCaml code bases?
- Can we offer a better user experience, especially in terms of type error messages?
- Do we need a wider variety of ownership mechanisms, as shown e.g. by Cyclone [19]?

If we were to re-think *Mezzo* along different lines, there are several paths we could explore.

- One might wish to layer a permission analysis on top of an existing typed language, such as a subset of OCaml. The system would be less expressive, but the analysis would be simpler, and violations would be easier to explain. *Mezzo*'s unification of types and permissions was more radical and in a sense more elegant, but this simpler approach is perhaps more reasonable.
- We could design a system that is unsound but still pragmatically useful. Take Dart, for instance; the language features mutable, covariant containers and takes responsibility for it. Yet, the mere fact that it checks lexical scoping of variables is a massive improvement over JavaScript. We could imagine a variant of *Mezzo* that features, for instance, OCaml's weak references without additional checks. This would be unsound, but would help convert existing programs.

Looking back on *Mezzo*, we remain happy with the design of the type system. We believe it is expressive, concise, and that the adoption/abandon mechanism strikes a nice balance between expressiveness and complexity. We also believe that reasoning in terms of ownership is natural for beginners. We do remain unsatisfied, though, with the shortcomings of our implementation, and with the lack of interoperability with existing OCaml code.

References

1 Thibaut Balabonski and François Pottier. A Coq formalization of *Mezzo*, take 2, July 2014. `http://gallium.inria.fr/~fpottier/mezzo/mezzo-coq.tar.gz`.

2 Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. Submitted for publication, July 2014.

3 Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. Submitted for publication, July 2014.

4 Thibaut Balabonski, François Pottier, and Jonathan Protzenko. Type soundness and race freedom for Mezzo. In *Proceedings of the 12th International Symposium on Functional and*

Logic Programming (FLOPS 2014), volume 8475 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2014.

5 Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

6 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–320, 2007.

7 Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *International Conference on Functional Programming (ICFP)*, pages 213–224, 2008.

8 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Principles of Programming Languages (POPL)*, pages 207–218, 2013.

9 Armaël Guéneau, François Pottier, and Jonathan Protzenko. The ins and outs of iteration in Mezzo. Higher-Order Programming and Effects (HOPE), 2013. `http://goo.gl/NrgKc4`.

10 Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

11 Niko Matsakis. Focusing on ownership, 2014.

12 François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23(1):38–144, 2013.

13 François Pottier. Type soundness for Core Mezzo. Unpublished, January 2013.

14 François Pottier and Jonathan Protzenko. *Mezzo*. `http://protz.github.io/mezzo/`, July 2014.

15 Jonathan Protzenko. *Mezzo: a typed language for safe effectful concurrent programs*. PhD thesis, Université Paris Diderot-Paris 7, 2014.

16 Jonathan Protzenko. *Mezzo*-web: try *Mezzo* in your browser, 2014.

17 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

18 Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2000.

19 Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming*, 62(2):122–144, 2006.

Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems

Tiark Rompf[1], Kevin J. Brown[2], HyoukJoong Lee[2], Arvind K. Sujeeth[2], Manohar Jonnalagedda[4], Nada Amin[4], Georg Ofenbeck[5], Alen Stojanov[5], Yannis Klonatos[3], Mohammad Dashti[3], Christoph Koch[3], Markus Püschel[5], and Kunle Olukotun[2]

1 **Purdue University, USA,** `{first}@purdue.edu`
2 **Stanford University, USA,** `{kjbrown,hyouklee,asujeeth,kunle}@stanford.edu`
3 **EPFL DATA, Switzerland,** `{first.last}@epfl.ch`
4 **EPFL LAMP, Switzerland,** `{first.last}@epfl.ch`
5 **ETH Zürich, Switzerland,** `{ofgeorg,astojanov,pueschel}@inf.ethz.ch`

Abstract

Most performance critical software is developed using very low-level techniques. We argue that this needs to change, and that generative programming is an effective avenue to enable the use of high-level languages and programming techniques in many such circumstances.

1998 ACM Subject Classification D.1 Programming Techniques

Keywords and phrases Performance, Generative Programming, Staging, DSLs

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.238

1 The Cost of Performance

Performance critical software is almost always developed in C and sometimes even in assembly. While implementations of high-level languages have come a long way, programmers do not trust them to deliver the same reliable performance. This is bad, because low-level code in unsafe languages attracts security vulnerabilities and development is far less agile and productive. It also means that PL advances are mostly lost on programmers operating under tight performance constraints. Furthermore, in the age of heterogeneous architectures, "big data" workloads and cloud computing, a single hand-optimized C codebase no longer provides the best, or even good, performance across different target platforms with diverse programming models (multi-core, clusters, NUMA, GPU, . . .).

1.1 Abstraction Without Regret

We argue for a radical rethinking of the role of high-level languages in performance critical code: developers should be able to leverage high-level programming abstractions without having to pay the hefty price in performance. The shift in perspective that enables this vision of "abstraction without regret" is a properly executed form of *generative programming*: instead of running the whole system in a high-level managed language runtime, we advocate to focus the abstraction power of high level languages on composing pieces of low-level code, making runtime code generation and domain-specific optimization a fundamental part of the program logic. This design fits naturally with a distinction into control and data paths, which already exists in many systems.

1.2 Towards a Discipline of Generative Performance Programming

While the general idea of program generation is already well understood and many languages provide facilities to generate and execute code at runtime (e.g., via quotation in the LISP tradition and "eval" even in JavaScript), generative programming remains somewhat esoteric – a black art, accessible only to the most skilled and daring of programmers. We believe that generative programming should become a part of every performance-minded programmer's toolbox. What is lacking is an established discipline of practical generative performance programming, including design patterns, best practices, and well-designed teaching material. To make up for this deficiency, we advocate for a research program and education effort, and describe our ongoing work in this direction:

- Compiling queries in database systems (Section 3)
- Protocol and data format parsers (Section 4)
- Delite: A DSL compiler framework for heterogeneous hardware (Section 5)
- Spiral: Synthesis of very high-performance numeric kernels (Section 6)

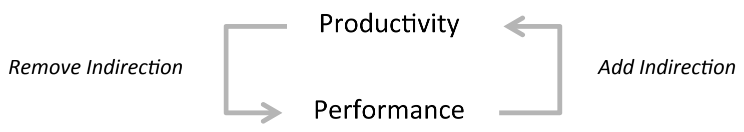
Along the way, we discuss programming patterns such as staged interpreters, mixed-stage datastructures, and how certain language features such as type classes enable powerful generative abstractions. We survey related work in Section 7, and we attempt to synthesize lessons learned and discuss limitations and challenges in Section 8.

2 Background and Context

A famous quote, attributed to David Wheeler, says that:

> "Any problem in computer science can be solved by adding a level of indirection."

What is less widely appreciated is the second part of this quote: "Except problems that are caused by too many levels of indirection". In practice, many of these problems are performance problems. More generally, there appears to be a fundamental conflict between performance and productivity: to increase productivity we need to add indirection, and to increase performance, we need to remove indirection.



2.1 About Performance

The business of program optimization is one of diminishing returns: more effort leads to increased efficiency, but further gains come at an exponentially higher price. Often, the biggest gains can be achieved by choosing a better algorithm. Thus, in the most common case, a programmer implements a work-optimal algorithm, and relies on a compiler to create an efficient mapping to architectural and microarchitectural details.

The problem with the former is that "work-optimal" is often only established asymptotically, thus ignoring constants, the choice of data structure, or locality. The problem with the latter is that compilers are automatic systems that need to work for all programs, and thus cannot be expected to deliver the best possible results for any particular program. The ability of a compiler to optimize a given program depends very much on the programming

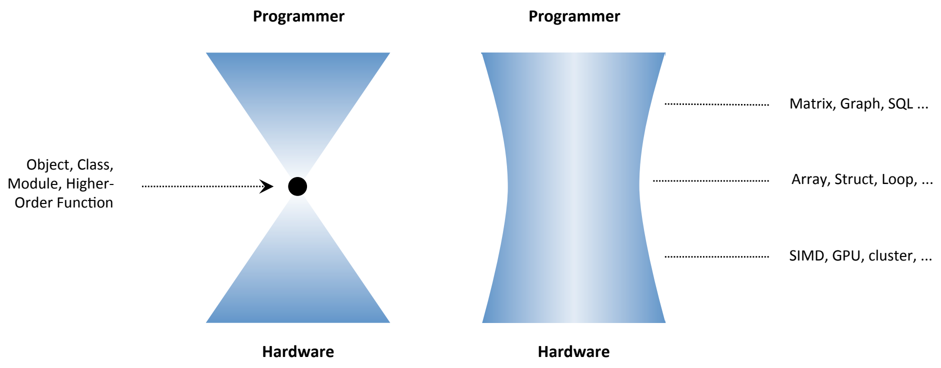


Figure 1 General purpose compilers (left) vs. DSL compiler toolchains (right).

language and on the particular programming style. In general, high-level languages are partly interpreted, with just-in-time compilers of varying sophistication. Programs in high-level languages can easily be 10x to 100x slower than equivalent C programs, and the individual coding style makes a big difference. In general, the more high-level features and indirection is used (objects, classes, higher-order functions, ...) the bigger the price in performance, because indirection fundamentally stands in the way of automatic program analysis. Thus, after algorithmic changes, a big performance boost can be obtained by rewriting a program in a low-level style, and moving "closer to the metal" by eliminating other overheads such as unnecessary data transfers. To give one example, a recent study [74] shows that simple single-threaded Rust programs can outperform big data processing frameworks running on a cluster of several hundred nodes on important graph applications. This inefficiency translates directly into higher energy consumption and data center bills.

But even reasonably optimized C implementations can be suboptimal by a large margin [12, 75], and speed-ups of sometimes 2x, 5x, or more can be achieved with careful tuning to the microarchitecture and employment of the right coding style – a non portable approach. The situation worsens considerably when parallelizing or distributing computation to homogeneous or heterogeneous processors. Here, a single C codebase is no longer sufficient as each class of devices comes with its own specific programming model (Pthreads, MPI, OpenMP, CUDA). Effectively, separate implementations are needed for each architecture, and for best performance, different heterogeneous devices may need to be combined (e.g. clusters of machines with CPUs and GPUs), which exacerbates the problem.

Compilers are unable to target this multitude of architectures automatically, mostly because they lack domain knowledge: general purpose languages provide generic abstractions such as functions, objects, etc. on top of which program-specific concepts are built, but they do not expose these more specific concepts to the compiler. Thus, the compiler is unable to restructure algorithms or rearrange data layouts to map well to a hardware platform, and cope with the large number of choices in transformations, data representations, and optimizations. This "general purpose bottleneck" is visualized in Figure 1 and contrasted with a domain-specific approach, which enables a compiler to reason about and optimize programs on multiple levels of abstraction.

2.2 About Staging

Generative or multi-stage programming (*staging* for short), goes back at least to Jørring and Scherlis [53], who observed that many programs can be separated into stages, distinguished by frequency of execution or by availability of data. Taha and Sheard [109] made these stages explicit in the programming model and introduced MetaML as a language for multi-stage programming. A staged computation does not immediately compute a result, but it returns a program fragment that represents the computation, and that can be explicitly executed to form the next computational stage. MetaML's staging operators *bracket*, *escape* and *run* are modeled after the quote, unquote and eval facilities known from LISP and Scheme.

The presentation in this paper uses Scala and LMS (Lightweight Modular Staging) [95], a staging technique based on types. LMS is implemented as a library (hence *lightweight*), and instead of syntactic quotations, it uses Scala's overloading facilities to build staged computations. Staged expressions can be further processed and unparsed into source code in Scala or a different language, including C. Since all staged operations are defined in library code, it is easy to restrict operations to those with a C counterpart (hence *modular*). Even C constructs with no Scala counterpart (e.g. pointers) can be represented abstractly, to get the same low-level behavior and performance. The key benefit of staging is that the present-stage code can be written in a high-level style, yet generate future-stage code that is very low-level and efficient. Staging is a programmatic way to remove indirection – when generating code in one step, Scala becomes a glorified macro system to generate C code.

2.3 About DSLs

Traditionally, the appeal of DSLs is in increasing productivity by providing a higher level, more intuitive programming model for domain experts, who are not necessarily expert programmers ("user-facing" DSLs). While this is an important direction and good tool support exists (e.g. language workbenches like Spoofax [57]), our interest in DSLs is as a vehicle for exposing knowledge about high-level program structures to a compiler. The effectiveness of a DSL-based program generation approach was demonstrated, among others, by the original Spiral system, which used a complete DSL-based generative approach for the automatic parallelization and locality optimization of linear transforms [83, 84]. In such systems, DSLs are implementation details, much like intermediate representations (IRs) in a compiler, that enable specific analyses and transformations at different levels of abstractions (Figure 1).

Staging in the style of LMS is a natural fit for these "internal" DSLs: instead of generating target code directly, one simply generates expressions in a domain-specific intermediate language. The key benefit of staging remains: computation at staging time, while the DSL program is assembled, is "free", i.e. without cost at DSL runtime. Thus, the DSL does not need to include features like higher order functions, objects, etc. which are hard to analyze, but it can focus exclusively on the elements of interest (e.g. matrices, graphs, SQL queries).

The second benefit of staging in a DSL context comes in when we consider how to translate from one DSL representation to the next lower-level one. In the partial evaluation community, it is well known that specializing an interpreter yields a compiler (the first Futamura projection [35]). Thus, a staged interpreter is a translator from one language to another: in other words, we can translate from one intermediate DSL to the next by defining an interpreter for the first language, and staging it [96, 81]. This drastically reduces the effort required to implement sophisticated multi-pass compiler toolchains.

With a properly designed layered approach, building new DSLs becomes simpler as well. Instead of starting from scratch, it is sufficient to implement a DSL front-end, consisting of the required datatypes, optimizations, and translation to an already existing backend [107].

2.4 Convergence: Generative Performance Programming

DSLs and generative approaches have been around for quite some time, with a number of successes but with few examples of more mainstream adoption. SQL is perhaps the biggest DSL success story: by restricting their interface to a well defined, restricted, language, database systems are able to leverage elaborate query optimization techniques. Traditionally, database systems stop short of generating machine code to run queries, but recently query compilation is seeing a surge of interest. We will come back to this in Section 3. A recent system that is rapidly making an impact in industry is Halide [88, 87], which generates fast image processing kernels from high-level algorithmic descriptions.

We believe that generative techniques for performance optimization are an exciting area of research with the potential to deliver important tools for real world software development. The reason is in the concurrent evolution of three trends:

- **Hardware**: With contemporary and emerging architectures, the pain involved in achieving high performance has increased dramatically. Frequency scaling, and with it free speed-up, has ceased a decade ago. Systems are becoming increasingly parallel, heterogeneous, and distributed, with diverse programming models. This means that mapping algorithms optimally is more difficult than ever, and likely requires new solutions, even if domain-specific.
- **Applications**: With a shift towards big data workloads in the cloud and a proliferation of mobile devices and embedded systems, there is a growing demand for highly efficient software. Mobile users expect an “always on” experience and are growing accustomed to applications based on sophisticated machine learning algorithms that process data in near realtime. In addition to the traditional latency and throughput requirements, this demand for efficiency is increasingly driven by concerns of energy consumption, which often dominates the operating costs of a system.
- **Programming Languages**: High-level languages focus increasingly on generality and abstraction, allowing programmers to build large systems from simple but versatile parts. On the one hand, this makes it even harder for compilers to translate high-level programs to efficient code. But on the other hand, these highly expressive languages enable the sophisticated meta-programming techniques that are needed for effective generative programming.

The bottom line of our approach can be summarized succinctly as follows:

Use Indirection and Abstraction to Solve Performance Problems, too.

3 Case Study: Databases and Query Processors

Popular open-source and commercial database systems have been shown [123, 104] to perform 10 to 100x worse on certain queries than specialized, hand-written C implementations of the same query. At the same time such systems contain hundreds of thousands of lines of optimized C code, with a lot of manual specialization. On last count in a recent version of the PostgreSQL server, there were about 20 distinct implementations of the memory page abstraction and 7 implementations of B-trees [61]. This form of human inlining and specialization creates a maintenance nightmare but is probably justified by improved performance.

Why, then, are database systems still falling short of hand-written queries? One reason is that most systems interpret query plans, operator by operator and record by record. Clearly this layer of interpretation can be a bottleneck. So why aren't databases using compilers? Because compilers are way too hard to implement! In fact this is a well-known part of database folklore: The first relational DBMS, IBM's System R, initially compiled its query plans, but before the first commercial release, changed to interpretation. The reason was that code generation for the large set of query techniques being investigated was incredibly painful. Nowadays, among mainstream DBMS, only data stream processing systems such as IBM Spade or StreamBase use compilation. The reason here is that ultra-low latencies are necessary and justify the inconvenience of creating a compiler. Still, compilation has received renewed interest [60, 61, 78, 76] and very recently, compilation based systems have started to appear again (e.g. Microsoft Hekaton, Cloudera Impala and MemSQL).

The good news is that generative programming offers a generic recipe to turn interpreters into compilers: staging an interpreter enables us to specialize it with respect to any program. The result of specialization is that the interpreter is dissolved and only the computation of the interpreted program remains as residual generated code [36, 1]. We discuss two research databases systems and one tutorial system that use this technique next.

3.1 DBToaster and LegoBase

DBToaster incrementalizes query evaluators and generates low-level C++ or Scala code. The first compilation stage turns SQL queries into update event triggers. These triggers prescribe how to efficiently refresh a materialized view of the query as the base data in the database changes. The second compiler stage optimizes the event triggers and generates efficient code. Experimental results show that DBToaster improves the performance of IVM by several orders of magnitude compared to the state of the art [2]. The second-stage compiler was originally written in about 15k lines of OCAML code; recently, it was rewritten in 2k lines of Scala/LMS code. Combined with other optimizations, the query running times improved by one to two further orders of magnitude compared to the results from [2].

LegoBase (a joint project between Oracle Labs and EPFL) is an analytic query engine [59] developed from scratch using Scala and LMS. With just about 3000 lines of Scala code, it runs all 22 queries from the industry-standard TPCH benchmark, achieving up to 20x speedups over a commercial database system, and competetive performance in comparison to other state-of-the-art query compilers that are implemented using LLVM, in a much more low-level style [78]. LegoBase removes all interpretive overhead by performing whole-query optimization, and generates specialized low-level datastructures (e.g. hash tables) based on the query and data schema.

3.2 The Essence of a SQL Compiler in 500 LOC

The LegoBase design went through several iterations, and we have distilled the essence into a generative programming tutorial (presented at CUFP'15[1]). By removing functionality that was just a variation of a common theme or which did not offer specific insights (no sorting, only inner joins, ...) we were able to implement an end-to-end SQL compiler in just under 500 lines of Scala [92].

Our starting point–without any compilation–is a generic library function to read CSV files. A CSV file contains tabular data, where the first line defines the schema, i.e. column names. We would like to iterate over all rows in a file and access data fields by name:

[1] `http://scala-lms.github.io/tutorials/query.html`

```
processCSV("data.txt") { record =>             // sample data:
  if (record("Flag") == "yes")                 // Name, Value, Flag
    println(record("Name"))                    // A, 7, no
}                                              // B, 2, yes
```

Records are objects of the following class, which carries both the field data and the schema, and enables lookup by key:

```
class Record(fields: Array[String], schema: Array[String]) {
  def apply(key: String) = fields(schema indexOf key)
}
```

This record class enables a nice high-level programming style but unfortunately it comes at a high price. The code above runs much slower than just writing a specialized `while` loop:

```
while (lines.hasNext) {
  val fields = lines.next().split(",")
  if (fields(2)) == yes) println(fields(0))
}
```

Being generic means that a system contains interpretive structure. In this example, we are interpreting the schema that is read from the file. In a DBMS, queries are optimized at the SQL level and then translated to low-level execution plans, which are interpreted, operator by operator.

3.2.1 Interpreter + Staging = Compiler

Let us turn our CSV reader into a query engine that can run simple SQL queries. Our example above would be expressed as

```
SELECT Name FROM data.txt WHERE Flag == 'yes'
```

and we assume that we already receive a parsed (and possibly optimized) structured representation, the execution plan. In this case:

```
Print(
  Project(List("Name"))(
    Filter(Eq(Field("Flag"),Const("yes")))(
      Scan("data.txt"))))
```

The operators are arranged in a tree, and apart from `Scan`, each of them has a parent from which it obtains a stream of records. We can implement a query interpreter as follows:

```
def eval(p: Predicate)(rec: Record): Boolean = ... // elided
def exec(o: Operator)(yld: Record => Unit) = o match {
  case Scan(file)             => processCSV(file)(yld)
  case Filter(pred)(parent)   => exec(parent) { rec => if (eval(pred)(rec)) yld(rec) }
  case Project(fields)(parent) => exec(parent) { rec => yld(fields map (k => rec(k))) }
  case Print(parent)          => exec(parent) { rec => println(rec.fields mkString ",") }
}
```

We now turn this interpreter into a compiler by adding staging using LMS (Lightweight Modular Staging) [94]. As we will see, the necessary modifications are rather minor.

3.2.2 Mixed-Stage Data Structures and Functions

LMS is a staging technique driven by types. The type `Rep[T]` represents a delayed computation of type `T`. Thus, during staging, a bare "static" type `T` means "executes now", while a wrapped "dynamic" type `Rep[T]` means "generate code to execute later". We tweak the `Record` class so that only the fields are "dynamic":

```
class Record(fields: Rep[Array[String]], schema:Array[String]) {
  def apply(key:String) = fields(schema indexOf key)
}
```

Now record objects exist purely at staging time, and never become part of the generated code. For our example, the generated code will be exactly the efficient `while` loop shown above, even if we are starting from a SQL query as our input. Let us look at the definition of procedure `processCSV`, which forms the core of our data processing engine:

```
def processCSV(file: String)(yld: Record => Rep[Unit]) = {
  val lines = FileReader(file); val schema = lines.next.split(",")
  run { while (lines.hasNext) {
      val fields = lines.next().split(",")
      yld(new Record(fields,schema))
  }}}
```

The type of the closure argument `yld` is `Record => Rep[Unit]`, a present-stage function. Invoking it will inline the computation, a key difference to a staged function of type `Rep[A=>B]`, which results in a function call in the generated code. The `while` loop is *virtualized* [93], i.e. overloaded to yield a staged expression when the condition or loop body is a `Rep` expression.

This is essentially all that is required to convert our query interpreter into a query compiler. Using types to drive staging decisions enables us to mix present-stage and future-stage code quite freely, without the syntactic noise introduced by quotation brackets. The full code adds data structure specialization, join and aggregation operators, and optimized input processing using memory-mapped IO, all in about 500 lines of Scala [92].

4 Regular Expressions and Parser Combinators

Data is not only stored but also transferred. Fast encoding and decoding of data formats is therefore a key concern in data processing pipelines. The accepted standard is to write protocol parsers by hand. Parser generators, which are common for compilers, are not frequently used. One reason is that many protocols require a level of context-sensitivity (e.g. read a value n, then read n bytes), which is not readily supported by common grammar formalisms. Many open-source projects, such as Joyent/Nginx and Apache have hand-optimized HTTP parsers, which span over 2000 lines of low-level C code. From a software engineering standpoint, big chunks of low-level code are never a desirable situation. First, there may be hidden and hard to detect security issues like buffer overflows. Second, the low-level optimization effort needs to be repeated for new protocol versions or if a variation of a protocol is desired: for example, a social network mining application may have different parsing requirements than an HTTP load balancer, although both process the same protocol.

Parser combinators could be a very attractive implementation alternative, but are usually simply too slow. Their main benefit is that the full host language can be used to compose parsers, so context sensitivity and variations of protocols are easily supported. We have recently shown how parser combinators can be implemented in a generative style [52], yielding parser generator combinators. Clever use of present-stage and staged functions enables recursive grammars and prevents code size explosion.

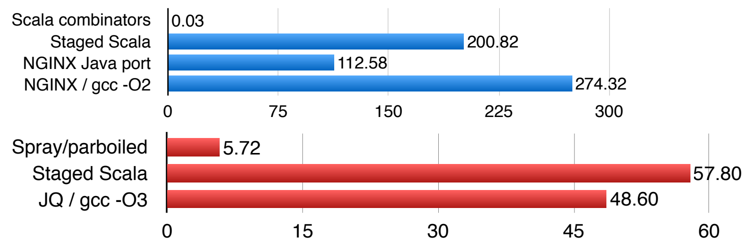


Figure 2 Parser performance (throughput in MB/s) for HTTP (top) and JSON (bottom).

We evaluate staged parser combinators against hand-written parsers for HTTP and JSON data from the NGINX and JQ projects. Our generated Scala code, running on the JVM, achieves HTTP throughput of 75% of NGINX's low-level C code, and 120% of JQ's JSON parser. Other Scala tools such as Spray are at least an order of magnitude slower. The standard Scala combinator implementation is 4 orders of magnitude slower (Figure 2).

Staged parser combinators can be implemented for different parsing styles. Communication protocols are designed to be unambiguous, so recursive decent works well and does not cause backtracking. For ambiguous grammars, we have also studied bottom up parsers [52].

An interesting special case of parsing are regular expressions. We have shown that NFA to DFA conversion can be expressed as a staged interpreter. We achieve speedups of 2x over optimized automata libraries, 100x over the JDK implementation, and more than 2000x over unstaged Scala code [96].

5 Delite: DSLs for Heterogeneous Targets

While SQL and C code generation are important use cases, contemporary "big data" workloads require more flexibility. On the application side, we need to deal with machine learning algorithms and graph processing. On the target side, we need to support GPUs, NUMA machines, and clusters. General purpose compilers are a bottleneck, unable to translate all these applications efficiently to different targets.

Delite [15, 97, 67, 68] is an extensible compiler framework for building embedded domain specific languages (DSLs). Delite supports a versatile intermediate language of parallel patterns called "Delite Ops" (including e.g. data-parallel map, filter, and reduce abstractions) to which domain-specific operations are translated, and code generation as well as runtime support for heterogeneous targets. Multiple DSLs can be used together in a single program, since all of them are translated to the common IR [107]. On the IR level, sophisticated optimizations like loop fusion and loop nesting optimizations are shared by all DSLs.

To give an example of the types of transformations that can be performed, here are two different ways of writing the computation loop (one iteration) of k-means clustering using data-parallel patterns. For shared-memory execution, the rows of the large dataset (`matrix`) are clustered, with data implicitly shuffled via the indexing operation `matrix(as)`:

```
val assigned = matrix.mapRows { row =>
  oldClusters.mapRows(centroid => dist(row, centroid)).minIndex
}
val newClusters = oldClusters.mapIndices { i =>
  val as = assigned.filter(a => a == i)
  matrix(as).sumRows.map(s => s / as.size)
}
```

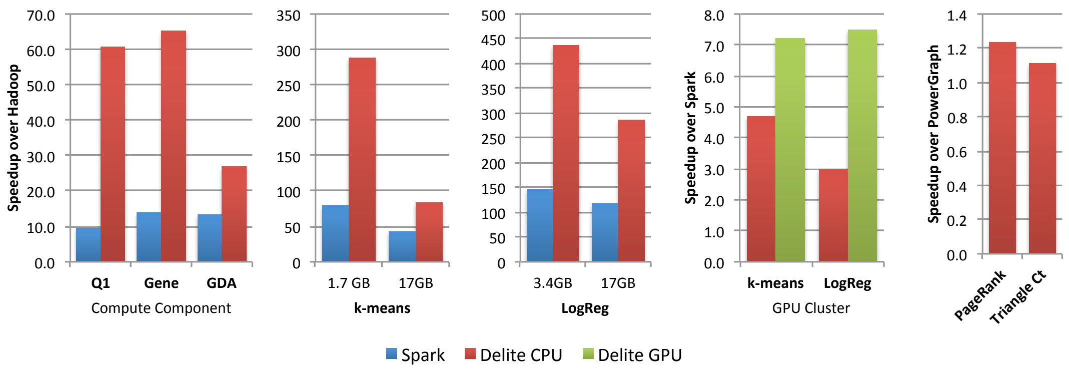


Figure 3 Delite performance for machine learning and graph apps on heterogeneous clusters.

For distributed-memory execution, however, we shuffle data explicitly via `groupBy`:

```
val clusteredData = matrix.groupRowsBy { row =>
  oldClusters.mapRows(centroid => dist(row, centroid)).minIndex
}
val newClusters = clusteredData.map(e => e.sum / e.size)
```

This way the required data movement is explicitly visible to the compiler and Delite can then reason about how to automatically partition `matrix` and the intermediate results across distributed memories and perform the required data movement. We have developed a number of DSLs for different domains, including OptiML (machine learning), OptiGraph (graph processing), OptiQL (SQL-like data querying), and OptiWrangler (data cleaning). All of these exhibit very competetive performance compared to stand-alone systems and other DSLs [106, 107]. Delite DSLs achieve high sequential performance via LMS optimizations and systematic removal of high-level domain abstractions. Parallel scalability is provided by efficient implementations of Delite Ops for multiple hardware platforms as well as the locality-improving loop transformations Delite performs on these Ops.

We show experiments comparing Delite, Spark [122], and Hadoop [11] performance on a 20 node Amazon cluster for machine learning applications, and comparing to PowerGraph [40] for graph analytics applications on a 4 node cluster connected within a single rack. The results are shown in Figure 3.

Extremely large speedups compared to Hadoop are possible for applications that perform multiple passes over a single dataset. While Hadoop loads the data from disk every iteration, Delite and Spark both keep the data in memory. Delite also shows large speedups over Spark (a Scala library) due to efficient code generation provided by LMS compared to Scala library overheads. Speedup over PowerGraph is very modest due to the graph applications being bound by network communication rather than computational efficiency.

6 Synthesis of High-Performance Numeric Kernels

For some ubiquitous computation kernels (e.g. basic linear algebra routines, FFTs, filters, Viterbi decoders), there is high demand for not only good, but absolute peak performance. The difference between a regular optimized implementation and the best known one can be staggering: for FFT kernels more than 10x, using the same algorithm and the same number of floating point operations. The difference is mostly due to low-level details such as locality optimizations, SIMD vectorization, and instruction scheduling.

```
type T = Double
add[Rep,Real,NoRep,SISD,T]
```

```
#define T double
#define N 4
void add(T* x, T* y, T* z) {
  int i = 0;
  for(; i < N; ++i) {
    T x1 = x[i];
    T y1 = y[i];
    T z1 = x1 + y1;
    z[i] = z1;
  }
}
```

(a) Staged SISD Array

```
type T = Double
add[NoRep,Real,Rep,SISD,T]
```

```
#define T double
void add(T* x, T* y, T* z) {
  T x1 = x[0]; T x2 = x[1];
  T x3 = x[2]; T x4 = x[3];
  T y1 = y[0]; T y2 = y[1];
  T y3 = y[2]; T y4 = y[3];
  T z1 = x1 + y1;
  T z2 = x2 + y2;
  T z3 = x3 + y3;
  T z4 = x4 + y4;
  z[0] = z1; z[1] = z2;
  z[2] = z3; z[3] = z4;
}
```

(b) SISD Array of Staged Doubles

```
type T = Double
add[Rep,Real,NoRep,SIMD,T]
```

```
#define T double
#define N 1
void add(T* x, T* y, T* z) {
  int i = 0;
  for(; i < N; ++i) {
    __m256d x1, y1, z1;
    x1 = _mm256_loadu_pd(x + i);
    y1 = _mm256_loadu_pd(y + i);
    z1 = _mm256_add_pd(x1, y1);
    _mm256_storeu_pd(z + i, y1);
  }
}
```

(c) Staged SIMD Array

```
type T = Double
add[NoRep,Real,Rep,SIMD,T]
```

```
#define T double
void add(T* x, T* y, T* z) {
  __m256d x1, y1, z1;
  x1 = _mm256_loadu_pd(x + 0);
  y1 = _mm256_loadu_pd(y + 0);
  z1 = _mm256_add_pd(x1, y1);
  _mm256_storeu_pd(z + 0, y1);
}
```

(d) SIMD Array of Staged Doubles

Figure 4 Different code styles generated from single vector add function (taken from [103]).

Producing optimized kernels for such numeric operations is one of the established use cases for program generation [75] with early examples including ATLAS [120], FFTW codelet generator [34], and the aforementioned Spiral [83, 84]. All three were developed from scratch as stand alone tools, which are hard to extend, reuse, combine, or maintain.

To investigate whether a) these generators can be implemented using embedded DSLs and staging, and b) what proper language support can offer for this application, we reimplemented a subset of Spiral using Scala and LMS, leading to a draft proposal for a more systematic approach to constructing this kind of program generators [81]. Spiral was chosen because it is almost completely DSL-based. In particular, Spiral uses three internal DSLs: one to capture recursive algorithms as breakdown rules, one to mathematically express loops and access patterns, and one a `C`-like intermediate representation. Algorithmic choice arises from different combinations of breakdown rules and structural optimizations for locality and parallelism are performed by rewriting on the second DSL.

The main results of this work were a) a staging approach to translate a DSL into another DSL, b) the use of type classes to abstract over data type and code style used in the generated code. We briefly survey the last contribution.

6.1 Type Classes and Generic Programming

Generated libraries often need to support multiple input/output data formats, such as interleaved, split, or C99 format of complex number arrays. A key result of the Spiral-LMS prototype has been to show that established generic programming techniques such as Haskell-style type classes [119], applied in a generative style, are well suited to model this diversity.

But type classes are even more powerful in the context of LMS, as they make it possible to abstract over staging decisions. In particular, some necessary but tedious low-level transformations such as loop unrolling with scalar replacement, selective precomputation or specialization to partially known input can all be expressed as instantiations of a generic `CVector` class with suitable type parameters.

More recent work has shown that this approach can also cover abstraction over SIMD architectures [103]. In Figure 4, we show how a generic `add` function can generate different code shapes depending on the type parameter instantiations. The definition of the `CVector` class and `add` function is given below:

```
class CVector[V[_], E[_], R[_], P[_], T](...) {
  type Element = E[R[P[T]]]
  def  size   (): V[Int]
  def  apply  (i: V[Int])  : Element
  def  update (i: V[Int], v: Element)
}
```

```
def add[V[_], E[_], R[_], P[_], T] (
  (x, y, z) : Tuple3[CVector[V,E,R,P,T]]
) = {
  for ( i <- 0 until x.size() )
    z(i) = x(i) + y(i)
}
```

- `T`: array primitive (double, float, etc).
- `P[_]`: SIMD or SISD array.
- `R[_]`: staged or non-staged elements of the array.
- `E[_]`: complex or real array elements.
- `V[_]`: encodes whether array accesses will be visible in the target code element operations.

Each type parameter, defined in the `CVector` class, is accompanied by a corresponding implicit type class instance, which defines the shape of `CVector` upon instantiation:

```
type NoRep[T] = T
class Staged_SISD_Double_Vector extends CVector[Rep  , Real, NoRep, SISD, Double]
class Scalar_SISD_Double_Vector extends CVector[NoRep, Real, Rep,   SISD, Double]
class Staged_SIMD_Double_Vector extends CVector[Rep  , Real, NoRep, SIMD, Double]
class Scalar_SIMD_Double_Vector extends CVector[NoRep, Real, Rep,   SIMD, Double]
```

The main prerogative of the `CVector` class is to generate code in lower level DSLs, `C-IR` and `I-IR`. The `C-IR` DSL represents a `C`-like intermediate language representation, and `I-IR` represents ISA-specific `C` intrinsics. Both DSLs model low level variable and array manipulations as well as arithmetic and binary operations, provided as extension to LMS.

```
val IR = new CIR (); IR.setISA(AVX2); import IR._
```

Once concrete `CVector` classes are created, the abstraction layer deals with a concrete ISA, specified in the `CIR` class, concrete staging decisions, and a concrete array type.

```
var x, y, z = new Staged_SISD_Double_Vector ()     var x, y, z = new Staged_SIMD_Double_Vector ()
var x, y, z = new Scalar_SISD_Double_Vector ()     var x, y, z = new Scalar_SIMD_Double_Vector ()
```

The `add` function operates on `CVector` elements constrained by the provided types. Type information propagates through each arithmetic operation, generating ISA-specific `C-IR` and `I-IR`. As a result, once C code is emitted, we obtain different versions (Figure 4) from a single source:

```
emitCode(add(x, y, z))
```

Abstracting over SIMD architectures enables the generator developer to deal with the higher-level specification using SIMD agnostic and architecture agnostic expressions. While SIMD-specific optimizations are required to achieve the highest performance, the abstraction layer is able to automatically invoke the particular architecture optimization, when the lower `I-IR` / `C-IR` is generated. As these optimizations are implemented in a modular fashion, they can be reused in the context of the abstraction layer, and also as stand-alone optimizations for building generators directly.

7 Related Work

Embedded DSLs. Embedded languages have a long history [64]. Hudak introduced the concept of embedding DSLs as pure libraries [45, 46]. Steele proposed the idea of "growing" a language [102]. The concept of linguistic reuse goes back to Krishnamurthi [63]; Language virtualization to Chafi et al. [18]. The idea of representing an embedded language abstractly as methods (finally tagless) is due to Carette et al. [16] and Hofer et al. [44], going back to much earlier work by Reynolds [89]. Compiling embedded DSLs through dynamically generated ASTs was pioneered by Leijen and Meijer [69] and Elliot et al. [31]. All these works greatly inspired the development of LMS. Haskell is a popular host language for embedded DSLs [108, 39], examples being Accelerate [73], Feldspar [7], Nikola [72]. Recent work presents new approaches around quotation and normalization for DSLs [77, 22]. Other performance oriented DSLs include Firepile [80] (Scala), Terra [28, 29] (Lua). Copperhead [17] (Python). Rackets macros [112] provide full control over the syntax and semantics: Typed Racket [111] is implemented entirely as macros. DSLs are also used to target hardware generation, for example Lime [6], Chisel [8], Bluespec [5], or based on Delite/LMS [38, 37]. Related work on low-level systems oriented programming in high-level languages includes [33] and the Singularity operating system [65].

C++ Templates. Metaprogramming facilities exist in many languages, including C++ templates [113]. Expression Templates [114] can produce customized generation, and are used by Blitz++ [115]. Veldhuizen introduced active libraries [116], which are libraries that participate in compilation. Kennedy introduced telescoping languages [58], efficient DSLs created from annotated component libraries. TaskGraph [10] is a meta-programming library that sports run-time code generation in C++. Bassetti et al. review some of the challenges and benefits of expression templates[9]. Examples of successful packages are the Matrix Template Library [99], POOMA [56], ROSE [26] and the portable expression template engine PETE [25]. Many of these contain reusable generic components.

Compiler Optimizations. Compiler transformations in LMS [96] leverage work on rewrite rules [14], on combing optimizations in a modular fashion [70, 117, 23], on loop fusion [41], and on eliminating intermediate results [118, 24]. Other work focuses on tiling and loop nests[121][3]. Extensible compiler tools include Polyglot [79] and JastAdd [30]. The Glasgow Haskell Compiler also supports custom rewrite rules [51]. ROSE [86] derives source-to-source optimizations from semantics annotations on library code. COBALT [71] is a domain-specific language for optimizations amenable to automated correctness reasoning. Tate et al. [110] generate compiler optimizations automatically from program examples.

Cluster and Parallel Programming. There are many cluster programming frameworks, including MapReduce [27], Hadoop [11], Spark [122], Dryad [48]. Data parallel programming languages include NESL [13], SISAL [32], SaC [98], Fortress [55], Chapel [19], Data-Parallel Haskell [50], High Performance Fortran [43], and X10 [21]. Implicit parallelization framworks include Array Building Blocks [47], DryadLINQ [49], FlumeJava [20]. Another line of research focuses on irregular workloads [82].

Program Generators. A number of high-performance program generators have been built, for example ATLAS [120] (linear algebra), FFTW [34] (discrete fourier transform), and Spiral [85] (general linear transformations). Other systems include PetaBricks [4], CVXgen [42] and Halide [88, 87].

8 Outlook and Conclusions

In the preceding sections, we have presented several case studies of DSL-based generative performance programming with Scala and LMS. We could have mentioned more projects, e.g. generative programming for the web [62, 90] and domain-specific hardware generation [37, 38], but considered them beyond the scope of this paper.

What has enabled these results? First, we built on decades of prior art in DSLs, staging, and performance optimization, some of which we surveyed in Section 1 and Section 7. Second, all surveyed projects are collaborative efforts between experts in PL, architecture, DB, and performance optimizations. Third, the use of a common infrastructure enabled cross pollination and reuse of knowledge and code. Fourth, the embedding in Scala as host language (versus external DSLs) gave us the necessary flexibility to customize and the ability to reuse prior Java and Scala libraries. We discuss these and other important aspects below.

8.1 Modularity, Reuse, and Low Cognitive Overhead

One size does not fit all: all systems we presented use LMS with some slight variations. It is important that a system provides good core functionality out of the box but can easily accommodate extensions and modifications.

It is also important that the subjective overhead of generative programming is low, i.e. program generation should “feel” natural to programmers. While it may appear as a small detail, we have found that being able to freely mix present-stage and future-stage code with low syntactic overhead is extremely helpful. In LMS, the types in method signatures provide guidance as to which arguments are staged and which are not, as for other types in normal Scala code. As always, reading code is at least as important as writing code.

In Scala we achieve almost seamless language virtualization: allowing built-ins such as conditionals or loops to be overloaded like ordinary methods, so that they can create staged expressions. The necessary language support (Scala-Virtualized [93]) can be implemented as a modified compiler or using Scala macros.

8.2 Staging Should Preserve Semantics

Let us consider our `Record` abstraction from Section 2 with fields of type `Rep[T]`, this time using explicit quotation syntax:

```
val fields = <| lines.next().split(",") |>
yld(new Record(fields,schema))
```

If `Rep[T]` merely represents a quoted code fragment, every access to a record field may duplicate the computation! This is not only costly, but also not semantics-preserving with respect to termination and side effects. In this example:

```
processCSV("data.txt") { rec =>   <| print(${ rec("Name") }) + rec(${ "Flag") }) |>  }
```

The code fragment containing `lines.next()` would be executed twice for each record – clearly not the intended behavior. The problem is that quotation is usually a purely syntactic mechanism. To achieve semantic guarantees for realistic, call-by-value, computations we have proposed to make quotation context-sensitive [91]. We express quotation using two operators, `reflect` and `reify`, with the following reduction semantics:

```
(1)  <| foo ${ bar } baz |>         -->  reflect(" foo {" + reify { bar } + "} baz ")
(2)  reify { E[ reflect("str") ] }  -->  "val fresh = str; " + reify { E[ Code("fresh") ] }
(3)  reify { Code("str") }          --> "str"
```

Here, `E[.]` denotes a `reify`-free execution context and `fresh` a fresh identifier. As we can see, each quotation is immediately bound to a fresh identifier in the generated code, and only identifiers are passed around as `Rep` values. Thus, the evaluation order in the generated code mirrors the evaluation order of the quotations.

This development is a natural extension of previous work on quotations: Lisp introduced unhygienic `quote`/`unquote`/`eval` operators; MetaML [109] provided guarantees about the binding structure; LMS additionally maintains evaluation order.

8.3 One-Pass Code Generation is not Enough

Most previous work on staging and generative programming focused on code generation as a single pass. However, a single-pass code generator is fundamentally disadvantaged compared to a multi-pass compiler pipeline. As indicated in Section 2, low-level code is not enough to achieve best performance – for this one needs to build compiler systems around domain-specific intermediate languages. Potentially there are multiple levels of such DSLs to capture different optimization opportunities.

The key insight here is that staging greatly simplifies building such multi-pass DSL compilers: just as one can stage interpreters for regular expressions or SQL queries, many program transformations can be expressed as interpreters for intermediate languages, which are potentially annoted with auxiliary information derived by program analysis passes.

8.4 Challenges and Limitations

Understanding the limitations of a proposed techniques is key to using it effectively. First of all, we wish to note that we do not propose a silver bullet that will magically accelerate programs without help from the programmer.

Program generation shares some ideas with program synthesis, but is in a sense less ambitious. Whereas program synthesis promises to come up with an efficient and correct program automatically, generative programming still requires programming, i.e. the implementation of an algorithm. However, the tedium of manually applying performance optimizations is removed. Staging is also less ambitious than automatic just-in-time (JIT) compilation. Whereas JIT compilation aspires to identify chunks of code that are profitable to compile automatically based on runtime profiling, staging delegates such decisions to the programmer. The result is a deterministic performance model and strong guarantees about the shape of compiled code and when compilation happens.

Staging and DSLs work well if there is indeed a stage distinction: some piece of code is run much more often than another or after certain crucial data becomes available that enables optimization. A challenge in the deployment in real systems is that generated code must amortize the code generation and compilation time. Thus, staging only pays off if compiled code is used often or a single run offsets the generation cost. In Spiral, for example, code is generated offline and used many times. Many real-world systems have a natural separation into control and data paths. In these cases staging is a very good fit. In SQL databases, some queries from the TPCH benchmark take minutes to run and query compilation needs at most seconds. OptiMesh, a Delite DSL, performs iterative computations on large meshes, which would be altogether infeasible without compilation.

When going beyond single-pass code generation, a suitable DSL or potentially a stack of DSLs is needed that captures the essential degrees of freedom that can be exploited for optimization. Designing such DSLs is a creative process, and no tool can completely eliminate this effort. However, tools can help building these DSLs and implementing the corresponding transformations. Defining a DSL in LMS requires some boilerplate, which has lead us to build Forge [105], a meta-DSL that can generate DSL implementations from declarative specifications. YinYang [54] is another approach to generate necessary definitions automatically.

Generative programming itself has a learning curve. Our experience suggests that students can be productive within a semester without prior experience with LMS. While effective generative programming demands a certain way of thinking, and teasing apart what should be static and what dynamic requires cleverness at times, we believe that generative programming is not fundamentally harder than other kinds of programming. It is easy to forget that in the early days of OOP, concepts like objects and inheritance were difficult to grasp for many programmers.

8.5 A Call to Action

As mentioned in Section 1 and throughout the paper, we argue for a rethinking of the role of high-level languages in performance critical code. In particular, as our work and the work of others have shown, generative abstractions and DSLs can help in achieving high performance by programmatically removing indirection, enabling domain-specific optimizations, and mapping code to parallel and heterogeneous platforms. Our examples used Scala, but other, similarly expressive modern languages can be used just as well, as demonstrated by Racket macros [112], DSLs Accelerate [73], Feldspar [7], Nikola [72] (Haskell), Copperhead [17] (Python), or Terra [28, 29] (Lua).

What will it take to realize the full potential of generative performance programming on a broad scale? We believe that concerted efforts along three major lines are necessary:

- **Research**: While the mechanics of code generation are well understood, we have only begun to understand what practical generative programming means. What are the programming language features that can be used to good effect in a generative style that targets performance? We have identified type classes as an important feature, others have worked with polytypic programming [100, 101] and metaobject protocols [29]. Specializing interpreters is another well known technique. What are key programming patterns beyond those?
- **Engineering**: For effective practical use, we need toolchains and frameworks that come with “batteries included”, but are at the same time accessible for customization. Building and maintaining such systems takes considerable engineering effort, but pays off in the long run through reuse. A case in point is the LLVM compiler framewok [66].
- **Teaching**: Today, generative programming is somewhat of a black art. We need learning materials, textbooks, and university courses that teach effective meta-programming techniques to drive adoption beyond the perhaps 1% of programmers that are intrinsically inclined towards such techniques.

We believe that this is an exciting opportunity for PL research, with a big potential impact on the whole computing industry. If high-level programming becomes more often the tool of choice for performance critical systems, and less code is written in low-level, unsafe, languages, the overall reliability and safety of software will improve, with transitive benefits to the whole society.

Acknowledgements. The work presented in this paper has profited from numerous interactions with other researchers in the area, for example at Shonan Meetings 2012 and 2014, and annual IFIP WG 2.11 meetings. Many other people have contributed to the development of our tools, including Nithin George, Vojin Jovanovic, Sandro Stucki, Vera Salvisberg, Adriaan Moors, Thierry Coppey, Julien Richard-Foy, Hassan Chafi, Chris De Sa, Christopher R. Aberger, Jithin Thomas, Alexey Romanov, Alexander Slesarenko, Lewis Brown.

This work is supported, among other sources, by ERC grant 587327 DOPPLER, DARPA Contract- Air Force, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army Contract AHPCRC W911NF-07-2-0027-1; NSF Grant, BIGDATA: Mid-Scale: DA: Collaborative Research: Genomes Galore - Core Techniques, Libraries, and Domain Specific Languages for High-Throughput DNA Sequencing, IIS-1247701; NSF Grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Dept. of Energy- Pacific Northwest National Lab (PNNL)- Integrated Compiler and Runtime Autotuning Infrastructure for Power, Energy and Resilience-Subcontract 108845; NSF Grant- EAGER- XPS:DSD:Synthesizing Domain Specific Systems-CCF-1337375; Stanford PPL affiliates program, Pervasive Parallelism Lab.

References

1 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, PPDP'03, pages 8–19, New York, NY, USA, 2003. ACM.

2 Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

3 Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 31–31. IEEE, 2000.

4 Saman P. Amarasinghe. Petabricks: a language and compiler based on autotuning. In Manolis Katevenis, Margaret Martonosi, Christos Kozyrakis, and Olivier Temam, editors, *High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC 2011, Heraklion, Crete, Greece, January 24-26, 2011. Proceedings*, page 3. ACM, 2011.

5 Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. In *1st ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2003), 24-26 June 2003, Mont Saint-Michel, France, Proceedings*, page 249. IEEE Computer Society, 2003.

6 Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA, pages 89–108, New York, NY, USA, 2010. ACM.

7 Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of feldspar: An embedded language for digital signal processing. In *Proceedings of the 22nd international conference on Implementation and application of functional languages*, IFL'10, pages 121–136, Berlin, Heidelberg, 2011. Springer-Verlag.

8 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC'12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225. ACM, 2012.

9 Federico Bassetti, Kei Davis, and Daniel J. Quinlan. C++ expression templates performance issues in scientific computing. In *IPPS/SPDP*, pages 635–639, 1998.

10 Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul H.J. Kelly. Runtime code generation in c++ as a foundation for domain-specific optimisation. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 77–210. Springer Berlin / Heidelberg, 2004.

11 Andrzej Bialecki, Michael Cafarella, Doug Cutting, and Owen O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 11, 2005.

12 Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing (ICS)*, pages 340–347, 1997.

13 Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.

14 Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundam. Inf.*, 69:123–178, July 2005.

15 Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.

16 Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

17 Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 47–56, New York, NY, USA, 2011. ACM.

18 H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. In *Onward!*, 2010.

19 Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

20 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI. ACM, 2010.

21 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

22 James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 403–416. ACM, 2013.

23 Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.

24 Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP*, pages 315–326, 2007.

25 James Crotinger, Scott Haney, Stephen Smith, and Steve Karmesin. PETE: The portable expression template engine. *Dr. Dobb's J.*, October 1999.

26 Kei Davis and Daniel J. Quinlan. Rose: An optimizing transformation system for c++ array-class libraries. In Serge Demeyer and Jan Bosch, editors, *ECOOP Workshops*, volume 1543 of *Lecture Notes in Computer Science*, pages 452–453. Springer, 1998.

27 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, OSDI, pages 137–150, 2004.

28 Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'13, Seattle, WA, USA, June 16-19, 2013*, pages 105–116, 2013.

29 Zachary DeVito, Daniel Ritchie, Matthew Fisher, Alex Aiken, and Pat Hanrahan. First-class runtime generation of high-performance types using exotypes. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 11. ACM, 2014.

30 Torbjörn Ekman and Görel Hedin. The jastadd system - modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

31 Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

32 J. McGraw et. al. SISAL: Streams and iterators in a single assignment language, language reference manual. Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

33 Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: high-level low-level programming. In Antony L. Hosking, David F. Bacon, and Orran Krieger, editors, *Proceedings of the 5th International Conference on Virtual Execution Environments, VEE 2009, Washington, DC, USA, March 11-13, 2009*, pages 81–90. ACM, 2009.

34 Matteo Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.

35 Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

36 Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

37 Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–8. IEEE, 2014.

38 Nithin George, David Novo, Tiark Rompf, Martin Odersky, and Paolo Ienne. Making domain-specific hardware synthesis tools cost-efficient. In *2013 International Conference on Field-Programmable Technology, FPT 2013, Kyoto, Japan, December 9-11, 2013*, pages 120–127. IEEE, 2013.

39 Andy Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12(4):30:30–30:43, April 2014.

40 Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

41 Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. With-loop fusion for data locality and parallelism. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages*, IFL, pages 178–195. Springer Berlin / Heidelberg, 2006.

42 Martin Hanger, Tor Arne Johansen, Geir Kare Mykland, and Aage Skullestad. Dynamic model predictive control allocation using CVXGEN. In *9th IEEE International Conference on Control and Automation, ICCA 2011, Santiago, Chile, December 19-21, 2011*, pages 417–422. IEEE, 2011.

43 High Performance Fortran. `http://hpff.rice.edu/index.htm`.

44 Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In Yannis Smaragdakis and Jeremy G. Siek, editors, *GPCE*, pages 137–148. ACM, 2008.

45 P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

46 Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

47 Intel. Intel array building blocks. `http://software.intel.com/en-us/articles/intel-array-building-blocks`.

48 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys'07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys, pages 59–72, New York, NY, USA, 2007. ACM.

49 Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD'09: Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD, pages 987–994, New York, NY, USA, 2009. ACM.

50 Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008.

51 Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, 2001.

52 Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. Staged parser combinators for efficient data processing. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 637–653. ACM, 2014.

53 Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 86–96. ACM Press, 1986.

54 Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-yang: concealing the deep embedding of dsls. In Ulrik Pagh Schultz and Matthew Flatt, editors, *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, pages 73–82. ACM, 2014.

55 Guy L. Steele Jr. Parallel programming and parallel abstractions in fortress. In *IEEE PACT*, page 157, 2005.

56 Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William Humphrey, John Reynders, Stephen Smith, and Timothy J. Williams. Array design and expression evaluation in pooma ii. In *ISCOPE*, pages 231–238, 1998.

57 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'10, pages 444–463, New York, NY, USA, 2010. ACM.

58 Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, 2005.

59 Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

60 Christoph Koch. Abstraction without regret in data management systems. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

61 Christoph Koch. Abstraction without regret in database systems building: a manifesto. *IEEE Data Eng. Bull.*, 37(1):70–79, 2014.

62 Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. Javascript as an embedded dsl. In *ECOOP*, pages 409–434, 2012.

63 Shriram Krishnamurthi. *Linguistic reuse*. PhD thesis, Computer Science, Rice University, Houston, 2001.

64 Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

65 James R. Larus and Galen C. Hunt. The singularity system. *Commun. ACM*, 53(8):72–79, 2010.

66 C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

67 HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

68 HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, and Kunle Olukotun. Locality-aware mapping of nested parallel patterns on gpus. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Micro, 2014.

69 Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

70 Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. *SIGPLAN Not.*, 37:270–282, January 2002.

71 Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, pages 220–231, 2003.

72 Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell'10, pages 67–78, New York, NY, USA, 2010. ACM.

73 Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP'13, pages 49–60, New York, NY, USA, 2013. ACM.

74 Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? Technical report, Microsoft Research, 2015.

75 José M. F. Moura, Markus Püschel, David Padua, and Jack Dongarra. Scanning the issue: Special issue on program generation, optimization, and platform adaptation. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):211–215, 2005.

76 Fabian Nagel, Gavin M. Bierman, and Stratis D. Viglas. Code generation for efficient query processing in managed runtimes. *PVLDB*, 7(12):1095–1106, 2014.

77 Shayan Najd, Sam Lindley, Josef Svenningsson, and Philip Wadler. Everything old is new again: Quoted domain specific languages. Technical report, University of Edinburgh, 2015.

78 Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

79 Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *CC*, pages 138–152, 2003.

80 Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for GPUs in Scala. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE, pages 107–116, New York, NY, USA, 2011. ACM.

81 Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in scala: towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 125–134, 2013.

82 Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 12–25. ACM, 2011.

83 M. Püschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232 –275, feb. 2005.

84 Markus Püschel, Franz Franchetti, and Yevgen Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

85 Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

86 Daniel J. Quinlan, Markus Schordan, Qing Yi, and Andreas Sæbjørnsen. Classification and utilization of abstractions for optimization. In *ISoLA (Preliminary proceedings)*, pages 2–9, 2004.

87 Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32, 2012.

88 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530, 2013.

89 J.C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. *CMU Technical Report*, 1975.

90 Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel. Efficient high-level abstractions for web programming. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 53–60, 2013.

91 Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming.* PhD thesis, EPFL, 2012.

92 Tiark Rompf and Nada Amin. A SQL to C compiler in 500 lines of code. Technical report, Purdue University, 2015.

93 Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, pages 1–43, 2013.

94 Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE, pages 127–136, New York, NY, USA, 2010. ACM.

95 Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

96 Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs. In *POPL*, 2013.

97 Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented dsls. In Olivier Danvy and Chung-chieh Shan, editors, *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011.*, volume 66 of *EPTCS*, pages 93–117, 2011.

98 Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.

99 Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, number 1505 in Lecture Notes in Computer Science, pages 59–70, 1998.

100 Alexander Slesarenko. Lightweight polytypic staging: a new approach to an implementation of nested data parallelism in scala. In *Scala Workshop*, 2012.

101 Alexander Slesarenko, Alexander Filippov, and Alexey Romanov. First-class isomorphic specialization by staged evaluation. In *Workshop on Generic Programming (WGP)*, 2014.

102 G.L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

103 Alen Stojanov, Georg Ofenbeck, Tiark Rompf, and Markus Püschel. Abstracting vector architectures in library generators: Case study convolution filters. In Laurie J. Hendren, Alex Rubinsteyn, Mary Sheeran, and Jan Vitek, editors, *ARRAY'14: Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, United Kingdom, June 12-13, 2014*, page 14. ACM, 2014.

104 Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *VLDB*, pages 1150–1160. ACM, 2007.

105 Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: generating a high performance DSL implementation from a declarative specification. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 145–154, 2013.

106 Arvind K. Sujeeth, HyoukJoong. Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

107 Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *European Conference on Object Oriented Programming*, ECOOP, 2013.

108 Bo Joel Svensson, Mary Sheeran, and Ryan Newton. Design exploration through code-generating dsls. *Queue*, 12(4):40:40–40:52, April 2014.

109 Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

110 Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *POPL*, pages 389–402, 2010.

111 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In George C. Necula and Philip Wadler, editors, *POPL*, pages 395–406. ACM, 2008.

112 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI'11, pages 132–141, New York, NY, USA, 2011. ACM.

113 D. Vandevoorde and N.M. Josuttis. *C++ templates: the Complete Guide*. Addison-Wesley Professional, 2003.

114 Todd L. Veldhuizen. Expression templates, C++ gems. SIGS Publications, Inc., New York, NY, 1996.

115 Todd L. Veldhuizen. Arrays in blitz++. In *ISCOPE*, pages 223–230, 1998.

116 Todd L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

117 Todd L. Veldhuizen and Jeremy G. Siek. Combining optimizations, combining theories. Technical report, Indiana University, 2008.

118 Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

119 Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

120 R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

121 Michael E Wolf and Monica S Lam. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):452–471, 1991.

122 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI, 2011.

123 Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. Monetdb/x100 - a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

Hardware–Software Co-Design: Not Just a Cliché

Adrian Sampson, James Bornholt, and Luis Ceze

University of Washington, US
`{asampson,bornholt,luisceze}@cs.washington.edu`

Abstract

The age of the air-tight hardware abstraction is over. As the computing ecosystem moves beyond the predictable yearly advances of Moore's Law, appeals to familiarity and backwards compatibility will become less convincing: fundamental shifts in abstraction and design will look more enticing. It is time to embrace *hardware–software co-design* in earnest, to cooperate between programming languages and architecture to upend legacy constraints on computing. We describe our work on *approximate computing*, a new avenue spanning the system stack from applications and languages to microarchitectures. We reflect on the challenges and successes of approximation research and, with these lessons in mind, distill opportunities for future hardware–software co-design efforts.

1998 ACM Subject Classification C.5 Computer System Implementation

Keywords and phrases approximation, co-design, architecture, verification

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.262

1 Introduction

Generations of computer scientists and practitioners have worked under the assumption that computers will keep improving themselves: just wait a few years and Moore's Law will solve your scaling problems. This reliable march of electrical-engineering progress has sparked revolutions in the ways humans use computers and interact with the world and each other. But growth in computing power has protected outdated abstractions and encouraged layering even more abstractions, whatever the cost.

The free lunch seems to be over: single-thread performance has stagnated, Dennard scaling has broken down, and Moore's Law threatens to do the same. The shift to multi-core designs worked as a stopgap in the final years of frequency advancements, but physical limits have dashed hopes of long-term exponential gains through parallelism.

Hardware–software co-design presents significant performance and efficiency opportunities that are unavailable without crossing the abstraction gap. For example, embedded systems depended on co-design from their inception. The embedded domain uses software for flexibility while specialized hardware delivers performance. General-purpose computing features many hardware extensions employed to better serve software – virtual memory, ISA extensions for security, and so on. While these mechanisms have been successful, they are ad hoc responses to trends in software design.

Hybrid hardware–software research cannot just be a cliché: now more than ever, true cooperation is crucial to improving the performance and efficiency of future computing systems. Over the past five years, our research group has been exploring *approximate computing*, a classic example of a hardware–software co-design problem. Approximate computing trades accuracy for performance or efficiency, exploiting the fact that many applications are robust to some level of imprecision. Our experience has shown that neither software nor hardware alone can unlock the full potential of approximation; optimal solutions require co-design between programming models and hardware accelerators or ISA extensions. We discuss our experience with approximate computing and highlight lessons for future co-design efforts.

We believe we can go much further by truly designing hardware and software in unison. We survey the opportunities for hardware software co-design in four broad categories. The first is effective hardware acceleration, where the best algorithm for general-purpose hardware and the best custom hardware for generic algorithms both fall short. For example, the Anton [12] special machine for molecular dynamics, co-designed a simulation algorithm with the hardware. The second category is reducing redundancy and reassigning responsibilities. For example, if all programs were guaranteed to be free of memory bugs, can we drop support for memory protection? Or if programs were free of data-races by construction, can we rethink cache coherence? The third category is hardware support for domain-specific languages, whose booming popularity democratizes programming by providing higher-level semantics and tooling, but today must still compile to low-level general-purpose hardware. The final category is research beyond the CPU: unexplored opportunities abound for hardware–software cooperation in networks, memory, storage, and less glamorous parts of computer hardware such as power supplies.

2 Approximate Computing

For the last five years, our group has explored one instance of a hardware–software co-design opportunity: *approximate computing*. The idea is that current abstractions in computer systems fail to incorporate an important dimension of the application design space: not every application needs the same degree of accuracy all the time. These applications span a wide range of domains including big-data analytics, web search, machine learning, cyber-physical systems, speech and pattern recognition, augmented reality, and many more. These kinds of programs can tolerate unreliable and inaccurate computation, and approximate computing research shows how to exploit this tolerance for gains in performance and efficiency [1, 7, 10, 11, 25, 26, 28, 30, 35].

Approximate computing is a classic cross-cutting concern: its full potential is not reachable through software or hardware alone, but only through changing the abstractions and contracts between hardware and software. Advances in approximation require co-design between architectures that expose accuracy–efficiency trade-offs and the programming systems that make those trade-offs useful for programmers. We have explored projects across the entire system stack – from programming languages and tools down through the hardware – that enable computer systems to trade off accuracy of computation, communication, and storage for gains in efficiency and performance.

Our research direction spans languages [38], runtime systems, programmer tools including debuggers [36], compilers [41], synthesis tools for both software and hardware, microarchitecture [13], hardware techniques [43, 29], data stores [40], and communication services.

Safety and Quality-of-Results

A core requirement in writing programs with approximate components is *safety*: approximate components must not compromise safe execution (e.g., no uncontrolled jumps or critical data corruption) and must interact with precise components only in well-defined ways allowed by the programmer. Our work met this need with language support in the form of type qualifiers for approximate data and type-based static information-flow tracking [38]. Other work from MIT consists of a proof system for deriving safety guarantees in the face of unreliable components [5]. These crucial safety guarantees allow systems to prove at compile time that approximation cannot introduce catastrophic failures into otherwise-good programs.

Beyond safety, another key requirement is ways to specify and ensure acceptable quality-of-results (QoR). Languages must enable programmers to declare the magnitude of acceptable approximation in multiple forms and granularities. For example, QoR can be set for a specific value (X should be at most Y% from its value in a fully precise execution), or one could attach QoR to a set of values (at most N values in a set can be in error). One can provide a QoR specification only for the final output of a program or for intermediate values. QoR specifications can then guide the compiler and runtime to choose and control the optimal approximate execution engine from a variety of software and hardware approximation mechanisms. While quality constraints are more general and therefore more difficult to enforce statically than safety requirements, initial tactics have seen success by limiting the kinds of approximation they can work with [41, 6] or by relying on dynamic checks [36, 17].

Approximation Techniques

The purpose of allowing approximation is to trade accuracy for energy savings. At the highest level, there are three categories of approximation techniques: algorithmic, compiler/runtime, and hardware. Algorithmic approximation can be achieved by the programmer providing multiple implementations of a given computation and the compiler/runtime choosing among them based on QoR needs. A compiler can generate code for approximate execution by eliding some computations [28] or reducing value precision whenever allowed by the approximation specification. Approximation research has explored several approximate execution techniques with hardware support, among them: compiler-controlled voltage overscaling [13]; using learning-based techniques such as neural networks or synthesis to approximate kernels of imperative programs in a coarse-grain way [14]; adopting analog components for computation [43]; designing efficient storage systems that can lose bits [40]; and extending architecture mechanisms with imprecise modes [27].

While approximation only at the algorithm level together with compiler/runtime support applies to off-the-shelf hardware (and we intend to further explore that space too), our experience has shown that the greatest energy benefit comes from hardware-supported approximation with language/architecture co-design [29].

Tools

A final key component for making approximate programming practical is software-development tools. We need tools to help programmers identify approximation opportunities, understand the effect of approximation at the application level, assist with specifying QoR requirements, and help test and debug applications with approximate components. Our first steps in this direction are a debugger and a post-deployment monitoring framework for approximate programs [36].

2.1 Next Steps in Approximation

Controlling Quality

The community has allocated more attention to assuring safety of approximate programs than to controlling quality. Decoupling safety from quality has been crucial to enabling progress on that half of the equation [38, 5] but more nuanced quality properties have proven more challenging. We have initial ways to prove and reason about limited probabilistic quality properties [6, 4, 41], but we still lack techniques that can cope with arbitrary approximation strategies and still produce useful guarantees.

We also need ways to measure quality at run time. If approximate programs could measure how accurate they are without too much overhead, they could offer better guarantees to programmers while simultaneously exploiting more aggressive optimizations [17, 36]. But there is not yet a general way to derive a cheap, dynamic quality check for an arbitrary program and arbitrary quality criterion. Even limited solutions to the dynamic-check problem will amplify the benefits of approximation.

Defining Quality

Any application of approximate computing rests on a *quality metric*. Even evaluations for papers on approximation need to measure their effectiveness with some accuracy criterion. Unlike traditional criteria – energy or performance, for example – the right metric for quality is not obvious. It varies per program, per deployment, and even per user. The community does not have a satisfactory way to decide on the right metric for a given scenario: we are so far stuck with guesses.

A next step in approximation research should help build confidence that we are using the right quality metrics. We should adopt techniques from software engineering, human-computer interaction, and application domains like graphics to help gather evidence for good quality metrics. Ultimately, programmers need a sound methodology for designing and evaluating quality metrics for new scenarios.

The Right Accelerator

Hardware approximation research has fallen into two categories: extensions to traditional architectures [13, 27] and new, discrete accelerators [47, 14]. The former category has yielded simpler programming models, but the fine-grained nature of the model means that efficiency gains have been limited. Coarser-grained, accelerator-oriented approaches have yielded the best results to date. There are still opportunities for co-designing accelerators with programming models that capture the best of both approaches. The next generation of approximate hardware research should co-design an accelerator design with a software interface and compiler workflow that together attack the programmability challenges in approximation: safety and quality. By decoupling approximation from traditional processors, new accelerators could unlock new levels of efficiency while finally making approximate computing palatable hardware vendors.

2.2 Lessons from Approximation

Our group's experience with approximate computing as a cross-cutting concern has had both successes and failures. The path through this research has yielded lessons both for approximation research specifically and hardware–software co-design generally.

The von Neumann Curse

When doing approximation at the instruction granularity in a typical von Neumann machine, the data path can be approximate but the control circuitry likely can't. Given that control accounts for about 50% of hardware resources, gains are fundamentally limited to 2×, which is hardly enough to justify the trouble. We therefore have more hopes for coarse-grain approximation techniques than fine-grain.

And that was just an example. Many other promising avenues of our work have fallen afoul of the conflation of program control flow and data flow. For example, if we want to approximate the contents of a register, we need to know whether it represents pixel data amenable to approximation or a pointer, which may be disastrous to approximate.

Separation problems are not unique to approximation. Secure systems, for example, could profit from a guarantee that code pointers are never manipulated as raw data. Future explorations of hardware–software co-design would benefit from architectural support for separating control flow from data flow.

The High Cost of Hardware Design

In our work on approximation with neural networks, we achieve the best energy efficiency when we use a custom hardware neural processing unit, or NPU [14, 43]. Our first evaluations of the work used cycle-based architectural simulation, which predicted an average 3× energy savings. Later, we implemented the NPU on an FPGA [29]. On this hardware, we measured an average energy savings of 1.7×. The difference is due partially to the FPGA's overhead and clock speed and partially due to the disconnect between simulation and reality. Determining the exact balance would require a costly ASIC design process. Even building the FPGA implementation took almost two years.

Hardware–software co-design involves an implied imbalance: software is much faster to iterate on than hardware. Future explorations of hardware–software co-design opportunities would benefit from more evolution of hardware description languages and simulation tools. We should not have to implement hardware three times – first in simulation, second in an HDL for an efficient FPGA implementation, and again for a high-performance ASIC.

Trust the Compiler

Hybrid hardware–software research constantly needs to divide work between the programmer, the compiler, and the hardware. In our experience, a hybrid design should delegate as much as possible to the compiler. For example, the Truffle CPU [13] has dual-voltage SRAM arrays that require every load to match the precision level of its corresponding store. This invariant would be expensive to enforce with per-register and per-cache-line metadata, and it would be unreasonable for programmers to specify manually. Our final design leaves all the responsibility to the compiler, where enforcement is trivial.

Relegating intelligence to the compiler comes at a cost in safety: programming to the Truffle ISA directly is dangerous and error-prone. Fortunately, modern programmers rarely write directly to the CPU's hardware interface. Researchers should treat the ISA as a serialization channel between the compiler and architecture – not a human interface. Eschewing direct human–hardware interaction can pay off in fundamental efficiency gains.

3 Opportunities for Co-Design

3.1 Programming Hardware

The vast majority of programming languages research is on languages that target a very traditional notation of "programming." Programs must eventually be emitted as a sequence of instructions, meant to be interpreted by processors that will load them from memory and execute them in order. The programming languages and architecture communities should not remain satisfied with this traditional division of labor. The instruction set architecture abstraction imposes limits on the control that programmers can have over how hardware works, and compatibility concerns limit the creativity of architecture designs. Hybrid hardware–software research projects should design new hardware abstractions that violate the constraints of traditional ISAs.

Some recent work revived interest in languages for designing hardware and FPGA configurations [3, 31] or applied language techniques to special-purpose hardware like networking equipment [15] and embedded processor arrays [33]. But we can do more. A new story for programmable hardware will require radically new architecture designs, hardware interfaces that expose the right costs and capabilities, programming models that can abstract these interfaces' complexity, and careful consideration of the application domains.

Rethinking Architectures from the Ground Up

The central challenge in designing programmable hardware is finding the right architectural trade-off between reconfigurability and efficiency. Co-design is only possible if we expose more than what current CPUs do, but providing too much flexibility can limit efficiency. Current field-programmable gate arrays (FPGAs) are a prime example that misses the mark today: FPGAs strive for bit-level reconfigurability, and they pay for it in both performance and untenable tool complexity. New architectures will need to carefully choose the granularity of reconfiguration to balance these opposing constraints.

Exposing Communication Costs

The von Neumann abstraction is computation-centric: the fundamental unit of work is computation. But the costs in modern computers, especially in terms of energy, are increasingly consumed by *communication* more than computation. New programmable hardware will need to expose abstractions that reflect this inversion in the cost model. Unlike earlier attempts to design processors with exposed communication costs [16], these new abstractions need not be directly comprehensible for humans: we do not need to expect programmers to write this "assembly language" directly. Instead, we should design abstractions with their software toolchains in mind from the beginning.

Managing Complexity

New programmable hardware will need to work in concert with programming languages and compiler workflows that can manage their inherent complexity. Crucially, the toolchain will need to balance convenient automation with programmer control. It can be extremely powerful to unleash programmers on the problem of extracting efficiency from hardware, but history has also shown that overly complex programming models do not go mainstream. New toolchains should combine both language ideas, which can safely expose complexity, compiler techniques like program synthesis, which can hide the complexity.

General vs. Domain-Specific Architectures

There is a well-known trade-off in architecture between generality and efficiency [18]. The best reconfigurable hardware may never be fully general purpose: domain-specific languages, tools, and chips can have critical advantages for their sets of applications. At every level in the stack, new designs should provide tools for building domain-specific components.

3.2 Simpler Hardware With Software Verification

Modern computer architectures spend considerable resources protecting software from itself, with kernel/user mode, process isolation through virtual memory, sophisticated cache coherency and memory consistency models to extract maximum performance, and others. These protections add complexity to hardware design and implementation, and cost both time and energy during execution.

Rapid improvement in software verification has greatly expanded both the scope and the strength of the safety properties we can (automatically or manually) prove about programs. In particular, a system that is verified from the microkernel up to the application appears within reach [24]. The power of verification presents an opportunity to design new hardware architectures specifically for verified software. These architectures would avoid the energy and time overhead of dynamic protection mechanisms and simplify future scaling efforts.

Virtual Memory

Major architectures provide virtual memory to help operating systems provide process isolation. Each process sees its own address space and has no way of accessing the physical memory allocated to other processes. While this abstraction is convenient for operating system and application programmers, virtual memory protection entails considerable hardware complexity and cost.

Aiken et al. showed in 2006 that the cycle overhead of hardware virtual memory protection is up to 38% on the SPECweb99 benchmark [2]. Half this cost (19 pp) is due to switching between multiple address spaces (kernel and application) on control transfers, and the pressure these address spaces place on the TLB. An additional 12 pp is due to switching protection modes between ring 0 (kernel) and ring 3 (application) on control transfers. Working sets are growing much faster than the size of TLBs (combining L1 and L2, TLB entries have grown from 560 pages or 2,240 kB on Nehalem (2008) to 1,088 pages or 4,352 kB on Haswell (2014), and so TLB pressure and virtual memory overhead will only continue to increase.

The Singularity project explored the opportunity to abandon virtual memory in favor of *software-isolated processes (SIPs)* [22]. Instead of asking virtual memory hardware to provide process isolation, Singularity guarantees isolation between SIPs through static verification and some runtime checks. The verification is made feasible by Singularity's Sing# language for applications, which is type- and memory-safe. The software runtime checks are very low overhead; for example, a ping-pong message between two processes on Singularity is 7× faster than Linux and 8× faster than Windows [21].

Concurrency

A substantial portion of multiprocessor hardware is dedicated to cache coherence and memory consistency. These mechanisms work with the programming model, compilers and operating systems to preserve a simple memory model for programmers. But coherence protocols and consistency enforcement are neither simple nor cheap.

Memory models are complex to understand, and even more complex to implement correctly. Verification efforts have uncovered bugs in both hardware implementations of relaxed consistency models [19] and in the software memory models that attempt to abstract over them [46]. Given these difficulties, some have advocated for abandoning shared-memory hardware altogether (for example, Intel's Single-Chip Cloud Computer (SCC) has 48 cores without coherence), but this seems extreme.

Coherence and consistency come with significant performance and hardware overheads. DeNovo is a coherence protocol that works together with a disciplined shared-memory programming model to reduce coherence overheads [8]. Compared to MESI, DeNovo executes benchmarks up to 60% faster due to reduced network traffic, read misses and stall times.

Verification of data race freedom would enable simpler hardware designs and therefore increase the potential for future scaling. Traditional work in verifying race freedom (including the DeNovo programming model) has required explicit program annotations [23]. But recent work in formalizing hardware memory models [42] suggests that we should work towards *automatically* verifying race freedom. A hardware architecture for verified software would take advantage of provable race freedom to reduce coherence traffic and state overhead, and so save energy and time.

3.3 Hardware-backed DSLs

Today's programming languages are too complex for many of the tasks today's programmers want to complete. We ask programmers to reason about subtle issues like memory management, concurrency and race conditions, and large standard libraries. Domain-specific languages (DSLs) are a way to democratize programming by providing languages with higher-level semantics and simpler tooling, and have seen considerable success [45]. But traditional DSLs are often implemented as slow, interpreted languages, reducing their usefulness for many programmers.

Delite is a compiler framework for implementing high-performance DSLs [44]. The framework provides components common to high-performance implementations, such as parallel versions of higher-order functions, compiler optimizations, and the ability for DSL implementers to provide domain-specific optimizations for the code generator. Delite compiles DSLs to an intermediate representation which can then be compiled to multiple targets such as C++ and OpenCL.

While Delite makes for high-performance DSLs, they ultimately execute on general-purpose hardware. There is the opportunity to build hardware accelerators specialized to particular DSLs, or even to individual programs written in a given DSL. For example, the Halide DSL for image processing compiles to SIMD or CPU+GPU code and sees considerable performance gains [34]. But Darkroom, another DSL targeting the same domain, compiles directly to pipelined ASIC implementations for even higher performance [20].

Previous work on compiling DSLs to hardware has required custom design and implementation work for each DSL compiler. Future work should target a more general compiler from DSL implementation to hardware. We should take motivation from recent work on lightweight modular staging [37], which builds high-performance compilers by staging interpreter code. Existing high-level synthesis (HLS) tools for synthesizing circuits from higher-level programming languages are difficult to use and even more difficult to debug, because they are required to implement the rich and unsafe semantics of C. We should aim to build DSL frameworks with hardware synthesis in mind to unlock the potential of ASIC performance.

3.4 Beyond the CPU

Researchers is both architecture and programming languages tend to focus on one component above all others: the CPU. While the heart of a computer has special importance, it is not the only piece that deserves attention. When the goal is energy efficiency, for example, it is crucial to bear in mind that most of a smartphone's battery capacity goes to powering the screen and radio – the CPU is a comparatively small drain [39]. And in the datacenter, network and storage performance can outweigh the importance of raw compute power.

Hardware–software research is in a good position to seek out opportunities beyond the CPU: in memory, storage, networking, and even more mundane systems such as displays, sensors, and cooling infrastructure.

Non-volatile main memory is an example of an emerging architectural trend in desperate need of attention from a programming-model perspective. Initial work has shown that traditional abstractions are insufficient [32] and has proposed new approaches for managing non-volatile state [9, 48].

Language research should also seek out opportunities in networking. Recent software-defined networking work [15] has shown how to build better abstraction for networking equipment, but that work focuses on traditional roles for the network. We should also consider new ways to program the network. For instance, in the data center, what computation would be best offloaded from servers into the network? How can we productively program mobile device radio modems while preventing disastrous network attacks?

More experimental research should consider collaboration with the less glamorous parts of computer hardware. Power supply systems are one such example that are typically hidden at all costs from the end programmer. Researchers should question this separation. For example, can datacenter applications extract more energy efficiency by programming their power distribution systems? How can we re-design the battery and charging systems in a mobile phone to specialize them to a particular use patterns? Can applications and operating systems collaborate with hardware to fairly divide energy resources? Peripheral, traditionally non-architectural components like power systems are ripe for rethinking, but only in the context of co-design with a new programming model.

4 Conclusion

The air-tight hardware abstraction continues to serve innovation in programming languages well. But as physical challenges begin to threaten Moore's law, the biggest leaps in performance and efficiency will come from true hardware–software co-design. We believe our community can go far beyond the current primitive cooperation between hardware and software by designing them *together*. We can take inspiration from successes in embedded system design and our experiences with approximate computing. Exciting opportunities abound for hardware accelerators co-designed with software, reduced redundancy between hardware and software mechanisms, and entirely new classes of optimizations. We hope that our experiences and successes with hardware–software co-design will encourage researchers to cross the abstraction boundary to design more integrated and thoughtful systems.

References

1 Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, MIT, 2009.

2 Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Workshop on Memory System Performance and Correctness (MSPC)*, 2006.

3 J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a Scala embedded language. In *DAC*, June 2012.

4 James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: A First-Order Type for Uncertain Data. In *ASPLOS*, 2014.

5 Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.

6 Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.

7 Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE*, 2006.

8 Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarite V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

9 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

10 M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *SELSE*, 2009.

11 Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

12 Martin M. Deneroff, David E. Shaw, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, and Cliff Young. Anton: A specialized ASIC for molecular dynamics. In *Hot Chips*, 2008.

13 Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

14 Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *International Symposium on Microarchitecture (MICRO)*, 12 2012.

15 N. Foster, A. Guha, M. Reitblatt, A. Story, M.J. Freedman, N.P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for software-defined networks. *Communications Magazine, IEEE*, 51(2):128–134, February 2013.

16 Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS*, 2002.

17 Beayna Grigorian and Glenn Reinman. Improving coverage and reliability in approximate computing using application-specific, light-weight checks. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

18 Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

19 Sudheendra Hangal, Durgam Vhia, Chaiyasit Manovit, Jiun-Weu Joseph Lu, and Sridhar Narayanan. TSOtool: A program for verifying memory systems using the memory consistency model. In *31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

20 James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4), 2014.

21 Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawbliztel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. In *EuroSys*, 2007.

22 Galen Hunt and James Larus. Singularity: Rethinking the software stack. In *21st ACM Symposium on Operating System Principles (SOSP)*, 2007.

23 Rajesh K. Karmani, P. Madhusudan, and Brandon M. Moore. Thread contracts for safe parallelism. In *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

24 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

25 Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. ERSA: error resilient system architecture for probabilistic applications. In *DATE*, 2010.

26 Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*, 2011.

27 Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load value approximation. In *MICRO*, 2014.

28 Sasa Misailovic, Stelios Sidiroglou, Hank Hoffman, and Martin Rinard. Quality of service profiling. In *ICSE*, 2010.

29 Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *HPCA*, 2015.

30 Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. Scalable stochastic processors. In *DATE*, 2010.

31 Rishiyur S. Nikhil and Arvind. What is bluespec? *SIGDA Newsl.*, 39(1):1–1, January 2009.

32 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ISCA*, 2014.

33 Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

34 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

35 Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Onward!*, 2010.

36 Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and debugging the quality of results in approximate programs. In *ASPLOS*, 2015.

37 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, 2012.

38 A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

39 Adrian Sampson, Calin Cascaval, Luis Ceze, Pablo Montesinos, and Dario Suarez Gracia. Automatic discovery of performance and energy pitfalls in html and css. In *IISWC*, 2012.

40 Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *MICRO*, 2013.

41 Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn McKinley, Dan Grossman, and Luis Ceze. Expressing and Verifying Probabilistic Assertions. In *PLDI*, 2014.

42 Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009.

43 Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. In *ISCA*, 2014.

44 Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s), 2014.

45 Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Onward!*, 2011.

46 Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: Checking axiomatic specifications of memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

47 Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *MICRO*, 2013.

48 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

Refined Criteria for Gradual Typing*

Jeremy G. Siek[1], Michael M. Vitousek[1], Matteo Cimini[1], and John Tang Boyland[2]

1 **Indiana University – Bloomington, School of Informatics and Computing**
150 S. Woodlawn Ave. Bloomington, IN 47405, USA
jsiek@indiana.edu

2 **University of Wisconsin – Milwaukee, Department of EECS**
PO Box 784, Milwaukee WI 53201, USA
boyland@cs.uwm.edu

Abstract

Siek and Taha [2006] coined the term *gradual typing* to describe a theory for integrating static and dynamic typing within a single language that 1) puts the programmer in control of which regions of code are statically or dynamically typed and 2) enables the gradual evolution of code between the two typing disciplines. Since 2006, the term *gradual typing* has become quite popular but its meaning has become diluted to encompass anything related to the integration of static and dynamic typing. This dilution is partly the fault of the original paper, which provided an incomplete formal characterization of what it means to be gradually typed. In this paper we draw a crisp line in the sand that includes a new formal property, named the *gradual guarantee*, that relates the behavior of programs that differ only with respect to their type annotations. We argue that the gradual guarantee provides important guidance for designers of gradually typed languages. We survey the gradual typing literature, critiquing designs in light of the gradual guarantee. We also report on a mechanized proof that the gradual guarantee holds for the Gradually Typed Lambda Calculus.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs – Type structure

Keywords and phrases gradual typing, type systems, semantics, dynamic languages

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.274

1 Introduction

Statically and dynamically typed languages have complementary strengths. Static typing guarantees the absence of type errors, facilitates the generation of efficient code, and provides machine-checked documentation. On the other hand, dynamic typing enables rapid prototyping, flexible programming idioms, and fast adaptation to changing requirements. The theory of *gradual typing* provides both of these typing disciplines within a single language, puts the programmer in control of which discipline is used for each region of code, provides seamless interoperability, and enables the convenient evolution of code between the two disciplines. Gradual typing touches both the static type system and the dynamic semantics of a language. The key innovation in the static type system is the *consistency* relation on types, which allows implicit casts to and from the unknown type, here written $\star$, while still catching static type errors [5, 50, 27].[1] The dynamic semantics for gradual typing is based on the semantics

* This work was partially supported by NSF grant 1360694.
[1] The consistency relation is also known as compatibility.

of contracts [19, 24], coercions [33], and interlanguage migration [62, 40]. Because of the shared mechanisms with these other lines of research, much of the ongoing research benefits the theory of gradual typing, and vice versa [28, 39, 26, 16, 56, 15, 17, 25].

Gradual typing has a syntactic similarity to type inference [42], which also supports optional type annotations. However, type inference differs from gradual typing in that it requires static type checking for the whole program. However, there are many other lines of research that seek to integrate static and dynamic typing, listed below.

1. *Dynamic & Typecase* extends statically typed languages with a type, often named Dynamic or Any, together with explicit forms for injecting and projecting values into and out of the Dynamic type [38, 1, 12].
2. *Object-Oriented Languages* extend statically typed languages with a type, often called Object, with implicit conversions into Object and explicit conversions out of Object [23, 37].
3. *Soft Typing* applies static analysis to dynamically typed programs for the purposes of optimization [13] and debugging [21].
4. *Type Hints* in dynamically typed languages enable type specialization in optimizing compilers, such as in Lisp [55] and Dylan [47].
5. *Types for Dynamic Languages* design static type systems for dynamically typed languages, such as StrongTalk [10] and Typed Racket [63].

While these lines of research share the broad goal of integrating statically and dynamically typed languages, they address goals different from those associated with gradual typing.

To a large degree, the practice of gradual typing preceded the theory. A number of languages provided a combination of static and dynamic type checking, with implicit casts to and from the unknown type. These languages included Cecil [14], Visual Basic.NET [41], Bigloo [46], and ProfessorJ [24]. The theory of gradual typing provides a foundation for these languages. Thatte's earlier theory of Quasi-static Typing [60] almost meets the goals of gradual typing, but it does not statically catch all type errors in completely annotated programs. (See Siek and Taha [50] for an in-depth discussion.)

Over the last decade there has been significant interest, both in academia and industry, in the integration of static and dynamic typing. On the industry side, there is Dart [59], TypeScript [32, 6], Hack [65], and the addition of Dynamic to C# [31]. On the academic side, there is a growing body of research [34, 51, 30, 40, 49, 67, 8, 54, 69, 68, 2, 18, 36, 44, 58, 3, 66, 61, 57, 45, 4, 9, 6, 22, 53]. The term *gradual typing* is often used to describe language designs that integrate static and dynamic typing. However, some of these designs do not satisfy the original intent of gradual typing because they do not support the convenient evolution of code between the two typing disciplines. This goal was implicit in the original paper; it did not include a formal property that captures convenient evolution. To this end, we offer a new criterion in this paper, the *gradual guarantee*, that relates the behavior of programs that differ only with respect to the precision of their type annotations.

In Section 2 we discuss several example programs that demonstrate gradual typing and motivate the need for the gradual guarantee. We review the semantics of the Gradually Typed Lambda Calculus (GTLC) (Section 3) and then state the formal criteria for gradually typed languages, including the gradual guarantee (Section 4). In Section 5 we survey some of the gradual typing literature, critiquing designs in light of the gradual guarantee. The last section before the conclusion reports on a mechanized proof that the GTLC satisfies the gradual guarantee (Section 6).

2 Examples of Gradual Typing

In this section, we highlight the goals of gradual typing by way of several examples and motivate the need for the gradual guarantee. The examples are written in Reticulated Python, a gradually typed variant of Python [66] using the syntax for type annotations specified in PEP 484 [29]. For example, a function type $T_1 \to T_2$ is written `Callable[[`T_1`],`T_2`]`.

2.1 Gradual Typing Includes Both Fully Static and Fully Dynamic

The first goal of gradual typing is to provide both fully static type checking and fully dynamic type checking. In other words, a gradually typed language can be thought of being a superset of two other languages, a fully static one and a fully dynamic one. For example, the GTLC is, roughly speaking, a superset of both the Simply Typed Lambda Calculus (STLC) and the (Dynamically Typed) Lambda Calculus (DTLC). We say that a program is *fully annotated* if all variables have type annotations and if the type $\star$ does not occur in any of the type annotations. A fully annotated program of the GTLC should behave the same as in the STLC, and a program without type annotations should behave the same as in the DTLC. An important aspect of a program's behavior that we take into account is the error cases, of which there are several varieties: 1) a program may fail to type check, 2) a program may encounter a runtime error and the language definition requires that the program halt or raise an exception, i.e., a *trapped error* [11]) and 3) a program may encounter a runtime error that the language definition says nothing about, i.e., an *untrapped error*. The STLC, GTLC, and even DTLC are all strongly typed so they are free of untrapped errors. Furthermore, the STLC is free of trapped errors and so is the GTLC on fully annotated programs.

Consider the examples in Figure 1. The $\textsc{gcd}_{1a}$ and $\textsc{gcd}_{1b}$ functions at the top have no type annotations whereas $\textsc{gcd}_{3a}$ and $\textsc{gcd}_{3b}$ at the bottom are fully annotated. The un-annotated versions should behave just like they would in Python. Indeed, with $\textsc{gcd}_{1a}$, the call `gcd(15, 9)` returns `(3,-1,2)`. Can you spot the error in $\textsc{gcd}_{1b}$? The second return statement is returning a pair instead of a 3-tuple. But that error is not caught statically because the programmer has asked for dynamic checking. Turning to the fully annotated versions $\textsc{gcd}_{3a}$ and $\textsc{gcd}_{3b}$, they should behave just as they would in some hypothetical statically typed variant of Python. Indeed, with $\textsc{gcd}_{3a}$, the call `gcd(15, 9)` returns `(3,-1,2)` and furthermore, the gradual type system guarantees that $\textsc{gcd}_{3a}$ is free of runtime type errors. On the other hand, with $\textsc{gcd}_{3b}$, a static error is reported to indicate that returning a pair conflicts with the return type `Tuple[int,int,int]`.

2.2 Gradual Typing Provides Sound Interoperability

With gradual typing, fully static programs behave the same as if they were written in a statically typed programming language. As a result, they are guaranteed not to encounter type errors at runtime. But what about partially typed programs?

Consider the following algorithm for computing the modular inverse. We have not annotated the parameters of `modinv`, so it is dynamically typed, but suppose it calls the statically typed $\textsc{gcd}_{3b}$. What happens if someone forgets a conversion and passes a string as parameter `m` of `modinv`?

```
def modinv(a, m):
    (g, x, y) = gcd(a, m)
    if g != 1: raise Exception()
    else: return x % m
```

$\boxed{\text{GCD}_{1a}}$

```
def gcd(a, b):
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y, y)
```

$\boxed{\text{GCD}_{1b}}$

```
def gcd(a, b):
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y)
```

$\boxed{\text{GCD}_{3a}}$

```
def gcd(a:int, b:int)
      -> Tuple[int,int,int]:
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y, y)
```

$\boxed{\text{GCD}_{3b}}$

```
def gcd(a:int, b:int)
      -> Tuple[int,int,int]:
  if a == 0:
    return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y)
```

Figure 1 Static and dynamic variants of the extended greatest-common divisor algorithm.

Does the string flow into the `gcd` function and trigger a runtime error in the expression `b % a`? That would be unfortunate, because `gcd` is statically typed and one would hope that `gcd` is guaranteed to be free of runtime errors, of both the trapped and untrapped variety. Further, there would be a string masquerading as an integer and programmers would not be able to trust their type annotations. With gradual typing, the runtime system protects the static typing assumptions by casting values as they flow between statically and dynamically typed code. So in this example, there is a cast error in `modinv` just before the call to `gcd`. In fact, gradual typing ensures that statically typed regions of code are free of runtime type errors.

Next let us consider the following fully annotated version of `modinv` with a call to the dynamically typed GCD_{1a}. Because this function refers to a variable (`gcd`) that is dynamic, this function is only partially typed.

```
def modinv(a : int, m : int):
    (g, x, y) = gcd(a, m)
    if g != 1: raise Exception()
    else: return x % m
```

However, one would like to understand which parts of `modinv` are safe versus which parts might result in a runtime type error. We accomplish this by analyzing the implicit casts in `modinv`. In the call `gcd(a,m)`, the arguments are cast from `int` to `Any`. In general, gradual typing guarantees that upcasts like these are safe. (We define upcast in terms of subtyping in Section 4.2.) On the other hand, the return type of GCD_{1a} is unspecified, so it defaults to `Any`. Thus, the assignment to the tuple `(g, x, y)` requires a downcast, which is unsafe. One thing worth noting is that some partially typed code can be completely safe. For example, if we changed GCD_{1a} to have return type `Tuple[int,int,int]` (but leave the parameter types unspecified), then the `modinv` function would be safe because the only implicit casts would be the upcasts on the arguments in the call to `gcd`. We envision that IDE's for gradually typed languages will provide feedback to programmers, identifying the locations of unsafe casts.

Next we consider the situation in which a dynamically typed function is used as a callback inside a statically typed function. In the following we compute the derivative of a function `fun` at two different points. However, there is an error during the second call to `deriv` because `fun` returns the `None` value when the input is not positive.

```
def deriv(d: float, f: Callable[[float],float], x: float) -> float:
  return (f(x + d) - f(x - d)) / (2.0 * d)

def fun(y):
  if y > 0: return y ** 3 - y - 1

deriv(0.01, fun, 3.0)
deriv(0.01, fun, -3.0)
```

As described above, gradual typing performs runtime casts to ensure that values are consistent with their static types. Here the cast needs to check that `fun` has type `Callable[[float], float]`. However, determining the return type of an arbitrary dynamically typed function is undecidable. (The halting problem reduces to this problem.) Instead, gradual typing draws on research for contracts [19] and delays the checking until the function is called. Thus, a cast error occurs inside `deriv` when parameter `f` is applied to a negative number.

If this were the end of the story, it would be unfortunate; as we stated above, statically typed code should be free of runtime type errors. In this example, the code that called `deriv` is at fault but the error occurs inside `deriv`. However, thanks to the blame tracking technique of Findler and Felleisen [19], the cast error can point back to the call to `deriv` and explain that `fun` violated the expected type of `Callable[[float],float]` returning `None`. Thus, in general, the soundness guarantee for gradual typing is stated in terms of blame: upcasts never result in blame, only downcasts or cross-casts.

2.3 Gradual Typing Enables Gradual Evolution

So far we demonstrated how gradual typing subsumes static and dynamic typing and provides sound interoperability between the two, but we have not demonstrated the *gradual* part of gradual typing. That is, programmers should be able to add or remove type annotations without any unexpected impacts on their program, such as whether it still typechecks and whether its runtime behavior remains the same. In Figure 2 we show several points in the evolution of the `gcd` function with respect to dynamic versus static typing. These versions of the `gcd` function have bodies identical to GCD_{3a}; they only differ in their type annotations.

One might naively want all of the versions in Figure 2 to have exactly the same behavior with respect to type checking and execution. However, GCD_{3c} has the *wrong* annotation, with a return type of `Tuple[int,int]`. To ensure type soundness, a gradual type system must reject this program as ill typed. Thus, when a programmer adds annotations, they can sometimes trigger a static type error. Similarly, adding the wrong annotation can sometimes trigger a runtime error, which is good because it make sure that annotations stay consistent with the code. For example, when using GCD_{2a} (whose second parameter has unknown type), adding `str` as the annotation for the `m` parameter of `modinv`, as shown below, does not trigger a static error, but it does trigger a cast error at the recursive call to `gcd`.

```
def modinv(a, m : str):
  (g, x, y) = gcd(a, m)
  ...
modinv(3, 'hi %s')
```

GCD_{1a}

```
def gcd(a, b): ...
```

GCD_{2a}

```
def gcd(a:int, b): ...
```

GCD_{3a}

```
def gcd(a:int, b:int)
      -> Tuple[int,int,int]:
  if a == 0: return (b, 0, 1)
  else:
    (g, y, x) = gcd(b % a, a)
    return (g, x - (b // a) * y, y)
```

GCD_{2b}

```
def gcd(a, b:int): ...
```

GCD_{3c}

```
def gcd(a:int, b:int)->Tuple[int,int]:
  ...
```

Figure 2 Evolutions of the extended greatest-common divisor algorithm.

What about the reverse? What does removing type annotations do to the behavior of a gradually typed program? The *gradual guarantee* says that if a gradually typed program is well typed, then removing type annotations always produces a program that is still well typed. Further, if a gradually typed program evaluates to a value, then removing type annotations always produces a program that evaluates to an equivalent value.

One of the primary use cases for gradual typing is to enable the evolution of programs from untyped to typed. Thus, one might be disappointed that the graduate guarantee is not as strong when moving in that direction. However, the gradual guarantee has more to say about this direction: a program remains well typed so long as only correct type annotations are added. We take *correct* to mean that the annotation agrees with the corresponding annotation in some fully annotated and well-typed version of the program. With this definition, one can apply the gradual guarantee to show that the program of interest remains well typed with the addition of correct type annotations.

3 The Gradually Typed Lambda Calculus

Here we review the GTLC [50] (Figure 3), which extends the STLC with the unknown type $\star$ and un-annotated functions. The blame labels ℓ represent source position information from the parser, so blame labels are unique. The typing rules for constants, variables, and functions are the same as in the STLC. The two key aspects of the GTLC type system can be seen in the rule for application. First, the consistency relation, written $T_1 \sim T_2$, is used in GTLC where the STLC would check for type equality. The consistency relation is more liberal when it comes to the unknown type: it relates any type to the unknown type. For example, consistency is responsible for the the call `gcd(15, 9)`, with version GCD_{1a}, being well typed. The argument types are `int`, the parameter types are $\star$, and `int` $\sim \star$. The consistency relation is responsible for rejecting `gcd(32, true)`, with GCD_{3a}, because `bool` $\not\sim$ `int`. In contrast to subtyping, consistency is symmetric but not transitive. Gradual typing can be added to object-oriented languages by combining subtyping and consistency in a principled fashion [51, 6]. Another important aspect of the GTLC is the metafunction *fun*. The GTLC allows a term in function position to be of type $T_1 \to T_2$ or of type $\star$. The *fun* metafunction extracts the domain and codomain type, treating $\star$ as if it were $\star \to \star$ [22].

Blame labels	ℓ			Constants	k	$::=$	$\texttt{true} \mid 0 \mid \texttt{inc} \mid \cdots$
Base types	B	$::=$	$\texttt{int} \mid \texttt{bool}$	Expressions	e	$::=$	$k \mid x \mid \lambda x{:}T.\, e \mid (e\ e)^\ell$
Types	T	$::=$	$B \mid T \to T \mid \star$				$\lambda x.\, e \equiv \lambda x{:}\,\star\,.\, e$

Consistency $\boxed{T \sim T}$

$$\frac{}{\star \sim T} \qquad \frac{}{T \sim \star} \qquad \frac{}{B \sim B} \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$$

Expression Typing $\boxed{\Gamma \vdash e : T}$

$$\ldots \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad \mathit{fun}(T_1) = T_{11} {\to} T_{12} \qquad T_2 \sim T_{11}}{\Gamma \vdash (e_1\ e_2)^\ell : T_{12}}$$

Function Matching $\boxed{\mathit{fun}(T) = T \to T}$

$$\mathit{fun}(T_{11} \to T_{12}) = T_{11} \to T_{12}$$
$$\mathit{fun}(\star) = \star \to \star$$

Dynamic Semantics: $e \Downarrow r \equiv \emptyset \vdash e \rightsquigarrow f : T$ and $f \longmapsto^* r$ for some f and T.

Figure 3 The Gradually Typed Lambda Calculus (GTLC).

The dynamic semantics of the GTLC is defined by translation into an internal cast calculus much like the Blame Calculus [67]. The internal cast calculus extends the STLC with the unknown type $\star$ but it replaces the implicit casts of the GTLC with explicit casts. The cast calculus and translation is defined in Figure 4. The cast expression has the form $f : T \Rightarrow^\ell T$, which enables a left-to-right reading. We abbreviate a sequence of two casts $(e : T_1 \Rightarrow^{\ell_1} T_2) : T_2 \Rightarrow^{\ell_2} T_3$ as follows to avoid repetition: $e : T_1 \Rightarrow^{\ell_1} T_2 \Rightarrow^{\ell_2} T_3$. The translation from the GTLC to the cast calculus is defined by the judgment $\Gamma \vdash e \rightsquigarrow f : T$. Again, application is the interesting case. We insert a cast on the expression in function position to make sure it has type $T_1 \to T_2$ and we insert a cast around the argument to make sure it has type T_1. Each inserted cast corresponds to the use of *fun* or $\sim$ in the type system.

The dynamic semantics of the cast calculus is given in Figure 4. The first things to note are the two value forms specific to casts. A value enclosed in a cast between two function types is itself a value, which we call a *wrapped* function. A value that is cast from a ground type G to $\star$ is also a value, which we call an *injection*. Ground types include only base types and the function type $\star \to \star$.[2]

Now we turn to the reduction rules for casts. Identity casts on base types and $\star$ are discarded (rule IdBase and IdStar). When an injection (a cast from a ground type to $\star$) meets a projection (a cast from $\star$ to a ground type) then either the ground types are equal and the casts are discarded (rule Succeed) or the ground types differ and the program halts and assigns blame (rule Fail). A cast to or from $\star$ that involves a non-ground type is decomposed into two casts with a ground type in the middle (rules Ground and Expand). The Ground rule is necessary to ensure that injections are restricted to ground types. The Expand rule is not strictly necessary but allows Succeed and Fail to focus on ground type.

Perhaps the most interesting rule concerns casts between function types (rule AppCast). Recall the example in Section 2.2 in which the function `fun` of type $\star \to \star$ is passed to `deriv` at type `float` $\to$ `float` and applied to 3.01. The cast from $\star \to \star$ to `float` $\to$ `float` applied to `fun` is a value. The application of a wrapped function breaks the cast into two parts: a

[2] Restricting injections to ground types gives the UD semantics [49, 67]. To instead obtain the D semantics, this restriction can be lifted [49].

Ground Types	G	$::=$	$B \mid \star{\to}\star$
Expressions	f	$::=$	$k \mid x \mid \lambda x{:}T.\,f \mid f\;f \mid f : T \Rightarrow^{\ell} T \mid \texttt{blame}_T\,\ell$
Values	v	$::=$	$k \mid \lambda x{:}T.\,f \mid v : T_1{\to}T_2 \Rightarrow^{\ell} T_3{\to}T_4 \mid v : G \Rightarrow^{\ell} \star$
Results	r	$::=$	$v \mid \texttt{blame}_T\,\ell$
Frames	F	$::=$	$\square\,f \mid v\,\square \mid \square : T_1 \Rightarrow^{\ell} T_2$

Expression Typing $\boxed{\Gamma \vdash_C f : T}$

$$\ldots \qquad \frac{\Gamma \vdash_C f : T_1 \qquad T_1 \sim T_2}{\Gamma \vdash_C (f : T_1 {\Rightarrow^{\ell}} T_2) : T_2} \qquad \frac{}{\Gamma \vdash_C \texttt{blame}_T\,\ell : T}$$

Cast Insertion $\boxed{\Gamma \vdash e \rightsquigarrow f : T}$

$$\ldots \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow f_1 : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow f_2 : T_2 \\ \mathit{fun}(T_1) = T_{11} \to T_{12} \qquad T_2 \sim T_{11}\end{array}}{\Gamma \vdash (e_1\;e_2)^{\ell} \rightsquigarrow (f_1 : T_1 \Rightarrow^{\ell} T_{11}{\to}T_{12})\;(f_2 : T_2 \Rightarrow^{\ell} T_{11}) : T_{12}}$$

Dynamic Semantics $\boxed{f \longmapsto f}$

$$\begin{array}{rll}
(\lambda x{:}T.\,f)\;v \longmapsto [x := v]f & & (\textsc{Beta}) \\
v : B \Rightarrow^{\ell} B \longmapsto v & & (\textsc{IdBase}) \\
v : \star \Rightarrow^{\ell} \star \longmapsto v & & (\textsc{IdStar}) \\
v : G \Rightarrow^{\ell_1} \star \Rightarrow^{\ell_2} G \longmapsto v & & (\textsc{Succeed}) \\
v : G_1 \Rightarrow^{\ell_2} \star \Rightarrow^{\ell_2} G_2 \longmapsto \texttt{blame}_{G_2}\,\ell_2 & \text{if } G_1 \neq G_2 & (\textsc{Fail}) \\
(v_1 : T_1{\to}T_2 \Rightarrow^{\ell} T_3{\to}T_4)\;v_2 \longmapsto v_1\;(v_2 : T_3 \Rightarrow^{\ell} T_1) : T_2 \Rightarrow^{\ell} T_4 & & (\textsc{AppCast}) \\
v : T \Rightarrow^{\ell} \star \longmapsto v : T \Rightarrow^{\ell} G \Rightarrow^{\ell} \star & \text{if } T \neq \star, T \neq G, T \sim G & (\textsc{Ground}) \\
v : \star \Rightarrow^{\ell} T \longmapsto v : \star \Rightarrow^{\ell} G \Rightarrow^{\ell} T & \text{if } T \neq \star, T \neq G, T \sim G & (\textsc{Expand}) \\
F[f] \longmapsto F[f'] & \text{if } f \longmapsto f' & (\textsc{Cong}) \\
F[\texttt{blame}_{T_1}\,\ell] \longmapsto \texttt{blame}_{T_2}\,\ell \quad \text{if } \vdash F : T_1 \Rightarrow T_2 & & (\textsc{Blame})
\end{array}$$

Figure 4 Cast insertion and the internal cast calculus.

cast on the argument and on the return value. The following shows the important steps.

$$\begin{array}{ll}
& (\texttt{fun} : \star \to \star \Rightarrow^{\ell} \texttt{float} \to \texttt{float})\,3.01 \\
\longmapsto & (\texttt{fun}\,(3.01 : \texttt{float} \Rightarrow^{\ell} \star)) : \star \Rightarrow^{\ell} \texttt{float} \\
\longmapsto^{*} & 5.02 : \texttt{float} \Rightarrow^{\ell} \star \Rightarrow^{\ell} \texttt{float} \quad \longmapsto \quad 5.02
\end{array}$$

Recall that the second call to `deriv` resulted in a cast error. In that case `fun` is applied to -2.99. The result of the function is a `None` value, which cannot be cast to `float`. Thanks to the blame tracking, the error $\texttt{blame}_{\texttt{float}}\,\ell$ includes the source information for the cast that arose from passing `fun` into `deriv`.

$$\begin{array}{ll}
& (\texttt{fun} : \star \to \star \Rightarrow^{\ell} \texttt{float} \to \texttt{float}) \;\; -2.99 \\
\longmapsto & (\texttt{fun}\,(-2.99 : \texttt{float} \Rightarrow \star)) : \star \Rightarrow^{\ell} \texttt{float} \\
\longmapsto^{*} & \texttt{None} : \texttt{NoneType} \Rightarrow^{\ell} \star \Rightarrow^{\ell} \texttt{float} \quad \longmapsto \quad \texttt{blame}_{\texttt{float}}\,\ell
\end{array}$$

4 Criteria for Gradual Typing

We begin by reviewing the formal criteria for gradually typed languages that appear in the literature. These criteria cover the first three subsections of Section 2. We then develop a formal statement of our new criterion, the gradual guarantee. For the sake of precision, we state each criterion as a theorem about the GTLC. Our intent is that one would adapt these theorems to other gradually typed languages.

4.1 Gradual as a Superset of Static and Dynamic

As discussed in Section 2.1, a gradually typed language is intended to include both an untyped language and a typed language. For example, GTLC should encompasses both DTLC and the STLC. Siek and Taha [50] prove that the GTLC type system is equivalent to the STLC on fully annotated programs. We extend this criterion to require the dynamic semantics of the GTLC to coincide with the STLC on fully annotated programs in the theorem below. Let $\vdash_S$ and $\Downarrow_S$ denote the typing judgment and evaluation function of STLC, respectively. We say that a type is *static* if the unknown type does not occur in it.

▶ **Theorem 1** (Equivalence to the STLC for fully annotated terms).
Suppose e is fully annotated and T is static.
1. $\vdash_S e : T$ *if and only if* $\vdash e : T$. *(Siek and Taha [50]).*
2. $e \Downarrow_S v$ *if and only if* $e \Downarrow v$.

The relationship between the GTLC and DTLC is more nuanced by necessity because there exist terms that are both fully annotated and un-annotated. Suppose the constant `inc` has type `int`→`int` and the constant `true` has type `bool`. Then the application of `inc` to `true` is ill-typed in the GTLC even though it is a well-typed program in the DTLC (trivially, because programs are). Similar issues arise when extending the GTLC with other constructs, such as conditionals, where one must choose to lean towards either static typing or dynamic typing. Nevertheless, there is a simple encoding of the DTLC into the GTLC, here written as $\lceil\cdot\rceil$, that casts constants to the unknown type. Let $\Downarrow_D$ denote evaluation of DTLC.

▶ **Theorem 2** (Embedding of DTLC). *Suppose that e is a term of DTLC.*
1. $\vdash \lceil e \rceil : \star$ *(Siek and Taha [50]).*
2. $e \Downarrow_D r$ *if and only if* $\lceil e \rceil \Downarrow r$.

The two theorems of this section characterize programs at the two extremes: fully static and fully dynamic. However, these theorems say nothing about partially typed programs which is the norm for gradually typed languages. The next section describes notions of soundness that make sense for partially typed programs.

4.2 Soundness for Gradually Typed Languages

Siek and Taha [50] prove that the GTLC is sound in the same way that many dynamically typed languages are sound: execution never encounters untrapped errors (but trapped errors could be ubiquitous). Let $e \Uparrow$ indicate that e diverges.

▶ **Theorem 3** (Type Safety of GTLC, Siek and Taha [50]). *If* $\vdash e : T$, *then either* $e \Downarrow v$ *and* $\vdash v : T$ *for some* v, *or* $e \Downarrow \mathtt{blame}_T\, \ell$ *for some* ℓ, *or* $e \Uparrow$.

This theorem is unsatisfying because it does not tell us that statically typed regions are not to blame for trapped errors. Thankfully, blame tracking provides the right mechanism

$$\begin{tikzcd}
 & (\lambda y{:}\star\,.\,y)\ ((\lambda x{:}\star\,.\,x)\ 42) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
(\lambda y{:}\texttt{int}.\,y)\ ((\lambda x{:}\star\,.\,x)\ 42) \arrow[d, dash] & (\lambda y{:}\star\,.\,y)\ ((\lambda x{:}\texttt{int}.\,x)\ 42) \arrow[dl, dash] & (\lambda y{:}\texttt{bool}.\,y)\ ((\lambda x{:}\star\,.\,x)\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{int}.\,y)\ ((\lambda x{:}\texttt{int}.\,x)\ 42) & & (\lambda y{:}\texttt{bool}.\,y)\ ((\lambda x{:}\texttt{bool}.\,x)\ 42)
\end{tikzcd}$$

Figure 5 A lattice of differently annotated versions of a gradually typed program.

for formulating type soundness for gradually typed programs. The Blame Theorem [62, 67] characterizes the safe versus possibly unsafe casts. Adapting these results to gradual typing, we arrive at the Blame-Subtyping Theorem, which states that if the cast insertion procedure inserts a cast from a type to another and the former is a subtype of the latter, then the cast is guaranteed not to fail. To state this theorem, we give the following definition of subtyping.

$$\boxed{T <: T}$$

$$B <: B \qquad \star <: \star \qquad \frac{T <: G}{T <: \star} \qquad \frac{S_1 <: T_1 \qquad T_2 <: S_2}{T_1 \to T_2 <: S_1 \to S_2}$$

▶ **Theorem 4** (Blame-Subtyping Theorem). *If* $\emptyset \vdash e \rightsquigarrow f : T$ *,* f *contains a cast* $f' : T_1 \Rightarrow^{\ell} T_2$*,* $T_1 <: T_2$*, and* $e \Downarrow \texttt{blame}_T\, l'$ *then* $l \neq l'$*.*

The practical implication of this theorem is that a programmer (or automated tool) can easily analyze a region of code to tell whether they region is safe or whether it might trigger a cast error. Furthermore, the Blame-Subtyping Theorem guarantees that fully-static regions of code are never to blame for cast errors.

4.3 The Gradual Guarantee

So far we have four theorems to characterize gradually typed languages, but none of them address the requirement of Section 2.3. Roughly speaking, changes to the annotations of a gradually typed program should not change the static or dynamic behavior of the program.

For example, suppose we start with the un-annotated program at the top of the lattice in Figure 5. One would hope that adding type annotations would yield a program that still evaluates to 42. Indeed, in the GTLC, adding the type annotation `int` for parameters x and y does not change the outcome of the program. On the other hand, the programmer might insert the wrong annotation, say `bool` for parameter y, and trigger a trapped error. Even worse, the programmer might add `bool` for x and cause a static type error. So we cannot claim full contextual equivalence when going down in the lattice, but we can make a strong claim when going up in the lattice: the less precise program behaves the same as the more precise one except that it might have fewer trapped errors.

The partial order at work in Figure 5 is the *precision relation* on types and terms, defined in Figure 6. Type precision [52] is also known as naive subtyping [67]. Term precision is the natural extension of type precision to terms. Here we write $T \sqsubseteq T'$ when type T is more precise than T' and $e \sqsubseteq e'$ when term e is more precisely annotated than e'.[3] We give the definition of these relations in Section 6. We characterize the expected static and dynamic behavior of programs as we move up and down in precision as follows.

[3] We apologize that the direction of increase in precision is to the left instead of to the right. We settled on this directionality to be consistent with subtyping.

Type Precision $\boxed{T \sqsubseteq T}$

$$\frac{}{T \sqsubseteq \star} \qquad \frac{}{B \sqsubseteq B} \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4}$$

Term Precision for the GTLC $\boxed{e \sqsubseteq e}$

$$\frac{}{k \sqsubseteq k} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{T_1 \sqsubseteq T_2 \quad e_1 \sqsubseteq e_2}{\lambda x{:}T_1.\, e_1 \sqsubseteq \lambda x{:}T_2.\, e_2} \qquad \frac{e_1 \sqsubseteq e_2 \quad e_1' \sqsubseteq e_2'}{(e_1\ e_1')^{\ell} \sqsubseteq (e_2\ e_2')^{\ell}}$$

Figure 6 Type and Term Precision.

▶ **Theorem 5** (Gradual Guarantee). *Suppose $e \sqsubseteq e'$ and $\vdash e : T$.*
1. $\vdash e' : T'$ *and* $T \sqsubseteq T'$.
2. *If* $e \Downarrow v$, *then* $e' \Downarrow v'$ *and* $v \sqsubseteq v'$.
   If $e \Uparrow$ *then* $e' \Uparrow$.
3. *If* $e' \Downarrow v'$, *then* $e \Downarrow v$ *where* $v \sqsubseteq v'$, *or* $e \Downarrow \mathtt{blame}_T\, l$.
   If $e' \Uparrow$, *then* $e \Uparrow$ *or* $e \Downarrow \mathtt{blame}_T\, l$.

Now that we have stated the gradual guarantee, it is natural to wonder how important it is. Of course, there are many pressures at play during the design of any particular programming language, such as concerns for efficiency, safety, learning curve, and ease of implementation. However, if a language is intended to support gradual typing, that means the programmer should be able to conveniently evolve code from being statically typed to dynamically typed, and vice versa. With the gradual guarantee, the programmer can be confident that when removing type annotations, a well-typed program will continue to be well-typed (with no need to insert explicit casts) and a correctly running program will continue to do so. When adding type annotations, if the program remains well typed, the only possible change in behavior is a trapped error due to a mistaken annotation. Furthermore, it is natural to consider tool support (via static or dynamic analysis) for adding type annotations, and we would not want the addition of types to cause programs to misbehave in unpredictable ways.

5 Critiques of Language Designs in Light of the Gradual Guarantee

Researchers have explored a large number of points in the design space for gradually typed languages. A comprehensive survey is beyond the scope of this paper, but we have selected a handful of them to discuss in light of the gradual guarantee.

5.1 GTLC

As discussed above, the Gradually Typed Lambda Calculus [50] satisfies the gradual guarantee (the proof is in Section 6).

5.2 GTLC with Mutable References

Siek and Taha [50] treat mutable references as invariant in their type system, disallowing implicit casts that change the pointed-to type. Consider that design in relation to the lattice of programs in Figure 7. The program at the top is well-typed because `Ref int` may be implicitly cast to $\star$ (anything can). The program at the bottom is well-typed; it contains no

$$\begin{tikzcd}
(\lambda y{:}\star.\,y)\ ((\lambda x{:}\star.\,x)\ \texttt{ref}\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{Ref int}.\,y)\ ((\lambda x{:}\texttt{Ref}\ \star.\,x)\ \texttt{ref}\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{Ref int}.\,y)\ ((\lambda x{:}\texttt{Ref int}.\,x)\ \texttt{ref}\ 42)
\end{tikzcd}$$

Figure 7 A lattice of varying-precision for a program with mutable references.

$$\begin{tikzcd}
(\lambda f : \star.\, f\ \texttt{true})\ (\lambda x : \star.\, x) \arrow[d, dash] \\
(\lambda f : \texttt{bool}{\to}\texttt{bool}.\, f\ \texttt{true})\ (\lambda x : \star.\, x) \arrow[d, dash] \\
(\lambda f : \texttt{bool}{\to}\texttt{bool}.\, f\ \texttt{true})\ (\lambda x : \texttt{bool}.\, x)
\end{tikzcd}$$

Figure 8 A lattice of varying-precision for a higher-order program.

implicit casts. However, the program in the middle is not well-typed because one cannot cast between `Ref int` and `Ref` $\star$. These programs are related by precision and the bottom program is well-typed, so part 1 of the gradual guarantee is violated.

Herman et al. [34, 35] remedy the situation with a design for mutable references that allows implicit casts between reference types so long as the pointed-to types are consistent (in the technical sense of Siek and Taha [50]). We conjecture that their design satisfies the gradual guarantee. Likewise, we conjecture that the new monotonic approach [53] to mutable references satisfies the gradual guarantee.

5.3 TS*

The TS* language [57] layers a static type system over JavaScript to provide protection from security vulnerabilities. TS* is billed as a gradually typed language, but it does not satisfy the gradual guarantee. For example, consider the program variations in Figure 8. The function with parameter f is representative of a software framework and the function with parameter x is representative of a client-provided callback.

A typical scenario of gradual evolution would start with the untyped program at the top, proceed to the program in the middle, in which the framework interface has been statically typed (parameter f) but not the client, then finally evolve to the program at the bottom which is fully typed. The top-most and bottom-most versions evaluate to `true` in TS*. However, the middle program produces a trapped error, as explained by the following quote.

> "Coercions based on `setTag` can be overly conservative, particularly on higher-order values. For example, trying to coerce the identity function $id : \star \to \star$ to the type `bool` $\to$ `bool` using `setTag` will fail, since $\star \not<:$ `bool`." [57]

The example in Figure 8 is a counterexample to part 2 of the gradual guarantee; the bottom program evaluates to `true`, so the middle program should too, but it does not. The significance of not satisfying the gradual guarantee, as we can see in this example, is that programmers will encounter trapped errors in the process of refactoring type annotations, and will be forced to make several coordinated changes to get back to a well-behaved program.

5.4 Thorn and Like Types

The Thorn language [69] is meant to support the evolution of (dynamically typed) scripts to (statically typed) programs. The language provides a dynamic type **dyn**, nominal class types, and *like* types.

We revisit parts of Figure 5 under several scenarios. First, suppose we treat $\star$ as **dyn** and `int` is a concrete type (i.e. a class type). Then we have the following counterexample to part 1 of the gradual guarantee; the bottom program is well-typed but not the top program.

$$\begin{tikzcd}(\lambda y{:}\texttt{int}.\,y)\ ((\lambda x{:}\mathbf{dyn}.\,x)\ 42) \arrow[d, dash] \\ (\lambda y{:}\texttt{int}.\,y)\ ((\lambda x{:}\texttt{int}.\,x)\ 42)\end{tikzcd}$$

Second, suppose again that $\star$ is **dyn** but that we replace `bool` with `like bool` and `int` with `like int`. Now every program in Figure 5 is well-typed, even the bottom-right program that should not be:

$$(\lambda y{:}\texttt{like bool}.\,y)\ ((\lambda x{:}\texttt{like bool}.\,x)\ 42)$$

So in this scenario the gradual guarantee is satisfied, but not the correspondence with a fully static language (Theorem 1). Finally, suppose we replace $\star$ with `like int` and treat `int` as a concrete type. Similar to the first scenario, we get the following counterexample to part 1 of the gradual guarantee; the bottom program is well-typed but not the top program. (It would need an explicit cast to be well-typed.)

$$\begin{tikzcd}(\lambda y{:}\texttt{int}.\,y)\ ((\lambda x{:}\texttt{like int}.\,x)\ 42) \arrow[d, dash] \\ (\lambda y{:}\texttt{int}.\,y)\ ((\lambda x{:}\texttt{int}.\,x)\ 42)\end{tikzcd}$$

We note that efficiency is an important design consideration for Thorn and that it is challenging to satisfy the gradual guarantee and efficiency at the same time. For example, only recently have we found a way to support mutable references without using wrappers [53].

5.5 Grace and Structural Type Tests

The Grace language [7] is gradually typed and includes a facility for pattern matching on structural types. Boyland [9] observes that, depending on the semantics of the pattern matching, the gradual guarantee may not hold for Grace. Here we consider an example in an extension of the GTLC with a facility for testing the type of a value: the expression e `is` T returns `true` if e evaluates to a value of type T and `false` otherwise. Boyland [9] considers three possible interpretations what “a value of type T” means: an optimistic semantics that checks whether the type of the value is *consistent* with the given type, a pessimistic semantics that checks whether the type of the value is *equal* to the given type, and a semantics similar in spirit to that of Ahmed et al. [2], which only checks the top-most type constructor.

Consider the following example where g is a function that tests whether its input is a function of type `int→int`. On the right we show a lattice of several programs that apply g to the identity function.

$$g \equiv (\lambda f:\star.\,f\ \texttt{is int}\to\texttt{int}) \qquad \begin{tikzcd}g\ (\lambda x{:}\star.\,x) \arrow[d, dash] & g\ (\lambda x{:}\star.\,x) \arrow[d, dash] \\ g\ (\lambda x{:}\texttt{int}.\,x) & g\ (\lambda x{:}\texttt{bool}.\,x)\end{tikzcd}$$

- Under the optimistic semantics, $g\ (\lambda x{:}\star.\,x)$ and $g\ (\lambda x{:}\texttt{int}.\,x)$ evaluate to `true` but $g\ (\lambda x{:}\texttt{bool}.\,x)$ evaluates to `false`. So part 2 of the gradual guarantee is violated.
- Under the pessimistic semantics, $g\ (\lambda x{:}\star.\,x)$ evaluates to `false` whereas $g\ (\lambda x{:}\texttt{int}.\,x)$ program evaluates to `true`, so this is a counterexample to part 2 of the gradual guarantee.
- Under the semantics of Ahmed et al. [2], this program is disallowed syntactically. We could instead have $g \equiv (\lambda f:\star.\,f\ \texttt{is}\ \star\to\star)$ and then all three programs would evaluate to `true`. We conjecture that this semantics does satisfy the gradual guarantee.

5.6 Typed Racket and the Polymorphic Blame Calculus

We continue our discussion of type tests, but expand the language under consideration to include parametric polymorphism, as found in Typed Racket [28, 63] and the Polymorphic Blame Calculus [2]. Consider the following programs in which a test-testing function is passed to another function, either simply as $\star$ or at the universal type $\forall\alpha.\,\alpha{\to}\alpha$.

$$\begin{array}{c}
(\lambda f:\star.\, f[\texttt{int}]\ 5)\ (\Lambda\alpha.\,\lambda x:\star.\, x\ \texttt{is}\ \texttt{int}) \\
| \\
(\lambda f:\forall\alpha.\,\alpha{\to}\alpha.\, f[\texttt{int}]\ 5)\ (\Lambda\alpha.\,\lambda x:\star.\, x\ \texttt{is}\ \texttt{int})
\end{array}$$

In the bottom program, 5 is sealed when it is passed to the polymorphic function f. In Typed Racket, a sealed value is never an integer, so the type test returns `false`. The program on the top evaluates to `true`, so this is a counterexample to part 2 of the gradual guarantee.

In the Polymorphic Blame Calculus, applying a type test to a sealed value always produces a trapped error, so this is not a counterexample under that design. We conjecture that one could extend the GTLC with polymorphism and compile it to the Polymorphic Blame Calculus to obtain a language that satisfies the gradual guarantee.

5.7 Reticulated Python and Object Identity

During the evaluation of Reticulated Python, a gradually typed variant of Python, Vitousek et al. [66] encountered numerous problems when adding types to third-party Python libraries and applications. The root of these problems was a classic one: proxies interfere with object identity [20]. (The standard approach to ensuring soundness for gradually typed languages is to create proxies when casting higher-order entities like functions and objects.) Consider the GTLC extended with mutable references and an operator named `alias`? for testing whether two references are aliased to the same heap cell. Then in the following examples, the bottom program evaluates to `true` whereas the top program evaluates to `false`.

$$\begin{array}{c}
\texttt{let}\ r:\texttt{ref}\,\texttt{int} = \texttt{ref}\ 0\ \texttt{in}\ \ (\lambda x:\texttt{ref}\,\texttt{int}.\,\lambda y:\texttt{ref}\ \star.\,\texttt{alias?}\,x\,y)\ r\ r \\
| \\
\texttt{let}\ r:\texttt{ref}\,\texttt{int} = \texttt{ref}\ 0\ \texttt{in}\ \ (\lambda x:\texttt{ref}\,\texttt{int}.\,\lambda y:\texttt{ref}\,\texttt{int}.\,\texttt{alias?}\,x\,y)\ r\ r
\end{array}$$

There are several approaches to mitigate this problem, such as changing `alias`? to see through proxies, use the membrane abstraction [64], or avoid proxies altogether [69, 66, 53]. One particularly thorny issue for Reticulated Python is that the use of the foreign-function interface to C is common in Python programs and the foreign functions are privy to a rather exposed view of Python objects.

6 The Proof of the Gradual Guarantee for the GTLC

Here we summarize the proof of the gradual guarantee for the GTLC. All of the definitions, the proof of the main lemma (Lemma 7), and its dependencies, have been verified in Isabelle [43]. They are available at the following URL:
`https://dl.dropboxusercontent.com/u/10275252/gradual-guarantee-proof.zip`
Part 1 of the gradual guarantee is easy to prove by induction on $e \sqsubseteq e'$. The proof of part 2 is interesting and will be the focus of our discussion. Part 3 is a corollary of part 2.

Because the semantics of the GTLC is defined by translation to the cast calculus, our main lemma concerns a variant of part 2 that is adapted to the cast calculus. For this we

Term Precision for the Cast Calculus $\boxed{\Gamma, \Gamma' \vdash f \sqsubseteq f'}$

$$\ldots \quad \frac{\Gamma, \Gamma' \vdash f \sqsubseteq f' \quad T_1 \sqsubseteq T_1' \quad T_2 \sqsubseteq T_2'}{\Gamma, \Gamma' \vdash (f : T_1 \Rightarrow^{\ell_1} T_2) \sqsubseteq (f' : T_1' \Rightarrow^{\ell_2} T_2')} \quad \frac{\Gamma' \vdash f' : T' \quad T \sqsubseteq T'}{\Gamma, \Gamma' \vdash \mathtt{blame}_T\, \ell \sqsubseteq f'}$$

$$\frac{\begin{array}{cc}\Gamma, \Gamma' \vdash f \sqsubseteq f' & \Gamma' \vdash f' : T' \\ T_1 \sqsubseteq T' & T_2 \sqsubseteq T'\end{array}}{\Gamma, \Gamma' \vdash (f : T_1 \Rightarrow^{\ell} T_2) \sqsubseteq f'} \quad \frac{\begin{array}{cc}\Gamma, \Gamma' \vdash f \sqsubseteq f' & \Gamma \vdash f : T \\ T \sqsubseteq T_1' & T \sqsubseteq T_2'\end{array}}{\Gamma, \Gamma' \vdash f \sqsubseteq (f' : T_1' \Rightarrow^{\ell} T_2')}$$

Abbreviation: $f \sqsubseteq f' \equiv \emptyset, \emptyset \vdash f \sqsubseteq f'$

Figure 9 Term Precision for the Cast Calculus.

need a notion of precision for the cast calculus. Further, the translation to the cast calculus needs to preserve precision. Figure 9 defines precision for the cast calculus in a way that satisfies this need by adding rules that allow extra casts on both the left and right-hand side.

▶ **Lemma 6** (Cast Insertion Preserves Precision). *Suppose* $\Gamma \vdash e \rightsquigarrow f : T$, $\Gamma' \vdash e' \rightsquigarrow f' : T'$, $\Gamma \sqsubseteq \Gamma'$, *and* $e \sqsubseteq e'$. *Then* $\Gamma, \Gamma' \vdash f \sqsubseteq f'$ *and* $T \sqsubseteq T'$.

This lemma is interesting in that it was, in fact, not true for the original formulation of the GTLC [50]. In that version, there were two cast insertion rules for function application, one where the term in function position had an arrow type and one where it had type $\star$. The latter case used the following rule.

$$\frac{\Gamma \vdash e_1 \rightsquigarrow f_1 : \star \quad \Gamma \vdash e_2 \rightsquigarrow f_2 : T}{\Gamma \vdash (e_1\ e_2) \rightsquigarrow ((f_1 : \star \Rightarrow T \to \star)\ f_2) : \star}$$

Using this rule, if we take $e_1 = (((\lambda g{:}\star \to \star . g)\ (\lambda x{:}\star . x))\ 42)$ and $e_2 = (((\lambda g{:}\star . g)\ (\lambda x{:}\star . x))\ 42)$, we have that $e_1 \sqsubseteq e_2$. However, when we obtain f_1, f_2 by $\emptyset \vdash e_1 \rightsquigarrow f_1 : \star$ and $\emptyset \vdash e_2 \rightsquigarrow f_2 : \star$, we get

$$f_1 = (((\lambda g{:}\star \to \star . g)\ (\lambda x{:}\star . x))\ (42 : \mathtt{Int} \Rightarrow \star))$$
$$f_2 = ((((\lambda g{:}\star . g)((\lambda x{:}\star . x){:}\star \to \star \Rightarrow \star)){:}\star \Rightarrow \mathtt{Int} \to \star)\ 42)$$

and $f_1 \not\sqsubseteq f_2$.

The following is the statement of the main lemma, which establishes that less-precise programs simulate more precise programs.

▶ **Lemma 7** (Simulation of More Precise Programs). *Suppose* $f_1 \sqsubseteq f_1'$, $\vdash f_1 : T$, *and* $\vdash f_1' : T'$. *If* $f_1 \longmapsto f_2$, *then* $f_1' \longmapsto^* f_2'$ *and* $f_2 \sqsubseteq f_2'$ *for some* f_2'.

The proof of the Lemma 7 is by induction on the derivation of $f_1 \sqsubseteq f_1'$ followed by case analysis on $f_1 \longmapsto f_2$. The proof required four major lemmas (and numerous minor lemmas).

Because the precision relation of the cast calculus allows extra casts on the right-hand side, we prove the following lemma by case analysis on T_1' and T_2'. The precision relation also allows extra casts on the left, but we handled those cases in-line in the proof of Lemma 7.

▶ **Lemma 8** (Extra Cast on the Right). *Suppose* $\vdash_C v : T$, $\vdash_C v' : T_1'$, $T \sqsubseteq T_1'$, *and* $T \sqsubseteq T_2'$. *If* $v \sqsubseteq v'$, *then* $v' : T_1' \Rightarrow^{\ell} T_2' \longmapsto^* v''$ *and* $v \sqsubseteq v''$ *for some* v''.

To handle cases where the more-precise program is already a value, we prove that the less-precise program can reduce to a related value. This proof is by induction on the derivation of $v \sqsubseteq f'$.

▶ **Lemma 9** (Catchup to Value on the Left). *Suppose* $\vdash_C v : T$, $\vdash_C f' : T'$, *and* $v \sqsubseteq f'$. *Then* $f' \longmapsto^* v'$ *and* $v \sqsubseteq v'$.

The most complex part of the proof involved application, as functions may be wrapped in a series of casts. We prove the following lemma by induction on the derivation of $(\lambda x : T_1.\, f) \sqsubseteq v'_1$.

▶ **Lemma 10** (Simulation of Function Application). *Suppose* $\vdash_C (\lambda x : T_1.\, f) : T_1 {\to} T_2$, $\vdash_C v : T_1$, $\vdash_C v' : T_1$, $\vdash_C v'_1 : T'_1 {\to} T'_2$, $\vdash_C v'_2 : T'_1$, *and* $T_1 {\to} T_2 \sqsubseteq T'_1 {\to} T'_2$. *If* $(\lambda x : T_1.\, f) \sqsubseteq v'_1$ *and* $v \sqsubseteq v'_2$, *then* $v'_1\ v'_2 \longmapsto^* f'$, $[x := v]f \sqsubseteq f'$, *and* $\vdash_C f' : T'_2$.

We prove the next lemma by induction on $(v_1 : T_1 {\to} T_2 \Rightarrow^{\ell} T_3 {\to} T_4) \sqsubseteq v'_1$.

▶ **Lemma 11** (Simulation of Unwrapping). *Suppose* $\vdash_C v_1 : T_1 {\to} T_2$, $\vdash_C v_2 : T_1$, $\vdash_C v'_1 : T'_1 {\to} T'_2$, $\vdash_C v'_2 : T'_1$, *and* $T_1 {\to} T_2 \sqsubseteq T'_1 {\to} T'_2$. *If* $(v_1 : T_1 {\to} T_2 \Rightarrow^{\ell} T_3 {\to} T_4) \sqsubseteq v'_1$ *and* $v_2 \sqsubseteq v'_2$, *then* $v'_1\ v'_2 \longmapsto^* f'$ *and* $v_1\ (v_2 : T_3 \Rightarrow^{\ell} T_1) : T_2 \Rightarrow^{\ell} T_4 \sqsubseteq f'$.

With Lemma 7 in hand, we prove part 2 of the gradual guarantee by induction on the number of reduction steps. Part 3 is a corollary of part 2, as follows. Assume that e' evaluates to v'. Because e is well typed, it may either evaluate to a value v, evaluate to a trapped error, or diverge. If it evaluates to some v, then we have $v \sqsubseteq v'$ by part 2 and because reduction is deterministic. If e results in a trapped error, we are done. If e diverges, then so does e' by part 2, but that is a contradiction.

7 Conclusion

Gradual typing should allow programmers to straightforwardly evolve their programs between the dynamic and static typing disciplines. However, this is only available to the programmer if the language designer formulates their language in a specific way. In this paper, we emphasize the need for formal criteria for gradually typed languages and offer a new criterion, the gradual guarantee. This formal property captures essential aspects of the evolution of code between typing disciplines. The current landscape of gradually typed languages reveals that this aspect has been either silently included or unfortunately overlooked, but never explicitly taken into consideration. That we could formally prove that GTLC obeys the gradual guarantee is a promising step and indicates that it is a realistic goal for researchers designing gradually typed systems. It remains to be investigated whether the gradual guarantee can be proven for a full-blown language with modern features (such as polymorphism, recursive types, type inference, etc.). We look forward to working with the research community to address this challenge.

Acknowledgments. We thank Sam Tobin-Hochstadt, Philip Wadler, Andrew Kent, Ambrose Bonnaire-Sergeant, Andre Kuhlenschmidt, and Ronald Garcia for their input. The gradual guarantee was discovered independently by John Boyland and the Gradual Typing Group at Indiana University in 2014. The gradual guarantee is sketched in Boyland's paper at the FOOL workshop [9] and in Siek's presentation at the NII Shonan meeting on Contracts [48].

References

1 Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

2 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. In *Symposium on Principles of Programming Languages*, January 2011.

3 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, August 2013.

4 Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'14, pages 251–270, New York, NY, USA, 2014. ACM.

5 Christopher Anderson and Sophia Drossopoulou. BabyJ – From Object Based to Class Based Programming via Types. *ENTCS*, 82(8), 2003.

6 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2014.

7 Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: The absence of (inessential) difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 85–98, New York, NY, USA, 2012. ACM.

8 Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the jvm. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 117–136, 2009.

9 John Tang Boyland. The problem of structural type tests in a gradual-typed language. In *Foundations of Object-Oriented Langauges*, FOOL, 2014.

10 Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA'93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.

11 Luca Cardelli. *Handbook of Computer Science and Engineering*, chapter Type Systems. CRC Press, 1997.

12 Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Not.*, 27(8):15–42, August 1992.

13 Robert Cartwright and Mike Fagan. Soft typing. In *PLDI'91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

14 Craig Chambers and the Cecil Group. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, 2004.

15 Olaf Chitil. Practical typed lazy contracts. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP'12, pages 67–76, New York, NY, USA, 2012. ACM.

16 Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'11, pages 215–226, New York, NY, USA, 2011. ACM.

17 Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *ESOP*, 2012.

18 Tim Disney and Cormac Flanagan. Gradual information flow typing. In *Workshop on Script to Program Evolution*, 2011.

19 R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, ICFP, pages 48–59, October 2002.

20 R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *European Conference on Object-Oriented Programming*, 2004.

21 Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999.

22 Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Symposium on Principles of Programming Languages*, POPL, pages 303–315, 2015.

23 James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Sun Developer Network, 1996.

24 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA'05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.

25 Michael Greenberg. Space-efficient manifest contracts. In *POPL*, 2015.

26 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL) 2010*, 2010.

27 Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

28 Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium*, 2007.

29 Łukasz Langa Guido van Rossum, Jukka Lehtosalo. Type hints. `https://www.python.org/dev/peps/pep-0484/`, September 2014. draft.

30 Lars T. Hansen. Evolutionary programming and gradual typing in ECMAScript 4 (tutorial). Technical report, ECMA TG1 working group, November 2007.

31 Anders Hejlsberg. C# 4.0 and beyond by anders hejlsberg. Microsoft Channel 9 Blog, April 2010.

32 Anders Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, 2012.

33 Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

34 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

35 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167–189, 2010.

36 Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'11, 2011.

37 International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*, September 1998.

38 Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU reference manual. Technical Report LCS-TR-225, MIT, October 1979.

39 Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing, or, theorems for low, low prices! In *Proceedings of the 17th European Symposium on Programming (ESOP'08)*, March 2008.

40 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2007.

41 Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

42 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

43 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, November 2007.

44 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *Symposium on Principles of Programming Languages*, POPL, pages 481–494, January 2012.

45 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. Technical Report MSR-TR-2014-99, Microsoft Research, 2014.

46 Manuel Serrano. *Bigloo: a practical Scheme compiler*. Inria-Rocquencourt, April 2002.

47 Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

48 Jeremy G. Siek. Design and evaluation of gradual typing for Python. `https://dl.dropboxusercontent.com/u/10275252/shonan-slides-2014.pdf`, May 2014.

49 Jeremy G. Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, ESOP, pages 17–31, March 2009.

50 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

51 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, volume 4609 of *LCNS*, pages 2–27, August 2007.

52 Jeremy G. Siek and Manish Vachharajani. Gradual typing and unification-based inference. In *DLS*, 2008.

53 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *European Symposium on Programming*, ESOP, April 2015.

54 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, POPL, pages 365–376, January 2010.

55 Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

56 T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, 2012.

57 Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *ACM Conference on Principles of Programming Languages (POPL)*, January 2014.

58 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, pages 793–810, 2012.

59 The Dart Team. *Dart Programming Language Specification*. Google, 1.2 edition, March 2014.

60 Satish Thatte. Quasi-static typing. In *POPL 1990*, pages 367–381, New York, NY, USA, 1990. ACM Press.

61 Peter Thiemann and Luminous Fennell. Gradual typing for annotated type systems. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 47–66. Springer Berlin Heidelberg, 2014.

62 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium*, 2006.

63 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, January 2008.

64 Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS'10, pages 59–72, New York, NY, USA, 2010. ACM.

65 Julien Verlaguet. Facebook: Analyzing PHP statically. In *Commercial Users of Functional Programming (CUFP)*, 2013.

66 Michael M. Vitousek, Jeremy G. Siek, Andrew Kent, and Jim Baker. Design and evaluation of gradual typing for Python. In *Dynamic Languages Symposium*, 2014.

67 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP, pages 1–16, March 2009.

68 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, ECOOP'11. Springer-Verlag, 2011.

69 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages*, POPL, pages 377–388, 2010.

None, One, Many – What's the Difference, Anyhow?

Friedrich Steimann

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen, Germany
`steimann@acm.org`

Abstract

We observe that compared to natural and modelling languages, the differences in expression required to deal with no, one, or many objects in programming languages are particularly pronounced. We identify some problems inherent in type-based unifications of different numbers, and advocate a solution that builds on the introduction of multiplicity as a new grammatical category of programming languages.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases objects, collections, relationships, pointers, multiplicity, null

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.294

1 Introduction

Programming languages like Smalltalk, Java, and C$^\sharp$ abstract from pointers in that variables holding (heap) objects have reference semantics by default. However, the pointers (or references) become apparent, and need to be dealt with explicitly, in two not so rare cases: when a variable holds *no* object, and when a variable holds *many* objects. The first is usually represented by the null pointer; since its dereferencing causes the notorious null pointer exception, accessing variables that may hold `null` must be explicitly guarded, requiring stereotypical (and hence annoying) coding patterns. The second is usually implemented using a collection, which reifies pointers to many objects as one object; accessing the many objects then requires going through the collection, an indirection that is not unlike the explicit dereferencing of a pointer (and that also requires stereotypical code). It follows that each of the three cases, namely that a variable holds no, one, or many objects, must be handled differently.

By looking at the differences in handling comparable cases in other disciplines, notably natural language and modelling (Section 2), we find that the differences are not necessarily grounded in the nature of the matter. Indeed, as we discuss in some detail in Section 3 (using the programming languages Xen [11], Cω [1], and C$^\sharp$ with LINQ [2] as examples), there have been previous attempts at smoothing out the differences of none, one, and many in object-oriented programming, but as we find, these attempts (which are mostly type-based) have been only partly successful. In Section 4, we therefore advocate our own solution based on the separation of type and multiplicity (previously studied in [19, 20]) which, given the new insights presented here, amounts to introducing the grammatical category *number* to programming. Section 5 then discusses some questions that may arise in the context of our solution. We conclude by reporting on how we have come to depart from our previous view on how number should be handled by programming languages (Section 6), and by outlining research questions for future work (Section 7).

2 The Differences from Different Perspectives

2.1 Linguistic Perspective

In the English language, the difference between *one* and *many* materializes in the grammatical category *number*, whose values are *singular* and *plural*. Number surfaces in the declension of nouns ("person" – "persons"), and also in the conjugation of verbs ("laugh" – "laughs"); in a sentence, the number of the verb representing the predicate must match the number of the noun representing the subject. However, singulars can be combined to form plurals. For instance, "Peter and Mary laugh." is a grammatically well-formed sentence.

Interestingly, in English the word "many" can only be used with nouns that refer to separate items, or individuals, such as "person", "car", etc. These nouns are called *countable*; as their name suggests, only countable nouns can be used with cardinals (e.g., "one person", "two cars"). To make *uncountable* nouns such as "information" or "rain" countable nevertheless (so that we can use them with "one" or "many"), they can be wrapped by countable nouns, as in "piece of information" or "occasion of rain".

Converting singulars to plurals, and uncountables to countables, is complemented by the possibility to cast plurals to singulars, using the means of reification: e.g., "Peter and Mary are a couple." is grammatically well-formed. Reifications of many individuals, like "couple", are usually themselves countable, meaning that there can be many instances of them, which can in turn be reified. It is characteristic for these kinds of reifications that they have properties of their own, i.e., properties that are not (derived from) properties of the individuals they reify. Neither nesting nor the attachment of properties are available for plurals; instead, plurals are unstructured ("flat"), and properties associated with plurals are lifted over the individual entities they represent: e.g., "Peter and Mary are happy." means that both are happy – a happy couple is something else.

There is however one property associated with plurals which is not lifted, and this is the *number* of individuals being denoted. Few would dissent that when determining this number (via counting), each individual is counted only once; e.g., for the sentence "Two people laughed." it is clear that the two are different entities. This appears to give plurals set semantics, and yet counting each individual only once is different from saying that each individual is contained only once, as the latter would require a container and hence a reification of the plural. Also note that countability does not imply order; in fact, there is nothing in a plural that would suggest an ordering – this is either an extrinsic property implied by properties of the individuals (such as their age, position in space, etc.), or an intrinsic property of the container (e.g., a sequence).

2.2 Modelling Perspective

Natural language is highly informal: Using it, misunderstandings are difficult to avoid and resolve. Philosophers have therefore long strived for "the perfect language" (see, e.g., [5]).

The language of mathematics can be considered perfect: it has been used with great success in modelling much of our world. Somewhat surprisingly, however, it was not before the end of the 19^{th} century that mathematicians formalized the notion of *many* as what is today known under the term "set". Interestingly, set is a reification, not a plural (a set of mathematical entities is itself a mathematical entity)[1], and indeed, there are sets of sets; at

[1] but note that mathematical entities are not considered to have identity; sets in particular are extensionally defined, i.e., two sets are the same (or identical) if their elements are the same

the same time, however, there are multitudes that cannot be reified as sets – for instance, what would naively be referred to as "the set of all sets" (which would need to contain itself) is commonly not considered to be a set.

While there are multitudes that cannot be reified as sets, sets are sufficient to represent many other kinds of collections that are in use in mathematics today (including vectors, tuples, and sequences). Sets also capture the notion of *none*, namely as the empty set, and hence can be used to express non-existence (for instance, that an equation has no solution). Regarding *one*, however, mathematics distinguishes between a singleton set and its (sole) member, as evidenced by the different operations available for each (unless of course the member is itself a set).

In software modelling, various languages are in use. Interestingly, in several of these languages, *many* appears as a plural (i.e., the many objects are not reified): for instance, entity-relationship diagrams [4] (and in its wake also the class diagrams of the UML [12]) use multiplicity annotations at the places of relationships (or ends of associations) to distinguish numbers. In fact, it is these modelling languages that make the least difference between *none*, *one*, and *many* – multiplicities are uniformly represented by (intervals of) cardinals[2], and changing a multiplicity never affects more than a single place. This ease of change is intriguing; it is not found in mathematics (where a change from scalar to vector or vice versa means a change of type) and not even in natural language (where, e.g., a change of number propagates from the predicate to its subject or vice versa).

Other modelling languages have embraced this smooth transition from *none* to *many*, and extended it to the expression of constraints and specifications. For instance, Alloy [8] does not distinguish between single objects and sets, but treats the former as singletons. Variables in Alloy are multiplicity-annotated, e.g. with *one* (for precisely one object), or with *set* (for any number of objects). A navigation expression $x.y$, where x and y are variables, always evaluates to a flat set, independently of the multiplicities of x and y. Navigation expressions of OCL [13] are evaluated similarly: even though OCL does not represent single objects as singletons, a navigation expression can mix multiplicities freely, and the resulting collection (if any) is always flat. The difference in type of an expression with multiplicity *one* (the type of the object) and with multiplicity *many* (a collection) that exists in OCL is mitigated by allowing collection operations (such as *forAll*) on single objects also.

2.3 Computing Perspective

In the pre-object-oriented era of programming, when data items did not have identity, the difference between *one* and *many* was basically that between a value and an array of values.[3] However, this distinction was already quite prominent, as an array represents a different data type than its values. Languages like APL eliminated this difference, by interpreting scalars as special arrays, and by lifting functions defined on scalars over arrays; however, the lifted functions may pose size constraints on their arguments, and may use padding to meet them (see, e.g., [15], for a recent account). While this approach is well-suited to remove boilerplate iterations from array computations, it seems less ideal for the handling of many objects (plurals) in object-oriented programs, which are mostly about updating object structures.

[2] Semantically, multiplicities as used in these languages can be interpreted as the cardinalities of sets, but these sets never surface, not even at the instance level, where many objects at a place of a relationship (at an association end) appear as just that: many objects (e.g., in a UML instance diagram).

[3] There were of course dynamic data structures and pointers (including the null pointer), but the notion of identity – a prerequisite of reifying many as one – had yet to emerge.

Table 1 Seven differences to be observed by the object-oriented programmer when implementing relationships to many, rather than one or no, objects using collections (see [19, 20] for details).

Manifestation of Difference in	Multiplicity: *none* or *one*	Multiplicity: *many*

1: type of expression	Account a;
	a = new Account(); //OK
	... a.owner ... //OK
		Set<Account> as;
		as = new Account(); //type error
		... as.owner ... //type error

2: "no object"	a == null? ...	a.isEmpty()? ...
3: subtyping conditions	Account a = new SavAccount();	Set<? extends Account> as = new HashSet<SavAccount>();

4: encapsulation	Account getA() {
	return a;
	}
		Set<Account> getAs() {
		return as.clone();
		}

5: assignment semantics	Account backup = a;
	a = null; // oops!
	a = backup; // phew!
		Set<Account> backups = as;
		as.clear(); // oops!
		as = backups; // what??!

6: call semantics	swap(a, backup); //cannot work	sort(as); //no problem

7: meaning of final	final Account forLife =
	new Account();
	forLife = null; //compile error
		final Set<Account> allForLife =
		Arrays.asSet(forLife);
		allForLife.remove(forLife); //OK

With the advent of object-oriented programming languages, the differences between *one* and *many* became considerably more pronounced. At first glance, this may appear paradoxical, as in object-oriented programming, collections are objects in the same right as all others. Indeed, as mentioned in the introduction, with all objects residing on the heap, a relationship to no object can be represented as the null pointer, a relationship to one object as a pointer to that object, and a relationship to many objects as a pointer to a collection (i.e., a relationship to one object) reifying the multitude. However, the indirection introduced for implementing relationships to many objects does not only dominate the type of the expressions representing the relationship (the dominance of the container over the content noted in [19]), the reification of the relationship (in the form of the collection object) also allows its aliasing. The same kind of aliasing is not possible for a relationship to one object, unless this relationship is also reified (e.g., by using a singleton container type such as `Option<T>`).

In previous work, we have identified seven practically relevant differences between implementing relationships to no or one object on the one hand, and any number of (here referred to as *many*) objects on the other [19, 20]. The differences are summarized in Table 1; for reasons of space, we do not explain them further here, but point the reader to our earlier works. Note that all differences not only have to be fully understood by programmers (which may be a problem especially for beginners, or for people with a strong modelling background); they also present tough maintenance problems, namely when multiplicities change.

3 Type-based Reconciliation of None, One, and Many

There have of course been previous attempts of reconciling representations of no, one, and many objects in object-oriented programming (see, e.g., [19, 20] for overviews). However, except for the indexed instance variables of Smalltalk, which let objects possess an arbitrary number of pointers to other objects [7], the uniform solution seems to be that of going through some kind of container. To highlight the problems this causes, we use the Xen/Cω/C$^\sharp$ with language-integrated querying (LINQ) line of programming languages [1, 2, 11] as a representative here. Note that the scope of these languages is bridging the gap between object-oriented programming and relational and XML-based representations of data, which is much wider, but here, we focus on their contributions to our problem, the reconciliation of *none*, *one*, and *many*.

3.1 The Streams of Xen

In Xen [11], occurrences of no, one, and many objects are unified as streams, i.e., ordered homogeneous collections of objects that can be iterated over. Streams are immutable, i.e., elements of a stream cannot be overwritten. Like other reifications of many objects, streams have identity; however, like plurals, they are always flat, meaning that there are no streams over streams. Hence, from a linguistic perspective, they are hybrids of reified multitudes and plurals (cf. Section 2.1).

Xen introduces three different types of streams: $T!$, a stream of one object of type T, $T?$, a stream of no or one object (also called an *optional*), and T^*, a stream of any number of objects. In addition, `null` is interpreted as the empty stream, resolving Item 2 in Table 1. The subtyping relationship of Xen (and with it assignment compatibility) observes the axioms

$$T! <: T \qquad T <: T? \qquad T? <: T* \tag{1}$$

[11], which suggests that literals and constructor invocations of type T have type $T!$ instead (otherwise, type $T!$ would have no instances). The insertion of the element type T in the hierarchy of stream types seems awkward[4], yet provides for convenient assignment compatibilities, partly resolving the difference noted under Item 1 in Table 1. Because streams are immutable, covariance of streams in their element types is safe, resolving Item 3; at the same time, immutability makes the remaining differences of Table 1 obsolete. However, we note that immutability is not generally a useful property for variables holding (one or many) objects; we will therefore drop it in our own approach (Section 4).

One of the most salient features of streams in Xen is that operations – including member access – defined for a stream's element type are lifted over the stream, resolving the remaining difference of Item 1. For instance, if `as` has type `Account*` and `owner` is a field of `Account`, `as.owner` evaluates to a stream of the owners of the accounts in `as`. At the type level, the flattening of streams that is required when accessing stream-typed members on stream-typed receivers is achieved in Xen by a type equivalence relation $\cong$ defined so that

$$\begin{array}{lll} T!! \cong T! & T!? \cong T? & T!* \cong T* \\ T?! \cong T? & T?? \cong T? & T?* \cong T* \\ T*! \cong T* & T*? \cong T* & T** \cong T* \end{array} \tag{2}$$

Member access on an empty stream (represented by the value `null`; cf. above) results in no member access rather than a null pointer exception; if the member is non-void, it returns an empty stream of the member type. Since optionals are special streams, they share this behaviour, gracefully handling the null case that makes other solutions (including the use of

[4] In particular, it means that stream types cannot be subtypes of *Object*, since otherwise, *Object* $<:$ *Object*$*$ $<:$ *Object*. Also, $T! <: T$ seems strange, since it allows a stream to be its own element (the same would hold for *Object*$*$, if it were a subtype of *Object*; note that neither recursion can be cured by flattening).

`Nullable<T>` in C♯; see below) so awkward. This kind of propagation of `null` is also known from other programming languages; however, the subsumption of member access on no object and on one object under the general member access on streams goes further, and in any case is very much in favour of unifying *none*, *one*, and *many* as we aspire for.

Unary and binary operators are lifted element-wise over streams in Xen [11]. For option streams on value types, particularly on Booleans and numbers, this means that operations can evaluate to "no value" (represented by `null`), effectively requiring introduction of a ternary logic and calculation with `null`. For streams with more elements, lifting of binary operators would either require that both streams have equal length (a dynamic multiplicity check), or mean that all elements of the left operand are combined with all elements of the right operand. In any case, one could argue that if binary operators receive special treatment, binary methods [3] should receive it as well. We will return to this at the end of Section 4.

3.2 The Streams of Cω

While the nature of Xen appears to be more that of a proposal, its successor language, Cω [1], comes with a formally specified type system which, compared to Xen, makes a number of simplifications. In particular, it drops the streams $T!$ containing precisely one element (whose full exploitation would require solving the problem of initializing circular structures; see, e.g., [14]); however, as we will see below, the multiplicity *one* of types $T!$ is actually useful in bypassing the problems with lifting operations on value types sketched above.

Except for the omission of $T!$, Cω adopts the subtyping relationship of Xen, and explicitly adds *Object* as a supertype of all stream types[5] [1]:

$$null <: T \qquad T <: T? \qquad T? <: T* \qquad T* <: Object \tag{3}$$

The type equivalence relation of the streams of Xen (Eq. 2) is retained in Cω, but remains implicit in the type rules, which code the flattening directly. These rules allow iterators to contain themselves; e.g., `int* i = {yield return i;}`, which lets iteration attempts over `i` recurse until stack overflow, is legal Cω code. Note that the same self-containment is not allowed for arrays and generic collections (unless their element type is `object`): e.g., `int[] i = new int[] {i}` is ill-typed (cf. Section 2.2; note how this type-based exclusion of self-containment, which is lifted for streams through flattening, is reminiscent of Russell's type theory).

The lifting of operators over streams, which we reviewed critically in Section 3.1, is not addressed in [1]; the Cω compiler appears to implement it only very selectively.

3.3 The Iterators of C♯, and Language-Integrated Querying (LINQ)

In C♯, the stream types $T*$ and $T?$ of Xen and Cω materialize as iterators (implementing the `IEnumerable<T>` interface) and nullables (instances of `struct Nullable<T> where T : struct`), respectively [2]. Unlike the streams of of Xen and Cω, however, iterators can be nested in C♯; their flattening can be enforced by using the `SelectMany` method (instead of `Select`) in LINQ expressions (see below). By contrast, nesting is illegal for `Nullable<T>`, even though its above definition would permit it: the declaration `Nullable<Nullable<int>> n`, for example, does not compile (another ad hoc type check required by coding multiplicity in the type).

[5] The Cω compiler, which can be obtained from `http://research.microsoft.com/comega`, refuses implicit conversion from `object` to `object*`, hence breaking the circularity noted in footnote 4.

Using the language-integrated querying (LINQ) facilities of C♯, member access on objects can be lifted over collections of objects. E.g., revisiting our running example, the expression `from a in as select a.owner`[6] yields an iterator over owners. If owners have a field `name`, `from a in as select a.owner.name` yields an iterator over names (or a null pointer exception if an account has no owner). On the other hand, the analogous query `from a in as select a.owners.name`, where `owners` is iterable, will not compile – instead, the query must be written as `from a in as from o in a.owners select o.name`, which is the sugared version of `as.SelectMany(a => a.owners).Select(o => o.name)`, and which differs from the *one* account case.

A related issue is the fact that in C♯, `Nullable<T>` is not a subtype of `IEnumerable<T>`, so that nullables cannot be the subject of language-integrated queries, losing much of the uniform access of *none*, *one*, and *many* that was granted by Xen and Cω. Instead, avoiding null pointer exceptions when accessing members on nullable receivers requires explicit tests for nullness and explicit retrieval of the value, exposing the wrapper nature of `Nullable<T>` to the programmer much like collections did before the introduction of LINQ.

3.4 Summary

To summarize, it appears that while Xen started off heading for a far-reaching reconciliation of *none*, *one*, and *many*, casting its respective contributions into the rules of a formal type system required certain concessions such as introducing type equivalences for the purpose of flattening and ad hoc type checks preventing circularity and ill-defined behaviour. Carrying over the contributions of Xen and Cω to C♯ and LINQ seems to have meant further abandonment of the reconciliation, to the extent that it has almost disappeared.

4 Separating Multiplicity from Type

To improve on this situation, we learn from natural language and complement the collections of programming (as reified multitudes) with (non-reified) plurals of countable entities. For this, we extend the grammars of programming languages by introducing *number* or, to use a more customary term, *multiplicity*, as a category that is largely orthogonal to *type*. In so-extended languages, expressions can evaluate to many, rather than just one or no, objects – they can be plural. The use and propagation of multiplicities is subject to specific rules that, like type rules, can be checked statically. For instance, multiplicities are not allowed for types whose instances do not have identity (value types); if multiplicity is required for these types nevertheless, the values need to be wrapped first (observe the analogy with natural language; cf. Section 2.1).

Following the example of Xen and several other (especially modelling) languages, we introduce three static multiplicity annotations, named *one* (for precisely one object), *option* (for no or one object), and *any* (for any number of, including many, objects). In addition, to cater for the absence of a multiplicity annotation, we introduce a pseudo-multiplicity *bare*, which, like *option*, means that there can be no or one object; its use will become clear below. Also, for completeness, we add *none* (for no object) as the multiplicity of `null` (which makes `null` the literal representation of "no object"). Like *bare*, *none* never appears literally in declarations. All non-*bare* expressions can be converted to iterables (or streams) so that they can be used in external and internal iterations.

[6] `as` is a keyword in C♯; however, we use it as an identifier here.

Table 2 Dependence of the multiplicity of a member access expression $r.m$ on the multiplicities of the receiver r and the member m.

MULTIPLICITY OF:	MEMBER m			
RECEIVER r	*one*	*bare*	*option*	*any*
one	*one*	*bare*	*option*	*any*
bare	*one*	*bare*	*option*	*any*
option	*option*	N/A	*option*	*any*
any	*any*	N/A	*any*	*any*

The numbers of objects each multiplicity represents give rise to the subsumption hierarchy

$$none \leq bare \qquad one \leq bare \qquad bare \leq option \qquad option \leq any \tag{4}$$

(note the partial ordering required by consideration of *none* and *one*; also note the absence of a type T; cf. this with Eqs. 1 and 3). This hierarchy, which is largely orthogonal to the type hierarchy, says that expressions having a lesser multiplicity can occur where a greater multiplicity is required (a multiplicity upcast is always safe and hence remains implicit)[7]. In the reverse direction, explicit multiplicity downcasts are required which, except for casts from *option* to *bare*, may fail at runtime (e.g., when an *any* expression which evaluates to two objects is cast to *option*, or when an *option* expression evaluating to no object is cast to *one*). A downcast from *option* to *bare* by itself cannot cause a runtime multiplicity error; however, accessing a member on an *option* receiver cast to *bare* can cause a null pointer exception (which is why *bare* is considered "less" than *option*). Note that such a cast may be required, namely when a *bare* member is to be accessed on an *option* receiver (see below).

Table 2 shows how multiplicities propagate through member access expressions. Note how the table corresponds to the type equivalence relation of Xen (Eq. 2), extending it with rules for *bare*, disallowing the access of *bare* members on receivers having multiplicity *option* or *any*. Together with the rule that all members having value types must be declared *bare* (i.e., without an explicit multiplicity annotation; cf. above), this ensures that expressions having value types always remain *bare*. This restriction of the use of multiplicities avoids the problems of lifting (built-in) binary operations on value types over multitudes of values (cf. Section 3.1); it is further justified in Section 5.2.

To be able to update variables having non-*bare* multiplicity, we provide three assignment operations: `=`, letting the variable on the left-hand side hold (pointers to) the objects the expression on the right-hand side evaluates to; `+=`, letting the variable hold the objects of the expression, plus any objects it held before; and `-=`, letting the variable hold the objects it held before, minus the objects of the expression (the latter two are reserved for *any* variables). Note that since we do not use containers, `=` cannot create an alias for a container; therefore, updating the right-hand side of an assignment after the assignment cannot affect the variable on its left-hand side. Hence, all differences of Table 1 that can be reduced to differences between value semantics and reference semantics dissolve.

With multiplicity and type separated as suggested, all differences of Table 1 that are related to type also disappear. In particular, the following declarations and updates are all well-typed:

[7] Note that this displaces *Object* (with *bare* multiplicity) from the top of the assignment compatibility hierarchy, which is now occupied by *any Object*.

```
         any SavAccount ss = new SavAccount();
         any Account as = ss;
         as += new Account();
         any Owner os = as.owner;
```

Note that the last statement assigns the owners of the accounts held by `as` (a savings account and an ordinary account) to `os`, which must therefore be annotated with `any`; as in Xen and Cω, member access is applied to all objects held by `as`.

The use of multiplicities allows a number of simplifications of programs. Obviously, an iteration like

```
for (Account a : as) a.changeToEuro();
```

can be replaced by

```
as.changeToEuro();
```

if `as` has multiplicity `any`. Also, guards for not null can be dropped if the null case means "no operation" or "evaluate to null":

```
owner = a.owner;
```

completely replaces for

```
if (a != null) owner = a.owner;
else owner = null;
```

A pair of overloaded methods such as

```
void add(Account a) { as.add(a); }
void add(Collection<Account> as) { this.as.addAll(as); }
```

can be replaced by the single method

```
void add(any Acccount as) { this.as += as; }
```

More generally, changing the multiplicity of formal parameters to `any` where this makes sense can save loops in method invocations in which the many objects occur in the argument rather than the receiver position:

```
for (Account a : as) registry.add(a.owner);
```

can be replaced with the more fluent[8]

```
registry.add(as.owner);
```

The inherent asymmetry in method invocations with respect to multiplicity (while a method invoked on many receivers is dispatched to each of the receivers separately, many arguments passed in a single argument position are always passed together) prevents the application of binary methods [3] to pairs of many receivers and many arguments in an APL-like manner. For instance, invocation of `equals(.)` on many receivers passing many objects as the argument will match each receiver individually with all arguments jointly. However, while the problems of applying binary methods to many objects are largely the

[8] `http://martinfowler.com/bliki/FluentInterface.html`

same as for lifting binary operations over streams as discussed in Section 3.1, we note here that it is possible to combine all receiver objects with all argument objects, by extending the technique of double dispatching to multiplicity:

```
void pair(one Object arg) {
  // this and arg make a pair
}
void pair(any Object args) {
  args.pair(this);
}
```

(note that the multiplicity of `this` is `one`).

5 Discussion

5.1 More Multiplicity Annotations

While the static multiplicity annotations *option*, *one*, and *any* appear to be necessary for achieving what we strive for, more can certainly be added. For instance, Alloy offers *some*, meaning one or more [8]. More generally, since static multiplicities represent possible numbers of objects, integer intervals can be used to express them (e.g., $[0,1]$ for *option*, $[1,1]$ for *one*, $[0,\infty)$ for *any*, etc.). Interval-based multiplicities can then be propagated through expressions, but due to the imprecision of static data-flow and control-flow analyses, it is difficult to see how this would work without ubiquitous use of dynamic multiplicity checks and casts. For instance, subtracting objects from a $[1,\infty)$ variable (multiplicity *some*) using `-=` would require a dynamic check that its multiplicity remains greater than zero. Given that numeric multiplicity intervals account for only a small fraction of all desirable invariants regarding the number of objects (others are non-contiguous intervals, predicates such as *odd*, relative multiplicities [9], or arbitrary constraints relating multiplicities of different expressions), the value they add seems somewhat limited (yet may pay off in certain domains).

5.2 Why *bare*?

English and other natural languages suggest that there are things of which one cannot have many, namely those that are uncountable (cf. Section 2.1). However, values do not seem to fall into this category: in computing, we have two Boolean values and countable numbers of characters, integers, dates, etc. So, why do we require value-typed expressions to have multiplicity *bare*, denying programmers the possibility to have many values in non-reified form?

There are various reasons for this. One is that lifting operations defined for single values (scalars) over many values (vectors) in a way that makes sense for these values requires mechanisms like those of APL, which do not seem to carry over well to objects (cf. Sections 2.3 and 3.1). Another is that values typically do not have identity, which appears to be a prerequisite for countability in natural language (cf. Section 2.1). While this argument may appear weak, it gives rise to a third one.

In object-oriented programming, pointers to objects are typically used to express relationships between objects, and annotations like *option* and *any* as put forward here are meant to constrain the multiplicities of these relationships. However, objects typically do not relate to values: a person for instance does not relate to her name, height, or date of birth. Data models like the entity relationship model acknowledge this by storing values in attributes, which are distinct from relationships. Wrapping values in objects may remove the distinction between values and objects technically, but conceptually, it remains intact: it is still unclear why an object should relate to stateless and propertyless data items. Hence, in the context of relationships, the distinction between objects and values is justified; given that natural languages distinguish between countables and uncountables, we should not be too worried about using different means for representing multitudes of objects and multitudes of values.

A last argument is technically motivated: *bare* introduces backward compatibility with code that does not use multiplicity annotations. In fact, in contexts where one object is expected, *bare* can by considered a dynamically checked variant of *one*, and the null pointer exception the result of a failed dynamic multiplicity check.

5.3 Multiplicity and Identity

Identity is a trait of single objects[9] – two objects are identical if they are in fact the same object. It follows that more than one object (a plural in non-reified form) cannot be identical to anything. For analogue reasons, it is meaningless to ask whether an object is identical to no object. Tests for identity (`==`) should therefore be reserved for arguments with multiplicity *one* (with `e == null` being shorthand for checking whether the dynamic multiplicity of expression `e` is *none*).

It is however reasonable to ask whether two expressions with multiplicity other than *one* evaluate to the same objects or, more specifically, to pairs of identical objects. This would be the case if mutually applied subtractive updates (i.e., `x1 -= x2` and `x2 -= x1`) both evaluate to no objects. Equality of two expressions (in the sense of `e1.equals(e2)`) is harder to define, since each expression by itself may evaluate to any number of equal (but not identical) objects.

6 Experience Gained from a Previous Case Study

Jesper Öqvist from Lund University implemented a variant of the multiplicities as described in Section 4 as an extension to the Java 7 programming language, which we evaluated in a case study [20]. The case study provided valuable insights, which in turn led to the writing of this paper.

The main differences between our original design and what has been proposed here are:

1. The original design and its implementation do not consider multiplicity *one*.
2. The original design and its implementation allow *any* annotations to be complemented with a collection type, which is used by the compiler to store the many objects internally.

The first proved to be a handicap for using multiplicities, since it required an explicit downcast of an *option* receiver to *bare* whenever a value-typed member was to be accessed on it (see Table 2), a constellation that occurred rather frequently in our case study. If the receiver would have had multiplicity *one* instead, no such cast would have been necessary. However, statically ensuring *one*-ness shares many of the problems of checking non-null annotations (see, e.g., [14]), which is why we left it to future work.

The second was a tribute to earlier reviewers' comments suggesting that in practice, the programmer would want to have some control over the nature of a multitude. However, from a theoretical stance, a non-reified multitude does not have a nature, which hence cannot be controlled; if anything, it appears that when counting many objects, each counts only once (and, as a corollary, iterating over many objects can yield each object only once). Any other

[9] In object-oriented programming, the term *object* is defined as an item that has identity.

property, such as sequence or order, would presuppose a container, suggesting that reification of the multitude (with all its associated problems) is more appropriate (cf. the discussion at the end of Section 2.1). Interestingly, the reverse dilemma exists in the relational database world: originally (i.e., in the pure relational calculus), many entities could not be reified as one entity, and all grouping and ordering had to be done in queries; only with the advent of object-relational extensions, entities could be wrapped in containers, and stored as attributes [6]. However, to our knowledge, these object-oriented extensions have not been embraced in practice, perhaps because with non-reified multitudes available, their reification appears unnecessary.

Concerning the utility of our multiplicities in practical programming, our case study showed that, while incurring no measurable computing overhead, multiplicities offer many opportunities for rewriting object-oriented code into more fluent style, avoiding the use of control structures (see [20] for examples). With hindsight, this should not have been surprising, given that we extended programming languages with a grammatical form that is quite useful in natural language: *the plural*.

7 Research Opportunities

The author's original motivation to conduct this work was to support his conception of object-relational programming, that is, to enhance object-oriented programming with relationships that are *not first-class*, but instead extend the object-oriented notion of a reference as a unidirectional pointer to no or a single object (which is likewise not first-class) to bidirectional to-many pointers.

7.1 Adding Bidirectionality

In object-oriented programming, a standard way of implementing bidirectional relationships using pointers is to identify "opposite" fields that implement the reverse direction of a pointer, a practice which is also adopted by object-relational mappers. For instance, to implement a bidirectional relationship between objects of classes `A` and `B`, the declaration of a field `B b` in class `A` would be annotated to indicate that field `A a` in class `B` implements its reverse direction. Updating field `a` or `b` would then be complemented under the hood with the necessary update of the opposite field. The to-many pointers advocated in this paper can readily be used to implement one half of non-first-class many-to-many relationships using fields; at the same time, extending multiplicities to other declared entities provides for straightforward integration of bidirectional relationships with arbitrary expressions, without having to resort to containers.

7.2 Connection with Roles

While the implementation of relationships using bidirectional to-many pointers is more or less a technical detail (yet one which raises many interesting questions!), the notion of *interfaces as roles* [17] is more central to the author's notion of object-relational programming. In the relational realm, roles are defined as the places of relationships, and every binary relationship comes with two roles, a role and its counter-role. By a suitable definition of role-playing [16], an object playing one role is necessarily related to one or more objects playing the counter-role. The behaviour associated with a role (as expressed by the methods offered by the corresponding interface) is then the behaviour available for the collaboration relying on the defining relationship [18]. Conceptually, an upcast of an object to a role it plays (an interface it implements) uniquely identifies the objects related to it by playing the counter-role; technically, this can be realized by letting each role define access and update operations for an implicit bidirectional to-many pointer to the objects playing the counter-role. Hence, binary relationships and the collaborations between the participating objects can be specified by pairing interfaces (a role with its counter-role), and by letting classes declare to implement these interfaces (meaning that the classes' instances can play the associated roles). Conceptually simple as this may seem, it raises many research questions, such as specializing relationships via subtyping roles and the relationship of roles to traits (see [21] for a proposal of stateful traits that would lend itself to implementing roles as envisioned here).

7.3 Adding Swarm Behaviour

Section 4 ended with a brief example of overloading methods with different argument multiplicities. The example suggests that, in analogy to type-based dispatch, static method dispatching always selects the method with the most specific argument multiplicities. However, differing from type-based dispatch, we do no dispatch on the receiver's multiplicity (the inherent asymmetry noted above): invoking a method on many receivers is implicitly resolved by invoking it on each.

The notion of *swarm behaviour* put forward in [10] suggests that there is a different interpretation of invoking a method on many receivers: that the receivers are expected to act as a swarm. Multiplicity-based swarm behaviour could be implemented by static methods that have access to a pseudo-variable `these` whose static multiplicity is *any* (or *many*) and which evaluates to the objects the method was invoked on.[10] In fact, one could assume a default implementation of swarm behaviour, which dispatches a method invocation on many objects to the individual objects and which can be overridden when needed (just like the implicit default constructor can be overridden). Using multiplicities, no container would be required to reify the swarm in swarm methods (as in [10]); `these` is simply the plural of `this`.

Pushing the idea of dispatching on multiplicities further, one could even provide methods for multiplicity *none* (i.e., no receiver object), replacing for the NULL OBJECT pattern [22]. However, this would require *dynamic dispatching* on the receiver multiplicity.

8 Conclusion

The addition of multiplicity to programming languages as a new grammatical category appears similarly fundamental as the introduction of types. It may prove a better foundation for ridding programming of null pointer exceptions than current type-based approaches (using containers), and can generalize non-null annotations to handling more multiplicities than just *bare* and *one*. At the same time, the dropping of containers promises to solve a number of anomalies that force code for handling many objects to look very different from code for handling one or no object. However, as the author himself experienced during the writing of this paper, there exists a mental obstacle to adopting multiplicities as proposed here, and this is rooted in the entire dismissal of reification: sets in particular have become so fundamental in our thinking that it is difficult to even talk about occurrences of many objects without implicitly or explicitly reifying them into a set. It will be interesting to see whether programming without collections comes more natural, at least in places where the container is irrelevant.

[10] Here it would pay that Java allows static member access on object expressions.

Acknowledgements. The writing of this paper was encouraged by discussions led during Dagstuhl Seminar 14211; the Summit on Advances in Programming Languages was brought to the author's attention at Dagstuhl Seminar 14412.

References

1 Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In Andrew P. Black, editor, *ECOOP 2005 – Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer, 2005.

2 Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C$^\sharp$. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 479–498. ACM, 2007.

3 Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *TAPOS*, 1(3):221–242, 1995.

4 Peter P. Chen. The entity-relationship model – Toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

5 Umberto Eco. *The Search for the Perfect Language*. Wiley-Blackwell, 1997.

6 Andrew Eisenberg and Jim Melton. SQL: 1999, formerly known as SQL 3. *SIGMOD Record*, 28(1):131–138, 1999.

7 Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

8 Daniel Jackson. *Software Abstractions – Logic, Language, and Analysis*. MIT Press, 2006.

9 Roman Knöll, Vaidas Gasiunas, and Mira Mezini. Naturalistic types. In Robert Hirschfeld and Eelco Visser, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2011, part of SPLASH'11, Portland, OR, USA, October 22-27, 2011*, pages 33–48. ACM, 2011.

10 Adrian Kuhn, David Erni, and Marcus Denker. Empowering collections with swarm behavior. *CoRR*, abs/1007.0159, 2010.

11 Erik Meijer, Wolfram Schulte, and Gavin M. Bierman. Unifying tables, objects and documents. In *Proceedings of Declarative Programming in the Context of OO Languages (DP-COOL 2003)*, August 2003.

12 OMG (Object Management Group). Unified Modeling Language 2.4.1. Specification, OMG (Object Management Group), 2011. `http://www.omg.org/spec/UML/2.4.1/`.

13 OMG (Object Management Group). Object Constraint Language 2.4. Specification, OMG (Object Management Group), 1 2012. `http://www.omg.org/spec/OCL/`.

14 Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix – safe modular circular initialisation with placeholders and placeholder types. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming – 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 205–229. Springer, 2013.

15 Justin Slepak, Olin Shivers, and Panagiotis Manolios. An array-oriented language with static rank polymorphism. In Zhong Shao, editor, *Programming Languages and Systems – 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 27–46. Springer, 2014.

16 Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.*, 35(1):83–106, 2000.

17 Friedrich Steimann. Role = interface – A merger of concepts. *Journal of Object-Oriented Programming*, 14:23–32, 2001.

18 Friedrich Steimann. Role + counter role = relationship + collaboration. In Stephen Nelson and Stephanie Balzer, editors, *RAOOL'08: Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages, co-located with OOPSLA 2008, Nashville, TN, USA*, 2008.

19 Friedrich Steimann. Content over container: Object-oriented programming with multiplicities. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH'13, Indianapolis, IN, USA, October 26-31, 2013*, pages 173–186. ACM, 2013.

20 Friedrich Steimann, Jesper Öqvist, and Görel Hedin. Multitudes of objects: First implementation and case study for Java. *Journal of Object Technology*, 13(5):1: 1–33, 2014.

21 Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22–27, 2011*, pages 959–972. ACM, 2011.

22 Bobby Woolf. Null object. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 5–18. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

A Complement to Blame

Philip Wadler

University of Edinburgh, UK
`wadler@inf.ed.ac.uk`

Abstract

Contracts, gradual typing, and hybrid typing all permit less-precisely typed and more-precisely typed code to interact. Blame calculus encompasses these, and guarantees blame safety: blame for type errors always lays with less-precisely typed code. This paper serves as a complement to the literature on blame calculus: it elaborates on motivation, comments on the reception of the work, critiques some work for not properly attending to blame, and looks forward to applications. No knowledge of contracts, gradual typing, hybrid typing, or blame calculus is assumed.

1998 ACM Subject Classification D.3.2 Multiparadigm Languages

Keywords and phrases contracts, gradual typing, hybrid typing, blame calculus

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.309

1 Introduction

Findler and Felleisen [5] introduced two seminal ideas: higher-order *contracts* to dynamically monitor adherence to a type discipline, and *blame* to indicate which of two parties is at fault if the contract is violated. Siek and Taha [17] introduced gradual types to integrate untyped and typed code, while Flanagan [6] introduced hybrid types to integrate simple types with refinement types. Both used similar source languages, similar target languages with explicit casts, and similar translations between source and target. Both depended upon contracts, but neither used blame.

Motivated by the similarities between gradual and hybrid types, Wadler and Findler [25] introduced blame calculus, which unifies the two by encompassing untyped, simply-typed, and refinement-typed code. As the name indicates, it also restores blame, which enables a proof of *blame safety*: blame for type errors always lays with less-precisely typed code – "well-typed programs can't be blamed". Blame safety may seem obvious (of course the blame lies with the less-precisely typed code!), so why bother with a proof? Because programming language researchers have learned that formal statement and proof help guide sound language design.

Findler and Felleisen [5] introduce blame, but give no general results on when blame can or cannot arise. Tobin-Hochstadt and Felleisen [22] and Matthews and Findler [14] both offer proofs that when integrating untyped and typed code, blame must always lay with the untyped code; each requires a sophisticated proof based on operational equivalence. Siek and Taha [17] and Flanagan [6] say nothing about blame safety, since they ignore blame. Wadler and Findler [25] generalise blame safety to levels of precision (untyped, simply-typed, and refinement typed), and introduce a new proof technique that avoids operational equivalence, instead using the standard progress and preservation approach to type soundness of Wright and Felleisen [27].

Arguably, gradual and hybrid types can have a stronger foundation if based on blame calculus (or some other formulation of blame). Subsequent work on gradual typing does exactly this; Siek is a coauthor of several of the blame papers cited below. However, further work on hybrid types continues to ignore blame.

Section 2 summarises blame calculus. Section 3 critiques two studies that relate contracts to hybrid types. Section 4 makes some observations about notation. Section 5 discusses related work, notably in industry. Section 6 speculates on who might benefit from blame.

2 Blame calculus in a nutshell

Blame calculus is developed in a series of papers. Wadler and Findler [25] introduces the calculus, which is based on casts, and applies it to gradual and refinement types. Siek and Wadler [18] observes that every cast can be decomposed into an upcast and a downcast, and applies this to derive a new space-efficient implementation. Ahmed et al. [1] extends blame calculus to polymorphic types; remarkably, an untyped program cast to a polymorphic type is guaranteed to satisfy relational parametricity – "theorems for free" [15, 23]. Siek et al. [16] relates the blame calculus to two new calculi, one inspired by the coercion calculus of Henglein [11] and one inspired by the space-efficient calculus of Herman et al. [12], and shows that translations between the three are fully abstract.

From untyped to typed. The blame calculus provides a framework for integrating less-precisely and more-precisely typed programs. One scenario is that we begin with a program in an untyped language that we wish to convert to a typed language.

Here is an untyped program.

$$\begin{array}{l}\lceil \texttt{let}\ x = 2 \\ \ \texttt{let}\ f = \lambda y.\, y + 1\ \texttt{in} \\ \ \texttt{let}\ h = \lambda g.\, g\ (g\ x)\ \texttt{in} \\ \ h\ f \rceil\end{array}$$

The term evaluates to $\lceil 4 \rceil : \star$.

By default, our programming language is typed, so we indicate untyped code by surrounding it with ceiling brackets, $\lceil \cdot \rceil$. Untyped code is typed code where every term has the dynamic type, $\star$. Dana Scott [19] and Robert Harper [9] summarise this characterisation with the slogan "untyped is uni-typed".

As a matter of software engineering, when we add types to our code we may not want to do so all at once. Of course, it is trivial to rewrite a four-line program. However, the technique described here is intended to apply also when each one-line definition is replaced by a thousand-line module. We manage the transition between untyped and typed code by a new construct with an old name, "cast".

Here is our program, mostly typed, with one untyped component cast to a type.

$$\begin{array}{l}\texttt{let}\ x = 2 \\ \texttt{let}\ f = \lceil \lambda y.\, y + 1 \rceil : \star \Longrightarrow^{p} \texttt{Int} \to \texttt{Int} \\ \texttt{let}\ h = \lambda g{:}\texttt{Int} \to \texttt{Int}.\, g\ (g\ x) \\ \texttt{in}\ h\ f\end{array}$$

The term evaluates to $4 : \texttt{Int}$.

In general, a cast has the form $M : A \Longrightarrow^{p} B$, where M is a term, A and B are types, and p is a blame label. Term M has type A, while the type of the cast as a whole is B. The blame label p is used to indicate where blame lies in event of a failure. For instance, if we replace

$\lceil \lambda y.\, y > 0 \rceil$ by $\lceil \lambda y.\, y + 1 \rceil$

then the term evaluates to $\texttt{blame}\ p$, indicating that the party at fault is the term contained in the cast labeled p. Cast failures may not be immediate: in this case, the cast fails only when the function contained in the cast is applied and returns a boolean rather than an integer.

Here is our program, mostly untyped, but with one typed component cast to untyped.

$\texttt{let}\ x = \lceil 2 \rceil$
$\texttt{let}\ f = (\lambda y{:}\texttt{Int}.\, y + 1) : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \star$
$\texttt{let}\ h = \lceil \lambda g.\, g\,(g\,x) \rceil$
$\texttt{in}\ \lceil h\, f \rceil$

Variables bound to untyped code or appearing in an untyped region must have type $\star$. The term evaluates to $\lceil 4 \rceil : \star$.

Blame in a cast may lie either with the *term contained* in the cast or the *context containing* the cast. For instance, if we replace

$\lceil \lambda g.\, g\, 2 \rceil$ by $\lceil \lambda g.\, g\, \texttt{true} \rceil$

then the term evaluates to $\texttt{blame}\ \overline{p}$, indicating that the party at fault is the context containing the cast labeled p. The complement of blame label p is written $\overline{p}$. Complementation is involutive, $\overline{\overline{p}} = p$.

When blame lies with the *term contained* in the cast (that is, a term with a cast labeled p evaluates to $\texttt{blame}\ p$), we say we have *positive blame*, and when blame lies with the *context containing* the cast (that is, a term with a cast labeled p evaluates to $\texttt{blame}\ \overline{p}$), we say we have *negative blame*.

In our examples, positive blame arose when casting from untyped to typed, and negative blame arose when casting from typed to untyped. In both cases blame lies on the less precisely typed side of the cast. This is no coincidence, as we will see.

From simply typed to refinement typed. Refinement types are base types that additionally satisfy a predicate. For instance, $(x : \texttt{Int})\{x \geq 0\}$ is the type of integers that are greater than or equal to zero, which we abbreviate $\texttt{Nat}$. In general, refinement types have the form $(x : \iota)\{M\}$ where ι is a base type, and M is a term of type $\texttt{Bool}$.

Just as casts take untyped code to typed, they also can take simply-typed code to refinement-typed. For example

$\texttt{let}\ x = 2 : \texttt{Int} \Longrightarrow^{q} \texttt{Nat}$
$\texttt{let}\ f = (\lambda y{:}\texttt{Int}.\, y + 1) : \texttt{Int} \to \texttt{Int} \Longrightarrow^{p} \texttt{Nat} \to \texttt{Nat}$
$\texttt{let}\ h = \lambda g{:}\texttt{Nat} \to \texttt{Nat}.\, g\,(g\,x)$
$\texttt{in}\ h\, f$

returns $4 : \texttt{Nat}$.

As before, casts can fail. For instance, if we replace $\lambda y{:}\texttt{Int}.\, y + 1$ by $\lambda y{:}\texttt{Int}.\, y - 2$ then the term evaluates to $\texttt{blame}\ p$, indicating that the party at fault is the term contained in the cast labeled p. In this case, the cast fails when the function contained in the cast returns -2, which fails the dynamic test applied when casting $\texttt{Int}$ to $\texttt{Nat}$.

Analogous to previously, positive blame arises when casting from simply-typed to refinement-typed, and negative blame arises when casting from refinement-typed to simply-typed. As before, in both cases blame lies on the less precisely typed side of the cast.

How blame works. The blame calculus consists of typed lambda calculus plus casts $M : A \Longrightarrow^p B$ and failure `blame` p. Let L, M, N range over terms, V, W range over values, A, B range over types, and p, q range over blame labels.

We used the construct $\lceil\cdot\rceil$ to indicate untyped code, but it is easily defined in terms of the other constructs. For instance, $\lceil \lambda g.\, g\, 2 \rceil$ translates to

$$(\lambda g{:}\star\,.\,(g : \star \Longrightarrow^{p_1} \star \to \star)\,(2 : \mathtt{Int} \Longrightarrow^{p_2} \star)) : \star \to \star \Longrightarrow^{p_3} \star$$

where p_1, p_2, p_3 are fresh blame labels introduced by the translation.

The most important reduction rule of the blame calculus is the (Wrap) rule, which descends directly from the handling of higher-order contracts in Findler and Felleisen [5].

$$(V : A \to B \Longrightarrow^p A' \to B')\, W \quad \longrightarrow \quad (V\,(W : A' \Longrightarrow^{\overline{p}} A)) : B \Longrightarrow^p B' \qquad (\textsc{Wrap})$$

A cast of a function applied to a value reduces to a term that casts on the domain, applies the function, and casts on the range. To allocate blame correctly, the blame label on the cast of the domain is complemented, corresponding to the fact that the argument W is supplied by the context. In the cast for the domain the types are swapped (A' to A) and the blame label is complemented ($\overline{p}$), while in the cast for the range the order is retained (B to B') and the blame label is unchanged (p), corresponding to the fact that function types are contravariant in the domain and covariant in the range.

It is straightforward to establish type safety, using preservation and progress as usual.

▶ **Theorem 1** (Type safety).
1. *If* $\vdash M : A$ *and* $M \longrightarrow N$ *then* $\vdash N : A$.
2. *If* $\vdash M : A$ *then either*
   a. *there exists a term* N *such that* $M \longrightarrow N$*, or*
   b. *there exists a value* V *such that* $M = V$*, or*
   c. *there exists a label* p *such that* $M =$ `blame` p.

Here type safety is a weak result, as it cannot rule out the possibility that a term reduces to the form `blame` p, which is akin to a type error. How to guarantee blame cannot arise in certain circumstances is the subject of the next section.

Blame safety. The statement and proof of blame safety is relatively straightforward. However, it requires four different subtyping relations: $<:$, $<:^+$, $<:^-$, and $<:_n$, called ordinary, positive, negative, and naive subtyping respectively.

Why do we need *four* different subtyping relations? Each has a different purpose. Relation $A <: B$ characterizes when a cast from A to B *never* yields blame; relations $A <:^+ B$ and $A <:^- B$ characterize when a cast from A to B cannot yield *positive* or *negative* blame, respectively; and relation $A <:_n B$ characterizes when type A is more *precise* than type B.

We omit the details of the definitions here; see Wadler and Findler [25]. As is standard, $<:$, $<:^+$, and $<:^-$ are contravariant in the domain and covariant in the range of function types, while $<:_n$ is (perhaps surprisingly) covariant in both the domain and the range.

These four relations are closely connected: ordinary subtyping decomposes into positive and negative subtyping, which can be reassembled to yield naive subtyping.

▶ **Lemma 2** (Tangram).
1. $A <: B$ *iff* $A <:^+ B$ *and* $A <:^- B$.
2. $A <:_n B$ *iff* $A <:^+ B$ *and* $B <:^- A$.

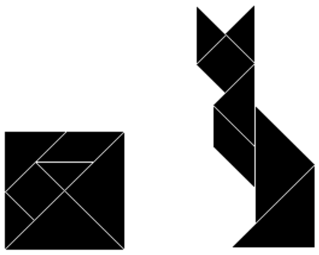


Figure 1 Tangram as metaphor.

I named the result by analogy to the well-known tangram puzzle, where a square decomposes into parts that recombine into a different shape (Figure 1).

We can now characterise under what circumstances reduction of a term can never blame p.

▶ **Definition 3.** Term M is *safe* for label p, written M safe p, if every cast in M of the form $A \Longrightarrow^p B$ satisfies $A <:^+ B$, and every cast of the form $A \Longrightarrow^q B$ where $q = \overline{p}$ satisfies $A <:^- B$.

If M safe p then $M \not\longrightarrow^* \texttt{blame}\ p$. This is established via a variant of preservation and progress.

▶ **Theorem 4** (Blame Safety)**.**
1. *If M safe p and $M \longrightarrow N$ then N safe p.*
2. *If M safe p then $M \not\longrightarrow \texttt{blame}\ p$.*

The proof is by straightfoward case analysis of each of the reduction rules, and involves checking that the reduction rules preserve the desired invariants. For instance, for (Wrap),

$$(V : A \to B \Longrightarrow^p A' \to B')\, W \longrightarrow (V\ (W : A' \Longrightarrow^{\overline{p}} A)) : B \Longrightarrow^p B'$$

we must check that if $A \to B <:^+ A' \to B'$ then $A' <:^- A$ and $B <:^+ B'$; and similarly with $<:^+$ and $<:^-$ interchanged. In fact, this easily follows from the definitions of $<:^+$ and $<:^-$. While the basic proof of blame safety is mine, the formulation in terms of preservation and progress is due to Robby Findler.

Let $\mathcal{C}$ range over contexts, terms containing a hole. Assume p and $\overline{p}$ do not appear in context $\mathcal{C}$ or term M. Combining the Tangram Lemma with Blame Safety we can characterise casts that never fail, and show that when casting between a less precise type and a more precise type that the more-precisely-typed side of the cast is never to blame.

▶ **Corollary 5** (Blame Theorem)**.**
1. *If $A <: B$ then $\mathcal{C}[M : A \Longrightarrow^p B] \not\longrightarrow^* \texttt{blame}\ p$ and $\mathcal{C}[M : A \Longrightarrow^p B] \not\longrightarrow^* \texttt{blame}\ \overline{p}$.*
2. *If $A <:_n B$ then $\mathcal{C}[M : A \Longrightarrow^p B] \not\longrightarrow^* \texttt{blame}\ p$.*
3. *If $B <:_n A$ then $\mathcal{C}[M : A \Longrightarrow^p B] \not\longrightarrow^* \texttt{blame}\ \overline{p}$.*

The last two points are summarised by the motto “well-typed programs can’t be blamed”.

For instance, in the earlier examples, the untyped term in a typed context can never blame the context because $\star <:^- \texttt{Int} \to \texttt{Int}$, while the typed term in an untyped context can never blame the term because $\texttt{Int} \to \texttt{Int} <:^+ \star$, and both of these follow from (and indeed are equivalent to) $\texttt{Int} \to \texttt{Int} <:_n \star$.

The result for ordinary subtyping is not nearly so useful as the results for naive subtyping. Cast between ordinary subtypes, such as $\texttt{Int} \to \texttt{Nat} <: \texttt{Nat} \to \texttt{Int}$, rarely arise, while casts between naive subtypes, such as $\texttt{Int} \to \texttt{Int} <:_n \texttt{Nat} \to \texttt{Nat}$ arise often. Despite this, the literature often presents the result for ordinary subtyping, while the results for naive subtyping are ignored.

An encouraging discovery and a disappointing reception. I was amazed and delighted to discover the Tangram Lemma. When defining the four relations, I had in mind that $<:$ should imply $<:^+$ and $<:^-$, and that $<:_n$ should imply $<:^+$ and the converse of $<:^-$. But it was a complete surprise to find that the implications I had in mind could be replaced by equivalences.

The Tangram Lemma convinced me that there must be mathematical substance to the Blame Calculus. I felt less like I was inventing, and more like I was discovering. Interestingly, I know of no analogues of the Tangram Lemma in other areas of mathematics, where two relations refactor into two other relations.

So far as I can tell, I am alone in my enthusiasm for the Tangram Lemma and the Blame Theorem. Others don't find them as exciting as I do. Perhaps one reason why is because the Tangram Lemma is stated in terms of four subtyping relations, which may be three too many to easily absorb.

I'm particularly disturbed that researchers tend to formulate systems without complement, which renders the Blame Theorem impossible to formulate or prove, and which discards useful debugging information, as described in the next section.

3 To complement or not to complement?

Gronski and Flanagan [8] relate the contracts of Findler and Felleisen [5] to the hybrid types of Flanagan [6]; and Greenberg et al. [7] build on these results.

Interestingly, the cited works relate contracts where blame is complemented to casts where blame is not complemented. This section begins by presenting a variation of their results, using our casts with complement; and then compares it to their version, where casts are not complemented.

I've translated the notation of the cited papers to the notation used here. Among other things, "casts with complement" here corresponds to "casts with two blame labels" in those papers; and "casts without complement" corresponds to "casts with one blame label"; see "Two against one" in Section 4.

We write contracts of Findler and Felleisen [5] as $M \mathbin{@^p} A$, indicating term M is subject to contract A with blame label p. The equivalent of the (Wrap) rule is

$$(V \mathbin{@^p} A \to B)\; W \quad \longrightarrow \quad (V\;(W \mathbin{@^{\bar{p}}} A)) \mathbin{@^p} B.$$

Observe the blame label p is complemented for the domain but not the range, just as with (Wrap) in the blame calculus.

Gronski and Flanagan [8] give a map ϕ from contracts to casts. It is defined using a supplementary function that removes refinements to yield ordinary types.

$$\begin{aligned} \lfloor (x : \iota)\{M\} \rfloor &= \iota \\ \lfloor A \to B \rfloor &= \lfloor A \rfloor \to \lfloor B \rfloor \end{aligned}$$

The key equation of the map from contracts to casts is

$$\phi(M \mathbin{@^p} A) \quad = \quad \phi(M) : \lfloor A \rfloor \Longrightarrow^p \phi(A) \Longrightarrow^p \lfloor A \rfloor$$

(Here $M : A \Longrightarrow^p B \Longrightarrow^q C$ abbreviates $(M : A \Longrightarrow^p B) : B \Longrightarrow^q C$.) The other equations are homomorphic; and $\phi(A)$ applies ϕ to each term M in a refined type in A. The explanation of this translation is straightfoward. The type system for contracts requires that a term M with contract A must have type $\lfloor A \rfloor$, and after applying the contract it must have the same type. Hence the two casts ensure the terms have the proper type, while the application of $\phi(A)$ in the middle enforces the conditions imposed by the contract A.

However, the actual mapping given in Gronski and Flanagan [8] and Greenberg et al. [7] use a variant of casts that does not complement the blame label, which we indicate by adding a subscript `no`. The system is essentially identical to ours, save that the equivalent of (Wrap) does not complement blame labels in the domain.

$$(V : A \to B \Longrightarrow^{p}_{\mathsf{no}} A' \to B')\, W \quad \longrightarrow \quad (V\,(W : A' \Longrightarrow^{p}_{\mathsf{no}} A)) : B \Longrightarrow^{p}_{\mathsf{no}} B'$$

The key equation of the translation is now

$$\phi(M \mathbin{@^{p}} A) \quad = \quad \phi(M) : \lfloor A \rfloor \Longrightarrow^{p}_{\mathsf{no}} \phi(A) \Longrightarrow^{\overline{p}}_{\mathsf{no}} \lfloor A \rfloor$$

The only difference in the equation is that the second cast, which was previously labelled p, is now labelled $\overline{p}$.

The reason why the version without complement works can be easily understood by an application of the Blame Theorem. It is easy to see that a refinement type A is more precise than the equivalent simple type $\lfloor A \rfloor$. Hence, in $\phi(M) : \lfloor A \rfloor \Longrightarrow^{p} \phi(A) \Longrightarrow^{p} \lfloor A \rfloor$ the leftmost cast can only blame p while the rightmost cast can only blame $\overline{p}$. This is why using casts without complement yields the same answer, by simply introducing the complement instead as part of the translation.

Gronski and Flanagan [8] observe that contracts with complement are potentially more expressive than casts without complement. The mapping ϕ shows that expressiveness is preserved, and they take this as a justification for limiting themselves to casts without complement. Note that, ironically, we require the Blame Theorem, which depends upon casts with complement, in order to show why the casts without complement are adequate for translating contracts with complement.

Greenberg et al. [7] introduce the name "latent" for the system with contracts and "manifest" for the system with casts. They note that either system can be formulated with or without complement, arguing as follows. [I have updated their notation to match this paper.]

> This difference is not fundamental, but rather a question of the pragramatics of assigning responsibility: both latent and manifest systems can be given more or less rich algebras of blame. Informally, a function contract check $f \mathbin{@^{p}} A \to B$ divides responsibility for f's behaviour between its body and its environment: the programming is saying "If f is ever applied to an argument that doesn't pass A, I refuse responsibility (`blame` $\overline{p}$), whereas if f's result for good arguments doesn't satisfy B, I accept responsibity (`blame` p)." In a manifest system, the programmer who writes $f : A \to B \Longrightarrow^{p}_{\mathsf{no}} A' \to B'$ is saying "Although all I know statically about f is that its results satisfy B when it is applied to arguments satisfying A, I assert that it's OK to use it on arguments satisfying A' (because I believe that A' implies A) and that its results will always satisfy B' (because I believe that B implies B')." In the latter case, the programmer is taking responsibility for *both* assertions (so `blame` p makes sense in both cases), while the additional responsibility for checking that arguments satisfy A' will be discharged elsewhere (by another cast with a different label).

However, this throws away useful information to no benefit. If one uses complement with manifest casts, then `blame` p tells us the claim that A' implies A is incorrect, while `blame` $\overline{p}$ tells us the claim that B implies B' is incorrect, a huge aid to debugging.

Whether one is interested in systems with complement depends on the the results one finds of interest. If one has a sophisticated notion of subtyping, where $(x : A)\{M\} <: (x : A)\{N\}$ if M implies N for all x, then $A \to B <: A' \to B'$ holds exactly when A' implies A and B implies B'. The system without complement is adequate if one's interest is only in clause 1 of the Blame Theorem (Corollary 5), showing that a cast $A \Longrightarrow^p B$ never fails if $A <: B$, while the system with complement is essential if one's interest is in clause 2 or clause 3, showing that a cast $A \Longrightarrow^p B$ never blames p if $A <:_n B$ and never blames $\overline{p}$ if $B <:_n A$.

I argue it is better to formulate a system with complement. Complement yields useful information for debugging; and without complement there is no way to formulate Blame Safety or the Blame Theorem, and hence to assure that blame never lies with the more-precisely-typed side of a cast. If for some reason one wishes to ignore the distinction between p and $\overline{p}$ (for instance, because we want to assign responsibility to the same programmer in either case), that is easy to do. But one should make the distinction and then ignore it; if one fails to make the distinction from the start, then Blame Safety and the Blame Theorem become impossible to prove. Above I've sketched a reformulation of Gronski and Flanagan [8] and Greenberg et al. [7] in a system with complement, and it would be nice to see this worked out in detail.

4 Notation

As observed by Whitehead [26, Chapter V], notation can crucially effect our thought. I expend much effort on choice of notation, yet often find I don't get it right the first time. This section reviews changes to notation for blame during a decade.

Two against one. Findler and Felleisen [5] indicate blame using a pair of labels, p and n, standing for positive and negative blame; their version of (Wrap) swaps p, n so the contract for the domain is labeled n, p. Wadler and Findler [25] indicate blame with a single label p; their (Wrap) complements p so the contract for the domain is labeled $\overline{p}$. The change is at best a small step forward, but nonetheless I am proud of it. Fewer variables is better, and the abstract notion of complement may open the way to further generalisation.

Casts. Siek and Taha [17] write $\langle B \rangle\ M$ to cast term M to type B. The type A of M is is easily inferred, and is omitted to save space. But a consequence is that their equivalent of (Wrap) must mix type inference and reduction. They use no blame labels.

Flanagan [6] writes $\langle A \rhd B \rangle^p\ M$ to cast term M from type A to type B. For me, the idea that one might explicitly list both source and target types of the cast was an eye opener. While too prolix for practice, it clarifies the core calculus by highlighting the contravariant and covariant treatment of the domain and range in (Wrap). As explained in Section 3, Flanagan's interpretation of blame label p differs from ours, in that it lacks complement.

Wadler and Findler [25] write $\langle B \Leftarrow A \rangle^p\ M$ to cast term M from type A to type B, a variation of Flanagan [6] that allows reading uniformly from right to left. Blame label p now supports complement.

Ahmed et al. [1] write $M : A \Longrightarrow^p B$ to cast term M from type A to type B. I remember the day we introduced the new notation: Amal Ahmed, Robby Findler, and myself were hosted by Jacob Matthews at Google's Chicago office, where we had an unused floor to ourselves. I didn't think direction of reading was a big deal, but I had underestimated the extent to which my reading of prose influenced my reading of formulas. When we reversed right-to-left to left-to-right it was as if a weight lifted from my brow!

The new notation suggests a natural abbreviation, replacing $(M : A \Longrightarrow^p B) : B \Longrightarrow^q C$ by $M : A \Longrightarrow^p B \Longrightarrow^q C$. Again, the abbreviation helped more than I might have imagined, rendering rules and calculations far easier to read.

Refinement types and dependent function types. I also want to draw attention to an improvement in notation that had nothing to do with me.

In standard notation (found in Flanagan [6] and many others), refinement types are written $\{x : A \mid M[x]\}$, where x is a variable of type A, and $M[x]$ is a boolean-valued term that may refer to x; and dependent function types are written $x : A \to B[x]$, where x is a variable of type A, and $B[x]$ is a type that may refer to x. For instance,

$$x : \{x : \texttt{Int} \mid x \geq 0\} \to \{y : \texttt{Int} \mid y \geq x\}$$

is the dependent function type that accepts a natural, and returns an integer greater than its argument. A drawback of this notation is duplication of x, bound once in the refinement type and again in the dependent function type.

An improved notation is suggested by Swamy et al. [20] for F*, where refinement types are written $(x : A)\{M[x]\}$, and dependent function types are written as before. For instance,

$$x : (x : \texttt{Int})\{x \geq 0\} \to (y : \texttt{Int})\{y \geq x\}$$

is the dependent function type that accepts a natural, and returns an integer greater than its argument. This notation lends itself naturally to the abbreviation

$$(x : \texttt{Int})\{x \geq 0\} \to (y : \texttt{Int})\{y \geq x\}$$

which avoids the annoying duplication of x.

Interplay with implementation. The choice between target-alone or source-and-target casts is not only about concision versus clarity, but also relates to implementation. Simply-typed lambda calculus is typically written with types in lambda abstractions, $\lambda x{:}A.\, N$, which ensures each closed term has a unique type. But types play no part in the reduction rules, and so can be erased before evaluation. As noted above, target-alone casts must mix type inference and reduction in (Wrap), so erasure is not an option. However, source-and-target casts need no type inference in (Wrap), and so support erasure. To be precise, types may be erased in lambda abstractions (and related constructs), but source and target types cannot be erased from casts.

Hence, a source language should use target-alone casts for concision, and translate to an intermediate language with source-and-target casts to aid implementation. So far as I know, this observation has not been written down previously. I think one reason why is that authors are reluctant to make a claim – even if the point is as obvious as the one made here – without either writing out the relevant bit of the theory or building an actual implementation, both of which can be costly.

5 Practice and theory

Nothing is so practical as a good theory.

C#. C# introduces a type `dynamic`, which plays a role almost identical to our $\star$ [3]. As with gradual typing, a translation introduces casts to and from `dynamic` where needed. However, the introduced casts are first-order; higher-order casts must be written by hand, implementing the equivalent of (Wrap).

TypeScript. TypeScript allows the programmer to specify in an `interface` declaration types for a JavaScript module or library supplied by another party [10]. The DefinitelyTyped repository contains over 150 such declarations for a variety of popular JavaScript libraries [28]. Interestingly, TypeScript is not concerned with type soundness, which it does not provide, but instead with exploiting types to provide effective prompts in Visual Studio, for instance to populate a pulldown menu with methods that might be invoked at a given point.

TS*. TS* is a variant of TypeScript which ensures type safety by assigning run-time type information (RTTI) to some JavaScript objects [21]. One desirable property of gradual typing is a *type guarantee*: any untyped program can be cast to any type so long as its semantic properties correspond to that type. The embedding of untyped programs into blame calculus satisfies the type guarantee, while the use of RTTI violates it. Assessing the tradeoffs between blame calculus and TS* remains interesting future work.

TypeScript TNG. In TypeScript, the information supplied by `interface` declarations is taken on faith; failures to conform to the declaration are not reported. Microsoft has funded myself and a PhD student to build a tool, TypeScript TNG, that uses blame calculus to generates wrappers from TypeScript `interface` declarations. The wrappers monitor interactions between a library and a client, and if a failure occurs then blame will indicate whether it is the library or the client that has failed to conform to the declared types. Our design satisfies the *type guarantee* described above.

Initial results appear promising, but there is much to do. We need to assess how many and what sort of errors are revealed by wrappers, and measure the overhead wrappers introduce. It would be desirable to ensure that generated wrappers never change the semantics of programs (save to detect more errors) but aspects of JavaScript (notably, that wrappers affect pointer equality) make this difficult in problematic cases; we need to determine to what extent these cases are an issue in practice.

Wide-spectrum language. Programming languages offer a range of type structures. Here are four of the most important, listed from weakest to strongest:

1. Dynamic types, as in Racket, Python, and JavaScript.
2. Polymorphic types, as in ML, Haskell, and F#.
3. Refinement types, as in Dependent ML and F7.
4. Dependent types, as in Coq, Agda, and F*.

A variant of blame calculus augmented with polymorphic types, refinement types, and dependent types might be used to mediate between each of these systems. Kinds or effects could delimit where programs may loop and where they are guaranteed to terminate, as required for the strongest form of dependent types. We hypothesise that a wide-spectrum language will increase the utility of the different styles of typing, by allowing dynamic checks to be used as a fallback when static validation is problematic.

6 Who needs gradual types?

I always assumed gradual types were to help those poor schmucks using untyped languages to migrate to typed languages. I now realise that I am one of the poor schmucks. My recent research involves session types, a linear type system that declares protocols for sending messages along channels. Sending messages along channels is an example of an effect. Haskell uses monads to track effects [24], and a few experimental languages such as Links [4], Eff [2], and Koka [13] support effect typing. But, by and large, every programming language is untyped when it comes to effects. To encourage migration from legacy code to code with effect types, such as session types, some form of gradual typing may be essential.

Acknowledgements. My thanks to Shriram Krishnamurthi, Sam Lindley, Don Sannella, Jeremy Siek, and the staff of the Western General Hospital. This work is supported by EPSRC Programme Grant EP/K034413/1.

References

1 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Principles of Programming Languages (POPL)*, pages 201–214, 2011.

2 Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014.

3 Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming*, ECOOP'10. Springer-Verlag, 2010.

4 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

5 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, October 2002.

6 Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, January 2006.

7 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Principles of Programming Languages (POPL)*, 2010.

8 Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming (TFP)*, April 2007.

9 Robert Harper. *Practical foundations for programming languages.* Cambridge University Press, 2012.

10 Anders Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, October 2012.

11 Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

12 David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23:167–189, 2010.

13 Daan Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming (MSFP)*, pages 100–126, 2014.

14 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Principles of Programming Languages (POPL)*, pages 3–10, January 2007.

15 John Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing*, pages 513–523. North-Holland, 1983.

16 Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and coercions: Together again for the first time. Technical report, University of Edinburgh, 2014.

17 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop (Scheme)*, pages 81–92, September 2006.

18 Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Principles of Programming Languages (POPL)*, pages 365–376, 2010.

19 Richard Statman. A local translation of untyped [lambda] calculus into simply typed [lambda] calculus. Technical report, Carnegie-Mellon University, 1991.

20 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP)*, September 2011.

21 Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *Principles of Programming Languages (POPL)*, January 2014.

22 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium (DLS)*, pages 964–974, October 2006.

23 Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture (FPCA)*, September 1989.

24 Philip Wadler. Comprehending monads. *Mathemetical Structures in Computer Science*, 2(4):461–493, 1992.

25 Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, March 2009.

26 Alfred North Whitehead. *An Introduction to Mathematics.* Henry Holt and Company, 1911.

27 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

28 Boris Yankov. Definitely typed repository, 2013. `https://github.com/borisyankov/DefinitelyTyped`.

Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development

Kunshan Wang[1], Yi Lin[1], Stephen M. Blackburn[1], Michael Norrish[2,1], and Antony L. Hosking[3]

1 **Research School of Computer Science, Australian National University***
108 North Road, Canberra, ACT, Australia
kunshan.wang@anu.edu.au, yi.lin@anu.edu.au, steve.blackburn@anu.edu.au

2 **Canberra Research Lab., NICTA†**
7 London Circuit, Canberra, ACT, Australia
michael.norrish@nicta.com.au

3 **Department of Computer Science, Purdue University‡**
305 N. University St., West Lafayette, IN, USA
hosking@purdue.edu

Abstract

Many of today's programming languages are broken. Poor performance, lack of features and hard-to-reason-about semantics can cost dearly in software maintenance and inefficient execution. The problem is only getting worse with programming languages proliferating and hardware becoming more complicated. An important reason for this brokenness is that much of language design is implementation-driven. The difficulties in implementation and insufficient understanding of concepts bake bad designs into the language itself. Concurrency, architectural details and garbage collection are three fundamental concerns that contribute much to the complexities of implementing managed languages.

We propose the *micro virtual machine*, a thin abstraction designed specifically to relieve implementers of managed languages of the most fundamental implementation challenges that currently impede good design. The micro virtual machine targets abstractions over memory (garbage collection), architecture (compiler backend), and concurrency. We motivate the micro virtual machine and give an account of the design and initial experience of a concrete instance, which we call Mu, built over a two year period. Our goal is to remove an important barrier to performant and semantically sound managed language design and implementation.

1998 ACM Subject Classification D.3.4 Processors

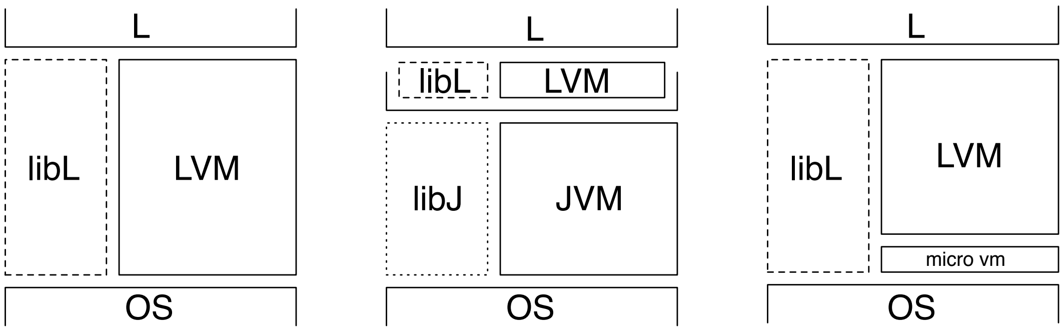
Keywords and phrases virtual machines, concurrency, just-in-time compiling, garbage collection

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.321

1 Introduction

Today's programming landscape is littered with otherwise important languages that are inefficient and/or hard to program. The proliferation of these languages is not symptomatic of a disease but evidence of a vibrant programming language ecosystem. The appeal of such languages has seen them become heavily used in critical settings. Unfortunately, the

* Supported by ARC grant no. DP0666059.
† NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.
‡ Supported by National Science Foundation grant no. CCF-0811691.



(a) Monolithic **(b)** VM on Macro VM **(c)** VM on Micro VM

Figure 1 Three implementation strategies for a managed language L. Traditional VMs (left) are typically monolithic designs, reusing little or nothing. Macro VMs such as the JVM and .NET (center) provide heavyweight reuse but offer far more than many language runtimes need. A micro virtual machine (right) provides only a thin abstraction over core concerns.

performance overheads in many cases are best measured on a log scale, leading to very real costs for the companies that have come to depend on them. Likewise, inscrutable semantics cost dearly as systems grow and maintenance becomes a nightmare. The vibrancy of the language ecosystem and rapid changes in the nature of the systems on which they are deployed suggests to us that this is a problem that is set only to get worse.

Our position is that many of the performance and semantic issues that befall such languages can be traced to fundamental implementation challenges. For example, PHP's confounding copy-on-write semantics [16], can be traced directly to a bug report dating to 2002, in which a user first observed the behavior [15]. Five days later, upon realizing that the fix would be challenging, the developers declared the "broken" semantics to be a feature of the language, and it has remained so to this day. The engineering challenge of implementing a garbage collector has led many languages to depend on naïve reference counting in their earliest implementations despite its well-known performance limitations and inability to collect cycles [11, 2]. When the owners of the language then make virtue of necessity and expose reference counting semantics in the language, this expensive but expedient implementation choice gets baked in. Similarly, the intellectual challenge of correctly supporting concurrency has led many languages to have a weak, broken, or absent model of concurrency, a limitation that grows increasingly embarrassing as stock hardware offers higher and higher degrees of parallelism.

Three fundamental concerns contribute to much of the complexity of language implementation: compiler back ends, garbage collection, and concurrency. Each are technical minefields in their own right but when brought together in a language runtime, their respective complexities combine in very challenging ways. Each of these concerns is rich enough to warrant a well developed research sub-community and rich literature of its own. We surveyed a handful of language maintainers and found a near-universal desire to be unburdened of the need to deal with these elements of language implementation, which cannot be ignored and yet frequently distract from the language implementer's principal interest in the higher levels of the language's design and implementation.

We propose a *micro virtual machine* as a robust, performant and lightweight abstraction over just three concerns: hardware (compiler back end), memory (garbage collector), and concurrency (scheduler and memory model for language implementers). Such a foundation attacks many of the issues that face existing language designs and implementations, leaving language implementers free to focus on the higher levels of the language design.

Unlike LLVM [13], a micro virtual machine is not a compiler framework, it natively supports garbage collection and concurrency, while performance-critical language-specific optimizations are performed outside the micro virtual machine in higher layers of the runtime stack. Unlike monolithic language implementations, a micro virtual machine explicitly supports cross-language reuse of demanding implementation details. Unlike the JVM, CLR, and LLVM, a micro virtual machine is minimalist, and explicitly designed to support the development of new languages, targeting a low level of abstraction, minimizing the semantic gap [5].

We have embarked on the ambitious project of designing and constructing a concrete instantiation of a micro virtual machine, with the goal of testing our hypothesis that it will advance the cause of language design and implementation. This paper reports on our motivation and two years of experience in designing and building a prototype micro virtual machine, which we call Mu. The current Mu specification is available online [14]. We will discuss a few of the more interesting aspects of the system design and implementation, including the type system, exact garbage collection, dynamism including OSR, the IR and API, and the pervasive design requirement of minimalism. We will include discussion of our preliminary experience in targeting Lua and Python implementations to Mu.

Ambitious as it is, our primary goal is that future languages will be less likely to have their semantics and performance dictated by fundamental implementation hurdles imposed upon the designers early in the language's life. A secondary goal is to improve existing language implementations, unburdening developers from elements of the language implementation that are critical but relatively uninteresting to them. It is not a goal of our project to improve upon language implementations that have benefited from massive commercial investments, such as Java.

2 Motivation

A large fraction of today's software is written in *managed languages*. Examples include JavaScript, PHP, Objective-C, Java, C#, Python, and Ruby. These languages are economically important. For example, Facebook depends on servers running PHP for its core business of efficiently delivering hundreds of billions of page views a month, Apple depends on Objective-C for every iPhone app, while Google depends on Java for its Android apps, and uses JavaScript to power its most widely used web applications including search and Gmail. Unfortunately, some of these languages are notoriously inefficient, imposing overheads as large as a factor of fifty compared to orthodox language choices such as C. The source of this inefficiency often lies in the language implementation (rather than the language itself), and this systemically impedes all applications written in that language. Moreover, early implementation choices often impede evolution of the language and/or implementation by baking implementation decisions into the language definition. A classic example is the transition from the early implementation "mistake" that resulted in dynamic scoping for LISP to the much saner static scoping. Similarly, PHP and Perl assume that their implementations use an extremely naïve reference counting memory management strategy [18]. As another example, Python's infamous "global interpreter lock" (GIL) [3] is a relic of an early implementation decision that limits Python and its GIL-infected peers (*e.g.*, Ruby) to sequential (non-parallel) execution, preventing programs from fully exploiting modern parallel hardware.

Implementing a new language can be easy when it is done naïvely, but taking the care to avoid early bad design decisions is hard. Enthusiastic implementers want to get their new language up and running without initially worrying about performance. But improving performance can require significant investment to achieve: witness the large investment by Sun/Oracle, IBM and others over many years to get Java to perform, and the competition among companies to make their JavaScript implementations outperform their competitors. And bad early language design and implementation decisions can cost even more to overcome.

The origins of systemic inefficiency of new languages often comes from the way they are implemented. When a language is implemented from the ground up in a monolithic fashion, developers must directly address every performance challenge. The problem is that different challenges often have subtle interactions. For example, the compiler, garbage collector, and thread subsystems must be designed to work together, but few developers have expertise in more than one of these subsystems. Fewer still will have expertise in their intersection, such as in the design and implementation of GC maps. An alternative approach is to avoid implementation challenges by building on top of existing language infrastructures, such as the JVM or .NET. These are large, heavily-invested platforms supporting advanced memory management, portability, aggressive just-in-time compilation, advanced support for concurrency, and come with extensive libraries. Both the JVM and CLR dictate a high level of abstraction tailored to the languages they support, which reduces the implementation effort, but which can be a poor match to the new language [5]. There are less obvious disadvantages. For example, the ease of integration of JRuby with Java leads developers to write performance-critical code in Java. Though inefficient compared to C, JRuby outperforms other Ruby implementations, so performance-critical Ruby applications inadvertently and unintentionally become dependent on Java.

Other infrastructures as a supporting substrate for language implementation, such as LLVM, have their own deficiencies. LLVM's focus on C means that support for garbage collection, concurrency, and dynamic typing are minimal or weak. Moreover, LLVM's focus on heavy-duty static optimizing compilation of C/C++ leaves its support for dynamic just-in-time (JIT) compilation as somewhat of an afterthought, receiving less attention to quality and maintenance. In contrast, dynamic languages rely heavily on on-going run-time (re)compilation to achieve good performance. Nonetheless, LLVM offers a good model with respect to intermediate language design and level of abstraction. Other infrastructures such as VMkit are composed from discrete library components that do not address the need for cross-cutting designs that span compilation, concurrency, and memory management, as discussed earlier.

The lack of suitable infrastructures drives us to propose *micro virtual machines* as a unifying substrate for language implementers that will support efficient execution of the abstractions it presents. We describe a micro virtual machine instance that will execute low-level code issued by high-level language compilers and/or run-time systems, sitting at the base of language implementations. It will take care of the most fundamental concerns of managed languages, with the bulk of the client run-time system above providing the personality of the specific client language. A micro virtual machine offers implementers a "third way," giving them access to state of the art foundations, whilst freeing them to implement language-specific semantics with maximum liberty and efficiency. Moreover, we hope that by keeping its code base as small as possible (approximately 25K lines of code), our micro virtual machine will present a suitable target for verification, allowing it to join the trusted computing base.

3 Mu: A Concrete Micro Virtual Machine

We now flesh out the design and implementation of Mu, our initial instantiation of a micro virtual machine. Space constraints limit the discussion here to a high level with a few key details, but Mu is open source, and both the specification and the source code of reference implementations are available on our website [14].

The broad architecture of Mu is reminiscent of other virtual machines such as the JVM or .NET. It executes dynamically loaded code by interpretation or compilation. A language-specific run-time system and supporting libraries sits above it. The principal difference is in the much lower level of abstraction at which micro virtual machines operate. The Mu instruction set is SSA-based rather than stack-based, the type system is much simpler, the concurrency primitives are low-level, and all high-level optimizations are the responsibility of the client run-time system, not the micro virtual machine.

A number of principles underpin the design of Mu: (1) Mu is explicitly *minimal*; any feature or optimization that can be deflected to the higher layers will be. (2) Minimalism will be compensated for by *client libraries* that sit above Mu, implementing higher level features, conveniences, transformations, and optimizations common to more than one language client. (3) The specification allows for *formally verifiable* instantiations, supporting our long term goal of a formally verified Mu instance. (4) Mu's client is *trusted*; it is allowed to shoot itself in the foot. (5) We use LLVM IR [13] as a *common frame of reference* for our IR, deviating only where we find compelling cause to do so. (6) We *separate specification and implementation*; Mu is an open specification against which clients can program and which different instantiations may implement. (7) We aim to support *diverse* language clients. (8) The Mu design will facilitate high performance language implementations.

Our goal is that Mu will facilitate the design and implementation of new and existing languages by abstracting over three of the most fundamental implementation concerns. Our focus is on languages most exposed to these concerns, namely dynamic managed languages. However, we are exploring clients for languages as diverse as Erlang, Haskell, Go, Lua and Python. It is *not* our goal to provide an implementation layer that will compete with mature, highly tuned runtimes such as the HotSpot JVM, which have benefited from enormous investment over a decade or more.

3.1 Mu Architecture

The Mu specification consists of the *Mu intermediate representation* and the *Mu client interface*. The Mu IR is the low-level language accepted and executed on Mu, while the Mu client interface defines the programming interface for client language runtimes. The client language runtime is responsible for (JIT-)compiling source code, bytecode or recorded traces into Mu IR. Mu IR code is delivered to Mu in the unit of *code bundles*, the counterpart of LLVM modules and Java class files. The Mu client interface specifies how the client may directly manipulate the state of Mu, including loading Mu IR bundles by sending messages to Mu, and how Mu-generated asynchronous events are handled by the client.

The abstract state of an executing Mu instance comprises some number of execution engines (`threads`), execution contexts (`stacks`), and memory accessed via references. Mu's abstract threads are similar to (and may directly map to) native OS/hardware threads. Stacks contain frames, each containing the context of a function activation, including its current instruction and values of local variables. Memory consists of a garbage-collected heap, a global memory, and memory cells allocated on the stacks. The abstract state can be changed by executing Mu IR code directly or by invocation of operations by the client through the Mu client interface.

The Mu project is under active development. The website [14] includes an initial Mu specification and source code for a reference implementation (which does not consider performance). We are concurrently developing a high performance Mu implementation. The reference implementation is intended for early evaluators to experiment with Mu.

3.2 Type System

The Mu type system is very simple, and designed to support both safe and unsafe memory operations. It features integer types in varying bit-widths, two floating point types, vector types for SIMD instructions, composite types in the form of `struct`s and arrays, multiple kinds of memory reference types and opaque reference types:

$$\begin{array}{rcl}
\tau_0 & ::= & \texttt{void} \mid \texttt{int}\langle n\rangle \mid \texttt{float} \mid \texttt{double} \\
& \mid & \texttt{vector}\langle\texttt{int}\langle n\rangle;\, m\rangle \mid \texttt{vector}\langle\texttt{float}; n\rangle \mid \texttt{vector}\langle\texttt{double}; n\rangle \\
& \mid & \texttt{struct}\langle\tau_0^+\rangle \mid \texttt{array}\langle\tau_0; n\rangle \\
& \mid & \texttt{func}\langle\tau_0^*; \tau_0\rangle \mid \texttt{thread} \mid \texttt{stack} \\
& \mid & \texttt{ref}\langle\tau\rangle \mid \texttt{iref}\langle\tau\rangle \mid \texttt{weakref}\langle\tau\rangle \mid \texttt{ptr}\langle\tau\rangle \mid \texttt{tagref64} \\
\tau & ::= & \tau_0 \mid \texttt{hybrid}\langle\tau_0; \tau_0\rangle
\end{array}$$

Types can be recursive under the reference types. The most basic types are scalar and vector integer and floating point types. Integers do not have signedness, but concrete operations, including `UDIV` and `SDIV`, may treat integer operands as signed or unsigned.

Object references `ref`$\langle\tau\rangle$ are references to *objects*[1] that have been allocated in the heap, and that will be managed by the garbage collector. Internal references `iref`$\langle\tau\rangle$ provide references to memory locations that may be internal to objects (*e.g.*, `array` elements or `struct` fields). Both object references and internal references are traced, and will keep their referents alive on the heap if the reference is itself reachable from GC roots. Weak references `weakref`$\langle\tau\rangle$ are object references that may be set to `NULL` when their referent is not otherwise (strongly) reachable. The `ptr`$\langle\tau\rangle$ type is an untraced pointer type, which can be used to reference memory locations potentially visible to native programs outside Mu.[2]

The type system also includes a number of opaque reference types, referring to Mu-specific entities: `thread`, `stack`, `func`$\langle\tau_0^*; \tau_0\rangle$ referring to threads, stacks and functions, respectively.

A type of the form `hybrid`$\langle F; V\rangle$ is akin to a `struct` with a fixed prefix F, followed by an array of unspecified size having elements from (non-hybrid) type V. The size of the variable-length array part in a `hybrid` is specified at allocation time. We expect, for example, that most language clients would implement their string types with a `hybrid` type. Similarly, a Java client might represent Java arrays as a fixed prefix `int` holding the size of the array and a variable-length part for the payload.

All variables and memory locations in Mu must hold values with well-defined Mu types. This restriction eliminates the option of letting the client customize the object layout and identify references in objects as VMKit does [9]. However, the Mu type system is powerful enough to express complex high-level types using a combination of nested aggregate types including `struct`s, `array`s and `hybrid`s. Mu does not have a "union" type because a union of reference and value types will make a memory location ambiguous to the garbage collector

[1] In Mu, an *object* is defined as the unit of memory allocation in the heap. We are deliberately agnostic about the sorts of languages and type systems implemented by clients; our use of the term *object* does not presuppose any sort of object-orientation. From the client's perspective, objects are headerless.

[2] The `ptr` type is a feature planned to be added in the next version of Mu. It may not be visible in the Mu specification when this paper appears.

with respect to its contents holding a reference or not. However, Mu provides a *tagged* reference type `tagref64`. It uses several bits of a 64-bit word to indicate whether it currently holds an object reference, an integer or a `double`. In this way, exact garbage collection is still possible.

3.3 Intermediate Representation

The Mu IR is low-level and language-neutral. It is similar to the SSA-based LLVM IR [13]. This grounds our design, providing a reference against which each Mu IR design decision can be measured and audited with respect to Mu design principles.

The top-level unit of the Mu IR is a *code bundle*, which contains definitions of types, function signatures, constants, global cells and functions. A function has basic blocks and instructions. The Mu instruction set contains LLVM-like primitive arithmetic and logical instructions as well as Mu-specific garbage-collection-aware memory operations, thread/stack operations, and traps.

3.3.1 Basic Instructions

A Mu instruction can be very simple. For example, an `ADD` instruction "`%c = ADD <@i64> %a %b`" adds two numbers and a `SITOFP` instruction "`%r = SITOFP <@i64 @double> %x`" converts an integer to a floating point number, treating the integer operand as signed. For the convenience of the micro virtual machine rather than the client, the types of the operands are explicitly written as type arguments so that Mu does not need to infer the type of any instruction from the types of its operands.

3.3.2 Function Calls and Exception Handling

A `CALL` instruction "`%rv = CALL <@sig> @func (%arg1 %arg2)`" calls a Mu function.[3] Mu IR programs must explicitly truncate, extend, convert or cast the arguments to match the signature. Mu also provides a `TAILCALL` instruction which directly replaces the stack frame of the caller with a frame of the callee rather than pushing a new frame. The client must explicitly generate `TAILCALL` instructions to utilize this feature. Mu implementations need not automatically convert conventional `CALL`s into `TAILCALL`s, though an implementation might.

Mu has built-in exception handling primitives that do not depend on system libraries, unlike LLVM. The `THROW` instruction generates an exceptional transfer of control to the caller of the current function.[4] The exception is caught by the nearest caller's `CALL` instruction with an *exception clause* of the form "`CALL <@sig> @func (%arg) EXC(%nor %exc)`", which branches to the designated basic block where a `LANDINGPAD` instruction receives the exception value. Unlike LLVM, an exception in Mu is an arbitrary object reference. This kind of `CALL` unconditionally catches all exceptions and the return type of `LANDINGPAD` is `ref<void>`. The client is responsible for implementing its own exception hierarchy which can be complex (like Java's and Python's) or simple (like Lua's and Haskell's, where an error is simply a

[3] Calling a native function requires a foreign function interface (FFI) that is still under design.

[4] Exception handling within a function, for example, a `throw` statement in a `try-catch` block in Java, should be translated to branching instructions (`BRANCH` and `BRANCH2`) in the Mu IR. In this case, Mu is not aware of any exceptions being thrown.

string message). The client should generate Mu IR code to check the run-time type of the exception object, and decide whether to handle, re-throw or clean up the current context.[5]

3.3.3 Memory Operations

Support for precise (exact) garbage collection is integral to the design of the instruction set. Heap memory allocation is a primitive operation in Mu. The `NEW` and the `NEWHYBRID` instructions allocate fixed and variable-length objects in the heap, respectively, automatically managed by the garbage collector. Memory can also be dynamically allocated on stacks or statically allocated in the global memory.

To implement exact garbage collection, Mu must be able to identify all references into the Mu heap. The GC root set is precisely defined as all references in live local variables, stack memory, global memory, and those explicitly held by the client. Because all values in Mu come from the Mu type system, which never confuses references and untraced values, the micro virtual machine can perform garbage collection internally without client intervention.

3.3.4 Atomic Instructions and Concurrency

Mu is designed with multi-threading in mind. Mu has threads and a C11/C++11-like memory model, allowing annotation of memory operations with the desired memory ordering semantics. Mu threads may execute simultaneously. Like LLVM, Mu has no "atomic data types", but defines a set of primitive data types eligible for atomic accesses. The supported memory orders are `NOT_ATOMIC`, `RELAXED`, `CONSUME`, `ACQUIRE`, `RELEASE`, `ACQ_REL` (acquire and release) and `SEQ_CST` (sequentially consistent). This gives the client the freedom and responsibility to implement whatever memory model is imposed (or not) by the client language. For example, the very weak `CONSUME` order is essential in the efficient implementation of the read-copy-update (RCU) pattern which facilitates extremely low-overhead lock-free concurrent memory accesses. Outdated records left by RCU updates can be garbage-collected when unused, which has been a major difficulty in implementing RCU in the Linux kernel without GC.

Supporting relaxed memory models is not trivial. As a design principle, the client is trusted, and can to shoot itself in the foot. Abusing the memory model may result in program errors or even undefined behaviors. However, Mu does not force all users to understand the most subtle memory orders. A novice language-client implementer can exclusively use the `SEQ_CST` order even though the Mu implementation supports weaker orderings. Conversely, a conservative implementer of Mu itself can always correctly implement a stronger memory model than required, for example, implementing `CONSUME` as `ACQUIRE` or implementing all memory models as `SEQ_CST`, which will trade performance for simplicity and perhaps ease of verification of the micro virtual machine.

In addition to the standard C11-like atomic operations (such as compare-and-swap) Mu provides a futex-like [8] wait mechanism, as well as basic thread park/unpark operations. The client is responsible for implementing other shared-memory machinery such as blocking locks and semaphores. As these can be difficult and tedious to implement, they may be provided by client-level *libraries*, enabling complex implementations to be shared among multiple language clients.

[5] There is no **`finally`** in Mu, but it can be implemented as an unconditional catch followed by the actions in the **`finally`** block and another `THROW` instruction.

3.3.5 Stack Binding and the Swap-stack Operation

Unlike many language runtimes, Mu clearly distinguishes between threads (executors) and stacks (execution contexts).

A *thread* is an abstraction of a processor, while a *stack* is the context in which a thread runs. Each stack includes abstract execution state such as the program counter and the values of local variables in each frame. A thread is not permanently bound to any particular stack.

The `SWAPSTACK` instruction unbinds a thread from one context and rebinds it to another context [6]. When rebound, the thread continues from the corresponding instruction (usually another `SWAPSTACK`) where the destination context paused when last active (bound to a thread). This semantics directly provides an implementation of symmetric co-routines, which can in turn implement high-level language features including user-level "green" threads, the $m \times n$ threading model, generators, and one-shot continuations.

Dolan *et al.* [6] showed that this lightweight context switching mechanism can be implemented fully in user space with only a few instructions, so it is more efficient than native threads which inevitably involve transitioning through the kernel. Mu also assumes a client code generator that knows and can specify the liveness of variables at each `SWAPSTACK`, so only the needed registers are saved. This is impossible for library-based approaches, including `setjmp`/`longjmp`, `swapcontext` or customized assembly code, which have no information from a compiler and must conservatively save all registers.

The same stack binding and unbinding mechanisms are also used in other places besides the `SWAPSTACK` instruction, and many Mu design choices are based on this mechanism. Unbound threads and stacks are inactive. Only when a thread is bound to a stack context does it begin/resume executing from that context. Stack binding and unbinding are also used in trap handling, which are discussed below as part of the Mu client interface.

3.3.6 Traps and Watchpoints

A Mu IR program can temporarily pause execution and transfer control to the client by executing a `TRAP` instruction, which is trivial in Mu: "`%trap_name = TRAP <@RetTy>`".[6] Traps give clients the opportunity to introspect execution state to adapt and optimize the running program. The `WATCHPOINT` instruction is a conditional variant of `TRAP` which is disabled in the common case but can be enabled asynchronously by a client thread. `WATCHPOINT` is particularly useful for invalidating speculatively optimized code.

3.4 Client Interface

The *Mu client interface* is bi-directional. The client can send messages to Mu for the purposes of: (1) loading Mu IR code bundles into Mu, (2) accessing the Mu memory, and (3) introspecting and manipulating the state of Mu threads and stacks. Mu sends messages to the client if a `TRAP` or `WATCHPOINT` instruction is executed, or a declared but not defined function is called.

All Mu values, including references, may be indirectly exposed to the client via opaque handles tracked by Mu. This makes exact GC easy to implement because all externally held

[6] Lameed *et al.* [12] implemented something similar in LLVM, but it required non-trivial work in both the JIT compiler and the runtime library in order to achieve the same goal as the `TRAP` instruction in Mu. For compatibility reasons, they did not modify LLVM itself, so could not introduce new instructions. Since Mu is designed from scratch, it has no such restriction.

references are tracked. Copying and concurrent GC is also easier with a level of indirection. This design, which is also used by JNI and the Lua C API [10], also segregates the different type systems of Mu and the language the client is written in and, thus, makes the interface cleaner.

3.4.1 Bundle Loading and Code Redefinition

The client submits Mu IR code bundles to Mu via the interface. When a declared but not defined function is called, Mu will send a message to the client, which will be handled by a registered handler. The client should define the function by submitting another bundle containing the function definition. The client can also redefine existing functions. All existing call sites remain valid and Mu always calls the newest version of any function. This feature allows optimizing compilers to replace functions with (re-)optimized versions.

3.4.2 Traps and Stack Operations

Traps and watchpoints use the same stack binding mechanism as the `SWAPSTACK` instruction. When a `TRAP` or `WATCHPOINT` instruction is executed, the current thread is unbound from its current stack just before entering the registered trap handler in the client, leaving the stack in a clean state ready for introspection and on-stack replacement (OSR). During execution of the trap handler, the client can introspect the state of the stack frames, including the current function, the current instruction and the value of local variables. At the end of the trap handler, the client designates an unbound stack to which the original thread will be rebound. This stack does not need to be the same stack to which the thread was bound before the trap, so the thread may continue in a brand new context.

Some optimizations involve *on-stack replacement*: replacing a stack frame with a new frame of an optimized version of a function and continuing from the equivalent place it paused at. The Mu client interface supports this by providing two primitive operations: (1) popping a frame from a stack, and (2) making a new frame for a given function and its arguments on the top of a stack, and continuing from the beginning of that function. The client can emulate continuing from the middle of a function by inserting branches in the high-level language; a well-established approach [7].

3.4.3 Miscellaneous Operations

Many operations available as Mu IR instructions are intentionally duplicated to be used *via* the Mu client interface. The client can allocate and access the Mu memory via this interface, too. With an layer of indirection, this interface is designed for infrequent introspection. The client can also create threads and stacks, which is the proper way to start new Mu IR programs.

4 Building Mu Clients

The *client* is the program sitting on top of Mu, the micro virtual machine. It is the user of Mu and the implementer of the concrete high-level programming language, ‘LVM’ in Figure 1c. The Mu specification defines an interface for clients to manipulate and control the virtual machine, but does not impose any other requirements on the client. Here, we describe a number of strategies for building Mu clients, client-level libraries, and other higher-level client-specific abstractions.

4.1 Strategies for Building Clients

Mu supports a number of different approaches for implementing client languages. A client might dynamically compile to Mu IR just-in-time, compile to Mu IR ahead-of-time, or execute as an interpreter against either the Mu API or running itself as a Mu-coded client. We now discuss these options.

4.1.1 Just-in-time Compiling

The Mu client interface gives the client the power and responsibility to deliver Mu IR code to Mu. The most intuitive strategy is a compile-only approach, just-in-time compiling the higher-level language source code or byte code into the Mu IR, possibly *via* several extra layers of higher-level intermediate representations and optimizations. It us up to the client to decide how much optimisation it does to balance between compile time and code performance.

The generated Mu functions need not mirror high-level language functions. A tracing JIT compiler may deliver a recorded trace as a Mu function which may cross the boundary of many high-level functions, or may be a single loop within a high-level function.

4.1.2 Ahead-of-time Compiling

The Mu specification does not require code to be generated at run time. In fact, a valid implementation could ahead-of-time compile the high-level language program into the Mu IR before execution. In this way, the client merely loads Mu IR code from the disk and delivers it directly to Mu without further processing.

A valid full ahead-of-time implementation could also generate a single binary which behaves like an amalgamation of the micro virtual machine, a set of bundles, and a client which loads the bundles, starts with a particular Mu function and handles specific trap events. This approach is similar to the "boot image" of JikesRVM [1]. In this way, a Mu implementation of a client language might behave much like traditional ahead-of-time compilers.

4.1.3 Interpreting

For the ease of engineering, the first implementation of many languages is usually an interpreter. Although implementing the interpreter in the client and using the Mu heap *via* the Mu client interface is possible, it is not recommended because the interface is not designed for frequent lightweight calls, so will introduce an overhead.

One strategy would be to implement the interpreter itself in the Mu IR, or in any language that already compiles to the Mu IR. For example, if there is a client for Java running on Mu, then interpreters written in Java, including Jython and JRuby, will run on Mu. Currently we are working on translating RPython into Mu IR. The PyPy interpreter (written in RPython) will then run on Mu, as will other languages that have an implementation in RPython, including Prolog, Scheme and Erlang. Like Jython, this implementation strategy makes use of the concurrency and GC provided by the underlying VM, but does not directly utilize the underlying JIT compiler.

The Truffle/Graal project demonstrated that it is possible to generate a specializing JIT compiler from an interpreter by partial evaluation. In this way, the language implementer only needs to write an interpreter, but ultimately makes use of the JIT compiler.

4.2 Client-level Libraries

A micro virtual machine is minimalist, providing only a thin layer on which a language runtime can be built. Mu aims to deal with three of the most fundamental and conceptually challenging concerns within this layer. However the client is left with much to do. Rather than succumbing to the temptation to grow Mu, we adopt the principle of lifting additional features and amenities to client-level libraries, sitting above Mu. These may include helper features for generating SSA-form Mu IR, and Mu IR to Mu IR optimization libraries for common higher-level optimizations.

For example, there may be library support for dynamic languages that provides common specialization mechanisms, support for functional languages that handles higher-order functions, a library for concurrent languages which provides code snippets that implement synchronization primitives, *etc.* High-level frameworks, similar to RPython, Truffle, Terra or Lancet, can also be provided as libraries on the client side. In this way, the high-level language implementer can work on a much higher level, and does not always need to work with Mu directly.

4.3 Metacircular Clients

The client is just a program interacting with the micro virtual machine and, in theory, can be implemented in any language. It is possible to implement a *metacircular client* which runs as a Mu IR program inside the same Mu instance hosting other high-level programs. In this approach, the border between the client and the micro virtual machine is blurred. The client can directly refer to values or objects in Mu, and API messages can be implemented as simple function calls or SWAP-STACK operations. Special Mu IR instructions will give Mu IR programs access to the internals of Mu, including loading bundles, handling traps and introspecting stack states.[7]

4.4 Higher-level Abstraction

Mu is minimal, and does not provide any abstractions of classes, type hierarchies, methods, virtual dispatching, run-time type checking, strings, character encoding, higher-order functions, closures, events or message passing. The client is supposed to map its high-level language elements to the Mu IR. Two clients may map these elements differently even if they implement the same language.

Mu is designed to support multiple languages by minimizing semantic mismatching. However, unlike the .NET framework, Mu does not provide an abstract platform where programs in different languages can call each other. A client may implement such a platform by mapping different languages in a uniform way so that they can exchange data and make cross-language calls on the micro virtual machine level. The client can also implement a specific common intermediate language (like Java bytecode or the .NET CIL) above the Mu IR, and different languages can interoperate on that level. Mu understands only the Mu IR and remains oblivious of the actual high-level languages.

[7] Mu IR instructions to facilitate metacircular clients are still under consideration at the time of writing. Even without special instructions, such Mu IR-initiated introspection and controlling operations can be supported by traps and the assistance of a “conventional” client. This allows some Mu IR programs, such as an interpreter, to manage the state of Mu in a non-metacircular Mu implementation.

5 Related Work

5.1 Java Virtual Machines

The Java Virtual Machine (JVM) was originally designed for the Java programming language, but its portable Java Bytecode, clearly specified behaviors and performance attracted a wide range of language implementations to be hosted on the JVM, including Jython, JRuby, Scala, X10, *etc.*

This approach – reusing the existing JVM for new languages – bears several fundamental problems. The obvious one is the semantic gap between the new language and Java. The JVM implements many Java-specific semantics which are irrelevant to other languages. Working around them introduces overheads. On the other hand, Mu takes the lesson and is carefully designed to be language-agnostic at a much lower level to avoid the semantic gap.

Another problem is the JVM's lack of optimizations for languages other than Java. These optimizations include type inference and specialization, which are vital to dynamic languages [5], and inline caches for languages with dynamic dispatching. Mu, on the other hand, assumes most optimizations will be done by a client above it. For this reason, Mu exposes many low-level mechanisms, including vector instructions, on-stack replacement, swap-stack, tagged references, traps and watchpoints, to the client. Those mechanisms are either private or non-existent in the JVM, but they enable many advanced implementation techniques. For example, it is expensive to map Erlang processes to Java threads, which are usually mapped to native threads. In Mu, we believe stacks would work well in this role. Similarly, Mu's tagged reference type is a good candidate for implementing Lua's values, which have reference semantics, but refer to floating point numbers most of the time.

Besides, Mu aims to be a thin substrate while the JVM is a monolithic VM.

5.2 LLVM

The Low Level Virtual Machine (LLVM) [13] is a compiler framework including a collection of modular and reusable compilation and toolchain technologies. The LLVM compiler framework and its code representation (LLVM IR) together provide a combination of key capabilities that are important for language implementations. LLVM is the main reference according to which we designed Mu.

Mu also includes a number of significant differences from the LLVM. Firstly, Mu is designed to support managed languages while the LLVM is designed for C-like languages. Like C, the LLVM IR type system contains raw pointer types but not reference types. The LLVM does not provide a garbage collector, instead defining intrinsic functions including read/write barriers and yieldpoints, on top of which garbage collection must be implemented or inserted by the language frontend. Secondly, Mu is designed to be minimal while the LLVM is maximal. The LLVM tries to minimize the job of language frontends and include many optimization passes, while the Mu pushes as much work to the client as possible.

5.3 VMKit

VMKit [9] is a common substrate for the development of high-level *managed runtime environments* (client language runtimes), providing abstractions for concurrency, JIT-compiling and garbage collection. VMKit glues together three existing libraries: LLVM as the JIT compiler, MMTk as the memory manager, and POSIX threads for threading. The VMKit developers built two clients, for the CLI and JVM respectively as a proof of concept.

As the name suggests, VMKit is not a self-contained virtual machine, but a toolkit that provides incomplete abstractions over certain features. For example, VMKit leaves the object layout to be implemented by the client. As a consequence, the client runtime developed by the high-level language developer, must participate in object scanning and the identification of GC roots. VMKit's solution to concurrency is the POSIX Threads library, but threads cannot be implemented as a library [4], and require a carefully defined memory model involving both garbage collection and the JIT compiler.

Nonetheless, VMKit demonstrates that a toolkit that abstracts over the common key features can ease the burden of the development of language implementations, which is also part of the motivation for a micro virtual machine.

5.4 Others

Common Language Infrastructure

The Common Language Infrastructure (CLI) is Microsoft's counterpart to the JVM. Its Common Intermediate Language (CIL) is designed for several languages with similar level to VB.NET, C#, *etc.*, but also hosts many different languages including Managed C++, F# and JavaScript. The CLI shares similar problems to the JVM in that it is a monolithic VM and was designed for certain kinds of languages.

Truffle/Graal

Truffle and Graal [17] are reusable VM components for code execution developed by Oracle. Truffle serves as an AST interpreter with speculative execution and AST rewriting. Graal takes stabled Truffle AST and uses partial evaluation to JIT compile AST nodes. They aim to provide a reusable code execution engine for implementations of object-oriented languages. This goal sits on a much higher level than Mu.

6 Status and Future Work

During the early prototyping phase, we implemented a subset of Lua on a prototype micro virtual machine. Such a prototype could run simple Lua programs. It proved feasible to offload basic control flow analysis, including the conversion to the SSA form, up to the client. It also showed the usefulness of tagged references in the implementation of a dynamic language.

Currently we are experimenting with using Mu as a backend of RPython, the lower level of the PyPy project. Currently Mu can run simple RPython programs. This effort will eventually bring about a high-performance micro VM-based implementation of Python as well as other languages on RPython.

In the future, we will evaluate a diverse collection of languages, with Python, Haskell and Erlang having high priority, to test the ability of Mu to accommodate different languages. We will also bring transactional memory to Mu, which is known to solve the global interpreter lock (GIL) problem in Python and Ruby. Ultimately, we hope to produce a verified micro VM, perhaps combining it with the verified seL4 micro-kernel, thereby bringing us one step closer to a completely verified system with a managed runtime.

Mu abstracts over hardware. Although our immediate focus has been abstraction over ISAs and memory models, increasing hardware diversity invites a number of important future considerations. These include optimizing for energy as well as performance, abstracting over single-ISA heterogeneous hardware, and more radical hardware heterogeneity such as GPUs and FPGAs. These are open topics for future research.

7 Conclusion

The current proliferation of new languages is evidence of a vibrant programming languages ecosystem. Unfortunately many languages are seriously inefficient, in terms of performance or maintenance, or both. We suggest that fundamental implementation challenges are a root cause of much of this inefficiency, and that a micro virtual machine may provide a solid foundation for the development of new languages. We describe a new micro virtual machine, Mu, developed over the past two years, with the specific goal of addressing this problem. Our hope is that micro virtual machines will change the way the next generation of languages are built, for the better.

References

1 Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Hummel Flynn, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'99, pages 314–324, Denver, Colorado, November 1999. doi: 10.1145/320384.320418.

2 Swift, 2015. `https://developer.apple.com/swift/`.

3 Shannon Behrens. Concurrency and Python. *Dr Dobb's*, February 2008. `http://www.drdobbs.com/open-source/concurrency-and-python/206103078`.

4 Hans-J. Boehm. Threads cannot be implemented as a library. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'05, pages 261–268, Chicago, Illinois, 2005. doi: 10.1145/1065010.1065042.

5 Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Toshio Nakatani, Takeshi Ogasawara, and Peng Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 195–212, Pittsburgh, Pennsylvania, 2012. doi: 10.1145/2398857.2384631.

6 Stephen Dolan, Servesh Muralidharan, and David Gregg. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization*, 9(4):36:1–36:25, January 2013. doi: 10.1145/2400682.2400695.

7 Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'03, pages 241–252, San Francisco, California, 2003. doi: 10.1109/CGO.2003.1191549.

8 Hubertus Franke and Rusty Russell. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, pages 479–495, Ottawa, Canada, June 2002. `http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf`.

9 Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: A substrate for managed runtime environments. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'10, pages 51–62, Pittsburgh, Pennsylvania, 2010. doi: 10.1145/1735997.1736006.

10 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, San Diego, California, June 2007. doi: 10.1145/1238844.1238846.

11 Ivan Jibaja, Stephen M Blackburn, Mohammad R. Haghighat, and Kathryn S McKinley. Deferred gratification: Engineering for high performance garbage collection from the get go. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC'11, San Jose, California, June 2011. doi: 10.1145/1988915.1988930.

12 Nurudeen A. Lameed and Laurie J. Hendren. A modular approach to on-stack replacement in LLVM. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'13, pages 143–154, Houston, Texas, 2013. doi: 10.1145/2451512.2451541.

13 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, San Jose, California, March 2004. doi: 10.1109/CGO.2004.1281665.

14 The Micro Virtual Machine Project. `http://microvm.org/`.

15 Doc Bug #20993 Element value changes without asking. `https://bugs.php.net/bug.php?id=20993`.

16 Akihiko Tozawa, Michiaki Tatsubori, Tamiya Onodera, and Yasuhiko Minamide. Copy-on-write in the PHP language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'09, pages 200–212, Savannah, Georgia, 2009. doi: 10.1145/1480881.1480908.

17 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, Indianapolis, Indiana, 2013. doi: 10.1145/2509578.2509581.

18 Owen Yamauchi. On garbage collection. HipHop Virtual Machine for PHP. `http://www.hhvm.com/blog/431/on-garbage-collection`.