2nd Summit on Advances in Programming Languages

SNAPL 2017, May 7–10, 2017, Asilomar, CA, USA

Edited by

Benjamin S. Lerner
Rastislav Bodík
Shriram Krishnamurthi

LIPICS

Editors

Benjamin S. Lerner	Rastislav Bodík	Shriram Krishnamurthi
Northeastern University	University of California Berkeley	Brown University
USA	USA	USA
blerner@ccs.neu.edu	bodik@cs.berkeley.edu	sk@cs.brown.edu

ACM Classification 1998
D.3 Programming Languages

ISBN 978-3-95977-032-3

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/978-3-95977-032-3.

Publication date
May, 2017

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.2017.0

ISBN 978-3-95977-032-3 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This is the second running of the Summit oN Advances in Programming Languages (SNAPL), a new venue for the programming languages community. The goal of SNAPL is to complement existing conferences by discussing big-picture questions. After the success of the first SNAPL in 2015, we hope to continue growing the venue into a place 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 planned to hold SNAPL every two years in early May in Asilomar, California, and this second running is consistent with that plan.

SNAPL has drawn on the elements from many successful meeting formats such as the database community's CIDR conference, the security community's NSPW workshop, and others, and continues to evolve its own particular flavor. The focus on SNAPL is not primarily on papers but rather on talks and interaction. Nevertheless, a short paper is the primary medium by which authors request and obtain time to speak. A good SNAPL entry, however, does not have the character of a regular conference submission – we already have plenty of venues for those. Rather, it is closer to the character of an invited talk, encompassing all the diversity that designation suggests: visionary idea, progress report, retrospective, analysis of mistakes, call to action, and more. Thus, a SNAPL submission should be viewed more as a "request to give an invited talk".

In the process of assembling SNAPL 2017, we also realized that SNAPL serves an additional useful role: serving as a discussion venue for programming languages akin to the role the Snowbird conference plays for computer science chairs and deans. In this spirit, we invited the community to suggest the names of junior researchers in programming languages who would be interesting to invite, and selected a few names from this long and impressive list. We also intend to invite a few senior researchers to address the gathering.

Overall, the submissions suggest SNAPL remains an interesting and valuable venue. Its main weakness was fewer submissions than we expected (28, but of sufficient quality that we were able to accept 18). We have heard three problems with the organization: the submission date was poorly timed (January 6 was too close to or amidst vacation time), there was insufficiently broad publicity, and the chosen date clashes with a few other venues. The first two of these, in particular, seem easy to remedy in the future.

Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi
May, 2017

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Everest: Towards a Verified, Drop-in Replacement of HTTPS

Karthikeyan Bhargavan[1], Barry Bond[2], Antoine Delignat-Lavaud[3], Cédric Fournet[4], Chris Hawblitzel[5], Cătălin Hriţcu[6], Samin Ishtiaq[7], Markulf Kohlweiss[8], Rustan Leino[9], Jay Lorch[10], Kenji Maillard[11], Jianyang Pang[12], Bryan Parno[13], Jonathan Protzenko[14], Tahina Ramananandro[15], Ashay Rane[16], Aseem Rastogi[17], Nikhil Swamy[18], Laure Thompson[19], Peng Wang[20], Santiago Zanella-Béguelin[21], and Jean-Karim Zinzindohoué[22]

1 Inria, Paris, France
2 Microsoft Research, Redmond, WA, USA
3 Microsoft Research, Cambridge, UK
4 Microsoft Research, Cambridge, UK
5 Microsoft Research, Redmond, WA, USA
6 Inria, Paris, France
7 Microsoft Research, Cambridge, UK
8 Microsoft Research, Cambridge, UK
9 Microsoft Research, Redmond, WA, USA
10 Microsoft Research, Redmond, WA, USA
11 Inria, Paris, France
12 Inria, Paris, France
13 Microsoft Research, Redmond, WA, USA
14 Microsoft Research, Redmond, WA, USA
15 Microsoft Research, Redmond, WA, USA
16 Microsoft Research, Redmond, WA, USA
17 Microsoft Research, Bangalore, India
18 Microsoft Research, Redmond, WA, USA
19 Microsoft Research, Redmond, WA, USA
20 Microsoft Research, Cambridge, UK
21 Microsoft Research, Cambridge, UK
22 Inria, Paris, France

Abstract

The HTTPS ecosystem is the foundation on which Internet security is built. At the heart of this ecosystem is the Transport Layer Security (TLS) protocol, which in turn uses the X.509 public-key infrastructure and numerous cryptographic constructions and algorithms. Unfortunately, this ecosystem is extremely brittle, with headline-grabbing attacks and emergency patches many times a year. We describe our ongoing efforts in *Everest*,[1] a project that aims to build and deploy a verified version of TLS and other components of HTTPS, replacing the current infrastructure with proven, secure software.

Aiming both at full verification and usability, we conduct high-level code-based, game-playing proofs of security on cryptographic implementations that yield efficient, deployable code, at the level of C and assembly. Concretely, we use F*, a dependently typed language for programming, meta-programming, and proving at a high level, while relying on low-level DSLs embedded within

[1] The *Everest* VERified End-to-end Secure Transport: `https://project-everest.github.io`.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 1; pp. 1:1–1:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

F⋆ for programming low-level components when necessary for performance and, sometimes, side-channel resistance. To compose the pieces, we compile all our code to source-like C and assembly, suitable for deployment and integration with existing code bases, as well as audit by independent security experts.

Our main results so far include (1) the design of Low⋆, a subset of F⋆ designed for C-like imperative programming but with high-level verification support, and KreMLin, a compiler that extracts Low⋆ programs to C; (2) an implementation of the TLS-1.3 record layer in Low⋆, together with a proof of its concrete cryptographic security; (3) Vale, a new DSL for verified assembly language, and several optimized cryptographic primitives proven functionally correct and side-channel resistant. In an early deployment, all our verified software is integrated and deployed within `libcurl`, a widely used library of networking protocols.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Security, Cryptography, Verification, TLS

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.1

1 Introduction

The Internet's core security infrastructure is frighteningly brittle. As more and more services rely on encryption, security best practices urge developers to use standard, widely-used components like HTTPS and SSL (the latter is now standardized as TLS). As a result, the same pervasive components are used for securing communications on the Web and for VoIP, email, VPNs, and the IoT.

Unfortunately, these standard components are themselves often broken in many ways. Even before recent headline-grabbing attacks like HeartBleed[2], FREAK[3], and Logjam[4] entire papers [21, 10] were published just to summarize all of the academically "interesting" ways TLS implementations have been broken, without even getting into "boring" vulnerabilities like buffer overflows and other basic coding mistakes. This tide of flaws shows no signs of abating; in the year after those papers were published, 54 new CVE security advisories were published just for TLS. These flaws are frequently found and fixed in all of the widely used TLS implementations, as well as in the larger HTTPS ecosystem. They span a wide gamut including memory management mistakes, errors in protocol state machines, lax X.509 certificate parsing and validation, weak or badly implemented cryptographic algorithms, side channels, and even genuine design flaws[5] in the standards. Furthermore, because many TLS implementations expose truly terrible APIs, HTTPS applications built on them regularly make devastating mistakes [12].

These persistent problems have generated sufficient industry concern that both Google and the OpenBSD project are building separate forks of OpenSSL (BoringSSL and LibreSSL, respectively) while Amazon is developing a brand new implementation (s2n). Many corporations have even joined the multi-million-dollar Core Infrastructure Initiative to support additional testing and security auditing of open-source projects, starting with OpenSSL.

[2] `https://heartbleed.com/`
[3] `https://freakattack.com/`
[4] `https://weakdh.org/`
[5] `https://mitls.org/pages/attacks/3SHAKE`

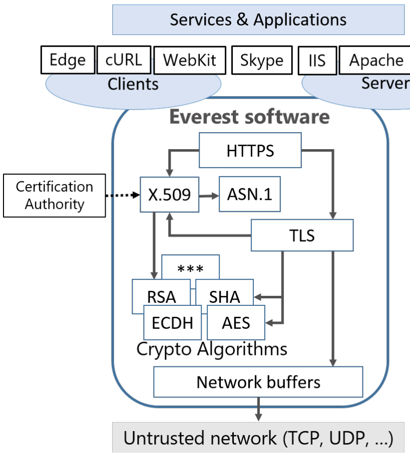


Figure 1 *Everest* architecture.

1.1 A Need for Verified Deployments Now

While the industry is taking incremental steps to try to stem the persistent tide of vulnerabilities, the programming-language community is uniquely positioned to definitively solve this problem. The science of software verification has progressed to a point where a large team of experts can reasonably expect to build a fully verified software stack, e.g., seL4 [16], Ironclad [14], and CertiKOS [13], with still more ambitious, broadly ranging efforts already underway (e.g., `http://deepspec.org/`). Yet, even when augmented with secure communication components like TLS, a fully verified stack would not meet today's pressing needs, since a wholesale replacement of the software stack is not in the offing.

Improving the current software landscape requires both a dramatic shift in software development methodology and an incremental approach to the deployment of verified software.

Everest is a new joint project between Microsoft Research and INRIA that aims to build verified software components and deploy them within the existing software ecosystem. Specifically, *Everest* develops new implementations of existing protocols and standards used for web communications. At a minimum, we prove our implementations functionally correct. Beyond functional correctness, we integrate cryptographic modeling and protocol analysis to prove, by reduction to computational assumptions on their core algorithms, that our implementations provide secure-channel abstractions between the communicating endpoints. Our verified code is implemented in F* [25, 1], a dependently typed programming language and proof assistant, and in several DSLs embedded within F*.

After verification, in support of incremental deployment, our code is extracted by verified tools to C and assembly, and compiled further by off-the-shelf C compilers (e.g., gcc and clang, but also, at a performance cost, verified compilers like CompCert [18]) and composed with adapters that interface our verified code with existing software components, like the web browsers, servers and other communication software shown in the Figure 1. Being only as strong as its weakest component, software systems that include verified *Everest* code may not be impervious to attack; yet, any attack such a system suffers will be attributable to a flaw in a component outside *Everest*, while simple, critical systems may be within reach of full verification with a reasonable marginal effort.

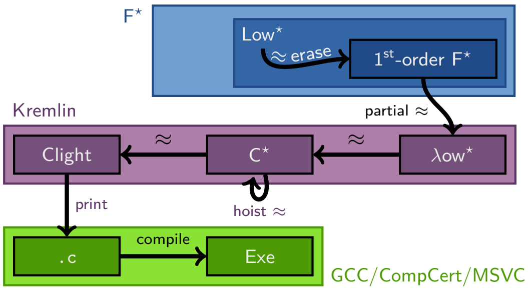


Figure 2 Low⋆ embedded in F⋆, compiled to C, with soundness and security guarantees.

1.2 Structure of this Paper

Everest is a work in progress – we present an overview of the methodology we have used so far. Our main guiding principle is to provide low-level, efficient implementations of various protocol standards by extracting them from fresh code, programmed and verified at a high-level of abstraction. This principle applies best for relatively small and complex code, such as a secure protocol stack.

To this end, in §2, we present Low⋆, an embedded sub-language of F⋆ geared towards programming against a C-like memory model, while specifying and proving these programs within F⋆'s dependent type theory. Low⋆ programs are extracted to C by a new tool called KreMLin: although we have yet to verify the implementation of KreMLin, we have formally modeled its translation and proved on paper that it preserves the functionality of programs to the level of CompCert's Clight. We have also showed that compilation from Low⋆ to Clight does not introduce any side-channels due to memory access patterns.

In §3, we sketch several examples of verified code and their specifications in Low⋆, showing how we state and prove memory safety, functional correctness, and cryptographic security.

In §4, we discuss a few strands of ongoing work. This includes the design of Vale, a new DSL for verified assembly language programming and its use in producing even lower level, efficient, functionally correct implementations of the AES and SHA256 algorithms whose performance is on par with OpenSSL, the mostly widely used implementation. We also discuss an early deployment of *Everest* software as a drop-in replacement for TLS in libcurl, and its use from within a command-line git client.

Finally, §5 presents some parting thoughts, covering some opportunities and challenges.

For more technical details, we refer the reader to three recent papers describing our verification of the TLS-1.3 record layer [6]; the design and implementation of Low⋆ and KreMLin [5]; and the design and implementation of Vale [9].

2 Low⋆: Verified Low-level Programming Embedded in F⋆

We aim to bridge the gap between high-level, safe-by-construction code, optimized for clarity and ease of verification, and low-level code exerting fine control over data representations and memory layout in order to achieve better performance. Towards this end, we introduce Low⋆, a DSL for verified, efficient low-level programming, embedded within F⋆, a verification-oriented, ML-like language with dependent types and SMT-based automation [11]. Figure 2 illustrates the high-level design of Low⋆ and its compilation to native code.

Libraries for low-level programming within F⋆. At its core, F⋆ is a purely functional language to which effects like state are added programmatically using monads. We instantiate the state monad of F⋆ to use a CompCert-like structured memory model [18, 19] that separates the stack and the heap, supporting transient allocation on the stack, and allocating and freeing individual reference cells on the heap – this is not the "big array of bytes" model systems programmers sometimes use. The heap is organized into disjoint logical regions, which enables stating separation properties necessary for modular, stateful verification. On top of this, we program a library of buffers – C-style arrays passed by reference – with support for pointer arithmetic and referring to only part of an array. By virtue of F⋆ typing, our libraries and all their well-typed clients are guaranteed to be memory safe, e.g., they never access out-of-bounds or deallocated memory.

Designing Low⋆, a subset of F⋆ easily compiled to C. We intend to give Low⋆ programmers precise control over the performance profile of the generated C code. As much as possible, we aim for the programmer to have control even over the syntactic structure of the target C code, to facilitate its review by security experts unfamiliar with F⋆. As such, to a first approximation, Low⋆ programs are F⋆ programs well-typed in the state monad described above, which, after all their computationally irrelevant (ghost) parts have been erased, must satisfy several requirements. Specifically, the code (1) must be first order, to prevent the need to allocate closures in C; (2) must not perform any implicit allocations; (3) must not use recursive datatypes, since these would have to be compiled using additional indirections to C structs; and (4) must be monomorphic, since C does not support polymorphism directly. We emphasize that these restrictions apply only to computationally relevant code: proofs and specifications are free to use arbitrary higher-order, dependently typed F⋆.

A dual interpretation of Low⋆, via compilation to OCaml or Clight. Low⋆ programs interoperate naturally with other F⋆ programs, and precise specifications of Low⋆ and F⋆ code are intermingled when proving properties of their combination. As usual in F⋆, programs are type-checked and compiled to OCaml for execution, after erasing computationally irrelevant parts of a program, e.g., proofs and specifications, using a process similar to Coq's extraction mechanism [20]. Importantly, Low⋆ programs have a second, equivalent but more efficient semantics via compilation to C, with a predictable performance model including manual memory management – this is implemented by KreMLin, a new compiler from the Low⋆ subset of F⋆ to C.

Justifying its dual interpretation as a subset of F⋆ and a subset of C, we give a translation from Low⋆, via two intermediate languages, to CompCert's Clight [8] and show that it preserves trace equivalence with respect to the original F⋆ semantics of the program. In addition to ensuring that the functional behavior of a program is preserved, our trace equivalence also guarantees that the compiler does not introduce unexpected side-channels due to memory access patterns, at least until it reaches Clight – a useful sanity check for cryptographic code.

KreMLin, a compiler from Low⋆ to C. Our formal model guides the implementation of KreMLin, a new tool that emits C code from Low⋆. Although the implementation of KreMLin is not verified yet, we plan to verify its main translation phased based on our formal model in the near future. KreMLin is designed to emit well-formatted, idiomatic C code suitable for manual review and audit by independent security experts unfamiliar with our verification methodology. The resulting C programs can be compiled with CompCert for greatest assurance, and with mainstream C compilers, including GCC and Clang, for greatest performance. We have used KreMLin to extract to C the 20,000+ lines of Low⋆ code we have written so far.

Our formal results cover the translation of standalone Low⋆ programs to C, proving that execution in C preserves the original F⋆ semantics of the Low⋆ program. More pragmatically, we have built several cryptographic libraries in Low⋆, compiled them to ABI-compatible C, allowing them to be used as drop-in replacements for existing libraries in C or any other language. The performance of our verified code after KreMLin extraction is comparable to existing (unverified) cryptographic libraries in C.

3 Proving Cryptographic Implementations in Low⋆

In this section, we sketch a few simple fragments of code from HACL⋆, a *high-assurance cryptographic library* programmed and verified in Low⋆ and used in our verified implementation of the TLS-1.3 record layer. First, we illustrate functional correctness properties proven of an efficient implementation of the Poly1305 Message Authentication Code (MAC) algorithm [2]. Then, we discuss our model of game-based cryptography in F⋆ and its use in proving security of the main authenticated encryption construction used in TLS-1.3.

3.1 Functional Correctness of Poly1305

Arithmetic for the Poly1305 MAC algorithm is performed modulo the prime $2^{130} - 5$, i.e., the algorithm works in the finite field $GF(2^{130} - 5)$. To specify modular arithmetic in this field in F⋆, we make use of refinement types, as shown below.

```
val p = 2ˆ130 − 5 (∗ the prime order of the field ∗)
type elem = n:nat {n < p} (∗ abstract field element ∗)
let ( + ) (x y : elem) : elem = (x + y) % p (∗ field addition ∗)
let ( ∗ ) (x y : elem) : elem = (x ∗ y) % p (∗ field multiplication ∗)
```

This code uses F⋆ infinite-precision natural numbers (nat) to define the prime order p of the field and the type of field elements, elem, inhabited by natural numbers n smaller than p. It also defines two infix operators for addition and multiplication in the field in terms of arithmetic on nat. Their result is annotated with elem, to indicate that these operations return field elements. The F⋆ typechecker automatically checks that the result is in the field; it would report an error if, for example, we omitted the reduction modulo p. These operations are convenient to specify polynomial computations but are much too inefficient for deployable code.

Instead, typical 32-bit implementations of Poly1305 represent field elements as mutable arrays of five unsigned 32-bit integers, each holding 26 bits. This representation evenly spreads out the bits across the integers, so that carries during arithmetic operations can be delayed. It also enables an efficient modulo operation for p.

We show below an excerpt of the interface of our lower-level verified implementation, relying on the definitions above to specify their correctness. The type repr defines the representation of field elements as buffers (mutable arrays) of five 32-bit integers. It is marked as abstract, to protect the representation invariant from the rest of the code.

```
abstract type repr = buffer UInt32.t 5 (∗ 5-limb representation ∗)
val ‘_.[_] ‘: memory → repr → Ghost elem (∗ m.[r] is the value of r in m ∗)
val multiply: e0:repr → e1:repr → ST unit (∗ e0 := e0 ∗ e1 ∗)
  (requires (λ m → e0 ∈ m ∧ e1 ∈ m ∧ disjoint e0 e1))
  (ensures (λ m0 _ m1 → modifies {e0} m0 m1 ∧ m1.[e0] = m0.[e0] ∗ m0.[e1]))
```

Table 1 Performance in CPU cycles: 64-bit HACL⋆, 64-bit Sodium (pure C, no assembly), 64-bit OpenSSL (pure C, no assembly). All code was compiled using `gcc -O3` optimized and run on a 64-bit Intel Xeon CPU E5-1630. Results are averaged over 1000 measurements, each processing a random block of 2^{14} bytes; Curve25519 was averaged over 1000 random key-pairs.

Algorithm	HACL⋆	Sodium	OpenSSL
ChaCha20	6.17 cy/B	6.97 cy/B	8.04 cy/B
Salsa20	6.34 cy/B	8.44 cy/B	N/A
Poly1305	2.07 cy/B	2.48 cy/B	2.16 cy/B
Curve25519	157k cy/mul	162k cy/mul	359k cy/mul

Functions are declared with a series of argument types (separated by →) ending with an effect and a return type (e.g., Ghost elem, ST unit, etc.). Functions may have logical pre- and post-conditions that refer to their arguments, their result, and their effects on the memory. If they access buffers, they typically have a pre-condition requiring their caller to prove that the buffers are live in the current memory.

The term m.[r] selects the contents of a buffer r from a memory m; it is used in specifications only, as indicated by the Ghost effect annotation on the final arrow of its type on line 2. The multiply function is marked as ST, to indicate a stateful computation that may read, write and allocate state. In a computation type ST a (requires pre) (ensures post), a is the result type of the computation, pre is a predicate on the input state, and post is a relation between the input state, the result value, and the final state. ST computations are also guaranteed to not leak any memory. The specification of multiply requires that its arguments are live in the initial memory (m) and refer to non-overlapping regions of memory; it computes the product of its two arguments and overwrites e_0 with the result. Its post-condition states that the value of e_0 in the final memory is the field multiplication of its value in the initial memory with that of e_1, and that multiply does not modify any other existing memory location.

Implementing and proving that multiply meets its mathematical specification involves hundreds of lines of source code, including a custom, verified Bignum library in Low⋆ [26]. Using this library, we implement poly1305_mac and prove it functionally correct.

```
val poly1305_mac:
  tag:nbytes 16ul →
  len:u32 → msg:nbytes len{disjoint tag msg} →
  key:nbytes 32ul{disjoint msg key ∧ disjoint tag key} → ST unit
  (requires (λ h → msg ∈ h ∧ key ∈ h ∧ tag ∈ h))
  (ensures (λ h0 _ h1 → let r = Spec.clamp h0.[sub key 0ul 16ul] in
                        let s = h0.[sub key 16ul 16ul] in
                        modifies {tag} h0 h1 ∧
                        h1.[tag] == Spec.mac_1305 (encode_bytes h0.[msg]) r s))
```

Its specification above states that the final value of the 16 byte tag (h1.[tag]) is the value of Spec.mac_1305, a polynomial of the message and the key encoded as field elements.

Performance of HACL⋆. Besides the Poly1305 MAC, HACL⋆ provides functionally correct, side-channel resistant implementations of the ChaCha20 [22] and Salsa20 [4] ciphers, and multiplication on the Curve25519 elliptic curve [3].

After verification, F⋆ types and specifications are erased during compilation and the compiled code only performs efficient low-level operations. Indeed, after extraction to C by KreMLin, our verified implementations are very slightly but consistently faster than unverified C implementations of the same primitives in libsodium [24] and OpenSSL (Table 1).

3.2 Game-Based Cryptography

Going beyond functional correctness, we sketch how we use Low* to do security proofs in the standard model of cryptography, using "authenticated encryption with associated data" (AEAD) as a sample construction. AEAD is the main protection mechanism for the TLS record layer; it secures most Internet traffic.

AEAD has a generic security proof by reduction to two core functionalities: a stream cipher (such as ChaCha20) and a one-time-MAC (such as Poly1305). The cryptographic, game-based argument supposes that these two algorithms meet their intended *ideal functionalities*, e.g., that the cipher is indistinguishable from a random function. Idealization is not perfect, but is supposed to hold against computationally limited adversaries, except with small probabilities, say $\varepsilon_{\mathrm{ChaCha20}}$ and $\varepsilon_{\mathrm{Poly1305}}$. The argument then shows that the AEAD construction also meets its own ideal functionality, except with probability say $\epsilon_{\mathrm{Chacha20}} + \epsilon_{\mathrm{Poly1305}}$.

To apply this security argument to our implementation of AEAD, we need to encode such assumptions. To this end, we supplement our real Low* code with ideal F* code. For example, ideal AEAD is programmed as follows:

- encryption generates a fresh random ciphertext, and it records it together with the encryption arguments in a log.
- decryption simply looks up an entry in the log that matches its arguments and returns the corresponding plaintext, or reports an error.

These functions capture both confidentiality (ciphertexts do not depend on plaintexts) and integrity (decryption only succeeds on ciphertexts output by encryption). Their behaviors are precisely captured by typing, using pre- and post-conditions about the ghost log shared between them, and abstract types to protect plaintexts and keys.

The abstract type of keys and the encryption function for idealizing AEAD is below:

```
type entry = cipher * data * nonce * plain
abstract type key = { key: keyBytes; log: if Flag.aead then ref (seq entry) else unit }
let encrypt (k:key) (n:nonce) (p:plain) (a:data)  =
  if Flag.aead
  then let c = random_bytes cipher_len in k.log := (c, a, n, p) :: k.log; c
  else encrypt k.key n p a
```

A module Flag declares a set of abstract booleans (*idealization flags*) that precisely capture each cryptographic assumption. For every real functionality that we wish to idealize, we branch on the corresponding flag.

This style of programming heavily relies on the normalization capabilities of F*. At verification time, flags are kept abstract, so that we verify both the real and ideal versions of the code. At extraction time, we reveal these booleans to be false. The F* normalizer then drops the then branch, and replaces the log field with (), meaning that both the high-level, list-manipulating code and corresponding type definitions are erased, leaving only low-level code from the else branch to be extracted.

Using this technique, we verify by typing e.g. that our AEAD code, when using *any* ideal cipher and one-time MAC, perfectly implements ideal AEAD. We also rely on typing to verify that our code complies with the pre-conditions of the intermediate proof steps. As a consequence of our proof, we are forced to establish various intensional properties of our code, e.g., that our code does not reuse nonces (a common cryptographic pitfall); that it has no buffer overruns (a common pitfall of low-level programming); etc.

4 Ongoing Work

4.1 Verified Assembly Language and Safe Interoperability with C

While Low* and KreMLin provide reasonably efficient C-like implementations of cryptography, for the highest performance, cryptographic code often relies on complex hand-tuned assembly routines that are customized for individual hardware platforms. For this we have designed Vale, a new DSL that supports foundational, automated verification of high-performance assembly code. The Vale tool transforms annotated assembly programs into deeply embedded terms in a verification language, together with proofs that the terms meet their specifications. So far, we have used Dafny [17] as the embedding language for Vale and used this tool chain to verify the correctness and security of implementations of SHA-256 on x86 and ARM, Poly1305 on x64, and hardware-accelerated AES-CBC on x86. Several implementations meet or beat the performance of unverified, state-of-the-art cryptographic libraries.

In ongoing work, we have begun to use F* as the embedding language for Vale, and are mechanizing a formal model of interoperability between Low* and Vale. By defining the deeply embedded semantics of Vale in F* as a Low* interpreter for assembly language terms, we aim to show that invocations from C to assembly can be accounted for within a single semantics for both DSLs. A key challenge here is to reconcile Low*'s CompCert-like structured memory model with Vale's "big array of bytes" view of memory.

4.2 An Early Deployment of *Everest* within libcurl

Emphasizing the incremental deployment of our verified code, we have integrated our verified TLS record layer extracted from Low* to C, as well as Vale implementations in assembly, within a new version of miTLS [7], implemented in F*, covering TLS-1.2 and TLS-1.3. Our eventual goal is for miTLS to be implemented entirely in the Low* subset of F* and extracted solely to C, with functional correctness and security proofs. However, as of now, miTLS is only partially verified (the handshake being the main, remaining unverified component) and extracts to OCaml. However, already, by virtue of basic type safety, our partially verified version of miTLS provides safety against low-level attacks (e.g. HeartBleed) similar to other OCaml-based implementations of TLS [15].

Relying on OCaml's foreign function interface, we integrate miTLS extracted to OCaml with the verified TLS record layer extracted to C-and-assembly. Dually, we provide C bindings for calling into a miTLS layer which handles the socket abstraction, fragmenting buffers, etc. The result is a `libmitls.dll`, which we integrate within `libcurl`, a popular open-source library, used pervasively in many tools. This early integration allows us to use our verified software from within popular command line tools, like `git`, providing immediate benefits.

5 Parting Thoughts

A significant novelty of our proposed work is that we aim to replace security-critical components of existing software with provably correct versions. Our careful choice of the problem domain is crucial: verified OS kernels and compilers can only succeed by replacing the software stack or development toolchain; verified, standardized, security protocols, and HTTPS and TLS in particular, can be deployed within the existing ecosystem, providing a large boost to software security at a small overall cost.

With the emergence of TLS 1.3, most TLS implementers will rewrite their code from scratch, and the world will be faced with migrating towards brand new implementations. History tells us that widespread adoption of a new version of TLS may take almost an entire decade. Given a similar time line for the adoption of TLS 1.3, if we distribute *Everest* within 2–4 years in a form where the cost of switching to it is negligible, then we are optimistic that it stands a chance of widespread adoption.

Despite this once-in-a-decade opportunity, several challenges remain. How will verified code authored in advanced programming languages be maintained and evolved going forward? Distributing our code as well-documented, source-like C may help somewhat, but to evolve the code while maintaining verification guarantees will require continued support from the *Everest* team, as well as outreach and education. Will the software industry at large be able to appreciate the technical benefits of verified code? How can we empirically "prove" that verified software is better? One direction may be to deploy, standalone, small-TCB versions of *Everest* and to demonstrate it is resistant to practical attacks – this raises the possibility of deployments of *Everest* within fully verified stacks [16, 14, 13] or sandboxes [23].

References

1 Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 515–529. ACM, January 2017. URL: `https://www.fstar-lang.org/papers/dm4free/`, `doi:10.1145/3009837.3009878`.

2 Daniel J. Bernstein. The Poly1305-AES message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005.

3 Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

4 Daniel J Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008.

5 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Cătălin Hriţcu, Jonathan Protzenko, Tahina Ramanandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoué. Verified low-level programming embedded in F*. Preprint, 2016. `https://arxiv.org/abs/1703.00053`.

6 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoué. Implementing and proving the tls 1.3 record layer. Cryptology ePrint Archive, Report 2016/1178, 2016. `http://eprint.iacr.org/2016/1178`.

7 Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

8 Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

9 Barry Bond, Chris Hawblitzel, Jay Lorch, Rustan Leino, Bryan Parno, Ashay Rane, and Laure Thompson. Verifying high-performance cryptographic assembly code. `https://project-everest.github.io/papers/`, 2016.

10 J. Clark and P. C. van Oorschot. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In *2013 IEEE Symposium on Security and Privacy*, pages 511–525, May 2013. `doi:10.1109/SP.2013.41`.

11 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. URL: `http://dx.doi.org/10.1007/978-3-540-78800-3_24`, `doi:10.1007/978-3-540-78800-3_24`.

12 Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2382196.2382204`, `doi:10.1145/2382196.2382204`.

13 Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Berkeley, CA, USA, 2016. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=3026877.3026928`.

14 Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 165–181, Berkeley, CA, USA, 2014. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=2685048.2685062`.

15 David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 223–238, 2015.

16 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1629575.1629596`, `doi:10.1145/1629575.1629596`.

17 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1939141.1939161`.

18 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

19 Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012. URL: `http://hal.inria.fr/hal-00703441`.

20 Pierre Letouzey. A new extraction for Coq. In *Types for proofs and programs*, pages 200–219. Springer, 2002.

21 Christopher Meyer and Jörg Schwenk. Sok: Lessons learned from ssl/tls attacks. In *Revised Selected Papers of the 14th International Workshop on Information Security Applications – Volume 8267*, WISA 2013, pages 189–209, New York, NY, USA, 2014. Springer-Verlag New York, Inc. URL: `http://dx.doi.org/10.1007/978-3-319-05149-9_12`, `doi:10.1007/978-3-319-05149-9_12`.

22 Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. IETF RFC 7539, 2015.

23 Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions.

In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 665–681, New York, NY, USA, 2016. ACM. URL: `http://doi.acm.org/10.1145/2908080.2908113`, `doi:10.1145/2908080.2908113`.

24 The Sodium crypto library (libsodium). URL: `https://www.gitbook.com/book/jedisct1/libsodium/details`.

25 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016. URL: `https://www.fstar-lang.org/papers/mumon/`.

26 Jean Karim Zinzindohoué, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, 2016.

Domain-Specific Symbolic Compilation

Rastislav Bodik[1], **Kartik Chandra**[2],
Phitchaya Mangpo Phothilimthana[3], **and Nathaniel Yazdani**[4]

1 **University of Washington, Seattle, WA, USA**
`bodik@cs.washington.edu`
2 **Gunn High School, Palo Alto, CA, USA**
`kartikchandra@acm.org`
3 **University of California, Berkeley, CA, USA**
`mangpo@cs.berkeley.edu`
4 **University of Washington, Seattle, WA, USA**
`nyazdani@cs.washington.edu`

Abstract

A *symbolic compiler* translates a program to symbolic constraints, automatically reducing model checking and synthesis to constraint solving. We show that new applications of constraint solving require domain-specific encodings that yield orders of magnitude improvements in solver efficiency. Unfortunately, these encodings cannot be obtained with today's symbolic compilation.

We introduce *symbolic languages* that encapsulate domain-specific encodings under abstractions that behave as their non-symbolic counterparts: client code using the abstractions can be tested and debugged on concrete inputs. When client code is symbolically compiled, the resulting constraints use domain-specific encodings.

We demonstrate the idea on the first fully symbolic checker of type systems; a program partitioner; and a parallelizer of tree computations. In each of these case studies, symbolic languages improved on classical symbolic compilers by orders of magnitude.

1998 ACM Subject Classification D.2.2 [Software Engineering] Design Tools and Techniques, D.3.3 [Programming Languages] Language Constructs and Features

Keywords and phrases Symbolic evaluation, constraint solvers, program synthesis

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.2

1 Introduction

A symbolic compiler translates a program `p` to constraints that model its behavior [3, 18, 20]. The unknowns in the constraints typically represent the symbolic inputs to `p`, and the solution to the constraints is an input that induces a particular program behavior. For example, with a symbolic compiler and a solver, by just writing program `p`, we obtain a program checker – producing an input to `p` that leads to an assertion failure – for free.

The applications of symbolic compilation become even more interesting when the input to the program `p` is itself a program:

- If the program `p` is an interpreter, then constraint solving can find a program that forces the interpreter into a violation due to unsoundness in its type system. In Section 3, we explore finding such witnesses to unsoundness by symbolically compiling interpreters.
- If `p` is a type checker, then constraint solving performs type inference. In Section 4, we partition a program onto a many-core processor. We model this program transformation with a hardware-specific place type system, relying on symbolic compilation of type checkers to produce constraints for type inference.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 2; pp. 2:1–2:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- If `p` is an "execution runtime" for parallel programs, then constraint solving finds a parallel execution strategy, effectively parallelizing `p` for us. In Section 5, we synthesize parallel evaluators for attribute grammars by modeling the strategies as schedules and symbolically compiling interpreters of such schedules.

More formally, assume we have access to a solver `sol` that accepts a constraint ϕ and returns a solution, *i.e.,* a value x such that $\phi(x)$ holds. If no such x exists, then `sol` returns $\perp$. A symbolic compiler `sym` translates a program `p` into a logical formula ϕ that models the input-output semantics of `p`. It is convenient to think of a symbolic compiler as an execution inverter: `sym` accepts `p` and an output value y and produces a formula ϕ over the program input variable `x` such that the solution x to ϕ makes the program output y.

Model checking and program synthesis are two common applications of symbolic compilation. In bounded model checking, we want to compute a program input that leads to a failure. First, we modify the program so that failed assertions exit the program, returning a special value *fail*. The call $\texttt{sol}(\texttt{sym}(\texttt{p}, \textit{fail}))$ produces the failing input if one exists.

In inductive synthesis, we have a sketch program $\texttt{def sk}(\texttt{x}, \texttt{h}) = e$ where e uses an (unknown) function `h`. We want to find a function `f` such that substituting `f` for `h` gives `sk` the desired behavior. The behavior is often given with an input-output pair of values (x_0, y_0), *i.e.*, we want $\texttt{sk}(x_0, \texttt{f})$ to evaluate to y_0. In many settings, y_0 is simply *success* or *fail*. Symbolic compilation produces such a function `f` with the call $\texttt{sol}(\texttt{sym}(\texttt{sk}(x_0), y_0))$. The notation $\texttt{sk}(\texttt{x}_0)$ is a partially applied function `sk`, *i.e.*, a function of `h`. Note that to produce a function, the solver need not be second-order; the function `f` can be represented as a list of constants that define a derivation of the syntax of `f` from a suitable grammar.

Revisiting the three case studies clarifies the task of symbolic compilation:

- *Checking soundness of type systems.* We want to check the type system of an interpreter `int` that is composed of a type checker and an evaluator. Assume that the interpreter outputs *fail* when a program passes the type checker but fails in the evaluator. The call $\texttt{sol}(\texttt{sym}(\texttt{int}, \textit{fail}))$ then finds a program that is deemed type-safe but encounters a run-time violation. (We assume the interpreted programs have no parameters.) The benefit of using symbolic compilation is that one needs just an implementation of the interpreter. There is no need for fuzzers or tools that translate language semantics to constraints.
- *Program partitioning.* A spatial type system maps variables and operations to CPU cores, partitioning the program. If the typechecker has two parameters, a program `r` and the values of `r`'s type variables, then the call $\texttt{sol}(\texttt{sym}(\texttt{typechecker}(\texttt{r}), \textit{success})$ produces the type assignment to the program that partitions the program satisfying all program and hardware constraints checked by the type checker. Symbolic evaluation produces type constraints where unknowns are the type variables. Solving performs type inference. Formulating type inference as constraint solving is nothing new, of course. Our goal is to make this idea easier to apply by automatically obtaining high-performance type inference from a type checker.
- *Parallelizing tree computations.* We want to efficiently compute the attributes of a tree `t` defined by an attribute grammar `G`. The parallel tree evaluator may need to perform multiple tree traversals, some bottom-up, some top-down, some in-order, subject to value dependences in `G`. The evaluation strategy can be described with a schedule of traversals. The schedule language is defined with an interpreter that reads the grammar, the tree, and the schedule. Symbolic evaluation of the interpreter can give us a legal schedule for free with the call $\texttt{sol}(\texttt{sym}(\texttt{int}(\texttt{G}, \texttt{t}), \textit{success})$. Constraint solving sidesteps the error-prone process of analyzing the grammar and operationally constructing a valid schedule.

2 Architectures for Symbolic Compilation

We discuss three approaches for generating constraints. We first compare two existing architectures – constraint generators and general-purpose specializing symbolic compilers – and then introduce domain-specific symbolic compilers.

We describe the architectures by composing interpreters, compilers, and specializers. Borrowing the notation from [4], we summarize here their definitions:

- interpreter int : $[\![\mathtt{int}]\!](\mathtt{p}, \mathtt{x}) = [\![\mathtt{p}]\!]\ \mathtt{x}$
- compiler comp : $[\![[\![\mathtt{comp}]\!]\ \mathtt{p}]\!]\ \mathtt{x} = [\![\mathtt{p}]\!]\ \mathtt{x}$
- specializer s : $[\![[\![\mathtt{s}]\!]\ (\mathtt{p}, \mathtt{x_s})]\!]\ \mathtt{x_d} = [\![\mathtt{p}]\!](\mathtt{x_s}, \mathtt{x_d})$
- symbolic compiler sym : $[\![\mathtt{p}]\!]\ ([\![\mathtt{sol}]\!]\ ([\![\mathtt{sym}]\!]\ (\mathtt{p}, \mathtt{y}))) = \mathtt{y}$

2.1 Constraint generators

A generator gen reads a problem instance and produces constraints whose solution solves the problem instance. As our running example, consider the synthesis of schedules for parallel tree evaluation that we introduced above. The problem instance is an attribute grammar G and the call sol(gen(G)) produces the schedule for G.[1]

Notice that a constraint generator gen is not asked to invert a program, unlike the symbolic compiler sym. This frees the author of the generator to employ a clever problem-specific encoding. For example, Gulwani *et al.* phrased synthesis of loop-free programs as constraint-based synthesis of a network that connects available instructions [6]. Kuchcinski solved scheduling and resource assignment by modeling a system as a set of finite-domain variables [10]. Hamza *et al.* synthesized linear-time programs by converting an automaton recognizing the input/output relation [7].

On the other hand, since the generator receives only a problem instance but not the program to be inverted, the semantics of generated constraints must entirely come from the author of the generator. Consider again the synthesis of schedules: sym received the schedule interpreter, which it can use to automatically produce constraints that encode schedules. In contrast, the semantics of schedules must be hard-coded into gen by the programmer.

Our running example shows why implementing generators is challenging. The programmer needs to wrangle the semantics of three languages – the language of attribute grammars AG, the language of schedules Sch, and the constraints language Φ – reasoning across a four-step indirection:

1. The programmer reads the specifications of the three languages and writes a constraint generator gen.
2. The generator gen reads the grammar G and outputs a constraint ϕ.
3. The solver sol reads ϕ and produces a solution σ.
4. The solution σ indirectly encodes a schedule $\mathtt{s} \in \mathtt{Sch}$.
5. The schedule s evaluates a tree according to the input grammar $\mathtt{G} \in \mathtt{AG}$.

The programmer must ensure that the generator written in Step 1 produces a schedule that in Step 4 correctly encodes, say, in-order traversals and in Step 5 evaluates the tree in accordance with the attribute grammar semantics. This reasoning may explain why in

[1] The solution to constraints must be typically converted back to the problem domain. For example, if using SAT constraints, a Boolean vector that solves the SAT problem is translated to a program in the scheduling language. This code-generation problem is important but we ignore its automation in this paper.

our previous work on synthesizing schedules, we failed to fully debug our generator once the schedule language became moderately sophisticated.

2.2 General-Purpose Specializing Symbolic Compilers

This is the architecture of Sketch [17] and to a large extent Rosette [20]. The architecture has three components and relies heavily on specialization:

1. $\texttt{int} : (D \to D)_L \to D \to D$, an interpreter implemented in a metaprogramming language L_m. The interpreter implements the language L of the input program `p`. It accepts the program $\texttt{p} : (D \to D)$ and `p`'s input, producing `p`'s output.
2. $\texttt{s} : (D \to D \to D)_{L_m} \to D \to (D \to D)_{L_s}$, a specializer of programs in L_m producing programs in L_s. In particular, `s` will specialize `int` with respect to `p`, producing a residual program $\texttt{int}_\texttt{p}$.
3. $\texttt{xlate} : (D \to D)_{L_s} \to D \to \Phi$ translates a symbolic program to the language of constraints Φ. `xlate` also receives the output value $y \in D$ and produces a formula $\phi(\texttt{x})$ that is satisfied iff `p(x)` outputs y.

The symbolic compiler is thus $\texttt{sym}(\texttt{p}, \texttt{y}) \triangleq \texttt{xlate}(\texttt{s}(\texttt{int}, \texttt{p}), \texttt{y})$. In Rosette, L_m is a subset of Racket maintaining many metaprogramming facilities of Racket; L_s is the symbolic expression language; and C can be one of several subsets of SMT languages, such as the bitvector language. Note that `s` and `xlate` are part of the framework, while `int` is developed by the user.

The core of symbolic evaluation happens in the specializer which explores all paths of the program, producing a functional residual program that reflects the shape of the final constraints. The translator `xlate` performs algebraic optimizations followed by local code generation from the residual program to the constraint language.

The downside of this architecture is that symbolic compilation must typically follow the forward symbolic execution that merges constraints under their path conditions [9]. This algorithm may suffer from path explosion and does not lend itself to constraints other than SAT or SMT. Thus, integer linear programming (ILP) constraints – often domain-specific and highly efficient – are often the constraints of choice produced by constraint generators.

2.3 Domain-specific symbolic compilers

We modify the specializing architecture by introducing a new abstraction for implementing the interpreter of L. This interpreter, $\texttt{int}_{Ld}$, now has two parts:

1. $\texttt{int}_L$, an interpreter of L implemented in the *domain-specific symbolic* language L_d.
2. $\texttt{int}_d$, an interpreter of L_d implemented in a metaprogramming language L_m.

The symbolic compiler pipeline is now $\texttt{xlate}(\texttt{s}(\texttt{int}_{Ld}, \texttt{p}), \texttt{y}))$. Ideally, the interpreter $\texttt{int}_d$ meets two informally stated properties: (1) symbolic evaluation of $\texttt{int}_d$ produces easier-to-solve constraints; and (2) symbolic evaluation of $\texttt{int}_d$ is faster than that of `int`, for example because L_d reduces path explosion.

This paper shows that domain-specific symbolic compilers can be implemented as a library on top of a classical symbolic compiler such as Rosette [19]. The library provides a *symbolic language* that implements the domain-specific encoding while hiding the encoding from both the programmer and the underlying symbolic compiler.

In Section 3, we check type systems for soundness errors. We introduce a *Bonsai tree* that serves as the symbolic input into a type checker and an interpreter, which are implemented on top of the Bonsai library. Symbolic evaluation of the two components produces constraints that encode a space of abstract syntax trees (ASTs). The solution is a witness: a tree that succeeds in the type checker but fails in the interpreter. The Bonsai tree has the usual interface but internally produces a special encoding that allows symbolically evaluating the interpreter on trees that are not necessarily syntactically correct or type correct. This is key to finding witnesses, for the first time, without enumerating or sampling the program space, allowing us to compute the witness for a tricky soundness bug [1].

In Section 4, we partition a program onto a many-core processor. Mapping of program operations to cores is modeled with a place type system ensuring that each code fragment fits into its core. Partitioning is thus type inference. To infer types, we symbolically evaluate the type checker with respect to a program whose type variables are symbolic. We design a symbolic language for querying properties of the symbolic location of a computation. Under this abstraction, we switch from the standard SMT encoding to an ILP encoding. The resulting ILP encoding solves previously inaccessible partitioning problems.

Finally, in Section 5, we synthesize parallel tree programs as used in page layout and data visualizations. The programs are formalized as schedules for evaluation of attribute grammars. We design a *symbolic trace language*, an abstraction for writing interpreters of such schedules. Under this abstraction, we can (1) sidestep the expensive symbolic state that is maintained by the standard symbolic evaluator and (2) switch from ensuring that all dependences are met to ensuring that all anti-dependences are avoided.

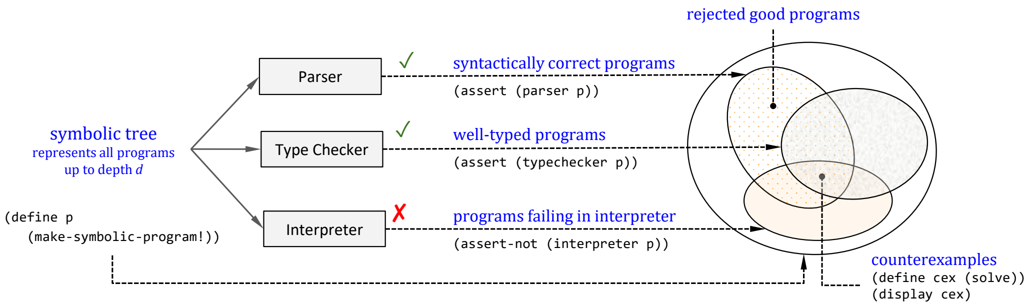
3 Checking Type Systems with Bonsai Trees

Model checking of type systems. Bonsai uses model checking to search for soundness errors in type systems. The user provides a typechecker and an interpreter for their language, and Bonsai searches for a *counterexample* program that passes the typechecker while causing the interpreter to crash. If such a counterexample can be found, then it is evidence of a soundness bug in the type system. Furthermore, such a counterexample provides helpful feedback for the user to understand and fix the bug. On the other hand, if *no* counterexample can be found, the user has some assurance that the typechecker is sound.

The most common existing typechecker-checking technique is *fuzzing*. A fuzzer generates random terms and uses them to test a typechecker and interpreter. Fuzzers may sample from the space of syntactically-correct terms or the space of well-typed terms; however, in both cases, the probability of generating a counterexample by chance is extremely low. Thus, fuzzers often need hours or days of guessing to find even simple type checker bugs (an example of a "simple" bug is assigning `cons` the return type `a` instead of `Listof a`).

Bonsai can be regarded as a final successor to typechecker fuzzing: rather than randomly sampling from the space of syntactically-correct or well-typed terms, Bonsai symbolically compiles an executable language specification to constraints, and then utilizes the backward reasoning of a constraint solver to sample *directly* from the space of counterexamples. This makes Bonsai much more efficient than traditional fuzzers: Bonsai finds the above bug in just 1.3 seconds compared to hours or days needed by fuzzers.

Bonsai consists of the algorithm shown in Figure 1. First, Bonsai initializes a symbolic representation of A, the set of all trees up to some maximum size m. Next, it computes symbolic representations of the subsets of A that (a) are syntactically valid; (b) pass the typechecker; (c) fail in the interpreter. Finally, it asks the solver to find a tree in the intersection of these three sets. If such a tree exists, it represents a counterexample.



```
(define p
   (make-symbolic-program!))
```

```
(assert (parser p))
(assert (typechecker p))
(assert-not (interpreter p))
```

```
(define cex (solve))
(display cex)
```

Figure 1 Bonsai performs three independent symbolic evaluations, interestingly executing the interpreter also on trees that are both syntactically and type-incorrect.

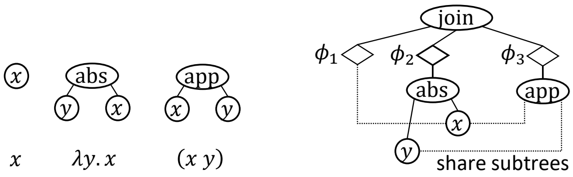
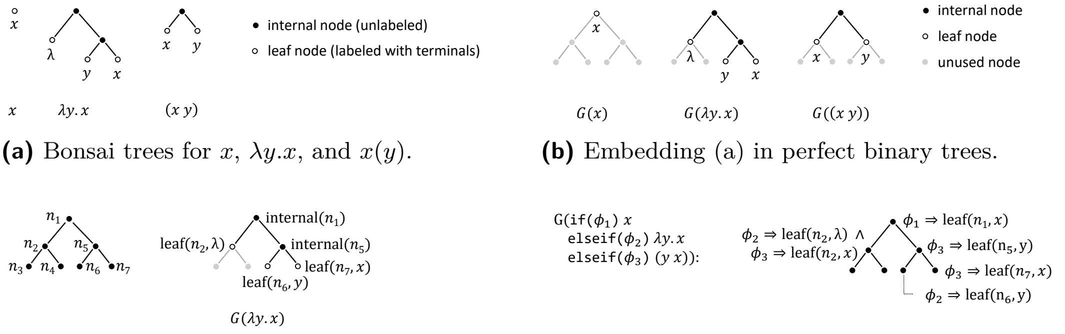


Figure 2 The classical symbolic representation of a set of trees grows very quickly even when small trees are merged, even if their subtrees are shared.


x
λ
y
x
x
y
● internal node (unlabeled)
○ leaf node (labeled with terminals)
x
$\lambda y.x$
$(x\ y)$
x
λ
y
x
x
y
● internal node
○ leaf node
● unused node
$G(x)$
$G(\lambda y.x)$
$G((x\ y))$
n_1
n_2
n_5
n_3
n_4
n_6
n_7
internal(n_1)
leaf(n_2, λ)
internal(n_5)
leaf(n_7, x)
leaf(n_6, y)
$G(\lambda y.x)$
G(if(ϕ_1) x
elseif(ϕ_2) $\lambda y.x$
elseif(ϕ_3) (y x)):
$\phi_1 \Rightarrow$ leaf(n_1, x)
$\phi_2 \Rightarrow$ leaf(n_2, λ) ∧
$\phi_3 \Rightarrow$ leaf(n_2, x)
$\phi_3 \Rightarrow$ leaf(n_5, y)
$\phi_3 \Rightarrow$ leaf(n_7, x)
$\phi_2 \Rightarrow$ leaf(n_6, y)

(a) Bonsai trees for x, $\lambda y.x$, and $x(y)$.

(b) Embedding (a) in perfect binary trees.

(c) Symbolic encoding of the tree for $\lambda y.x$.

(d) Merging three trees under path conditions ϕ_i.

Figure 3 A stepwise overview of the Bonsai tree encoding.

General-purpose symbolic evaluation. To perform symbolic evaluation of a type checker, we need to create a symbolic abstract syntax tree that represents A, the set of concrete abstract syntax trees. The standard approach would produce trees such as those shown in Figure 2, where sets of trees are merged by creating symbolic choices to select among potential children at each node. This merged symbolic tree could then be supplied to an existing type checker and interpreter by symbolically evaluating them on the tree.

Though this classical approach would work in principle, it fails to scale to trees that are deep enough to explore large programs. Furthermore, each operation on such a symbolic tree causes its representation to grow even more complex, and the large data structures prevent scalable symbolic execution. Thus, the challenge here is to optimize the speed of symbolic compilation, rather than the speed of solving.

Domain-specific symbolic evaluation. Bonsai solves this problem by creating a new encoding for sets of trees that limits growth by efficiently merging trees within the set. This "Bonsai tree" is compatible with a standard symbolic evaluator, and language engineers can use this symbolic tree nearly as if it were a concrete tree. Figure 3 gives a stepwise explanation of the Bonsai symbolic tree, starting from concrete Bonsai trees for three program terms (a). These concrete trees are embedded in a perfect binary tree (b). The embedding is represented with two predicates for each node: the first determines whether the node is internal or a leaf; the second determines the terminal for leaves (c). By allowing the predicates to be symbolic expressions, a single tree can represent multiple Bonsai trees. In (d), we show how symbolic Bonsai trees arise; here we merge three trees at an if-statement.

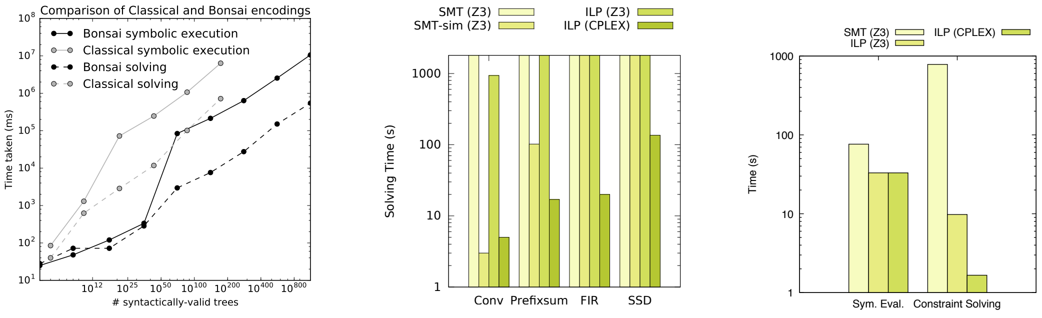
Despite having a different underlying representation, Bonsai trees can be easily manipulated by programmers, just as if they were concrete trees. Bonsai provides utilities for creating, modifying, and pattern-matching with symbolic trees, allowing programmers to implement typecheckers and interpreters without having to focus on the details of symbolic execution.

Evaluation. Figure 4a shows an empirical comparison between the classical and Bonsai encodings. Symbolic terms of various sizes were executed with identical typecheckers and interpreters, varying only the underlying encoding. Bonsai's encoding was consistently several orders of magnitude faster. In under an hour, Bonsai explores programs much larger than counterexamples created by human experts who report soundness bugs, thus providing users with a margin of assurance.

Bonsai has reproduced many soundness bugs in a variety of languages, notably including (1) unsound argument covariance in a model of Java, and (2) a subtle issue with Scala's existential types, discovered in 2016 by Nada Amin and Ross Tate [1]. Slight modifications to the algorithm also allow users to ask intriguing new questions that fuzzers cannot easily answer, such as "On what programs do typecheckers t_1 and t_2 disagree?" or "Does my typechecker reject programs that don't fail?" Finally, by making the *typechecker* symbolic, Bonsai can synthesize suggestions for how to fix an unsound type system.

4 Program Partitioning

The Code Partitioning Problem. Compilers for fine-grain many-core architectures must partition the program into tiny code fragments mapped onto physical cores. Chlorophyll [15, 16] is a language for GA144, an ultra-low-power processor with 144 tiny cores [5]. The Chlorophyll type system ensures that no fragment overflows the 64-word capacity of its core.



(a) Bonsai tree **(b)** Program partitioning **(c)** Tree scheduling

Figure 4 Experimental evaluations.

Listing 1 Original type checker, ensuring that code fragments fit into cores.

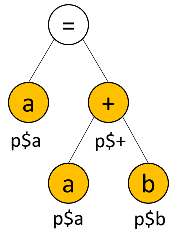
```
1  (define cores-space (make-vector n-cores 0)) ; space used up on each core
2  (define (inc-space p size)
3              (vector-set! cores-space p (+ (vector-ref cores-space p) size)))
4
5  ; Increase code size whenever core p sends a value to core r.
6  (define (comm p r) (when (not (= p r)) (begin (inc-space p 2) (inc-space r 2))))
7
8  ; Increase code size for broadcast communication from p to ps. ps may contain duplicates.
9  (define (broadcast p ps)
10   (define remote-ps (length (remove p (unique ps))));# of unique cores in ps excluding p
11   (inc-space p (* 2 remote-ps))    ; space used in the sender core
12   (for ([r ps]) (inc-space r 2))) ; space used in the receiver cores
13
14 (define (count-space node) ; Count space needed by an AST node.
15   (cond
16    [(var? node) (inc-space (place-type node) 1)]
17    [(binexpr? node) ; The inputs to this operation come from binexpr-e1 and binexpr-e2.
18     (define p (place-type node))
19     (inc-space p (size node)) ; space taken by the operation
20     (comm (place-type (binexpr-e1 node)) p)  ; Add space for communication code when
21     (comm (place-type (binexpr-e2 node)) p)] ; operands come from other cores.
22    [(if? node)
23     ; If is replicated on all cores that run any part of if's body.
24     ; We omit inc-space here.
25     ; The condition result is broadcast to all cores used in if's body.
26     (broadcast (place-type (if-test node))
27                (append (all-cores (if-then node)) (all-cores (if-else node))))]
28   ...))
29
30 (tree-map count-space ast)
31 (for ([space cores-space]) (assert (< space core-capacity)))
32 (minimize (apply + cores-space)) ; used during inference only
```

Each variable and operation have a *place type* whose value is a core ID. The type checker in Listing 1 computes the code size of each fragment. The `tree-map` function traverses a program AST in the post-order fashion and applies the function `count-space` on each node in the AST (line 28). The checker accumulates the space taken by each node (e.g. a `binexpr` node on line 18), and space occupied by communication code, for both one-to-one communication (e.g. sending operand values to an operator on lines 19–20) and broadcast communication (e.g. sending a condition result to all nodes in the body of `if` on lines 24–25).

Automatic Program Partitioning as Type Inference. When a program omits some place types, the compiler infers them, effectively partitioning the program. Chlorophyll implements the type inference using Rosette [19, 20], which symbolically evaluates the type checker in Listing 1 with respect to the program. The type checker needs no changes; we only need to

Listing 2 Type checker in resource language, producing ILP constraints.

```
 1 (define n (make-parameter #f)) ; a parameter procedure for dynamic binding
 2 (define (comm p r) (inc-space (n) (* 2 (+ (different? p r (n)) (different? r p (n))))))
 3 (define (broadcast p ps)
 4   (inc-space (n) (* 2 (+ (count-different p ps (n)) ;; space used in the sender core
 5                         (different? ps p (n)) )))) ;; space used in the receiver cores
 6
 7 ;; the function count-space is changed in one place (see text); inc-space is unchanged
 8
 9 (for ([i n-cores]) (parameterize ([n i]) (tree-map count-space ast)))
10 (for ([space cores-space]) (assert (< space core-capacity)))
11 (minimize (apply + cores-space))
```



(a) Example program AST. Each node is annotated with its place type below. The yellow nodes are the ones that have been interpreted.

```
(inc-space p$a 1) ;; Line 15, node = a [def]
(inc-space p$a 1) ;; Line 15, node = a [use]
(inc-space p$b 1) ;; Line 15, node = b
(inc-space p$+ 1)  ;; Line 18, node = +
(comm p$a p$+)     ;; Line 19, comm a -> +
(comm p$b p$+)     ;; Line 20, comm b -> +
```

(b) Residual type checking program after traversing the yellow nodes in the AST on the left (post-order). The line numbers in the comments indicate where the expressions come from from Listing 1.

Figure 5 Running example of program partitioning

initialize the (unknown) place types in the program to symbolic values (i.e., `p$0, p$1, ...`).

Figure 5a shows an example of a program AST with unknown places. Each node in the AST is annotated with its symbolic place type. Figure 5b shows the conceptual partially-evaluated type checker after checking the yellow nodes in the example AST; concrete expressions are fully evaluated, and the expressions with symbolic variables remain. After we symbolically evaluate the residual type checker in Figure 5b, we obtain `cores-space` shown in Figure 6a. Rosette then uses Z3 to solve the generated SMT constraints on `cores-space` (line 29 of Listing 1) and minimize the total code size (line 30 of Listing 1).

Hence, we obtain our type inference just by implementing a type checker. The development process requires little effort, but the type inference is slow at inferring place types.

Symbolic Evaluation to ILP Constraints. It is known that partitioning and scheduling problems can be solved efficiently using ILP [21, 14, 13, 8]. However, if we follow the standard way of generating ILP constraints, we will not be able to simply turn type checking into type inference. Here, we turn to our key idea and introduce a symbolic language that will generate ILP constraints. The programmer implements the type checker as before but in our *resource language*. The programmer is prohibited from writing programs with symbolic path conditions because these path conditions create non-linear constraints. If a program contains a symbolic path condition, the compiler will raise an exception. Resource language is embedded in Rosette. It provides additional operations: `mapped-to?`, `different?`, and `count-different`, as described in Table 1.

We make four minimal changes to our original type checker, shown in Listing 2. First, we traverse the AST once for every core (line 9). Each iteration i is responsible for accumulating space used in core i. Second, in the function `count-space`, we change the expression to increase the size of core p by the size of the operation of *node* from `(inc-space p (size node))` to `(inc-space (n) (* (size node) (mapped-to? p (n))))`. The previous call produces a non-

```
> cores-space
#(;; core 0
  (+ (ite (= p$a 0) 1 0) ;; a [def]
     (ite (= p$a 0) 1 0) ;; a [use]
     (ite (= p$b 0) 1 0) ;; b
     (ite (= p$+ 0) 1 0) ;; +
     (ite
       (and (or (= p$+ 0) (= p$a 0))
            (! (= p$+ p$a)))
       2 0)                ;; a -> +
     (ite
       (and (or (= p$+ 0) (= p$b 0))
            (! (= p$+ p$b)))
       2 0))               ;; b -> +

  ;; core 1
  (+ ...)
)
```

(a) Original symbolic expression generated from the original type checker (Listing 1)

```
> cores-space
#(;; core 0
  (+ (* 1 Mpn(p$a,0)) ;; a [def]
     (* 1 Mpn(p$a,0)) ;; a [use]
     (* 1 Mpn(p$b,0)) ;; b
     (* 1 Mpn(p$+,0)) ;; +
     (* 2 Remote_prn(p$+,p$a,0)) ;; a -> +
     (* 2 Remote_prn(p$a,p$+,0))
     (* 2 Remote_prn(p$+,p$b,0)) ;; b -> +
     (* 2 Remote_prn(p$b,p$+,0)))
  ;; core 1
  (+ ...))

> (asserts) ;; global assertions
((and (<= 0 Mpn(p$a,0)) (>= 1 Mpn(p$a,0)))
 (and (<= 0 Mpn(p$a,1)) (>= 1 Mpn(p$a,1)))
 (= 1 (+ Mpn(p$a,0) Mpn(p$a,1))) ;; a
 ...
 (<= 0 Remote_prn(p$+,p$a,0))     ;; a -> +
 (>= 1 Remote_prn(p$+,p$a,0))
 (>= Remote_prn(p$+,p$a,0)
     (- Mpn(p$+,0) Mpn(p$a,0)))
 ...
)
```

(b) ILP symbolic expression generated from the modified type checker (Listing 2)

Figure 6 Symbolic expression of space occupied in each core after running a type checker on the yellow nodes in the example AST (Figure 5a).

Table 1 Description of resource language operations. Sym/conc stands for symbolic or concrete.

Function	Type	Description

`(mapped-to? p n)`	`p: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if place p is core n (i.e. $p = n$), otherwise returns 0

`(different? p r n)`	`p: sym/conc integer`
	`r: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if places p and r are different, and place p is core n (i.e. $(p \neq r) \wedge (p = n)$), otherwise returns 0

`(different? ps r n)`	`ps: list of sym/conc integers`
	`r: sym/conc integer`
	`n: concrete integer`
	`return: sym/conc integer`
		returns 1 if there is at least one place p in ps such that $(p \neq r) \wedge (p = n)$, otherwise returns 0

`(count-different p rs n)`	`p: sym/conc integer`
	`rs: list of sym/conc integers`
	`n: concrete integer`
	`return: sym/conc integer`
		returns a number of unique places in rs that differ from p if place p is core n, otherwise returns 0

linear equation because the first argument p, which is symbolic, to `inc-space` is used as a path condition. Third, we avoid symbolic path conditions inside the function `comm` by using `(different? p r (n))` to compute the size of code for sending data at core (n), and similarly for receiving data. Last, in the function `broadcast`, we utilize `count-different` to compute space taken by code for broadcasting a value to a set of cores.

Implementation. Table 2 details the implementation of the additional operations provided by our symbolic language. Under the abstraction, `(mapped-to? p n)` creates symbolic variables $M_{pn}(p, n')$ for all $n' \in N$ – where N is a set of values that p can take – and returns $M_{pn}(p, n)$;

Table 2 Implementation of resource language operations. $\mathit{sum}^{\dagger}$ and $\mathit{offset}^{\dagger}$ are temporary variables.

Function → return	**Created Variables**	**Additional Assertions**
`(mapped-to? p n)` $\rightarrow M_{pn}(p,n)$	$\forall n' \in N, M_{pn}(p,n')$	$\forall n' \in N, 0 \leq M_{pn}(p,n') \leq 1$ $\sum_{n' \in N} M_{pn}(p,n') = 1$
`(different? p r n)` $\rightarrow \mathit{Remote}_{prn}(p,r,n)$	$\mathit{Remote}_{prn}(p,r,n)$	$0 \leq \mathit{Remote}_{prn(p,r,n)} \leq 1$ $\mathit{Remote}_{prn(p,r,n)} \geq M_{pn}(p,n) - M_{pn}(r,n)$
`(different? p rs n)` $\rightarrow \mathit{Remote}_{prsn}(p,rs,n)$	$\mathit{Remote}_{prsn}(p,rs,n)$	$0 \leq \mathit{Remote}_{prsn(p,rs,n)} \leq 1$ $\forall r \in rs, \mathit{Remote}_{prsn}(p,rs,n) \geq$ $\mathit{Remote}_{prn}(p,r,n)$
`(count-different p rs n)` $\rightarrow \mathit{Count}_{prsn}(p,rs,n)$	$\mathit{Count}_{prsn}(p,rs,n)$ $\forall n' \in N, M^{*}_{rsn}(n')$	$\forall n \in N, 0 \leq M^{*}_{rsn}(n) \leq 1$ $\forall n \in N, r \in rs, M^{*}_{rsn}(n) \geq M_{pn}(r,n)$ $\mathit{sum}^{\dagger} = \sum_{n' \in \{N - \{n\}\}} M^{*}_{rsn}(n')$ $\mathit{offset}^{\dagger} = (M_{pn}(p,n) - 1) \times \mathit{MAX}_{INT}$ $\mathit{Count}_{prsn}(p,rs,n) \geq 0$ $\mathit{Count}_{prsn}(p,rs,n) \geq \mathit{sum}^{\dagger} + \mathit{offset}^{\dagger}$

$M_{pn}(p,n) = 1$ if $p = n$, and $M_{pn}(p,n) = 0$ otherwise. Since p can be mapped to only one value, the function adds the constraint $\sum_{n' \in N} M_{pn}(p,n') = 1$ into the global list of assertions. `(different? p r n)` creates and returns a variable $\mathit{Remote}_{prn}(p,r,n)$, as well as adds $\mathit{Remote}_{prn(p,r,n)} \geq M_{pn}(p,n) - M_{pn}(r,n)$ and $0 \leq \mathit{Remote}_{prn(p,r,n)} \leq 1$ to the global list of assertions. Note that $\mathit{Remote}_{prn(p,r,n)}$ can be either 0 or 1 when $p = r$, which is not what we want. However, this equation is valid if $\mathit{Remote}_{prn(p,r,n)}$ is (indirectly) minimized, so it is 0 when $p = r$ as we expect. The validity check happens when `minimize` is called.

Our approach requires no change in Rosette's internals. The additional operations simply generate Rosette assertions. We implement our custom `minimize` function, which performs the validity check before calling Rosette's `minimize`.

Figure 6b show the symbolic expression of `cores-space` along with additional assertions after symbolically executing the modified type checker on the yellow nodes of the AST in Figure 5a. Notice that the new expression is linear, while the original one is not.

Evaluation. The ILP encoding produced by our abstraction solves problems inaccessible to the SMT-based partitioner, and it is faster than the SMT encoding optimized for the domain of partitioning problems (namely, flattening deeply nested `ite` expressions). Figure 4b shows the median time to partition four benchmarks across three runs. We set the timeout to 30 minutes. In summary, SMT always timed out; domain-optimized SMT constraints solved half of the benchmarks; the ILP encoding solved all benchmarks.

5 Synthesis of Parallel Tree Programs

Attribute Grammars and Static Scheduling. Tree computations such as document layout for data visualization or CSS are naturally specified as attribute grammars [12]. For the sake of efficiency, high-performance layout engines in web browsers schedule these tree computations statically, by assigning the statement that computes an attribute to a predetermined position in a sequence of tree traversals. Static scheduling avoids the overhead of determining dynamically when an attribute is ready to be computed.

We express a static schedule for an attribute grammar as a program in a domain-specific language of *tree traversal schedules*, L_S. A schedule consists of tree traversal passes, each

Listing 3 The original interpreter of tree traversal schedules, $\texttt{int}_S$.

```
1  (define (int_S G t s)
2    (match s
3      [(seq s1 s2)
4       (int_S G t s1)
5       (int_S G t s2)]
6      [(par s1 s2)
7       ; check data independence of forward
8       ; and backward orders
9       (int_S G (copy t) (seq s2 s1))
10      (int_S G t        (seq s1 s2))]
11     [(pre visits)
12      (preorder (visitor G visits) t)]
13     [(post visits)
14      (postorder (visitor G visits) t)]))
15
16 (define ((visitor G visits) node)
17   (let ([class (get-class G node)])
18     (for ([slot (get-slots visits class)])
19       (eval class node slot))))
20
21 (define (eval class node slot)
22   (let* ([rule (get-rule class slot)]
23          [attr (target node rule)])
24     (for ([dep (get-deps node rule)])
25       (assert (ready? dep)))
26     (assert (not (ready? attr)))
27     (set-ready! attr)))
```

Listing 4 The interpreter $\texttt{int}_{ST}$, written with the symbolic trace language.

```
(define (int_ST G t s)
  (match s
    [(seq s1 s2)
     (int_ST G t s1)
     (int_ST G t s2)]
    [(par s1 s2)
     (parallel
       (int_ST G t s1)
       (int_ST G t s2))]
    [(pre visits)
     (preorder (visitor G visits) t)]
    [(post visits)
     (postorder (visitor G visits) t)]))

(define ((visitor G visits) node)
  (let ([class (get-class G node)])
   (for ([slot* (get-slots visits class)])
      (fork ([slot slot*])
        (eval class node slot)))))

(define (eval class node slot)
  (let* ([rule (get-rule class slot)]
         [attr (target node rule)])
    (for ([dep (get-deps node rule)])
      (read dep))
    (write attr)
    (step)))
```

of which executes statements from the attribute grammar. For instance, the schedule `post{ Inner{w, h}, Leaf{w, h} } ; pre{ Inner{x, y}, Leaf{x, y} }` performs a post-order traversal computing `w` then `h` at both `Inner` and `Leaf` nodes and then performs a pre-order traversal computing `x` then `y` at both `Inner` and `Leaf` nodes. A statement to execute is indicated by the name of the target attribute, since attributes and statements to compute them correspond one-to-one.

Listing 3 presents a definitional interpreter for the scheduling language L_S. The interpreter checks the correctness of a schedule on a given input tree. Among the checks are the absence of reads from uninitialized attributes (line 25) and single assignment (line 26), which together ensure that data dependencies are satisfied.

Schedule Synthesis. To synthesize a legal schedule, we first define the space of candidate schedules by creating a partially symbolic schedule, such as `post{ Inner{`$??_1$`, `$??_2$`}, Leaf{`$??_3$`, `$??_4$`} } ; pre{ Inner{`$??_5$`, `$??_6$`}, Leaf{`$??_7$`, `$??_8$`} }`, where $??_i$ indicates a symbolic choice ranging over the statements from the relevant node class (*e.g.*, $??_1$ ranges over the statements for `Inner` nodes). This schedule is desugared to a schedule-generating function $\texttt{sch} : D^8 \to L_{Sch}$ whose parameters control the symbolic choices $??_i$.

Note that the schedule is partly concrete; it specifies two concrete traversals, leaving symbolic only the *slots* in the node visitors. This limited symbolic nature simplifies symbolic compilation. To consider other traversal patterns, we can apply a standard technique for prioritized enumeration of sketches [2].

General-Purpose Symbolic Evaluation. With the schedule-generating function `sch` in hand, the call $\texttt{sol}(\texttt{sym}(\texttt{int}_S(\texttt{G}, \texttt{t}) \circ \texttt{sch}, \mathit{success}))$ synthesizes the slots for the partially concrete schedule correct on a tree `t`. When evaluating a symbolic choice $??_i$, symbolic evaluation considers each alternative concrete statement (line 18 in Listing 3), generates the constraints stating that the dependencies are ready and the target has not been computed (lines 25 and

Listing 5 Example L_T program.

```
(define ??1 (choose "x := y + z" "y := 2 * x" "x := 3"))
(define ??2 (choose "x := y + z" "y := 2 * x" "x := 3"))
(define ??3 (choose "x := y + z" "y := 2 * x" "x := 3"))
(define x (alloc))
(define y (alloc))
(define z (alloc))

(for ([??i (list ??1 ??2 ??3)])  ; execute a program with three slots
  (fork ([stmt ??i])  ; for each possible statement in this slot
    (let ([var (lookup (lhs stmt))]
          [expr (rhs stmt)])
      (for ([ref (refs expr)])  ; for all locations in the right-hand side
        (read ref))
      (write var)
      (step))))
```

Table 3 Operations of the symbolic trace language L_T (each returning `(void)` unless noted otherwise), where `host` refers to the pure subset of the host language.

Operation	Type	Description
`(choose`$v_1 \ldots v_n$`)`	v_i`: concrete value` `return: symbolic choice`	returns a symbolic choice from the given values, to construct a program hole ??
`(alloc)`	`return: concrete location`	returns a fresh concrete location
`(read l)`	`l: concrete location`	logs a read from the given location
`(write l)`	`l: concrete location`	logs a write to the given location
`(step)`		advances the program to the next statement
`(fork ([`x`c])`e`)`	x`: variable in host` `c: symbolic choice` e`: expression in host`	evaluates e for each concrete alternative of `c`, bound to x, under an appropriate guard
`(parallel`$e_1 e_2$`)`	e_1`: expression in host` e_2`: expression in host`	evaluates e_1 then e_2 while checking for conflicting usage of locations

26), sets the target attribute as ready, updates the program state (line 27), and then merges alternative states.

The constraints resulting from this evaluation present a challenge for the SMT solver. The symbolic state encodes whether a concrete attribute is ready at a given execution step as a function of symbolic choices (*i.e.*, which statement goes into which slot). We hypothesize that this state formulation prevents the solver to learn from failed guesses: if placing statement s_1 before s_2 leads to a dependence violation, the solver will happily try to place s_1 before s_2 into some other pair of slots.

Domain-Specific Symbolic Evaluation. To make constraint solving more efficient, we express the interpreter of schedules in the *symbolic trace language* L_T. The syntax of this language is summarized in Table 3, and a small example program in L_T is shown in Listing 5. The new version of the interpreter, `int`$_{ST}$, is in Listing 4.

The symbolic trace language understands only dependency relationships carried through *locations*, write-once memory objects with fully abstract contents. A location `l` is generated with `(alloc)` and used with `(read l)` and `(write l)`. In the context of attribute grammars, a location corresponds to a particular attribute somewhere in the tree.

As a trace program executes, we generate efficient ILP constraints for these correctness conditions:

Listing 6 Excerpt of the residual program in L_T from partial evaluation of int$_{ST}$ with respect to the symbolic schedule `sch`, showing the call to `(eval Inner root` $??_1$`)`.

```
; construct the symbolic schedule
(define $??_1$
  (choose "self.x := 0" ; guard = $b_{1,x}$
          "self.y := 0" ; guard = $b_{1,y}$
          "self.w := left.w + right.w" ; guard = $b_{1,w}$
          "self.h := left.h + right.h")) ; guard = $b_{1,h}$
...
; assign each attribute a freshly allocated location
(set! root.x (alloc))
(set! root.y (alloc))
...
; expansion of (fork ([slot slot*]) (eval $??_1$))
; case for $??_1$ = "self.x := 0"
(write root.x #:guard {$b_{1,x}$})
(step #:guard {$b_{1,x}$})
; case for $??_1$ = "self.y := 0"
(write root.y #:guard {$b_{1,y}$})
(step #:guard {$b_{1,y}$})
; case for $??_1$ = "self.w := left.w + right.w"
(read root.left.w #:guard {$b_{1,w}$})
(read root.right.w #:guard {$b_{1,w}$})
(write root.w #:guard {$b_{1,w}$})
(step #:guard {$b_{1,w}$})
; case for $??_1$ = "self.h := left.h + right.h"
(read root.left.h #:guard {$b_{1,h}$})
(read root.right.h #:guard {$b_{1,h}$})
(write root.h #:guard {$b_{1,h}$})
(step #:guard {$b_{1,h}$})
...
```

1. Every location is written at most once.
2. Every read to a location is preceded by the write to that location.
3. Concurrent threads are data-independent (*i.e.*, locations read by a thread are disjoint from locations written by any concurrent thread).

Note that the symbolic trace language requires annotating with `fork` those code fragments that must be explored under alternative symbolic choices. Each such choice evaluates under a *guard* (analogous to a path condition in traditional symbolic evaluation) that records the current set of assumptions about symbolic choices. For instance, when `(fork ([x (choose` x_1 x_2`)]) ...)` explores the path for x_1, the current guard will be extended (since uses of `fork` may be nested) with the assumption that `x =` x_1, and a similar process then happens for x_2. The `fork` is analogous to Rosette's `for/all` and serves the same role of controlling where alternative paths are merged.

Adopting the symbolic trace language requires only a handful of straightforward changes to the interpreter in Listing 3, to turn its checks into trace events. The modified interpreter is shown in Listing 4. Listing 6 shows an excerpt of the residual program generated by the call $\texttt{s}(\texttt{int}_{ST}(\texttt{G}, \texttt{t}) \circ \texttt{sch})$, where `s` is the program specializer.

Implementation. The symbolic trace language is implemented as an embedded domain-specific language in Rosette. We leverage the restricted nature of L_T to generate efficient ILP constraints. We mention in this paper only the encoding of dependences, which was responsible for the bulk of the improvement over the previous SMT encoding.

Collectively, the generated constraints must ensure that all *dependences* are satisfied, which means that all reads from a location follow the write into that location. A straightforward encoding is to require that the step counter of the read is higher than the step counter of the write.

However, a less obvious encoding improves the solver's performance by several orders of magnitude. Rather than ensuring that all dependences are met, we pose the equivalent constraints ensuring that no *antidependences* exist, which means that the write must not happen after any of the reads from the location. The advantage of ruling out antidependences over requiring dependences is that in ILP, it seems easier to solve the constraint "if a write happens here, then *none* of the reads must have happened before," than it is to solve "if a read happens here, then a write *must* have happened before." We hypothesize that this encoding wins over alternative ILP encodings because ruling out antidependences provides the solver the analogue of conflict clauses that it would need to learn itself.

We want to point that the concept of antidependences does not exist in the original schedule interpreter. A general symbolic compiler thus cannot switch from dependences to ruling out antidependences. Doing so would require relative deep and global reasoning.

Evaluation. Figure 4c compares our domain-specific symbolic evaluator against the general-purpose evaluator. We evaluate the performance of symbolic evaluation and constraint solving. The benchmarks synthesize tree traversal schedules for an attribute grammar that encodes the treemap data visualization [11]. For each attribute grammar, symbolic evaluation is done with a set of example trees chosen by an automated tool to sufficiently cover the grammar. An enumeration of candidate partially symbolic schedules is used. Each measurement is the median value from three runs. The domain-specific encoding on the CPLEX solver improves the solving time by three orders of magnitude.

6 Summary and Future Directions

Contribution to Solver-Aided DSLs. Our domain-specific symbolic compilation idea originated from solver-aided DSLs (SDSLs). SDSL is attractive because it is the most promising technique we have today for automatically constructing program synthesizers: simply write an interpreter, and obtain a symbolic compiler for free. SDSLs take advantage of the domain in two ways. First, DSL programs are compact, reducing the search space explored by synthesizers. Second, an interpreter for a DSL can be written to increase the opportunities for specialization.

Rosette, its meta-language, and SMT solvers together served as an excellent tool to build SDSLs. However, as we moved to larger and more complex problems, both the symbolic compilation and solving emerged as a bottleneck. SMT solvers also ran out of steam. Our proposed solution delivers the promise of building domain-specific synthesizers that can solve larger, more complex problems. We show that a domain-specific symbolic compiler, built on top of Rosette, can produce a custom encoding of constraints and utilize a more efficient solver for that particular domain, such as an ILP solver.

Evaluation. Our ultimate goal is to obtain efficient constraints generated by an intuitive, expressive abstraction. In this paper, we showed that our domain-specific symbolic compiler can generate efficient constraints, reducing the time of solving and symbolic evaluation, sometimes by orders of magnitude. We are encouraged that good constraints can be generated by symbolically evaluating a program, as that opens new ways to write specifications.

Our findings are preliminary when it comes to expressiveness. While we are happy with the experience of expressing our interpreters and checkers, we have not stressed the abstractions enough to identify their limits. For example, we did not try to use the Bonsai tree (Section 3) on type checkers that also perform inference; the resource language (Section 4) may need extensions to support runtime data migration; and the trace language (Section 5) may fail on incremental tree evaluation because it needs data-dependent traversals that visit only a subtree. These new applications may require new abstractions. However, we hope that the three designs will serve at as an inspiration.

Checking the usage. Symbolic languages are not intended to be used without care, and a type system should thus ensure proper usage of symbolic language operations. However, the code using the abstraction can sometimes break the abstraction even through seemingly unrelated code (because the client and the abstraction are symbolically evaluated together). For example, using a conditional expression in the partitioning type checker (Listing 2) may cause the symbolic evaluation to produce a symbolic path condition, which makes the program inexpressible in ILP constraints. It is an open question what restrictions we should impose on the client to avoid such surprises.

Combining symbolic languages. New challenges arise when we compose two symbolic languages. For example, to distribute nodes of an unbounded tree onto CPU cores, we may want to combine some variants of the trace language from Section 5 and the resource language from Section 4. The former would schedule traversals while the latter would map data onto cores. What do we expect when both languages are used in the same interpreter? If scheduling and data placement happen to be separate problems, it would be desirable if the symbolic compilation of the interpreter produce two independent sets of constraints. If the problems are intertwined, the interpreter writer should be able to control the approximation: for example, fix the tree distribution first, then find the best traversal strategy.

Converting solutions to programs. We have blissfully assumed that the solution returned by the solver *is* the desired program. Naturally, the solution must be converted to program syntax, and the conversion becomes trickier as we add more levels of abstraction. Specializing symbolic compilers (Rosette [20] and Sketch [17]) automate the conversion, but that functionality is broken by our insertion of the symbolic language layer. It seems possible to define the conversion as an inversion of symbolic compilation, and perhaps this view could lead us towards automatic construction of solution-to-program converters.

References

1 Nada Amin and Ross Tate. Java and scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA*, 2016.

2 James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *POPL*, 2016.

3 Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 – April 2, 2004. Proceedings*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

4 Robert Glück. Is there a fourth futamura projection? In *Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2009, Savannah, GA, USA, January 19-20, 2009*, pages 51–60, 2009.

5 GreenArrays. *Product Brief: GreenArrays Architecture*, 2010. URL: `http://www.greenarraychips.com/home/documents/greg/PB002-100822-GA-Arch.pdf`.

6 Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

7 Jad Hamza, Barbara Jobstmann, and Viktor Kuncak. Synthesis for regular specifications over unbounded domains. In *FMCAD*, 2010.

8 Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *NSDI*, 2015.

9 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

10 Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*, 8(3):355–383, July 2003.

11 Leo Meyerovich, Matthew Torok, Eric Atkinson, and Rastislav Bodik. Parallel schedule synthesis for attribute grammars. In *PPoPP*, 2013.

12 Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *WWW*, 2010.

13 Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *PLDI*, 2013.

14 Jens Palsberg and Mayur Naik. ILP-based Resource-aware Compilation, 2004.

15 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.

16 Phitchaya Mangpo Phothilimthana, Michael Schuldt, and Rastislav Bodik. Compiling a gesture recognition application for a low-power spatial architecture. In *Proceedings of Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, 2016.

17 Sketch: a synthesizer language and compiler. URL: `http://bitbucket.org/gatoatigrado/sketch-frontend`.

18 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, New York, NY, USA, 2006.

19 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

20 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, 2014.

21 Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *PLDI*, 2000.

The End of History? Using a Proof Assistant to Replace Language Design with Library Design*

Adam Chlipala[1], Benjamin Delaware[2], Samuel Duchovni[3], Jason Gross[4], Clément Pit-Claudel[5], Sorawit Suriyakarn[6], Peng Wang[7], and Katherine Ye[8]

1 **MIT CSAIL, Cambridge, MA, USA**
`adamc@mit.edu`

2 **Purdue University, West Lafayette, IN, USA**
`bendy@purdue.edu`

3 **MIT CSAIL, Cambridge, MA, USA**
`dukhovni@mit.edu`

4 **MIT CSAIL, Cambridge, MA, USA**
`jgross@mit.edu`

5 **MIT CSAIL, Cambridge, MA, USA**
`cpitcla@mit.edu`

6 **MIT CSAIL, Cambridge, MA, USA**
`sorawit@mit.edu`

7 **MIT CSAIL, Cambridge, MA, USA**
`wangp@mit.edu`

8 **Carnegie Mellon University, Pittsburgh, PA, USA**
`kqy@cs.cmu.edu`

Abstract

Functionality of software systems has exploded in part because of advances in programming-language support for packaging reusable functionality as libraries. Developers benefit from the uniformity that comes of exposing many interfaces in the same language, as opposed to stringing together hodgepodges of command-line tools. Domain-specific languages may be viewed as an evolution of the power of reusable interfaces, when those interfaces become so flexible as to deserve to be called programming languages. However, common approaches to domain-specific languages give up many of the hard-won advantages of library-building in a rich common language, and even the traditional approach poses significant challenges in learning new APIs. We suggest that instead of continuing to develop new domain-specific languages, our community should embrace library-based ecosystems within very expressive languages that mix programming and theorem proving. Our prototype framework Fiat, a library for the Coq proof assistant, turns languages into easily comprehensible libraries via the key idea of modularizing *functionality* and *performance* away from each other, the former via *macros that desugar into higher-order logic* and the latter via *optimization scripts* that derive efficient code from logical programs.

1998 ACM Subject Classification F.3.1 [Specifying and Verifying and Reasoning about Programs] Mechanical Verification, F.4.1 [Mathematical Logic] Mechanical Theorem Proving, Logic and Constraint Programming, I.2.2 [Automatic Programming] Program Synthesis

* This work has been supported in part by NSF grants CCF-1253229, CCF-1512611, and CCF-1521584; and by DARPA under agreement numbers FA8750-12-2-0293 and FA8750-16-C-0007. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 3; pp. 3:1–3:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases Domain-specific languages, synthesis, verification, proof assistants, software development

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.3

1 The Case for Replacing Languages with Libraries in a Proof Assistant

As a programmer today, it is hard to imagine getting anything done without constant reuse of libraries with rather broad APIs. Complex production software systems weave together many different libraries hosted in a single language where integration is eased by a shared vocabulary of concepts like objects, functions, types, and modules. We can imagine piecing together similar end products from Frankenstein's monsters of distinct languages for different styles of programming, tied together with command-line tools and heroic build processes. Most of us are glad not to live in a world where that is the best option, however, considering several key advantages of the language-integrated approach.

- **Learnability.** To learn an arbitrary programming language, one might turn to its reference manual, which is prone to imprecision and likely to become out-of-date; or one might try to read the language's implementation, which unavoidably mixes in implementation details irrelevant to the user. In contrast, a library in a statically typed language inherits "for free" a straightforward characterization of the valid programs: precisely those that type check against the API, which is written out in a common formalism.
- **Interoperability.** With all libraries defining first-class ingredients that live in a common formalism, it becomes easier to code, for example, polymorphic operations that generalize over any such ingredients.
- **Correctness.** The developers of the chosen language implementation need to worry about getting it right, but then authors of individual libraries may rely on the language's encapsulation features to limit how much damage bugs in their libraries can inflict on other libraries and their private state.

All in all, we programmers can pat ourselves on the backs for collectively coming up with such a satisfyingly effective approach to building big things out of smaller, very general things. *However*, we claim there remain many opportunities to improve the story. *Domain-specific languages (DSLs)* are a paradigm growing in popularity, as programmers find that the API formalisms of general-purpose languages are not flexible enough to fit important packages of reusable functionality. Instead, a new notation is invented that allows much more concise and readable descriptions of desired functionality in a particular domain. DSLs have found widespread success in a range of domains, including HTML+CSS for layout, spreadsheet formulas for tabular data, and Puppet for system configurations. For programmer-facing tasks, DSLs such as SQL and BNF have been adopted widely, to the point that they are *the* standard solutions for interacting with databases and building parsers, respectively. Despite these isolated success stories, DSLs have not reached the ubiquity of library-based solutions in the average programmer's toolbox. Simply implementing a new DSL can require considerable effort, if one chooses to build a freestanding compiler or interpreter. Alternatively, we might have an *embedded* DSL that actually *is* a library within a general-purpose language, but which delineates its own new subset of that language with a rather distinctive look, perhaps calling library combinators to construct explicit abstract syntax trees. We claim both strategies introduce substantial friction to widespread adoption of DSLs.

- **Learnability.** A DSL with a freestanding implementation is just as unlearnable as a new general-purpose programming language, thrown at the programmer out of the blue. An embedded DSL may be easier to learn, by reading the static type signature that defines it, though we claim that often this type signature is complicated by limitations of the host language, and it almost never expresses semantics (in fact, in many cases, the host language is a dynamically typed Lisp-like language).
- **Interoperability.** Freestanding DSL implementations bring on all the pain that we congratulated ourselves on avoiding above. For instance, freestanding parser generators force the use of Makefiles or similar to coordinate with the main program, while the database query language SQL has notoriously bad coupling to general-purpose languages, for instance relying on APIs that present queries as uninterpreted strings and invite code-injection vulnerabilities. Embedded DSLs tend to appear to the programmer as their own fiefdoms with their own types of syntax trees, which rarely nest naturally within each other without giving up the performance from clever compilation schemes.
- **Correctness.** The kind of metaprogramming behind a language implementation is challenging even for the best programmers, and it is only marginally easier for compact DSLs than for sprawling marquee general-purpose languages. It may make sense to invest in producing a correct Java compiler with traditional methods, but the necessary debugging costs may prove so impractical as to discourage the creation of new DSLs with relatively limited scopes.

How could we reach back toward the advantages of libraries as natural APIs within a single host language? Our core suggestion in this paper is to modularize *functionality* away from *performance*. So much of programming's complexity comes from performance concerns. Let us adopt an extreme position on the meaning of the word, taking it even to encompass concerns of computability, where typically it is not sufficient to write an unambiguous description of desired program behavior; we must express everything algorithmically. Imagine, instead, that the programmer is liberated from concerns of performance, being able to write truly *declarative* programs that state *what* is desired without *how* to achieve it. Every part of a program is expressed with its most natural notation, drawing on the libraries that take over for DSLs, exporting notations, among other conveniences. However, each notation desugars to a common language expressive enough to cover any conceivable input-output behavior, even uncomputable ones.

At this point, the programmer has codified a precise specification and is ready to make it runnable. It is time to introduce *performance* in a modular way. We follow the tradition of program derivation by stepwise refinement [8], but in a style where we expect the derivation process to be automatic. Every program mixes together derivation procedures drawn from different libraries, with each procedure custom-designed to handle the notations of that library. The end result is a proof-producing *optimization script* that strings together legal moves to transform specifications into efficient executable code. By construction, no optimization script can lead to an incorrect realization of an original program/specification.

This style promotes **learnability** with self-documenting macro definitions that are readable by the programmers who apply the macros, with each definition desugaring a parsing rule into a common higher-order logic, written for clarity and without any concern for performance; **interoperability** by the very use of that logic as the common desugaring target[1]; and **correctness** by codifying legal moves to transform specifications toward efficient code in optimization scripts.

[1] A caveat on the interoperability front, considering a popular alternative reading of that word, is that we are not primarily concerned with integrating with legacy systems. We are happy to design a clean-slate platform where libraries interface nicely, assuming that they commit to a new style of design.

Assuming one buys into this pitch, what would be the ideal platform for unifying all the ingredients? One proposal would be to leverage an existing language with metaprogramming features. Indeed, Racket's implementation of languages as libraries [38] and Scala's Lightweight Modular Staging (LMS) framework [32] enable programs to be written at a high level and then compiled to efficient implementations in order to achieve "abstraction without regret" through a combination of macros and syntax transformations. While both these approaches come close to achieving our goal, they both require that initial programs be executable, thereby imposing some algorithmic requirements on them, and they require the user to trust that the metaprograms implementing transformations are semantics-preserving.

In order to enable programmers to focus on *how* and not *what*, we propose using a very expressive logic and a macro system that desugars into it as the host language. We also need a way to code heuristics for transforming programs in that logic. In other words, we have a two-level language, with an object language of specifications and a metalanguage for manipulating specifications. The metalanguage should work in a correct-by-construction way, where we can be sure that no transformation breaks program semantics. The combination of these features enables a reimagination of embedded DSLs to have clean, declarative semantics unpolluted by concerns of executability, but which still result in efficient implementations.

Many readers will not be surprised at this point that we have described the core features of modern proof assistants! Such popular ones as Coq and Isabelle/HOL fit the bill; we chose Coq. All of the widely used proof assistants have their rough edges today, but we do imagine a future where more polished proof assistants serve as the IDEs of everyday programming in this style (and we hope that our experiments can help identify the best ways to go about that polishing). Our prototype framework **Fiat** [7] already makes it possible to code interesting programs inside of Coq, with extensibility plus strong separation of functionality from performance, generating assembly code with a proof of conformance to original functionality specifications. In the rest of this paper we review the core of Fiat, give some old and new examples of notation domains therein, and indulge in more philosophizing and speculation than we usually would in a conference paper.

2 A Flexible Core Language for Declarative Programs

The heart of the Fiat concept is a unified, flexible language for writing out the functionality of programs. We write these declarative programs using macros defined by domain libraries, but each macro has a simple syntax-mapping rule, so macros cannot be used directly to encode complex logic. Instead, we need a core language under the hood that we believe can be used to encode any reasonable program specification. Rather than reinventing the wheel, we start from Gallina, the logic of our favorite proof assistant Coq, which is already well-known from successful mechanized proofs from algebra [10] to compiler correctness [26].

However, the original program is not the end of the story, and there are reasons to add more superstructure on top of Gallina. We transform initial declarative programs gradually into efficient executable programs, and it is helpful to maintain the same core language for original programs, intermediate programs, and final executable programs. For that purpose, we chose the *nondeterminism monad*, also used in concurrent work by Lammich [21] on stepwise refinement in a different proof assistant.

We define our type family $\mathcal{P}$ of *computations* using the pun that the familiar notation for the powerset operator may also be thought of as standing for "program." The standard monad operators are defined as follows and can be proved to obey the monad laws, in terms of the standard semantics of the set-theory operators we employ.

$$\begin{aligned}
\texttt{return} &: \forall \alpha.\ \alpha \to \mathcal{P}(\alpha) \\
\texttt{return} &= \lambda x.\ \{x\} \\
\texttt{bind} &: \forall \alpha, \beta.\ \mathcal{P}(\alpha) \to (\alpha \to \mathcal{P}(\beta)) \to \mathcal{P}(\beta) \\
\texttt{bind} &= \lambda c_1, c_2.\ \bigcup_{x \in c_1} c_2(x)
\end{aligned}$$

We introduce the usual shorthand $x \leftarrow c_1; c_2$ for `bind` c_1 $(\lambda x.\ c_2)$. Now we can write a variety of useful (though potentially non-executable) programs that periodically pick elements from mathematical sets. For instance, here is a roundabout and redundant way to express the computation of any odd natural number.

```
a ← {n ∈ ℕ | ∃ k ∈ ℕ. n = 2 × k};
b ← {m ∈ ℕ | ∃ k ∈ ℕ. m = 1 + 2 × k};
return (a + b)
```

Similarly, here is how one might compute the sum of the integer zeroes of a polynomial:

```
zs ← {xs ∈ list ℕ | NoDuplicates xs ∧ ∀ x, P(x) = 0 ⇔ x ∈ xs};
return (foldl (+) 0 zs)
```

More ominously, here is a program (referring to the set $\mathbb{B}$ of Booleans) that we should probably not try too hard to refine into an executable version.

```
b ← {b ∈ 𝔹 | b = true ⇔ P = NP};
if b then return 42
else return 23
```

This example also illustrates the *relational* character of the nondeterminism monad: defining computations using logical predicates makes it trivial to integrate potentially uncomputable logical functionality with standard functional programming. For instance, we use the normal `if` construct of Gallina, rather than defining our own. Such natural integration may even lull the programmer into a false sense of security, as in the above example, where choosing a branch of a conditional requires resolving a major open question in theoretical computer science! (It is at least fairly straightforward to formalize the proposition "P = NP" in a general-purpose proof assistant like Coq.)

We choose the superset relation $\supseteq$ as our notion of refinement between computations. We also sometimes read $c_1 \supseteq c_2$ as "c_2 implements c_1." In general, c_2 should be more algorithmic or more performant than c_1, and we chain together many such steps on the path to a final efficient program. For instance, by this definition, our example with odd numbers refines into `return` 7, because $\{n \mid n \text{ is odd}\} \supseteq \{7\}$.

It is also crucial that we have effective tools for rewriting in computations, since rewriting is a convenient way to structure refinement steps. Theorems like this one justify the use of standard rewriting rules:

$$\forall \alpha, \beta, c_1 : \mathcal{P}(\alpha), c_1' : \mathcal{P}(\alpha), c_2 : (\alpha \to \mathcal{P}(\beta)).\ c_1 \supseteq c_1' \Rightarrow \texttt{bind}\ c_1\ c_2 \supseteq \texttt{bind}\ c_1'\ c_2\,.$$

Applying this theorem with the fact $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N}.\ n = 2 \times k\} \supseteq \{4\}$ lets us refine the odd-number example above into this form:

```
a ← return 4;
b ← {m ∈ ℕ | ∃ k ∈ ℕ. m = 1 + 2 × k};
return (a + b)
```

From here, the monad laws allow us to remove the `bind` for a, substituting the value 4 for bound occurrences of a. Instead of simplifying, we could also apply an analogous theorem that justifies rewriting under binders, in the second term argument of `bind`.

$$\begin{tikzcd}
\texttt{a.m}(r, i) \arrow[r, phantom, "\approx"] \arrow[d, phantom, "\rotatebox[origin=c]{90}{$\in$}"] & \texttt{b.m}(t, i) \arrow[d, phantom, "\rotatebox[origin=c]{90}{$\in$}"] \\
(r', o) & (t', o) \arrow[l, "\exists\, r'"'] \arrow[l, phantom, "\approx", shift left=3]
\end{tikzcd}$$

Figure 1 Refinement preserves similarity of internal values.

The nondeterminism monad provides a concise language for capturing a program's algorithmic content. To provide a complete declarative programming language, however, we need to add in the second element of the classic equation "programs = algorithms + data structures." To finish the story, Fiat also supports *data refinement* [14], allowing programs to operate over an abstract data model, introducing efficient data structures as part of the refinement process via an abstraction relation [15]. Consider the following declarative program, which filters out any number that is at least 100 in a set `s`:

```
return s ∩ {n | n < 100}
```

One reasonable implementation of this program replaces sets with splay trees and uses their `split` operation to perform the filter:

```
return fst(split(s, 100))
```

These two programs clearly return similar results, in the sense that the splay tree produced by the latter has the same elements as the set produced by the former, assuming the initial data had this relationship. We can get more formal with the following abstraction relation: $s \approx t \triangleq \forall n.\ n \in s \leftrightarrow n \in \texttt{elements}(t)$. Parameterizing the refinement relation over such relations captures this notion of similarity between a declarative program, succinctly stated with datatypes that are more abstract, and an implementation, which uses optimized data structures.

Under this approach, the behavior of an implementation depends on the abstraction relation used during refinement. As an example, every data type in the specification could be related to the unit type, to produce a valid but uninteresting refinement. In order to understand a refined program, a programmer cannot simply examine its initial specification – it is also necessary to consider the specific abstraction relation that produced that implementation. This requirement contradicts our proposed *learnability* criterion. It is also inherently antimodular, as any client of a refined program needs to be refined via the same abstraction relation. In order to enable modular reasoning while still permitting data-structure optimizations, Fiat exploits the familiar notion of data encapsulation provided by abstract data types (ADTs).

An ADT packages an internal *representation type* and a set of operations that build and manipulate values of that type. Fiat restricts data refinements to the representation types of ADTs. Clients are unable to observe the choice of data structure for an ADT's representation type thanks to *representation independence*, freeing an implementation to optimize its representation as it sees fit. In Fiat, an ADT specification uses an abstract model for its representation type and has operations that live in the nondeterminism monad, while an implementation is a valid refinement under an abstraction relation that is limited to optimizing the representation type. Intuitively, every implementation of an operation takes similar internal values to similar internal values, and its observable outputs are elements of the original specification, as illustrated by Figure 1. Thus, in contrast to other refinement frameworks that allow arbitrary data refinements [6, 22], in Fiat a client can understand the behavior of a refined ADT just by looking at its specification.

3 Integrating DSLs

To demonstrate the flexibility of this approach in action, we present the development of a simple packet filter in Fiat. At a high level, such a filter has two components: decoding the "on-the-wire" binary packet into an in-memory representation and then consulting a set of rules to decide whether to drop or forward the packet. Each of these two algorithmic tasks can be expressed using a domain-specific language implemented as a Fiat library; we begin with a brief overview of the two appropriate libraries.

The domain of the first library is the decoding of bitstrings into high-level in-memory datatypes, compatibly with some specified binary format. For simplicity, we consider deterministic binary formats, where every datatype has a single valid encoded representation. In this setting, a format can be captured as a function from the original datatype to its encoding, as in the following encoder for a record with two fields:

```
T ≜ {A : string, B : list int}
encode (t : T) ≜ encodeInt(len(t!B)) ++ encodeString(t!A) ++ encodeList(t!B)
```

This format combines together existing encoders for strings, lists, and integers, using the last to encode the length of the list in `t.B`, which the decoder will need to decode this field correctly. Given such a format, the specification of a decoder is straightforward:

```
decode (s : BitString) ≜ {t | encode(t) = s}
```

Here we have used the nondeterminism monad to capture the fundamental correctness condition succinctly for a binary decoder. The library also supports derivation of such an implementation via conditional refinement rules, examples of which are given in Figure 2. Each rule is a theorem in higher-order logic, and, to derive a particular decoder automatically, the optimization script chains together rule applications in rewriting style (with the crucial consequence that applying optimization scripts cannot lead to incorrect programs). The first two rules decode the head of the bitstring before decoding the rest under the assumption that some projection f of the encoded datatype is equal to the decoded value. Subsequent derivation steps can make use of this information, e.g. the correctness of DecodeList depends on a previously decoded length value. The final rule, FinishDecoding, is used to finish a derivation when enough information has been decoded to determine the original datatype uniquely. An implementation of a decoder can be derived automatically using these (generic) rules, plus a rule for decoding integers:

```
  {t | encodeInt(len(t.B)) ++ encodeString(t.A) ++ encodeList(t.B) = s}
⊇ let (n, s) = decodeInt(s) in                                    (DecInt)
  {t | len(t.B) = n ∧ encodeString(t.A) ++ encodeList(t.B) = s}
⊇ let (n, s) = decodeInt(s) in let (a, s) = decodeString(s) in    (DecString)
  {t | len(t.B) = n ∧ t.A = a ∧ encodeList(t.B) = s}
⊇ let (n, s) = decodeInt(s) in let (a, s) = decodeString(s) in    (DecList)
  let (l, s) = decodeList(s, n) in {t | len(t.B) = n ∧ t.A = a ∧ t.B = l ∧ [] = s}
⊇ let (n, s) = decodeInt(s) in let (a, s) = decodeString(s) in    (FinishDec)
  let (l, s) = decodeList(s, n) in if s = [] then {A ≜ a; B ≜ l} else fail
```

The key takeaways here are: given a binary format, writing an initial, declarative decoder is immediate and obvious, and while its implementation is more complicated, the correctness of

$$\frac{}{\{\mathtt{t} \mid \mathtt{P(t)} \land \mathtt{encodeString(f(t))} \mathbin{+\!\!+} \mathtt{s}' = \mathtt{s}\} \supseteq \begin{array}{l}\mathtt{let}\ (\mathtt{v},\mathtt{s}) = \mathtt{decodeString(s)}\ \mathtt{in}\\ \{\mathtt{t} \mid \mathtt{P(t)} \land \mathtt{f(t)} = \mathtt{v} \land \mathtt{s}' = \mathtt{s}\}\end{array}}\ (\textsc{DecodeString})$$

$$\frac{\forall \mathtt{x}.\ \mathtt{P(x)} \to \mathtt{len(f(x))} = \mathtt{n}}{\{\mathtt{t} \mid \mathtt{P(t)} \land \mathtt{encodeList(f(t))} \mathbin{+\!\!+} \mathtt{s}' = \mathtt{s}\} \supseteq \begin{array}{l}\mathtt{let}\ (\mathtt{v},\mathtt{s}) = \mathtt{decodeList(s,n)}\ \mathtt{in}\\ \{\mathtt{t} \mid \mathtt{P(t)} \land \mathtt{f(t)} = \mathtt{v} \land \mathtt{s}' = \mathtt{s}\}\end{array}}\ (\textsc{DecodeList})$$

$$\frac{\forall \mathtt{x}.\ \mathtt{P(x)} \leftrightarrow \mathtt{x} = \mathtt{v}}{\{\mathtt{t} \mid \mathtt{P(t)} \land [\,] = \mathtt{s}\} \supseteq \mathtt{if}\ \mathtt{s} = [\,]\ \mathtt{then}\ \mathtt{v}\ \mathtt{else}\ \mathtt{fail}}\ (\textsc{FinishDecoding})$$

Figure 2 Refinement rules for deriving binary decoders.

```
empty           ≜ ∅
For x in i b    ≜ table ← {l | i ~ l};
                    fold_R (λ a b ⇒ l ← a; l' ← b; return (l ++ l'))
                           (return []) (map (λ x ⇒ b) table)
Where P b       ≜ {l | P → l ∈ b ∧ ¬ P → l = []}
Return a        ≜ return [a]
Count b         ≜ results ← b; return length(results)
```

Figure 3 Notations for querying sets (relation ∼ constrains a list to contain some permutation of the elements of a set).

one built by an optimization script is guaranteed by (proof-producing) refinement. Note also that the process is extensible without expanding the trusted code base, in that incorporating a decoder for a new type is as simple as writing a new decoder and proving the corresponding refinement rule.

The next library used in our packet filter is a DSL for writing SQL-like programs. The notations provided by this library, examples of which are shown in Figure 3, desugar into basic set- and list-comprehension operations. The `Where` notation showcases the extensibility provided by our core framework, as a clause uses an arbitrary predicate to filter the set in contrast to, say, SQL. Consider a declarative function that finds the size of an island in a set:

```
island ≜ {name : string, size : int, temp : int}
islands : set of island
sizeOf (name) ≜ For i in islands Where i!name = name Return i!size
```

Just as in SQL, the notation provides for a concise description of both the program and its functionality, as it desugars into an expression using familiar set operations. Also as with SQL, the key challenge in executing this program is selecting data structures supporting the needed searches. A user of this library can write out only the abstract model of the representation type of an ADT and then rely on the optimization script to solve this implementation challenge via data refinement. A pleasant consequence of encapsulating the sets inside the ADT's representation type is that "whole-program analysis" becomes possible: we can write optimization scripts that examine exactly the queries (and updates) that we have exposed, automatically tailoring a data representation to efficient execution of those operations. Our relational-data library does just that, using plugins to incorporate user-provided (and user-proved) data-structure strategies, relying on the correctness guarantees provided by the core of the framework to ensure that they preserve the functionality of the original programs.

These two libraries demonstrate how our approach promotes *learnability* by enabling concise, declarative specifications of functionality, while also maintaining *correctness* in the face of extensibility via a machine-checked refinement trail. To round out our wish list with *interoperability*, we can see that they also play nicely with each other by combining them together to build a packet filter:

```
packet ≜ {src : word, name : list string, qtype : int}
rule   ≜ {name : list string, qtype : int, approve : boolean}
rules  ≜ set of rule
decide (s : BitString) ≜ p ← {p : packet | encodePacket(p) = s};
                            ans ← For r in rules
                                  Where r!name isPrefixOf p!name
                                  Where r!qtype = p!qtype
                                  Return r!approve;
                            return (head ans)
```

This example mixes the notations of the two libraries with normal functions (e.g. `head`), and it uses a custom `isPrefixOf` predicate in the `Where` clause of the query. More importantly, the optimization script that produces an implementation is also able to mix the implementation strategies provided by the libraries to handle the implementation tasks in both domains, automatically synthesizing the decoder for packets and selecting a data structure that supports prefix queries (tries, in this case).

4 Related work

There is a long history [8, 20, 29, 1] of using program transformations and stepwise refinement to obtain correct-by-construction, efficient implementations from specifications (albeit not necessarily in an automated fashion). Recent developments differ in guarantees obtained about the refined programs, intended application domains, degrees and styles of automation, and extensibility. Similarly, there is a rich line of academic work [12, 16, 2, 40] on the design and applicability of domain-specific languages: in fact, most early programming languages had domain-specific roots before they grew into general-purpose languages (LISP, the *list processor* for symbolic manipulations and AI; COBOL, the *common business-oriented language*; and FORTRAN, the *formula translator* for numerical computations). The following is a limited sampling of tools closely related to Fiat.

Stepwise Refinement Frameworks

The family of tools encompassing KIDS, DTRE, and Specware [34, 3, 37] allows users to decompose high-level specifications progressively into more and more concrete subproblems, until a concrete implementation can be supplied for each subproblem. The refinement style is similar to the one used by Fiat, with the main differences in how refinement steps are justified (Fiat is embedded in Coq and transparently exports a Coq proof obligation, while Specware relies on trusted proof-obligation generators to produce Isabelle/HOL goals justifying each transformation), target languages (Specware uses unverified transformations to extract C code, while the original Fiat system produces executable Gallina programs), composability (Fiat programs can be integrated into larger software developments verified in Coq), sound extensibility (Fiat tactics are proof-producing programs that run no risk of introducing unsoundness), and application domains (Fiat is mostly used for “simple” domains that lend themselves well to DSL development and admit clear specifications, allowing for a single refinement script to cover a large fraction of all programs expressible in the corresponding

DSL; Specware, on the other hand, has been used to synthesize correct-by-construction collections of complex algorithms, such as garbage collectors [30] or SAT solvers [35]).

Leon [19] is a deductive synthesis framework for deriving verified recursive functions on unbounded data types. Leon combines built-in recursion schemas, exhaustive enumeration, and counterexample-guided synthesis to generate the bodies of functional programs according to formally expressed postconditions. When the implementation chosen by the system is correct but not satisfactory, Leon users have the option to step in and perform refinement steps (verified refactorings) manually. Fiat has also been used to synthesize recursive programs [11] and uses less general automation: instead of a single synthesizer intended to cover all possible programs, Fiat specifications are refined using domain-specific optimization scripts that usually employ mostly deterministic strategies without backtracking. Users are free to introduce new rewriting steps and refinement strategies to achieve the desired performance characteristics.

Cohen et al. [6] used a notion of data refinement close to that of Fiat to develop and verify a rich algebra library in Coq: starting with high-level definitions written using "proof-oriented" data structures amenable to simple verification, the authors use data refinement to obtain an implementation with more efficient data structures satisfying the same guarantees. Our approach is different, in that we start from a potentially noncomputational, nondeterministic specification, which we refine to an implementation. We furthermore restrict data refinements to the representation types of ADTs, obviating the need for transporting proofs across an entire program.

Data-Structure Synthesis and Selection

Automatic data-structure selection was pioneered in SETL [33], a high-level language where programmers manipulate sets and associative maps through high-level primitives such as comprehensions and quantifiers, without committing to specific implementations of the underlying data structures. Instead, the SETL compiler employs a sophisticated static analysis to make concrete data-structure choices. Fiat's decoupling of specifications and performance yields a similar process. Unlike SETL, Fiat imposes no restrictions on the kind of data structures that can be used, the ways they can be combined, and the type of hints that programmers can give to the compiler to nudge it towards specific implementations. Fiat's sound extensibility makes it possible to substitute newly verified data structures at any step in the refinement.

More recently, Loncaric et al. [28] have built Cozy, a system for efficiently synthesizing a broad range of data structures, starting from a restricted DSL of data-retrieval operations and generating efficient object-oriented code using counterexample-guided inductive synthesis. Cozy synthesizes a high-level functional implementation of each operation using exhaustive enumeration augmented with a cost model to prune the search space, and from there deduces a good data representation, optionally using real-world benchmarks to autotune the selection. Though there are close similarities between the input language of Cozy and Fiat's SQL-style application domain, Fiat uses a mostly deterministic domain-specific compiler and hand-verified refinements instead of exhaustive enumeration and a general-purpose verifier. Fiat's SQL-style domain can in a sense be seen as an extensible proof-producing query planner, with strong extensibility granted by integration in a proof assistant. This vision provides an alternative answer to one of Cozy's original motivations, replacing unpredictable and hard-to-extend SQL engines.

Closely related to Cozy is Hawkins et al.'s RelC synthesizer [13], which decouples the relational view of the data from its in-memory representation (specified as a combination of basic data structures such as hash tables, vectors, or linked lists) by automatically deriving low-level implementations of user-specified relational accessors and mutators compatible with the chosen representation. Fiat has a similar input language but provides stronger correctness guarantees and allows for proof-producing extensions to the existing compilation logic (Fiat optimization scripts cannot perform unsound transformations). Fiat is additionally an open-ended system, allowing users to combine multiple DSLs and use their respective compilers to synthesize parts of a larger verified program covered by end-to-end guarantees.

Leino and Milicevic [25] proposed dividing the effort of programming a verified component into three parts: a public interface providing a mathematical model of the object; a data-structure specification describing the layout and invariants of the underlying implementation; and an executable implementation of the data structure. Jennisys, a prototype implementation of this idea, allows users to synthesize the code part of a component automatically by extrapolating from pre- and postcondition-conforming input and output examples generated using the Dafny [24] program verifier. Fiat shares some of Jennisys' synthesis objectives but applies to different domains, does not commit to specific data layouts and implementation details, and rejects the traditional regime of dividing a program into data structures and algorithms (phrasing problems in terms of functionality and performance).

Domain-Specific Synthesis

The binary encoders and decoders presented in Section 3 are similar in spirit to programs written using bidirectional lens combinators in the Boomerang [4] programming language. A single Boomerang program represents a pair of transformation functions between source and target domains. Compiling a Boomerang program produces both a map from source to target and an inverse function guaranteed to propagate changes from a target back to the generating source object.

Many domains beyond the ones that we have focused on are amenable to our approach. SPIRAL [9] is a framework for automatically deriving high-performance digital signal-processing code. Bellmania [17] is a recent framework for deriving cache-efficient implementations of divide-and-conquer algorithms. Bellmania uses a unified formalism to encompass both relatively high-level specifications of dynamic programs and their low-level implementations, allowing programmers to derive cache-efficient code through expert application of trusted *solver-aided tactics*, a carefully crafted set of built-in program transformations. Bellmania uses an SMT solver to ensure that each tactic is used soundly and to assist users by synthesizing code fragments from concrete inputs and traces.

Domain-Specific Languages

More broadly, there is a large body of work on DSL design and implementation. Leijen and Meijer [23] introduce and highlight the advantages of embedding DSLs in higher-order typed languages. Kats and Visser have developed the Spoofax [18] language workbench, a metaprogramming framework encompassing DSL parsers, compilers, and IDE support. Tobin-Hochstadt et al. [38] used Racket to implement the high-performance Typed Racket language. Van der Storm et al. [39] use *object grammars* to define compositional DSLs.

5 Discussion and Future Directions

Many past systems have done principled generation of code from specifications, using either combinatorial search (e.g., with a SAT solver in Sketch [36]) or deductive derivation (e.g., with Specware [37]). What is the secret sauce that distinguishes Fiat from these past systems? We claim it is the careful combination of *correct-by-construction automation* with *manual design of abstractions and decomposition of programs into modules.* Fundamentally, Fiat is a refinement of today's standard wisdom in software development: there is no silver bullet for solving all design and implementation problems. Instead, developers need to work hard to design proper abstractions (e.g., classes, libraries). In the best case, many abstractions are highly reusable. However, when working in an unfamiliar programming domain, we expect to develop a few new abstractions, probably in concert with reusing many familiar ones. Fiat is not a program-synthesis system that generates code automatically from specifications in a fixed domain. Such systems have inherent limitations and are unlikely to scale in isolation to the full software-development problem. At the same time, Fiat is not a manual-derivation system in the style of Specware. Instead, Fiat embodies a new style of modular program decomposition, where some modules are similar to traditional programs, though they support higher-order logic in place of algorithmic constructs; while other modules are more unusual, implementing automated strategies for deriving good code from the other modules. The programmer still faces a difficult and manual task in decomposing a program in this way, but the principled use of formal logic and correct-by-construction rewriting dramatically dampens the traditional pain points that we have emphasized throughout this paper.

We hope that the Fiat approach or one like it can earn a place on the standard list of abstraction and modularity techniques for practical programming. The central idea is to allow separate coding of the *functionality* and *performance* parts of a program, which we see as a natural evolution of the implementation/interface distinction of data abstraction [27]: the interface becomes the declarative program itself, one that is specific enough that we are happy with any compatible implementation, which we then derive automatically with a short optimization script that soundly combines nontrivial procedures from libraries. Of course, remembering all of the folk stories of genies run amok when their users wish incautiously, it is a tall order to design a specification discipline that minimizes unintended consequences. At a minimum, the technique needs to be extended with performance requirements as part of functionality, and no doubt some aspects of security should be added explicitly, too, though many of them are implied by functional correctness.

Our ongoing work gives library authors broad discretion in crafting high-performance optimization strategies by connecting to a proof-carrying-code system [5], admitting optimization rules that refine functional programs into assembly code, in concert with requirements to link against handwritten, low-level, verified implementations of imperative data structures [31]. We are also thinking about and prototyping a number of other domains with simple declarative starting points and effective correct-by-construction optimization strategies: textual formats specified by context-free grammars, SMT-style solvers specified by logical theories, and optimized big-integer cryptographic primitives specified by whiteboard-level math. There also seems to be no shortage of more far-out ideas that fit into the framework. We would like to, for instance, replace `make` and other build systems with use of a Fiat-style framework. Instead of writing "compile `a.c` into `a.o` using `gcc`," the build configuration would read "choose an element of the set of object files meeting a fixed semantic contract with the following C AST." That is, a (verified) compiler is just a relatively predictable kind of optimization script. Combinators could be used to mix together all such directives into build specifications for whole projects, oriented toward proving top-level project theorems, protecting against bugs in a variety of internal development tools.

Acknowledgments. The first author was inspired to start a project in this research area by conversations with Daniel S. Wilkerson and Simon Goldsmith about their unpublished work. For their helpful feedback, we thank the anonymous SNAPL reviewers and our shepherd Jean-Baptiste Tristan.

References

1 David R. Barstow. Domain-specific automatic programming. *IEEE Softw.*, 11(11):1321–1336, November 1985. doi:10.1109/TSE.1985.231881.

2 Jon Bentley. Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721, August 1986. doi:10.1145/6424.315691.

3 Lee Blaine and Allen Goldberg. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, pages 165–204. Elsevier, 1991.

4 Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *Proc. POPL*, pages 407–419, 2008. doi:10.1145/1328438.1328487.

5 Adam Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proc. ICFP*. Association for Computing Machinery (ACM), 2013. doi:10.1145/2500365.2500592.

6 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! *Lecture Notes in Computer Science*, pages 147–162, 2013. doi:10.1007/978-3-319-03545-1_10.

7 Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. POPL*, pages 689–700. Association for Computing Machinery (ACM), 2015. doi:10.1145/2676726.2677006.

8 Edsger W. Dijkstra. A constructive approach to the problem of program correctness. Circulated privately, August 1967. URL: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD209.PDF.

9 Sebastian Egner, Jeremy Johnson, David Padua, Jianxin Xiong, and Markus Püschel. Automatic derivation and implementation of signal processing algorithms. *SIGSAM Bull.*, 35(2):1–19, June 2001. doi:10.1145/511988.511990.

10 Georges Gonthier. Formal proof – the four-color theorem. *Not. ACM*, 55(11):1382–1393, 2008.

11 Jason Gross. An extensible framework for synthesizing efficient, verified parsers. Master's thesis, Massachusetts Institute of Technology, September 2015. URL: https://people.csail.mit.edu/jgross/personal-website/papers/2015-jgross-thesis.pdf, doi:1721.1/101581.

12 Michael Hammer. The design of usable programming languages. In *Proc. ACM*, ACM'75, pages 225–229, New York, NY, USA, 1975. ACM. doi:10.1145/800181.810327.

13 Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proc. PLDI*, PLDI'11, pages 38–49, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993504.

14 J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *Proc. ESOP*, volume 213, pages 187–196. Springer Berlin Heidelberg, 1986. doi:10.1007/3-540-16442-1_14.

15 C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972. doi:10.1007/BF00289507.

16 E. Horowitz, A. Kemper, and B. Narasimhan. A survey of application generators. *IEEE Softw.*, 2(1):40–54, January 1985. doi:10.1109/MS.1985.230048.

17 Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proc. OOPSLA*. Association for Computing Machinery (ACM), 2016. doi:10.1145/2983990.2983993.

18 Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proc. OOPSLA*, OOPSLA'10, pages 444–463, New York, NY, USA, 2010. ACM. doi:10.1145/1869459.1869497.

19 Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proc. OOPSLA*, pages 407–426, 2013. doi:10.1145/2509136.2509555.

20 Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974. doi:10.1145/356635.356640.

21 Peter Lammich. Refinement to Imperative/HOL. In *Proc. ITP*, volume 9236 of *Lecture Notes in Computer Science*, pages 253–269. Springer International Publishing, 2015. doi:10.1007/978-3-319-22102-1_17.

22 Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In Lennart Beringer and Amy Felty, editors, *Proc. ITP*, volume 7406 of *Lecture Notes in Computer Science*, pages 166–182. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-32347-8_12.

23 Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proc. DSL*, pages 109–122, 1999. doi:10.1145/331960.331977.

24 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. LPAR*, pages 348–370, 2010. doi:10.1007/978-3-642-17511-4_20.

25 K. Rustan M. Leino and Aleksandar Milicevic. Program extrapolation with Jennisys. In *Proc. OOPSLA*, pages 411–430, 2012. doi:10.1145/2384616.2384646.

26 Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54. Association for Computing Machinery (ACM), 2006. doi:10.1145/1111037.1111042.

27 Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proc. VHLL*, pages 50–59, New York, NY, USA, 1974. ACM. doi:10.1145/800233.807045.

28 Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In *Proc. PLDI*, pages 355–368, 2016. doi:10.1145/2908080.2908122.

29 H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, September 1983. doi:10.1145/356914.356917.

30 Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. Formal derivation of concurrent garbage collectors. In *Proc. MPC*, pages 353–376, 2010. doi:10.1007/978-3-642-13321-3_20.

31 Clément Pit-Claudel. Compilation using correct-by-construction program synthesis. Master's thesis, Massachusetts Institute of Technology, August 2016. URL: http://pit-claudel.fr/clement/MSc/, doi:1721.1/107293.

32 Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proc. POPL*, POPL'13, pages 497–510, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429128.

33 Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in SETL. In *Proc. POPL*. Association for Computing Machinery (ACM), 1979. doi:10.1145/567752.567771.

34 Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Softw.*, 16(9):1024–1043, September 1990. doi:10.1109/32.58788.

35 Douglas R. Smith and Stephen J. Westfold. Synthesis of propositional satisfiability solvers. Manuscript, 2008.

36 Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proc. PLDI*, PLDI'05, pages 281–294, New York, NY, USA, 2005. ACM. doi:10.1145/1065010.1065045.

37 Specware. URL: http://www.kestrel.edu/home/prototypes/specware.html.

38 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. PLDI*, PLDI'11, pages 132–141, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993514.

39 Tijs van der Storm, William R. Cook, and Alex Loh. Object grammars: Compositional and bidirectional mapping between text and graphs. In *Proc. SLE*, pages 4–23, 2012. doi:10.1007/978-3-642-36089-3_2.

40 Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000. doi:10.1145/352029.352035.

Natural Language is a Programming Language: Applying Natural Language Processing to Software Development

Michael D. Ernst

University of Washington Computer Science & Engineering, Seattle, WA, USA
mernst@cs.washington.edu

Abstract

A powerful, but limited, way to view software is as source code alone. Treating a program as a sequence of instructions enables it to be formalized and makes it amenable to mathematical techniques such as abstract interpretation and model checking.

A program consists of much more than a sequence of instructions. Developers make use of test cases, documentation, variable names, program structure, the version control repository, and more. I argue that it is time to take the blinders off of software analysis tools: tools should use all these artifacts to deduce more powerful and useful information about the program.

Researchers are beginning to make progress towards this vision. This paper gives, as examples, four results that find bugs and generate code by applying *natural language processing* techniques to software artifacts. The four techniques use as input error messages, variable names, procedure documentation, and user questions. They use four different NLP techniques: document similarity, word semantics, parse trees, and neural networks.

The initial results suggest that this is a promising avenue for future work.

1998 ACM Subject Classification D.2 Software Engineering, F.3.2 Semantics of Programming Languages, I.2.7 Natural Language Processing

Keywords and phrases natural language processing, program analysis, software development

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.4

1 Introduction

What is software? A reasonable definition – and the one most often adopted by the programming language community – is: a sequence of instructions that perform some task. This definition accommodates the programmer's view of source code and the machine instructions that the CPU executes. Furthermore, this definition enables formalisms: the execution model of the machine, and the meaning of every instruction, can be mathematically defined, for example via denotational semantics or operational semantics. By combining the meanings of each instruction, the meaning of a program can be induced.

This perspective leads to powerful static analyses, such as symbolic analysis, abstract interpretation, dataflow analysis, type checking, and model checking. Equally important and challenging theoretically – and probably more important in practice – are dynamic analyses that run the program and observe its behavior. These are at the heart of techniques such as testing, error detection and localization, debugging, profiling, tracing, and optimization.

Despite the successes of viewing a program as a sequence of instructions – essentially, of treating a program as no more than an AST (abstract syntax tree) – this view is limited and foreign to working programmers, who should be the focus of research in programming languages. Developers make use of test cases, documentation, variable names, program

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 4; pp. 4:1–4:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

structure, the version control repository, the issue tracker, conversations, user studies, analyses of the problem domain, executions of the program, and much more. The very successes of formal analysis may have blinded the research community to the bigger picture. In order to help programmers, and even to provide the program specifications that are essential to formal analysis, software analysis tools need to analyze all the artifacts that developers create. Tools that analyze the whole program will deduce more powerful and useful information about the program than tools that view just one small slice of it. These non-AST aspects of the program are also good targets for generation or synthesis approaches, especially since developers usually encode information redundantly: the information can be recovered from other (formal or informal) sources of information.

This paper focuses on one part of this vision: analysis of the natural language that is embedded in the program. In order to provide inspiration for further research, the paper discusses four initial results that find bugs and generate code by applying natural language processing techniques to software artifacts. The four techniques use as input error messages, variable names, procedure documentation, and user questions. They use four different NLP techniques: document similarity, word semantics, parse trees, and neural networks. In many cases, they produce a formal artifact from an informal natural language input. The initial results show the promise of applying NLP to programs.

This paper is organized as follows. First, Section 2 puts the use of NLP in the context of previous work that uses non-standard sources of specifications for formal analysis. In other words, Section 2 shows how using NLP to produce specifications can be viewed as the continuation of an existing line of research. The following four sections present four different approaches to applying natural language processing to English text that is associated with a program. Each one addresses a different problem, uses a different source of natural language, and applies a different natural language technique to the English to solve the problem. The following table overviews the four approaches.

		Problem	NL source	NLP technique
§3	Analyze existing code to find bugs	inadequate diagnostics	error messages	document similarity
§4		incorrect operations	variable names	word semantics
§5	Generate new code	missing tests	code comments	parse trees
§6		unimplemented functionality	user questions	translation

These few examples cover only a small number of problems, sources of natural language, and NLP techniques. Other researchers can take inspiration from these examples in order to pursue further research in this area. Section 7 discusses how researchers are already doing related work, via text analysis, machine learning, and other approaches.

2 Background: Mining specifications

Students sometimes ask whether a program is correct[1], but such a question is ill-posed. A program is never correct or incorrect; rather, the program either satisfies a specification or fails to satisfy a specification. It is no more sensible to ask whether a program is correct, without stating a specification, than to ask whether the answer is 42, without stating the question to be answered.

[1] There are many other important questions to be asked about a program beyond correctness. Does it fulfill a need in the real world? Is it usable? Is it reliable? Is it maintainable?

Many tasks, such as verification and bug detection, require a specification that expresses what the program is supposed to do. As a result, many papers start out by assuming the existence of a program and a specification; given these artifacts, the paper presents a program analysis technique. Unfortunately, most programs do not come with a formal specification. Furthermore, programmers are reluctant to write them, because they view the cost of doing so as greater than the benefit. Researchers and tool makers need to make specifications easier to write, and they need to create tools that provide value to workaday programmers. Until that happens, there is still an urgent need for specifications, in order to apply the research and tools that have been created.

One effective approach is to mine specifications – that is, to infer them from artifacts that programmers *do* create. Programmers embed rich information in the artifacts that they create. *Program analysis tools should take advantage of all the information in programs, not just the AST.* Too often, this is not done. For example, before formally verifying a program, the program is always tested, because testing is a more cost-effective way to find most errors. However, the formal verification process generally ignores all the effort that was put into testing, the test suites that were created, and the knowledge that was gained. This is a missed opportunity, in part caused by a parochial blindness toward "non-formal" artifacts.

Another way to express this intuition is to contrast two different views of a software artifact. Traditionally, programming language researchers have viewed it as an engineered artifact with well-understood semantics that is amenable to formal analysis. An alternative view is as a natural object with unknown properties that has to be probed and measured in order to understand it. These two perspectives contrast an engineer's blueprint with a natural scientist's explorations of the world. Considering a program as a natural object enables many powerful analyses, such as machine learning over executions, version control history analysis, prediction of upgrade safety, bug prediction, warning prioritization, and program repair.

As one example, consider specification mining: machine learning of likely specifications from executions. This technique transforms the implicit specifications that the programmer has embedded into a test suite, into a formal specification. A tool that performs this task is the Daikon invariant detector [16, 17, 18]; other tools also exist [2, 24, 40, 56, 9, 7, 8]. The software developer runs the program, and Daikon observes the values that the program computes. Daikon generalizes over the values via machine learning, in particular using a generate-and-check approach augmented by static and dynamic analyses and optimizations, because prior learning approaches had limitations that prevented them from being applied to this domain. The output is properties such as

- `x > abs(y)`
- `x = 16*y + 4*z + 3`
- array `a` contains no duplicates
- for each node `n`, `n = n.child.parent`
- graph `g` is acyclic

Like any good machine learning algorithm, the technique is unsound, incomplete, and *useful.* It is unsound because these are likely invariants: they were true over all executions and passed statistical likelihood tests, but there is no guarantee that they will be true during all possible future executions. It is incomplete because every machine learning algorithm has a bias or a grammar that limits its inferences. Nonetheless, it is useful. Some uses do not require soundness, such as optimization or bug-finding. Humans are known to make good use of imperfect information. The likely invariants can be used as goals for a verifier, yielding a sound system. Automatically-generated partial information is better than none at all. In practice, the inference process is surprisingly effective: the invariants are overwhelmingly correct, even when generalizing from little execution data.

Just as it is useful to process test suites to create formal artifacts, it is also useful to process natural language to create formal artifacts. The following sections give some examples.

3 Detection of inadequate diagnostic messages

Software configuration errors (also known as misconfigurations) are errors in which the software code and the input are correct, but the software does not behave as desired because an incorrect value is used for a configuration option [60, 57, 53, 55]. Diagnostic messages are often the sole data source available to a developer or user. Unfortunately, many configurable software systems have cryptic, hard to understand, or even misleading diagnostic messages [57, 28], which may waste up to 25% of a software maintainer's time [6]. We have built a tool, ConfDiagDetector [61], that tells a developer, before their application is fielded, whether the diagnostic messages are adequate.

More concretely, if a user supplies a wrong configuration option such as `-port_num=100.0`, the software may issue a hard-to-diagnose error message such as "unexpected system failure" or "unable to establish connection". Our goal is to detect such problems before shipping the code, so that the developer can substitute a better message, such as "`-port_num` should be an integer".

ConfDiagDetector combines two main ideas: configuration mutation and NLP text analysis. ConfDiagDetector works by injecting configuration errors into a configurable system, observing the resulting failures, and using NLP text analysis to check whether the software issues an informative diagnostic message relevant to the root-cause configuration option (the one related to the injected configuration error). If not, ConfDiagDetector reports the diagnostic message as inadequate.

ConfDiagDetector considers a diagnostic message as adequate if contains the mutated option name or value [29, 57], or if its meaning is semantically similar to the manual description of that configuration option. For example, if the `-fnum` option was mutated and its manual description says "Sets number of folds for cross-validation", then the diagnostic message "Number of folds must be greater than 1" is adequate.

Classical document similarity work uses TF-IDF (term frequency – inverse document frequency) to convert each document into a real-valued vector, then uses vector cosine similarity. This approach does not work well on very short documents, such as diagnostic messages, so ConfDiagDetector instead uses a different technique that counts similar words [35].

In a case study, ConfDiagDetector reported 25 missing and 18 inadequate messages in four open-source projects: Weka, JMeter, Jetty, and Derby. A validation by three programmers indicated that ConfDiagDetector has a 0% false negative rate and a 2% false positive rate on this dataset. This is a significant improvement over the previous best tool, which had a 16% false positive rate.

This approach differs from configuration error diagnosis techniques such as dynamic tainting [4], static tainting [42, 43], and Chronus [53] that troubleshoot an exhibited error, rather than proactively detecting inadequate diagnostic messages. It also differs from software diagnosability improvement techniques such as PeerPressure [52], RangeFixer [54], ConfErr [29], Spex-INJ [57], and EnCore [59] that require source code, a usage history, or OS-level support.

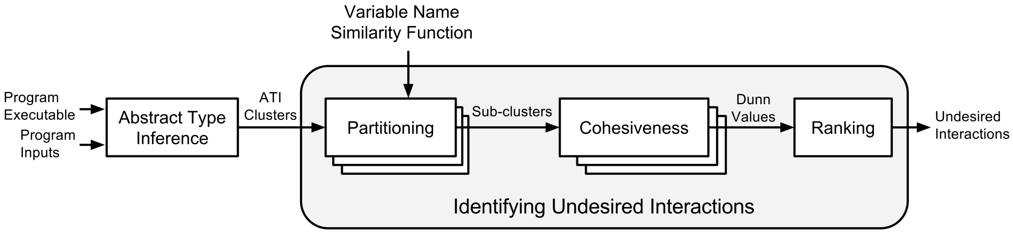


Figure 1 Ayudante architecture.

4 Identifying undesired variable interactions

A common programming mistake is for incompatible variables to interact, e.g., storing euros in a variable that should hold dollars, or using an array index with the wrong array. When a programmer commits an error, such as writing `totalPrice = itemPrice + shippingDistance;`, the compiler issues no warning because the two variables have the same programming language type, such as `int`. However, a human can tell that the abstract types are different, based on the variable names that the programmer chose.

We have developed an approach to detect such undesired interactions [51]. The approach clusters related variables, twice, using two different mechanisms. Natural language processing identifies variables with related names that may have related semantics. Abstract type inference identifies variables that interact with each other, which the programmer has treated as related. (For example, if the programmer wrote `x < y`, then the programmer must view `x` and `y` as having the same abstract type.) Any discrepancies between these two clusterings – that is, any inconsistency between variable names and program operations – may indicate a programming error, such as a poorly-named variable or an incorrect program operation.

Ayudante clusters variable names by tokenizing each variable name into dictionary words, computing word similarity based on WordNet or edit distance, and then arithmetically combining word similarity into variable name similarity. These variable name similarities can be treated as distances by a clustering algorithm. When a single ATI cluster can be split into two distinct variable-name clusters, it is treated as suspicious and presented to a user. Figure 1 shows the high-level architecture.

Abstract type inference can be computed statically [5, 36] or dynamically [22]; our tool, Ayudante, uses the dynamic approach, which is more precise in practice.

In an experiment, Ayudante's top-ranked report about the grep program indicated a interaction in grep that was likely undesired, because it discards information.

Previous work showed that reusing reusing identifier names is error-prone [32, 14, 3] and proposed identifier naming conventions [45, 30]. Languages like Ada and F# support a notation for units of measure. Our tokenization of variable names outperforms previous work [31, 21].

5 Generation of test oracles

Programmers are resistant to writing formal specifications or test oracles. Manually-written test suites often neglect important behavior. Automatically-generated test suites, on the other hand, lack test oracles that verify whether the observed behavior is correct. We have implemented a technique that automatically creates test oracles from something that programmers already write: code comments. In particular, it is standard practice for Java programmers to write Javadoc comments; IDEs even automatically insert templates for them.

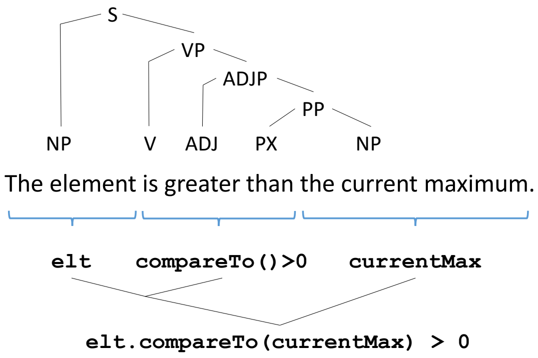

S
VP
ADJP
PP
NP
V
ADJ
PX
NP
The element is greater than the current maximum.
elt
compareTo()>0
currentMax
elt.compareTo(currentMax) > 0

Figure 2 Parsing a sentence and unparsing into an assertion.

We have built a tool, Toradocu [19], that converts English comments into assertions. For example, given

```
/** @throws IllegalArgumentException if the
 * element is not in the list and is not
 * convertible. */
void myMethod(Object element) { ... }
```

Toradocu might determine that `myMethod` should throw the exception iff

```
( !allFoundSoFar.contains(element) && !canConvert(element) ).
```

The intuition behind the technique is that when a sentence describes program behaviors, its nouns correspond to objects or values, and its verbs correspond to operations. This enables translation between English and code.

Toradocu works in the following steps.

1. Toradocu determines the nouns and verbs in a sentence from a Javadoc `@param`, `@return`, or `@throws` clause. It does so using the Stanford Parser, which yields a parse tree, grammatical relations, and cross-references. Toradocu uses pre- and post-processing to handle challenges such as the fact that the natural language is often not a well-formed sentence, it may use code snippets as nouns/verbs, and referents may be implicit.
2. Toradocu matches each noun/subject in the sentence to a code element from the program. It uses both pattern matching and lexical similarity to identifiers, types, and documentation.
3. Toradocu matches each verb/predicate to a Java element.
4. Toradocu reverses the parsing step: it recombines the identified Java elements, according to the parse tree of the original English sentence. The result is an assert statement.

Figure 2 gives an example.

In an experiment on 941 programmer-written Javadoc specifications, Toradocu achieved 88% precision and 59% recall in translating them to executable assertions. Toradocu can be tuned to favor either precision or recall.

Toraducu can automatically instrument test suites. Currently, automatic test generation tools have to guess whether a generated test fails or passes. Toradocu improved the fault-finding effectiveness of EvoSuite and Randoop test suites by 8% and 16% respectively, and reduced EvoSuite's false positive test failures by 33%.

Previously, test generation tools used heuristics to guess whether an exception was expected or unexpected [12, 13, 37, 38]. Property-based techniques that are similar to or can benefit from our approach include cross-checking oracles [10], metamorphic testing [11],

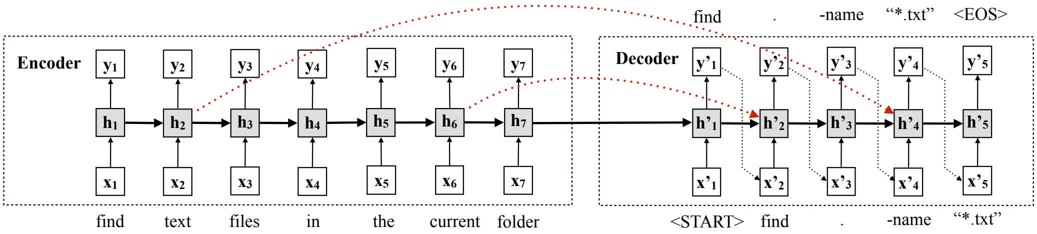

Encoder
y_1 y_2 y_3 y_4 y_5 y_6 y_7
h_1 h_2 h_3 h_4 h_5 h_6 h_7
x_1 x_2 x_3 x_4 x_5 x_6 x_7
find text files in the current folder
Decoder
find . -name "*.txt" <EOS>
y'_1 y'_2 y'_3 y'_4 y'_5
h'_1 h'_2 h'_3 h'_4 h'_5
x'_1 x'_2 x'_3 x'_4 x'_5
<START> find . -name "*.txt"

Figure 3 A sequence-to-sequence neural network translation model, applied to English and bash commands. The encoder reads the natural language description and passes its final hidden state to the decoder. The decoder takes the encoder's final hidden state and generates the output starting form a special symbol `<START>`. Notice that each decoder input symbol is the output symbol from the previous step. As is traditional, boxes are labeled by their outputs; for example, the lowest, leftmost box takes as input x_t (= "find") and applies I, producing as output $\mathbf{x}_t$. The red dotted lines mark the word alignments learned via the attention mechanism. While the neural network computes an alignment score for each pair of encoder hidden state and decoder hidden state, we illustrate only the alignments with high scores for readability.

and symmetric testing [20]. Previous work has used pattern-matching to extract simple properties, like whether a variable is intended to be non-null or nullable, from natural language documentation [50, 49, 48]; our approach is more general because it uses more sophisticated natural language processing techniques.

6 Generating code from natural-language specifications

The job of a software developer includes determining the customer's requirements and implementing a program that satisfies them. Part of this job is translating from a (usually informal) specification into source code.

One of the great successes of natural language processing is translation: for example, converting the English sentence "My hovercraft is full of eels" into the Spanish sentence "Mi aerodeslizador está lleno de anguilas." Recently, recurrent neural networks (RNNs) have come to dominate machine translation. The neural network is trained on a great deal of known correct data (English–Spanish pairs), and the network's input, hidden, and output functions are inferred using probability maximization.

If this approach works well for natural language, why shouldn't it work for programming languages? In other words, why can't we create a program – or, at least, get an initial draft – from natural language?

We have applied this approach to convert English specifications of file system operations into bash commands. Figure 3 shows a concrete example. We trained the RNN on 5,000 ⟨text, bash⟩ pairs that were manually collected from webpages such as Stack Overflow and bash tutorials. This domain includes 17 file system utilities, more than 200 flags, 9 types of open-vocabulary constants, and nested command structures such as pipelines, command substitution, and process substitution. Our system Tellina's top-1 and top-3 accuracy, for the structure of the command, was 69% and 80%.

No natural language technique will achieve perfect accuracy, due to the underlying machine learning algorithms. Tellina produces correct results most of the time, but produces incorrect results the rest of the time.[2] It is an important and interesting empirical question

[2] Classifying the usefulness of Tellina's output is not clear-cut. Even Tellina's correct results may not be perfect, and even its incorrect results can be helpful to programmers.

whether such a system that is useful in practice to programmers. In a controlled human experiment, programmers using Tellina spent statistically significantly less time ($p < .01$) while completing more file system tasks ($p < .1$). Even when Tellina's output was not perfect, it often informed the programmer about a command-line flag that the programmer didn't know about.

The most closely related work is in neural machine translation, which proposed both sequence-to-sequence learning with neural nets [47] and the attention mechanism [34]. Previous work on semantic parsing has translated natural language to a formal representation [58, 39], though one simpler than bash. Previous work on translating natural language to DSLs has also focused on simpler languages: if-this-then-that recipes [41], regular expressions [33], and text editing and flight queries [15].

7 Discussion

A minority of a software development team's is spent writing and changing the program, as opposed to participating in other activities, such as gathering requirements, design, documentation, and communicating with peers and stakeholders. Even when interacting with the program, a minority of a programmer's time is spent editing the programming language constructs in the source code, as opposed to testing, documenting, debugging, and reading it to understand it.

Researchers in software engineering and programming languages can find the most important challenges, do the most relevant work, and have the most impact by recognizing the needs of software developers. The programming language itself is an important but small part of this.

This paper advocates using natural language processing to analyze the textual parts of a program, in addition to the machine operations or AST that form its mathematical or operational core. Even the program including its natural language (the focus of this paper) still represents a minority of the concerns of a software developer! This paper focused on it because it is an important domain that permits use of a coherent set of research techniques. These techniques can apply ideas from both natural language processing and program analysis, and crucially, they can produce formal, executable specifications that feed back into many techniques that require specifications to express program semantics.

Our point of view is related to many previous lines of work. Previous researchers have applied pattern-matching or machine learning techniques (in some cases including NLP techniques), to software development artifacts that include the (formal) program, natural language in it, its tests, and its development history. We acknowledge their achievements, which have enabled and/or inspired our own.

The idea of analyzing the text that accompanies a program is not new. Up to now, much of this textual processing has been pattern-matching [48] rather than NLP. The same is true of many other approaches to processing program text, as described earlier. We believe that use of NLP will enable these techniques to become more general and achieve better results.

Statistical models can be used to model program text in similar ways to modeling natural language. Hindle et al. [26] hypothesize that "what people write and say is largely regular and predictable". This regularity is captured by n-gram models that capture how often a given sequence of n tokens occurs. This work ignores comments and applies these models to the executable program statements and expressions. The authors proposed that N-gram models can be used for code completion (such as stylized `for` loops). Subsequent work applied n-gram models to predicting common variable names and whitespace conventions [1].

Neither approach captures semantics other than incidentally by correlation, and neither was evaluated in terms of whether it would help programmers.

Another line of work focuses on creating the building blocks that from which NLP semantics could be obtained by future tools. Pollock and colleagues show how common variable-name patterns can be analyzed to assign a part of speech to each word that makes up the variable name [23], how rules and heuristics can match verbs to semantically-similar words by examining both code and comments [27], and how to mine abbreviation expansions such as “num” vs. “number” in variable names [25]. They also show how to generate summary comments for code [46], which is the dual of our goal of transforming less-formal into more-formal artifacts.

The JSNice system [44] represents a program AST in relational form for input to a learner. Given libraries/APIs that have known types and commonly-associated names, names and types can be inferred for new clients of those programs. This can regularize existing programs or suggest names for identifiers in new programs. It can also suggest types, without doing a standard type analysis. This work is notable for its uptake by industry. The variable names do not affect program semantics, the types are optional, and the compiler warnings can be suppressed; nonetheless, JSNice is useful in improving code style and gradually adding types to JavaScript code.

As the above examples show, natural language processing (NLP) is just one form of machine learning. NLP is applicable to the textual aspects of a program, such as messages, variable names, code comments, and discussions. Other types of data mining and machine learning can be applied to natural language in the text or to other artifacts, such as executions (e.g., Section 2), bug reports, version control history, developer conversations, and much more. The ideas presented in this paper could be extended to those other domains as well.

8 Analyzing the entire program

A program is more than source code, because a programming language – and more importantly, the programming system that surrounds it – is more than just a mathematical abstraction. In order to manage and understand the complexity of their programs, software developers embed important, useful information in test suites, error messages, manuals, variable names, code comments, and specifications. By paying attention to these rich sources of information, we can produce better software analysis tools and make programmers more productive. In addition to laying out this vision, this paper has overviewed a few concrete steps toward the vision: projects in which this extra information has proved useful. Many more opportunities exist, and I urge the community to grasp them.

Acknowledgements. This is joint work with Juan Caballero, Alberto Goffi, Alessandra Gorla, Mauro Pezze, Irfan Ul Haq, Kevin Vu, and Sai Zhang. The reviewers provided helpful comments that improved the paper.

References

1 Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 281–293, Hong Kong, November 18–20, 2014.

2 Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL 2002, Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 16–18, 2002.

3 Venera Arnaoudova, Laleh Eshkevari, Rocco Oliveto, Yann-Gael Gueheneuc, and Giuliano Antoniol. Physical and conceptual identifier dispersion: Measures and relation to fault proneness. In *26th IEEE International Conference on Software Maintenance*, pages 1–5, Timişoara, Romania, September 14–17, 2010.

4 Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX ATC*, pages 281–286, Boston, Massachusetts, 2008.

5 Henry Baker. Unify and conquer (garbage, updating, aliasing, ...). In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 218–226, Nice, France, June 27–29, 1990.

6 Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben M. Haber, Leila A. Takayama, and Madhu Prabaker. Field studies of computer system administrators: Analysis of system management tools and practices. In *Computer Supported Cooperative Work*, pages 388–395, Chicago, IL, USA, November 2004.

7 Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *ICSE'13, Proceedings of the 35th International Conference on Software Engineering*, pages 252–261, San Francisco, CA, USA, May 22–24, 2013.

8 Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, pages 468–479, Hyderabad, India, June 4–6, 2014.

9 Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE 2011: The 8th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 267–277, Szeged, Hungary, September 7–9, 2011.

10 Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. Cross-checking oracles from intrinsic software redundancy. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, pages 931–942, 2014.

11 Tsong Y. Chen, F.-C. Kuo, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing and beyond. In *STEP'03, Proceedings of the 11th International Workshop on Software Technology and Engineering Practice*, pages 94–100, 2003.

12 Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, September 2004.

13 Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE'05, Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, St. Louis, MO, USA, May 18–20, 2005.

14 Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, September 2006.

15 Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 345–356, 2016.

16 Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

17 Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

18 Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

19 Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis*, pages 213–224, Saarbrücken, Genmany, July 18–20, 2016.

20 Arnaud Gotlieb. Exploiting symmetries to test programs. In *ISSRE'03, Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pages 365–375, 2003.

21 Latifa Guerrouj, Philippe Galinier, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Massimiliano Di Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *WCRE*, 2012.

22 Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, Portland, ME, USA, July 18–20, 2006.

23 Samir Gupta, Sana Malik, Lori Pollock, and K. Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, pages 3–12, San Francisco, CA, USA, May 20–21, 2013.

24 Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, May 22–24, 2002.

25 Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR 2008: 5th Working Conference on Mining Software Repositories*, pages 79–88, Leipzig, Germany, May 10–11, 2008.

26 Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *ICSE'12, Proceedings of the 34th International Conference on Software Engineering*, pages 837–847, Zürich, Switzerland, June 6–8, 2012.

27 Matthew J. Howard, Samir Gupta, Lori Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *MSR 2013: 10th Working Conference on Mining Software Repositories*, pages 377–386, San Francisco, CA, USA, May 18–19, 2013.

28 Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A user survey of configuration challenges in Linux and eCos. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 149–155, Leipzig, Germany, January 25–27, 2012.

29 Lorenzo Keller, Prasang Upadhyaya, and George Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *DSN'08: The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 157–166, Achorage, AK, USA, June 25–27, 2008.

30 Doug Klunder. Naming conventions (Hungarian). Internal Microsoft document, January 18, 1988.

31 Dawn Lawrie, Christopher Morrell, and Dave Binkley. Normalizing source code vocabulary. In *2010 17th Working Conference on Reverse Engineering*, pages 3–12, Beverly, MA, USA, October 13–16, 2010.

32 Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, December 2007.

33 Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. Neural generation of regular expressions from natural language with minimal domain knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 1918–1923, 2016.

34 Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421, 2015.

35 Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *Proceedings of the 21st National Conference on Artificial Intelligence*, pages 775–780, Boston, MA, USA, July 18–20, 2006.

36 Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, MA, May 1997.

37 Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 – Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.

38 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

39 Panupong Pasupat and Percy Liang. Inferring logical forms from denotations. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

40 Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, pages 273–276, Ghent, Belgium, September 8–10, 2003.

41 Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL-15)*, pages 878–888, Beijing, China, July 2015.

42 Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *ASE 2011: Proceedings of the 26th Annual International Conference on Automated Software Engineering*, pages 193–202, Lawrence, KS, USA, November 2011.

43 Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 131–140, Waikiki, Hawaii, USA, May 25–27, 2011.

44 Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "Big Code". In *POPL 2015, Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–124, Mumbai, India, January 15–17, 2015.

45 Charles Simonyi. Hungarian notation. `https://msdn.microsoft.com/en-us/library/aa260976%28VS.60%29.aspx`.

46 Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *ASE 2010: Proceedings of the 25th Annual International Conference on Automated Software Engineering*, pages 43–52, Antwerp, Belgium, September 22–24, 2010.

47 Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

48 Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*iComment: Bugs or bad comments?*/. In *SOSP 2007, Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 145–158, Stevenson, WA, USA, October 14–17, 2007.

49 Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 11–20, 2011.

50 Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 260–269, Montreal, Canada, April 18–20, 2012.

51 Irfan Ul Haq, Juan Caballero, and Michael D. Ernst. Ayudante: Identifying undesired variable interactions. In *WODA 2015: 13th International Workshop on Dynamic Analysis*, pages 8–13, Pittsburgh, PA, USA, October 26, 2015.

52 Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *USENIX 6th Symposium on OS Design and Implementation*, pages 245–257, San Francisco, CA, USA, December 6–8, 2004.

53 Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *USENIX 6th Symposium on OS Design and Implementation*, pages 77–90, San Francisco, CA, USA, December 6–8, 2004.

54 Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *ICSE'12, Proceedings of the 34th International Conference on Software Engineering*, pages 58–68, Zürich, Switzerland, June 6–8, 2012.

55 Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki. Range fixes: Interactive error resolution for software configuration. *IEEE Transactions on Software Engineering*, 41(6):603–619, June 2014.

56 Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE'06, Proceedings of the 28th International Conference on Software Engineering*, pages 282–291, Shanghai, China, May 24–26, 2006.

57 Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, pages 159–172, Cascais, Portugal, 2011.

58 Luke S. Zettlemoyer and Michael Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, pages 678–687, 2007.

59 Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhangand Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 687–700, Houston, TX, USA, March 3–5, 2014.

60 Sai Zhang and Michael D. Ernst. Which configuration option should I change? In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, pages 152–163, Hyderabad, India, June 4–6, 2014.

61 Sai Zhang and Michael D. Ernst. Proactive detection of inadequate diagnostic messages for software configuration errors. In *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 12–23, Baltimore, MD, USA, July 15–17, 2015.

Fission: Secure Dynamic Code-Splitting for JavaScript*

Arjun Guha[1], Jean-Baptiste Jeannin[2], Rachit Nigam[3], Rian Shambaugh[4], and Jane Tangen[5]

1 **University of Massachusetts Amherst, Amherst, MA, USA**
2 **Samsung Research America, Mountain View, CA, USA**
3 **University of Massachusetts Amherst, Amherst, MA, USA**
4 **University of Massachusetts Amherst, Amherst, MA, USA**
5 **University of Massachusetts Amherst, Amherst, MA, USA**

Abstract

Traditional web programming involves the creation of two distinct programs: a client-side front-end, a server-side back-end, and a lot of communications boilerplate. An alternative approach is to use a *tierless* programming model, where a single program describes the behavior of both the client and the server, and the runtime system takes care of communication. Unfortunately, this usually entails adopting a new language and thus abandoning well-worn libraries and web programming tools.

In this paper, we present our ongoing work on Fission, a platform that uses dynamic tier-splitting and dynamic information flow control to transparently run a single JavaScript program across the client and server. Although static tier-splitting has been studied before, our focus on dynamic approaches presents several new challenges and opportunities. For example, Fission supports characteristic JavaScript features such as `eval` and sophisticated JavaScript libraries like React. Therefore, programmers can reason about the integrity and confidentiality of information while continuing to use common libraries and programming patterns. Moreover, by unifying the client and server into a single program, Fission allows language-based tools, like type systems and IDEs, to manipulate complete web applications. To illustrate, we use TypeScript to ensure that client-server communication does not go wrong.

1998 ACM Subject Classification D.3.2. Multiparadigm Languages

Keywords and phrases JavaScript, information flow control

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.5

1 Introduction

Two decades after the introduction of JavaScript, web application security remains a challenging problem that continues to grow in significance. For example, over the past three years, the CVE database has accumulated over 2,500 cross-site scripting (XSS) vulnerabilities in popular open-source web technologies, such as Wordpress and Ruby on Rails [58, 20]. Major technology companies, such as Google, Facebook, and Microsoft regularly award bug bounties worth several thousand dollars to white-hat hackers who find vulnerabilities in their websites [25, 22, 10]. These kinds of vulnerabilities have also been making headline

* This work is supported by the U.S. National Science Foundation under grants CNS-1413985 and CCF-1408745.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 5; pp. 5:1–5:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

news, since hackers exploited a web-based SQL injection vulnerability to gain access to the computer systems of the U.S. Election Assistance Commission [42].

Existing Approaches. The programming languages research community has addressed the web application security problem in several different ways. For example, there is a large body of work on type systems [36, 14, 50, 35, 29, 47, 30, 5, 55, 16, 56, 57, 11] and program analyses [37, 6, 27, 15, 26, 54, 33] for JavaScript. Since many security vulnerabilities are caused by JavaScript's confounding dynamic semantics [38, 23, 46, 28, 39, 49, 45, 44, 9], a security-conscious programmer could leverage a type system or program analysis to avoid JavaScript's pitfalls. Naturally, static disciplines impose restrictions: it is difficult for tools to reason statically about dynamically-loaded code (`eval`) and other characteristic features of JavaScript. Moreover, client-side JavaScript is only half of any web application. Many security vulnerabilities occur on the server, where a variety of languages, such as PHP, Perl, Python, Ruby, and even JavaScript (using NodeJS) are in use. These other scripting languages have also been subjected to type systems and static analyses [24, 4]. However, to formally reason about the behavior and security of a web application, a tool has to consider the client-side and server-side programs together. Reasoning about multi-lingual systems is a challenging problem that is an active area of research [1, 40, 43].[1]

An alternative approach is to abandon the traditional web programming model and use a *tierless programming language*, where a single program written in a single programming language describes the behavior of the client and the server. For example, Links [18] and Ur/Web [12] provide a unified language for programming the client, server, and the database. SELinks enhances the Links model with statically-checked, label-based security policies [19]. In contrast, Swift [13] automatically partitions security-typed programs written in JIF [32] into client-side and server-side components. Tierless languages thus enable a variety of creative security solutions, but their defining characteristic is that the entire application is written in a single language, which allows programmers and developer tools to reason algebraically about their code.

JavaScript Is Hard To Let Go. Unfortunately, a key shortcoming of the aforementioned tierless languages is that they are *not JavaScript*, thus they abandon the large and vibrant JavaScript ecosystem. For example, web programmers have grown dependent on libraries like jQuery to build cross-platform web applications; modern applications use frameworks like Facebook React, which bring elements of reactive programming to the mainstream; and language extensions like JSX that allow programmers to use XML notation within JavaScript; and tools like TypeScript bring a modicum of safety to everyday JavaScript code, despite deliberately abandoning soundness. If we ask programmers to use a newly designed language, we are asking them to give up this ecosystem of libraries and language-based tools.

Our Approach. In this paper, we present Fission, a new approach to web application security that is not only compatible with the JavaScript ecosystem, but can even make existing JavaScript tools more effective. Fission allows web applications to be written as a single JavaScript program and uses dynamic information flow control and dynamic tier-splitting to automatically and securely split it into client-side and server-side code. Although

[1] Even if JavaScript is also used on the server, the client and server operate as two independent programs that communicate over HTTP. Therefore, one cannot reason about these programs using only the semantics of JavaScript.

static information flow control and tier-splitting has been explored before, we show that doing both of these *dynamically* has several advantages:

- By eschewing static approaches, Fission places no restrictions on JavaScript. Fission is fully compatible with ECMAScript 5, including characteristic JavaScript features, such as prototype inheritance and `eval`.
- We apply Fission to several large programs that use ECMAScript 6 (via Babel), JSX, and React on the client and NodeJS on the server. By fusing the client-side and server-side into a single program, Fission lets us delete a lot of brittle serialization and communication boilerplate code.
- Fission's dynamic tier-splitting allows the client and server to share code. In particular, certain higher-order function can be evaluated on either the client or the server, based on the context in which they are applied.

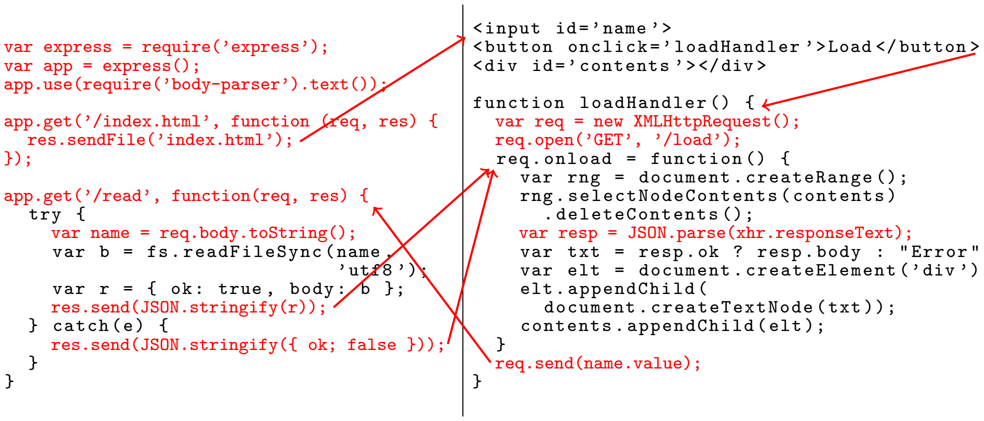
Beyond Security. Fission's information flow control can help programmers reason about the confidentiality and integrity of data. However, Fission's tierless design provides additional benefits. A shortcoming of the two-tier, web programming model is that it limits our ability to reason linguistically about program behavior. HTTP requests and web servers are an extra-linguistic feature and aren't part of the semantics of JavaScript. One way to address this issue is to add new language features to JavaScript [53]. However, since Fission makes web requests completely transparent, it allows us to reason about our code using just the semantics of JavaScript. Although it is rare for programmers to *reason about web programs*, there are several tools that do reason about JavaScript and these can be applied to Fission programs with minimal effort:

- We use Fission to develop server-side APIs (e.g., file I/O) for Elm [21]. Elm is a wonderful alternative to JavaScript, but without Fission, Elm programmers have to step outside the language to get any work done on the server. Furthermore, Fission could also be used with other languages that compile to JavaScript.
- We use Fission and TypeScript to write statically-typed web applications, where types ensure that client-server communication does not go wrong.
- Finally, we are able to use IDE features, such as refactoring tools, to consistently transform the client-side and server-side components of a program.

All these applications are possible because Fission faithfully implements JavaScript instead of requiring an entirely new programming language. Moreover, it is precisely because we do not design a new tierless language that Fission presents several new challenges. First, we introduce the Fission programming model in more depth.

2 A Fission Example

Figure 1 shows a two-tier program where both client and server are written in JavaScript. The purpose of this program is to display files stored on the server. When the user enters a filename and clicks "Load", the client sends a request to the server. The server either responds with the file contents or fails gracefully if the file is unreadable. Even in this trivial program, several shortcomings are apparent. First, the control-flow of the program is disjointed and bounces back and forth between the client and the server (indicated by the arrows in the figure). Second, about half the code is boilerplate needed to make requests, setup request handlers, and serialize data (highlighted in red). Finally, since the client and server are two logically distinct programs, it is difficult for programmers and tools to reason



```
var express = require(’express’);
var app = express();
app.use(require(’body-parser’).text());

app.get(’/index.html’, function (req, res) {
  res.sendFile(’index.html’);
});

app.get(’/read’, function(req, res) {
  try {
    var name = req.body.toString();
    var b = fs.readFileSync(name,
                            ’utf8’);
    var r = { ok: true, body: b };
    res.send(JSON.stringify(r));
  } catch(e) {
    res.send(JSON.stringify({ ok; false }));
  }
}
```

```
<input id=’name’>
<button onclick=’loadHandler’>Load</button>
<div id=’contents’></div>

function loadHandler() {
  var req = new XMLHttpRequest();
  req.open(’GET’, ’/load’);
  req.onload = function() {
    var rng = document.createRange();
    rng.selectNodeContents(contents)
      .deleteContents();
    var resp = JSON.parse(xhr.responseText);
    var txt = resp.ok ? resp.body : "Error";
    var elt = document.createElement(’div’);
    elt.appendChild(
      document.createTextNode(txt));
    contents.appendChild(elt);
  }
  req.send(name.value);
}
```

Figure 1 A canonical two-tier web application.

```
<input id=’name’>
<button onclick=’loadHandler’>Load</button>
<div id=’contents’></div>

function loadHandler() {
  var rng = document.createRange();
  rng.selectNodeContents(contents).deleteContents();
  var txt;
  try {
    txt = declassify(fs.readFileSync(name.value));
  }
  catch(e) { txt = "Error"; }
  var elt = document.createElement(’div’);
  elt.appendChild(document.createTextNode(txt));
  contents.appendChild(elt);
}
```

Figure 2 A tierless version of Fig. 1.

about their behavior. JavaScript tools cannot catch the trivial bug in the figure: the client requests `/load`, but the server has a handler for `/read`.

Figure 2 refactors the program to use Fission, which addresses the problems listed above. First, the boilerplate that was highlighted in the previous figure has been eliminated, since Fission handles serialization transparently. Second, the control-flow of the program is more natural since a single function can use both client and server APIs. Finally, since we no longer have two programs that explicitly communicate over HTTP, the bug in the previous program has been eliminated. Moreover, we can now leverage JavaScript tools to manipulate the entire program without breaking client/server consistency. For example, we could use an IDE to rename the `txt` variable which is written on the server but read on the client. With Fission, JavaScript developer tools can work on the whole program and aren’t limited to only the client-side code.

Attacker Model. Fission adopts the standard web attacker model [2] and assumes that an attacker can compromise the client and the network. Therefore, all values sent to the client are public to the attacker and all values received from the client are untrusted.

Information Flow Control. Fission uses dynamic information flow control, to securely partition code and data across the client and server. To a first approximation, all Fission values have two tags. A *secrecy* tag indicates whether a value is secret or public and an *integrity* tag indicates whether a value is trusted or untrusted. Therefore, Fission incorporates its attacker model as follows: Fission assumes that all client-side functions (i.e., all DOM

```
var found = 'Looking ...';
var txt = fs.readFileSync(
          '/etc/passwd', 'utf8');
if (txt.indexOf('alice') >= 0) {
  found = true;
} else {
  found = false;
}
```

Figure 3 Indirect information flow.

APIs) produce public, untrusted values and that all server-side functions (i.e., all NodeJS APIs) produce secret, trusted values.[2] Moreover, Fission will not send secret values to the client. Therefore, when a program needs to write a secret value to the client, it needs to be *declassified*. Untrusted values from the client can be *endorsed* in a similar way.

Fission asks programmers not to think about requests, responses, and serialization, but to think about the provenance of their data. This is not a new idea, but our experience with Fission shows that dynamic information flow control and dynamic tier-splitting is a powerful combination. Moreover, since Fission faithfully implements JavaScript, it supports several existing JavaScript libraries and tools with no changes required.

The Last Event Handler. Readers who enjoy functional reactive programming may be unhappy that the Fission code in Fig. 2 has one imperative event handler left. Instead of rehashing reactive programming for JavaScript, it is easy to reuse an existing JavaScript reactive programming library with Fission. Alternatively, the program could be rewritten in a language like Elm, using Fission to add support for transparent file I/O.

Overview. The rest of this paper summarizes Fission's technical approach, discusses some of the language design and implementation challenges we had to address, and some open research questions.

3 Faceted Execution

Faceted execution [7] is a form of termination-insensitive dynamic information flow control (IFC). The key idea in faceted execution is a *faceted value* (or facet for short), which is a pair of two values, where one is secret and the other is public. Which value is observed depends on the permissions of the observer. For example, when writing to the client, a facet is projected to its public component because all values on the client are visible to the attacker. Conversely, when reading from a secret file on the server, we create a facet where the private component has the file contents and the public component is a special unreadable value ($\perp$).

Let's consider a concrete example that has an indirect information flow. The program in Fig. 3 reads the user database from a server, tests if the account `alice` exists, and then sets the variable `found` to `true` or `false`. Since `readFileSync` is a NodeJS function, the variable `txt` holds a facet, where the secret component is the file contents and the public component is $\perp$, thus the client cannot directly read the file. Since the expression in the conditional uses the `txt` variable, the value of the conditional is a facet too, where the secret component is a boolean and the public component is $\perp$. Therefore, the assignment to `found` is affected by a secret value, even though no secret is directly written to `found`. Fortunately, faceted execution updates `found` to hold a facet, where the secret component is the boolean and

[2] A programmer can write more sophisticated, multi-principal policies, but these are reasonable defaults.

the public component is the original public value (the string `'Looking ...'`). Therefore, the server can observe the boolean which indicates whether the user `alice` exists, whereas the client sees the original value, thus cannot determine which branch was taken, unless the program is modified to declassify the value.

Therefore, to implement faceted execution, the control operators and primitive operations of JavaScript have to be lifted to manipulate faceted values appropriately. Facets can be nested and labeled to implement both confidentiality and integrity policies with several principals. Although Fission supports all these features, a typical Fission program only needs to reason about one principal, the server, and the programmer only needs to reason about whether values originated on the client or the server.

Cooperating Faceted Evaluators. A distinguishing characteristic of Fission is that it requires two faceted evaluators – one on the client and the other on the server – to cooperatively evaluate the program. Moreover, since the attacker model entails that the server-side evaluator cannot trust the client-side evaluator in any way, the server-side evaluator can neither send secrets to the client nor trust any information sent by the client.

Prior work on faceted execution has been based on a big-step semantics, which lends itself to a simple, direct-style interpreter. However, we had to develop a small-step faceted semantics with an explicit stack[3] because a context-switch requires an evaluator to examine its own stack, serialize it, and send it to the other evaluator. For example, if the server-side evaluator is running and the current stack frame is an application of a client-side function, at least that frame has to be serialized and transferred to the client.

It would be unsafe for the server to transfer stack frames that contain secrets to the client. So, what should happen if program applies a client-side function to a faceted argument that contain secrets? Since the Fission programming model assumes that client-side function only consume public values, a well-behaved client would simply discard the secret part of the argument and only use the public part. Instead of assuming that the client is well-behaved, the Fission server can instead project stack frames to only contain public values before transferring them to the client. This does not change the behavior of a well-behaved client, but prevents an attacker from observing secret values. Fission follows a similar approach with client-side state. The Fission programming model assumes that a web page's title, URL, cookies, etc. are always public. Therefore, when client-side state is updated to a new value, that value is projected to its public component before being transferred to the client.

Compilation and Taint Tracking. Our current implementation of Fission is an interpreter that is fast enough for interactive web pages. However, compiling Fission (or any faceted language) is challenging because each side of a facet may (or may not) follow a different branch. However, recent work [51] shows that faceted execution can be applied to taint tracking and that faceted taint tracking can be compiled in a relatively straightforward manner. Therefore, in situations where implicit flows are not a concern, a taint-tracking variant of Fission may be faster and easier to use.

4 Tier Splitting

The Fission programming model lends itself to several different implementations with a variety of tradeoffs. In particular, since the transfer of control between client and server is transparent to the program, tier splitting can be implemented in several ways.

[3] In essence, a faceted CEK machine.

Static Splitting. Although JavaScript is not statically typed, it should be possible to statically place expressions on the client or server. In brief, we could build a static control flow and data flow graph of the program, determine which expressions compute high-integrity values or consume secret values and ensure that these expressions are evaluated on the server. Swift [13] uses a richer variant of this approach, along with support for replicated data to keep the user interface responsive. We have yet to evaluate this approach in Fission, but we suspect that it may be too conservative for modern JavaScript that makes extensive use of higher-order functions. Even if programmers don't use higher-order functions themselves, they are generated by tools like Babel to implement modules. In these situations, a context-insensitive control flow graph may force too much code to needlessly run on the server. In addition, Fission supports `eval`, which gives fully static methods a lot of trouble.

Dynamic Splitting. In Fission, we tier-split the program dynamically because it produces better results for programs that use higher-order functions. The key idea is that the server can dynamically transfer control to the client (or vice versa) by transferring a prefix of the stack. However, since data sent to the client cannot contain secrets and data returned from the client cannot be trusted, the server cannot send an arbitrary portion of the stack. We use a lightweight, conservative analysis to determine which stack frames can be sent to the client without violating any of these requirements.

For example, suppose a program runs an expensive computation and displays its result on the client.

```
var x = fibonacci(50);
alert(x)
fs.writeFile(’result.txt’, x);
```

Since the value is not needed on the server, the expression can be evaluated entirely on the client, as long as the computation doesn't need to read secrets or update trusted values. However, if the value is also stored on the server, the computation needs to be performed on the server too.

Dynamic tier-splitting is particularly effective when a program uses higher-order functions that cannot be statically placed on either the client or the server. Consider the canonical *apply* higher-order function, which can be applied to either a function that must run on the client or a function that must run on the server.

```
function app(f, x) {
  return f(x);
}

app(fs.readFileSync, ’secret.txt’);
app(window.alert, ’hello’);
```

Therefore, we cannot statically determine where `app`(x) should be evaluated without considering the context where the evaluation occurs, and context-sensitive static analyses are very expensive. However, by examining the dynamic context, Fission can find a sequence of stack frames that do not read secrets, do not update trusted values, and do not execute operations like declassification and endorsement that have to be performed in a trusted context. Stack frames that meet these requirements can be evaluated on the client. These are broad requirements that allow a variety of tier-splitting mechanisms. We've developed a lightweight, conservative analysis that is effective on our benchmark programs, but it is straightforward to write contorted code that defeats the analysis and causes unnecessary context switching. A more precise analysis may be better at tier-splitting complicated code, but also have a longer running time. There is a large space of tier-splitting policies that can be explored.

5 JavaScript and Interoperability

Fission supports the ECMAScript 5.1 language standard, which is a close approximation of the JavaScript currently supported by major browsers. Unfortunately, ECMAScript is a fairly complicated language that includes getters and setters, object-oriented meta-programming features, and two language modes, in addition to well-known JavaScript pain-points, such as prototype inheritance, dynamic code-loading with `eval`, and more. Fission tackles this complexity by compiling JavaScript to a core language based on λ_{JS} [28] and S5 [45]. Fission's core language has additional features to support faceted execution as described above.

Let us now highlight Fission's implementation of `eval`, which is deeply affected by the attacker model. Note that Fission cannot implement `eval` by directly calling JavaScript's built-in `eval` function. If it did, the evaluated JavaScript code would not be able to interoperate with Fission's faceted JavaScript. Instead, Fission's implementation of `eval` builds a JavaScript AST, compiles it to a core language expression, and evaluates it using the Fission interpreter. If the call to `eval` is made in a trusted context, the JavaScript AST may even have sensitive operations like declassification and endorsement. Fission already ensures that the client is untrusted, therefore, a trusted call to `eval` may only occur on the server. In contrast, an untrusted call to `eval` may occur on either the client or the server. However, the code generated by an untrusted `eval` can neither neither access secret values nor endorse/declassify values. Notably, Fission does not need any additional mechanism to ensure that these properties hold when untrusted strings are evaluated. The same mechanisms in Fission that mediate interactions between the client and the server also ensure that untrusted strings can be safely evaluated. Although web programming best practices eschew using `eval`, it is still a commonly used construct [49], which is why put in the effort to support it.

Fission is carefully designed to support JavaScript's event-driven programming model. Therefore, a server-side event handler can transfer control to the client and vice versa. This requires full bidirectional communication, which Fission builds atop WebSockets. Moreover, if two events occur simultaneously on the client and the server, Fission serializes them to preserve JavaScript's single-threaded semantics. The current implementation of Fission suspends the client when the server is executing and vice versa, which is a reasonable default. However, there are scenarios where it is desirable to run client and server code in parallel. Other tierless languages expose server-side concurrency using libraries or special linguistic constructs [12, 18]. For the moment, we are evaluating Fission on existing NodeJS and Elm applications that do not require concurrency.

6 Applications

React-based Web applications. We have used Fission to refactor a handful of applications written for a final project in an undergraduate web programming class. All these applications use JavaScript on the client and NodeJS on the server, so refactoring involved directly calling request handlers at request sites (instead of making HTTP requests) and deleting serialization code. In addition, we had to insert declassification and endorsement operations, which is easy to do when server-side code is already factored into separate functions: we endorse all arguments and declassify the result. These applications have several hundred to two thousand lines of student-written code. However, all applications use Facebook React and several other JavaScript libraries. When they are linked together using Babel, each application has over 50,000 lines of code. Therefore, these are non-trivial examples that truly exercise the implementation. The Fission interpreter, which is written in JavaScript, takes a few seconds to load these programs, which start instantly without Fission. However, since these are interactive programs that are not compute-heavy, the slowdown is not noticeable when they are in use.

Elm. For readers who dislike JavaScript and would rather program in a statically-typed, ML-based language, we have used Fission to implement an Elm module that adds support for reading and writing files on the server. The library requires about five lines of code for each NodeJS function exported to Elm and leverages Fission to automatically context switch between the client and server. It would be straightforward to add wrappers for more NodeJS functions to enable pure Elm applications to seamlessly run on the client and server.

TypeScript and IDEs. There exist canonical TypeScript type definitions for both the NodeJS API and the Web browser DOM API. It usually does not make sense to import both type definitions in a single program, but the TypeScript compiler does not complain if you do so. With trivial type definitions for `endorse` and `declassify`, we can write statically-typed, tierless programs using TypeScript and Fission. Moreover, we can leverage IDEs like Visual Studio which have powerful support for TypeScript programming. Without Fission, TypeScript cannot reason about the client and server code in tandem, but Fission makes it trivial for TypeScript to do so.

7 Fission for Other Languages

Although Fission is engineered to support JavaScript, it is hopefully clear to the reader that the approach is not JavaScript-specific and could be applied to other programming languages too. The most natural candidates are other dynamic languages. For example Ruby and Python are not dissimilar to JavaScript and have support programming idioms that have been challenging to statically-check [24, 4, 34]. Therefore, the Fission approach is likely to suit these languages too.

The Fission approach is useful even with certain statically typed languages. Whereas languages like Links [18], Ur/Web [12], and Swift [13] have type systems that are explicitly designed to support tier-splitting, the Fission approach can be applied to statically languages that were not designed with tier-splitting and information flow control in mind. For example, Section 6 describes we applied Fission to (the JavaScript output of) programs written in TypeScript and Elm.

8 Related Work

Fission builds on a long line of research on tierless web programming languages, some of which we've already mentioned. Fission is directly inspired by Swift [13] which uses static IFC and static analysis to partition JIF programs across the client and server. Fission's emphasis on dynamic techniques and JavaScript makes it easy for us to support reams of existing JavaScript code and presents new challenges and opportunities.

Hop.js [53] is a tierless language that is also ECMAScript-compatible and supports both NodeJS and browser APIs. Unlike Fission, Hop.js is a syntactic superset of JavaScript because it uses a special quoting syntax to explicitly demarcate the boundary between client and server code. In contrast, Fission does not change the syntax of JavaScript and thus can be used in several ways that Hop.js cannot. First, a Hop.js program cannot be consumed by ordinary JavaScript analyses and refactoring tools. Second, Hop.js cannot be used to add server-side features to Elm (without changing the Elm compiler to generate Hop.js). Finally, Hop.js does not implement information flow control.

Fission does not syntactically distinguish client-side and server-side code, which is similar to the design of Swift and Links.[4] In contrast, in languages like Ur/Web [12], Hop [52], and Hop.js [53] it is syntactically evident when control crosses tiers. It is not clear to the authors which language design is intrinsically superior. However, since Fission does not add any new syntax to JavaScript, we get to reuse existing libraries and JavaScript developer tools.

Most web applications have three tiers: client, server, and database, and tierless languages like Links and Ur/Web unify all three tiers into a single language. Fission, with its emphasis on JavaScript, only unifies the client and the server. It is unclear if all three tiers can be unified satisfactorily for JavaScript without intentional language design (e.g., LINQ [41]).

Fission's tierless programming abstraction is built on top of an implementation of remote procedure calls [8] and distributed shared memory [3]. These abstractions require inter-machine communication, which can be optimized in several ways. For example, Remote Batch Invocation [31, 17] adds a language construct to batch several remote procedure calls together, which reduces the number of message round-trips incurred. These kinds of techniques are likely to improve Fission's performance, which currently uses very simple implementation techniques.

Fission's dynamic information flow control mechanism is based on faceted execution, which dynamically tracks implicit and explicit information flows in a fine-grained manner. However, there are alternative language-based approaches, such as Laminar [48], which is designed to make it easy to retrofit information flow control. For performance, the Laminar system requires virtual machine and operating system changes. However, Laminar's language abstractions could be adapted for JavaScript (with or without changing the VM). A possible future avenue for research may be to leverage Laminar-style "security regions" to make tier-splitting more efficient.

Acknowledgments. We thank the SNAPL'17 reviewers and our shepherd, Ranjit Jhala, for their thoughtful feedback and suggestions.

References

1 Amal Ahmed. Verified compilers for a multi-language world. In *Summit oN Advances in Programming Languages (SNAPL)*, 2015.

2 Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of Web security. In *IEEE Computer Security Foundations Symposium (CSF)*, 2010.

3 Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, February 1996.

4 Jong-hoon David An, Avik Chaudhuri, and Jeffrey S. Foster. Static typing for Ruby on Rails. In *IEEE International Symposium on Automated Software Engineering*, 2009.

5 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

6 Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2014.

[4] Links supports optional placement annotations.

7 Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

8 Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, February 1984.

9 Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014.

10 Bounty hunters: The honor roll. `https://technet.microsoft.com/en-us/security/dn469163.aspx`. Accessed Mar 24 2017.

11 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. Type inference for static compilation of JavaScript. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016.

12 Adam Chlipala. Ur/Web: A simple model for programming the web. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015.

13 Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

14 Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012.

15 Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

16 Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements for dynamic languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

17 William R. Cook and Ben Wiedermann. Remote batch invocation for SQL databases. In *International Symposium on Database Programming Languages (DBPL)*, 2011.

18 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods of Components and Objects*, 2006.

19 Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2009.

20 CVE-2016-6316: XSS vulnerability in Action View in Ruby on Rails. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6316`. Accessed Mar 24 2017.

21 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

22 Facebook bug bounty: $5 million paid in 5 years. `https://www.facebook.com/notes/facebook-bug-bounty/facebook-bug-bounty-5-million-paid-in-5-years/1419385021409053/`. Accessed Mar 24 2017.

23 Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013.

24 Michael Furr, Jong-hoon David An, and Jeffrey S. Foster. Profile-guilding static typing for dynamic scripting languages. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009.

25 Google security rewards–2015 year in review. `https://security.googleblog.com/2016/01/google-security-rewards-2015-year-in.html`. Accessed Mar 24 2017.

26 Salvatore Guarnieri and Benjamin Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009.

27 Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *World Wide Web Conference (WWW)*, 2009.

28 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

29 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming (ESOP)*, 2011.

30 Phillip Heidegger and Peter Thiemann. Recency types for dynamically-typed, object-based languages: Strong updates for JavaScript. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2009.

31 Ali Ibrahim, Yang Jiao an d Eli Tilevich, and William R. Cook. Remote batch invocation for compositional object services. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009.

32 JIF 3.5.0: Java information flow. `https://www.cs.cornell.edu/jif`. June 2016.

33 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A static analysis platform for JavaScript. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2014.

34 Jukka Lehtosalo. mypy. `http://mypy-lang.org`.

35 Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. Combining form and function: Static types for JQuery programs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2013.

36 Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting type systems for JavaScript. In *Dynamic Languages Symposium (DLS)*, 2013.

37 Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2015.

38 Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, 2008.

39 Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *European Symposium on Research in Computer Security*, 2009.

40 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

41 Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET Framework. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2006.

42 Joseph Menn. U.S. election agency breached by hackers after November vote. `http://www.reuters.com/article/us-election-hack-commission-idUSKBN1442VC`. Accessed Jan 2 2017.

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

44 Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

45 Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *Dynamic Languages Symposium (DLS)*, 2012.

46 Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: Type-based verification of javascript sandboxing. In *USENIX Security Symposium*, 2011.

47 Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Semantics and types for objects with first-class member names. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

48 Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. McKinley, and Emmett Witchel. Practical fine-grained information flow control using Laminar. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(1):4:1–4:51, 2014.

49 Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010.

50 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

51 Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. Let's face it: Faceted values for taint tracking. In *European Symposium on Research in Computer Security*, 2016.

52 Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the Web 2.0. In *Dynamic Languages Symposium (DLS)*, 2006.

53 Manuel Serrano and Vincent Prunet. A glimpse of Hopjs. In *ACM International Conference on Functional Programming (ICFP)*, 2016.

54 Ankur Taly, Úlfar Erlingsson, Mark S. Miller, John C. Mitchell, and Jasvir Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE Security and Privacy (Oakland)*, 2011.

55 Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming (ESOP)*, 2005.

56 Peter Thiemann. A type safe DOM API. In *International Workshop on Database Programming Languages*, 2005.

57 Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Trust, but verify: Two-phase typing for dynamic languages. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015.

58 WordPress 4.6.1 security and maintenance release. `https://wordpress.org/news/2016/09/wordpress-4-6-1-security-and-maintenance-release/`. Accessed Mar 24 2017.

I Can Parse You: Grammars for Dialogs

Martin Hirzel[1], Louis Mandel[2], Avraham Shinnar[3], Jérôme Siméon[4], and Mandana Vaziri[5]

1 **IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA**
hirzel@us.ibm.com

2 **IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA**
lmandel@us.ibm.com

3 **IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA**
shinnar@us.ibm.com

4 **IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA**
simeon@us.ibm.com

5 **IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA**
mvaziri@us.ibm.com

Abstract

Humans and computers increasingly converse via natural language. Those conversations are moving from today's simple question answering and command-and-control to more complex dialogs. Developers must specify those dialogs. This paper explores how to assist developers in this specification. We map out the staggering variety of applications for human-computer dialogs and distill it into a catalog of flow patterns. Based on that, we articulate the requirements for dialog programming models and offer our vision for satisfying these requirements using grammars. If our approach catches on, computers will soon parse you to better assist you in your daily life.

1998 ACM Subject Classification I.2.7 [Natural Language Processing] Discourse

Keywords and phrases Bots, virtual agents, dialog managers, domain-specific languages

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.6

1 Introduction

This paper is about authoring systems that allow dialogs between humans and computers. When humans can converse with computers using natural language, computers can assist them in real-life situations where traditional human-computer interfaces are cumbersome. Recent advances in natural language processing paved the way to bring conversational human-computer interfaces to the mainstream. However, facilities for authoring such interfaces are lagging behind. It is not easy to program a robust yet powerful human-computer dialog. On top of that, there is a confusing variety of use-cases for conversational interfaces. This paper takes a stab at organizing these use cases by cataloging recurring conversational flow patterns. Based on that, this paper outlines a vision for a new programming model in which developers specify human-computer dialogs via grammars.

Commercially successful prior approaches for specifying human-computer dialogs are the finite-state approach and the frame-based approach [13, 19]. In the finite-state approach, dialog control is determined by an explicitly-specified directed graph [12, 18], whereas in the frame-based approach, the dialog is driven by filling slots in a form [3, 18]. Unfortunately, both approaches suffer from what we term the *coherence-flexibility dilemma*: a dialog should be both coherent (yield the right outcome) and flexible (adapt to the human), but these

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 6; pp. 6:1–6:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

goals conflict. For instance, coherence can be maximized via confining prompts and explicit confirmations, but those can annoy humans or even keep them from reaching their goals.

Besides elaborating on the above problem statement, this paper also offers a vision towards a new programming model that solves it. First, we observe that the human utterances in a dialog constitute a linear sequences of inputs. Authoring a dialog means imposing structure over that sequence. A well-known and successful formalism that does exactly that is grammars. The idea is that the conversational agent is like a parser, and the natural-language understanding is like a lexer in a compiler. Continuing with the analogy, the outcome of the conversation is like an abstract syntax tree. The programming language community has plenty of expertise in using grammars to specify parsers along with their outcomes. Here, we put a twist to that by turning entire utterances by the human in a dialog into individual tokens in a grammar.

Overall, human-computer dialogs are a hot trend in computing. There is a lack of programming models for authoring them. This paper gives an overview of this trend and outlines a vision for a new programming model for it.

2 Why Human-Computer Dialog

Natural-language human-computer dialogs have been a mainstay of science fiction for a long time. In fiction, humans and computers or robots can carry wide-ranging conversations and interact almost as equals. Given this appeal, it is not surprising that fact follows fiction, although we must obviously keep reasonable expectations for how natural this interaction is [19]. Many early dialog systems focused on travel, where a human who is away from home can use a good old voice phone to book, for instance, flights [13].

Recently, human-computer dialog systems, also known as chat bots or virtual agents, received much renewed attention [14]. Cutting through the hype, there are several good reasons for that. One is that common-place devices such as cars, thermostats, or watches are becoming “smart”, but we typically do not want large displays on them to interact via graphical user interfaces. Another reason is that even on laptops or smart-phones with adequate displays, we do not always want graphical interfaces with many screens and clicks. For instance, when using a messaging platform such as Slack, it can be preferable to interact right there with bots, rather than context-switching to a different application. Such integrated interaction also benefits from other messaging features such as history. Another reason is that when computers understand the way humans speak, humans need not adopt a form of “machine-speak”.

Given this trend, several companies are starting to offer services that let customers author and run their own chat bots. For instance, the authors of this paper became involved in this trend because their employer started offering the Watson Conversation Service (WCS) [12]. WCS and its competitors can be put to immediate use to augment existing websites, messaging platforms, or mobile apps with a conversational interface. Furthermore, they are already being used for dialogs with robots. In the not-so-distant future, computers will enhance human cognition and ultimately work side-by-side with humans. When automated cognitive assistants become rich in features and reason in human concepts, they will eventually reach an inflection point where non-conversational interfaces no longer suffice to interact with them.

Ultimately, natural-language interfaces take advantage of our biology. Large areas of the human brain are devoted to natural-language communication [11]. It is easy for humans to use speech in circumstances that prevent using graphical user interfaces, such as in the dark, or hands-free (e.g. while cooking), or eyes-free (e.g. while driving). In fact, natural-language interfaces make computing technologies accessible to parts of the population that were previously excluded, such as small children, elderly, or visually impaired people.

Now that we have motivated why humans should converse with computers using natural language, let us explore what kinds of conversations they may have.

3 Use Cases and Flow Patterns

Use cases for human-computer dialogs abound in a staggering variety of domains, from travel to retail, entertainment to medical to automotive to technology trouble-shooting, and beyond. At first glance, it would seem like dialogs in each of these domains look very different. Yet as someone setting out to provide dialog authoring environments, we wanted to ensure we understand and capture the common patterns. To this end, this section contributes a catalog of flow patterns for bots. The catalog maps out the terrain, gives structure, and establishes terminology. Such an overview is a prerequisite to prioritization so we can arrive at the right scope of what to focus on and what to leave out.

A *flow pattern* is an interaction of a few back-and-forth turns in a dialog that either has a single well-defined outcome or handles a single deviation from the primary purpose of the bot. People with a programming-languages background can think of the *outcome* of a conversation as a data structure that can serve as a parameter to an external service call or simply as a record of what happened in the conversation. The outcome data structure fits the type for all conversations by that particular (part of a) bot. A flow pattern is finer-grained and lower-level than a *use case*, which captures an entire conversational agent for a particular purpose that may involve several flow patterns. In other words, flow patterns are intended to be domain-independent and indeed occur across many use cases in many domains. We distinguish two kinds of flow patterns: *outcome-driven patterns*, where the back-and-forth of conversation is directed at producing an agreed-upon outcome, and *add-on patterns*, which can occur during outcome-driven patterns but delay or possibly even derail the outcome. Another way to think of it is that outcome-driven patterns are about coherence whereas add-on patterns are about flexibility.

We conducted an informal survey and interviews with assorted product and client teams at IBM, and distilled our learning into the following set of flow patterns:

Outcome-driven flow patterns: Question answering, command and control, form filling, diagnosis, problem resolution, query formulation.

Add-on flow patterns: Anaphora and ellipsis, repair, disambiguation, elaboration, stopping a discussion, digression, interleaved dialogs.

Below, we elaborate on each of these patterns with brief definitions and discussions as well as example dialogs between a human (H) and a computer (C). Beyond this basic list, one can also consider bot features such as chit-chat, augmenting the bot with multi-modal input, vision, or touch, or making it empathetic by detecting and appropriately responding to the emotions of the human. To keep this paper focused, we elected not to elaborate on those other features here.

3.1 Outcome-Driven Flow Patterns

Question-answering flow pattern

Automatically answer questions posed by human in natural language based on a corpus of documents. From the perspective of the dialog flow, this is one of the simplest patterns.

One challenge is that questions often depend on context, such as the location. But the main challenge comes from dealing effectively with large numbers of questions. One issue is keeping the dialog in sync with the corpus. Another issue is that distinguishing between similar questions challenges current natural-language understanding technologies.

H: Where is the gift shop?
C: In the front lobby near the elevator.

Figure 1 Example for question-answering flow pattern in hotel concierge use-case.

Command-and-control flow pattern

Control a device through voice or natural language (e.g., radio, phone, thermostat, coffee machine, wheelchair). From the perspective of the dialog flow, this pattern is also simple. One challenge is detecting the conversation start, typically handled by anthropomorphizing the bot with a name such as "Olli" in the example. Another challenge is the integration with the device, which necessitates calling out from the dialog to external service code. And a further-reaching challenge is safety: if the device being controlled performs more serious actions than playing music, it is more important to ensure the human was understood correctly.

H: Olli, play some jazz music.
C: Okay, playing WBGO Jazz 88.3 FM.

Figure 2 Example for command-and-control flow pattern in car dashboard use-case.

Form-filling flow pattern

For each slot in a form, ask the human for the information to fill the slot. The dialog should catch mistakes without being too pedantic. One challenge is to allow the human to control the order in which slots get filled. Another challenge is that a single human utterance may contain multiple pieces of information. Doing this naturally in a finite-state approach is cumbersome. On the other hand, frame-based dialog interpreters are custom-tailored to this pattern [3, 18].

C: Welcome to the pizza dialog! What would you like?
H: A salami and mushrooms pizza.
C: Okay, salami and mushrooms. Any other toppings?
H: No thanks.
C: What size?
H: Medium, please.

Figure 3 Example for form-filling flow pattern in pizza-ordering use case.

Diagnosis flow pattern

Ask the human diagnostic questions following a decision tree to reach a leaf with a classification. The computer may be able to omit some questions if it can obtain the answers from context. One challenge is that the human may retract an earlier answer, either because they were uncertain or because they want to explore alternatives, thus requiring the bot to back-track. Another challenge is that questions posed by the computer are typically hand-crafted and cannot simply be derived from the name of a field in the outcome.

C: Welcome to the pizza dialog! What would you like?
H: A salami and mushrooms pizza.
C: Okay, salami and mushrooms. Any other toppings?
H: No thanks.
C: What size?
H: Medium, please.

Figure 4 Example for form-filling flow pattern in pizza-ordering use case.

Problem-resolution flow pattern

After diagnosis, walk human through how to fix their issue. The main challenge here is to actually come up with good instructions for fixing common issues, and refine the conversational agent so humans have success with it. Also, as in diagnosis, this may involve context and back-tracking.

C: Turn on the projector and connect your video cable.
H: Okay, it is on, but not showing my display!
C: Hit Fn-F7 on your laptop.
H: Good, now I can see my display. Thanks!

Figure 5 Example for problem-resolution flow pattern in technology trouble-shooting use case.

Query-formulation flow pattern

Query information in a database or knowledge base. One of the challenges with this pattern is that it is meta-data specific: human utterances mention pieces of the database schema. A related challenge is to enforce valid queries. Finally, if the human refines the query over the course of the dialog, they tend to use anaphora and ellipsis.

H: What was the median salary for working women in the US in 2009?
C: The median weekly earnings for women in 2009 was 657 US$.
H: Give me that number per race and ethnicity.
C: Here are the corresponding numbers: [. . .]

Figure 6 Example for query-formulation flow pattern in census use case.

3.2 Add-On Flow Patterns

Anaphora and ellipsis flow pattern

In linguistics, an anaphora is a reference, for instance, “it” in Figure 7 Line 3. An ellipsis is an omitted clause, for instance, the implied “from Croton-Harmon to Grand Central” in the last human (H) utterance in Figure 7 Line 5. Both anaphora and ellipsis must be understood from context. Furthermore, they involve linguistic subtleties that make natural-language understanding more challenging.

H: When is the next train from Croton-Harmon to Grand Central?
C: At 9:03am.
H: Is it peak or off-peak?
C: That's a peak-hour train.
H: When is the next off-peak train?
C: The next off-peak train from Croton-Harmon to Grand Central is at 9:28am.

Figure 7 Example for anaphora and ellipsis flow pattern in travel use case.

Repair flow pattern

Repair describes how parties in a conversation deal with problems in speaking, hearing, or understanding. These problems can occur both when uttering something and when receiving it, for both human and computer. Challenges include detecting the need for repair and wrapping up the repair in a small number of conversation turns, yet avoiding a repeat of the misunderstanding.

H: How much is full total coverage?
C: What kind of coverage? The choices are comprehensive or collision.
H: comprehensive
C: For your car, comprehensive is $983.70.
H: I meant collision.
C: For your car, collision is $685.40.

Figure 8 Example for repair flow pattern in insurance use case.

Disambiguation flow pattern

Agree upon one among alternative meanings of a word or phrase. As for repair, one challenge is detecting the need for disambiguation in the first place. Another challenge is to predict the likelihood that the default choice is good enough, to make a judgment call whether disambiguation is worth the effort.

H: Schedule a meeting with Alice at 10am.
C: Alice is in Texas. Do you mean 10am Central Time?
H: No, I meant 10am Eastern.

Figure 9 Example for disambiguation flow pattern in calendaring use case.

Elaboration flow pattern

Let the human obtain more information on the same topic. The challenge is to anticipate what the human might want to elaborate on, and to keep the system tractable despite the possible explosion of topics.

H: What's the weather tomorrow?
C: 30 degrees with snow.
H: How much snow?
C: 18 to 24 inches. Better stay at home!

Figure 10 Example for elaboration flow pattern in weather use case.

Stopping a discussion flow pattern

Let the human interrupt a conversation or a sub-dialog. One challenge is to decide on the right depth of interruption, i.e., how far to unwind nested levels of subdialogs. Another challenge is that the user might later change their mind about stopping the discussion, and want to resume it again.

C: Do you take anything in your coffee?
H: Skim milk.
C: I'm sorry, we're out of skim milk.
H: Then I don't want coffee.
C: Is there anything else I can get for you instead?

Figure 11 Example for stopping a discussion flow pattern in barista use case.

Digression flow pattern

Allow the human to take the conversation to a different, unrelated topic, and resume the prior topic afterwards. This requires the computer to keep the context of the main thread of conversation, and to re-establish the context for the human when switching back. A variant of digression is temporarily switching to a different part of the same dialog.

C: What's the algorithmic complexity of inserting an item in a B-tree?
H: What are B-trees used for?
C: B-trees are used when not all of the data fits in memory, for instance, in databases. What's the algorithmic complexity?
H: Hm, I guess logarithmic?

Figure 12 Example for digression flow pattern in tutoring use case.

Interleaved dialogs flow pattern

Conduct two or more dialogs concurrently. As in the example, this pattern comes up in use-cases where the conversation is about a situation that unfolds slowly in the real world. Beyond the challenges of digression, this requires keeping multiple contexts active. This makes it more challenge for both participants of the conversation to follow which dialog an utterance belongs to.

H: Correct course to the South.
C: How many degrees?
H: Where is the other vessel headed?
C: 20 degrees Northwest.
H: Okay, correct course 15 degrees South.
C: Initiating course correction.
C: The other vessel is no longer on collision course.

Figure 13 Example for interleaved dialogs flow pattern in naval use case.

The following section describes the most important add-on flow patterns from a different perspective that hints at a solution for them.

C: What size?
H: Mega-sized!
C: Okay, medium. What crust?
H: Uhm, I didn't say medium.
C: Okay, correcting the size. The choices are small, medium, or large. What size?
H: Large.
C: You picked large size. Is that correct?
H: Yes.
C: What crust?

Figure 14 Example dialog with correction.

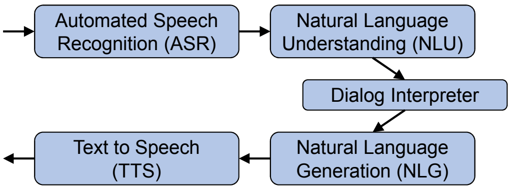

Automated Speech Recognition (ASR)
Natural Language Understanding (NLU)
Dialog Interpreter
Natural Language Generation (NLG)
Text to Speech (TTS)

Figure 15 Basic chat bot architecture.

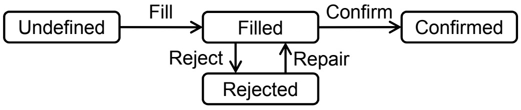


Figure 16 Outcome coherence state machine.

4 Coherence-Flexibility Dilemma

A dialog has *coherence* if its participants have common ground, i.e., they reach mutual belief that contributions have been understood sufficiently well for current purposes [6]. Coherence is essential to reaching the desired outcome of the dialog. Unfortunately, today's natural-language understanding (NLU) technologies have limitations that can put coherence at risk. When that happens, the add-on flow patterns repair and disambiguation from Section 3.2 are crucial.

A dialog has *flexibility* if its participants have both perceived and real control over it. If the dialog has too little flexibility, humans find it difficult to use, since they essentially have to learn its "machine-speak". Ultimately, humans will get frustrated and seek alternative communication channels. Flexibility relates to the add-on flow patterns elaboration, stopping a discussion, and digression from Section 3.2.

The *coherence-flexibility dilemma* is that these two goals are diametrically opposed. It is easy to accomplish one while ignoring the other. But techniques that improve coherence reduce flexibility and vice versa. Some of the early IVR (interactive voice response) systems were so inflexible that people referred to them as phone jail, trying to escape by immediately demanding a human operator.

The good news is that this dilemma is not unique to human-computer dialog. Coherence and flexibility are essential to human-human dialog as well, and human-human dialog has natural and effective ways to balance them. For instance, Clark and Schaefer argue that each utterance has two purposes: first, a backward-looking confirmation of the previous utterance, and second, a forward-looking question or statement advancing the conversation [7]. There is flexibility in how implicit or explicit the confirmation is, and humans adjust their style when they detect misunderstandings.

Consider for example the dialog in Figure 14. When the computer says "Okay, medium. What crust?", it attempts an implicit confirmation of what it understood (backward-looking) and asks the next question (forward-looking). The human corrects the computer. Next, the computer rephrases the question for the size by explicitly listing the choices, thus sacrificing some flexibility to improve coherence. After the human picks an option, the computer conducts a more explicit confirmation before going back to business as usual.

The good thing about this natural confirmation mechanism from human-human dialog is that it reduces the burden on the NLU technology. Instead of waiting for computers to get better at understanding natural language, we can work with the limitations of current technology if we are prepared to handle the occasional misunderstanding. And even if computers reach human parity in conversational NLU, that still does not imply zero errors, so repair capabilities remain necessary.

Figure 15 shows a simplified architecture for conversational agents, based on literature surveys [13, 19]. Human speech first gets converted to text, and then NLU extracts relevant inputs for the dialog interpreter. Symmetrically, the outputs from the dialog interpreter first get converted to text, and then synthesized back to speech. If the human interacts at the textual level, the speech components can be omitted from the architecture. While there is work on sophisticated NLU that understands parts-of-speech etc., this is brittle when human utterances defy grammar and is harder to port between different natural languages. Therefore, recent systems adopt a simpler and more robust approach to NLU, which merely extracts *intents* and *entities* from the human utterance [12, 23]. An intent is something like "turn on radio", and an entity is something like "jazz music". Intents can be detected via machine-learning classifiers, and entities via pattern-matching.

For coherence, we propose that each piece of the outcome data structure be subjected to the state machine in Figure 16. When NLU extracts an intent or entity, that can be used to fill a slot. But being merely filled is not enough. The computer gives the human an opportunity to confirm or reject a slot before it considers it part of the common ground. For flexibility, we propose enabling humans to take the *initiative* when they want to. The most operable definition of initiative we found comes from Derek Bridge, who simply says it belongs to whoever contributes the first part of a conversational adjacency pair [4].

Now that we have established the kinds of dialogs we want to enable, we will get back to the question of how to author them.

5 Grammars to the Rescue

Let us briefly summarize the requirements for a programming model for conversational agents. The agent needs to conduct a linear sequence of interactions with a human over time, consisting of utterances in a conversation. From this sequence of interactions, the agent needs to construct an outcome that adheres to a known type. However, it must detect and fix misunderstandings and allow the human to go off-script by grabbing the initiative where appropriate. Finally, the programming model should be easy to learn, ideally reusing widely-known, familiar, and well-understood programming-language concepts.

We hypothesize that grammars address these requirements well. Grammars specify parsers that process a linear sequence of tokens, produce an outcome, and can be made robust to certain kinds of errors. Most computer science students learn about grammars early in their education, so the concepts should be familiar to them. As an added benefit, in many cases, the outcome of a conversation gets transformed into a command or query for another system, which is itself also best characterized by a grammar. Therefore, our vision is to use grammars as a programming model for conversational agents. Finally, grammars are naturally compositional, thus facilitating modularity and reuse in dialog specification.

To be clear, in our vision, the grammar operates over tokens at the granularity of entire utterances by a human. In other words, we are not concerned with using grammars to parse an individual natural-language sentence, unlike some prior work [5, 20]. In terms of the architecture in Figure 15, we are proposing grammars to specify the dialog interpreter, not other components such as the NLU. Circling back to the paper title ("I can parse you"), we are envisioning a chat-bot as a parser for its human interlocutor, and we are thinking of the NLU component as merely the lexer that extracts tokens in the form of intents and entities from human utterances.

```
query        : select from where?;

select       : selectExpr+;
selectExpr   : selectColumn | selectAll | selectAs;
selectColumn : columnName;
selectAll    : "all" / "*" / "all␣columns" / "star";
selectAs     : expression columnName;

from         : tableName+;
where        : condition;

condition    : andCondition | orCondition | eqCondition;
andCondition : condition and condition;
and          : "and" / "conjunction";
orCondition  : condition or condition;
or           : "or" / "disjunction";
eqCondition  : expression eq expression;
eq           : "equals" / "=" / "is" / "equality" / "the␣same␣as";

expression   : STRING | INT | columnExpr;
columnExpr   : columnName;

tableName    : TABLE;
columnName   : COLUMN;
```

Figure 17 Grammar that guides database query dialog.

$$\begin{tikzcd}
\text{Grammar} \arrow[r, "\text{translates to}"] \arrow[d, "\text{guides}"'] & \text{Type} \arrow[d, "\text{validates}"] \\
\text{Dialog flow} \arrow[r, "\text{produces}"'] & \text{Outcome}
\end{tikzcd}$$

Figure 18 Commuting diagram. For examples see Figures 17 (grammar), 19 (dialog), 21 (outcome), and 20 (type).

We are still in the early stages of developing our programming model, but to make things more concrete, Figure 17 offers an example. The grammar guides a dialog that allows the human to issue a database query. It is based on a SQL subset and would be easy to extend for additional clauses (e.g., order-by) and expressions (e.g., arithmetic). As is customary in databases, the grammar distinguishes conditions from expressions, thus enforcing syntactically where one or the other can occur. By design, most of the grammar will look familiar to readers, with non-terminals (e.g., *query*), literal terminals (e.g., `"all"`), multi-valued terminals (e.g., *TABLE*), options (?), repetition (+ or `*`), choice (|), and products (e.g., *select from where*?). One new feature is synonyms (/), for instance, `"and"/"conjunction"`. Synonyms are viewed as samples for NLU tokens, and it is up to the implementation how literally to take them. In some cases, they just serve as training inputs for a machine-learning classifier. As one would expect from non-trivial grammars, this grammar includes nesting (*expression* within *selectAs* within *query*) and even recursion (an *andCondition* can be nested as a *condition* within another *andCondition*).

A single grammar such as the one in Figure 17 both guides the flow of the conversation and specifies the type of its outcome. As illustrated by the commuting diagram in Figure 18,

1 C: Which table do you want to query?
2 H: The Customers table please.
3 C: Okay, Customers. Any other tables?
4 H: Nope.
5 C: Is there a condition?
6 H: Yes: the country is "US".
7 C: Okay, the country equals "US". Any other conditions?
8 H: Yes: the zipcode is 55555.
9 C: Okay, the zipcode equals 55555. Any other conditions?
10 H: Nope.
11 C: What should the query return?
12 H: What are the available columns?
13 C: The available columns are: name, age, address, and zipcode.
14 H: The name
15 C: Ok, the name. Anything else being returned?
16 H: The address.
17 C: Ok, the address. Anything else being returned?
18 H: No, that's all.
19 C: Here is the query:

```
SELECT name, address
FROM Customers
WHERE country = "US" and zipcode = 55555;
```

Would you like to issue it?
20 H: Yes please.

Figure 19 Dialog for database query based on grammar in Figure 17.

the grammar guides the dialog flow. The grammar does not and should not explicitly specify every last detail of the dialog flow. Keeping the flow somewhat under-specified gives the dialog interpreter room to adapt, for instance, by making confirmations more or less explicit depending on whether there are many or few repairs. When the dialog interpreter uses the grammar to implement the dialog flow, it produces an outcome. The outcome could be represented as a JavaScript Object Notation (JSON) document. We implemented a translator from dialog grammars to TypeScript types that validate the final outcome. A coherent dialog yields a valid outcome, but a flexible dialog populates it in the order and style preferred by the human user.

Figure 19 shows a mock-up example dialog driven by the grammar in Figure 17. It occupies the lower left corner of Figure 18. On Line 1, the computer prompts for a table (to fill the *from* factor of the *query* product). On Line 2, the human gives a response in colloquial natural language, from which the NLU extracts a *TABLE* token with the value *Customers*. On Line 3, the computer first echoes back the table name (to establish common ground) and then asks whether there are other tables (implementing the repetition *tableName*+ in the grammar). On Line 4, the human says "Nope", from which the NLU extracts an intent of "no". On Line 5, the computer asks whether there is a condition (implementing the option *where*? in the grammar). On Line 6, the NLU can extract multiple tokens from a single human utterance: an intent of "yes", a *COLUMN* token "country", a synonym "is" for "equals", and a *STRING* token "US". The conversation continues to flow as specified in the grammar. Note that while the grammar specifies the select clause first, the order can be re-arranged to get a more natural dialog from a user standpoint, in this example by asking first about the SQL from clause rather than the select clause.

We have not yet implemented our ideas to support the example in Figure 19. Besides illustrating the aspiration of grammar-driven dialog flow, the example also illustrates additional desirable features for which we have yet to design integration points. For instance, on Line 12, the human goes off-script by requesting help. The computer replies with context-sensitive help driven by the underlying database schema. Later, on Line 19, the computer prints the outcome of the conversation using SQL syntax. Assuming that the raw outcome is a JSON document, this would require a simple pretty-printer. Both the schema-awareness and the pretty-printing are technically feasible but not specified by the grammar.

Figure 20 shows the TypeScript type for outcomes of our running example, and Figure 21 shows the concrete JSON outcome. The type and outcome occupy the top and bottom right corners of the commuting diagram in Figure 18. The type figure corresponds line-for-line to the grammar. For instance, Line 9 of the grammar, *from* : *tableName*+;, maps directly to Line 9 of the type, type FROM = { tableNames: TABLENAME[ ]; }. The JSON outcome in Figure 21 is essentially an abstract syntax tree for the SQL query in Figure 19 Line 19.

As we are embarking on the project to make this vision reality, it is healthy to also specify some success criteria. These can be found in the next section.

6 How Will we Know it Works?

When building a programming model for conversational agents, we aim at productivity for the dialog authors and quality for the actual human-computer dialogs. In a business context, the former decreases cost and the latter increases revenue. While monetary cost and revenue are concrete numbers, they will only become apparent when the system gets adopted. In the meantime, we need shorter-term metrics to guide our research.

For developer productivity, we are getting guidance from two sources. One is that, being in an industrial research lab, we have access to product and solution teams. We frequently solicit their opinion on the programming model we are proposing as our design is under way. Another source of guidance is to drive the development with several example programs of our own. We turned each flow pattern from Section 3.1 into a test case for our new programming model. In addition, we are extracting the essence from several customer use cases into test cases as well. The authors have had their share of experiences with programming language designs, some of which shipped in IBM products (e.g. [21]). Finally, some readers with a programming language background may appreciate the crazy idea of making our programming model meta-circular, by specifying a dialog whose outcome is the specification for another dialog.

For dialog quality, one might be tempted to adopt metrics from non-dialog tasks such as machine translation. Unfortunately, recent work demonstrates that these correlate very weakly with human judgment [17]. Such metrics are more appropriate to other components of the architecture in Figure 15 such as NLU, or to the simplest flow patterns such as question answering or command-and-control. Instead, a seminal paper on evaluating dialogs proposes ParaDiSE (Paradigm for Dialog System Evaluation) [22]. ParaDiSE postulates that the goal of a dialog is to maximize task success and minimize cost. In our terms, task success consists of producing the outcome the human wanted, and metrics for cost include the number of utterances, repair ratio, etc.

```
type TOP = { query: QUERY; };
type QUERY = { select: SELECT; from: FROM; where?: WHERE; };

type SELECT = { selectExprs: SELECTEXPR[ ]; };
type SELECTEXPR = SELECTCOLUMN | SELECTALL | SELECTAS;
type SELECTCOLUMN = { columnName: COLUMNNAME; };
type SELECTALL = "*";
type SELECTAS = { expression: EXPRESSION; columnName: COLUMNNAME; };

type FROM = { tableNames: TABLENAME[ ]; };
type WHERE = { condition: CONDITION; };

type CONDITION = ANDCONDITION | ORCONDITION | EQCONDITION;
type ANDCONDITION = { condition: CONDITION; and: AND; condition1: CONDITION; };
type AND = "and";
type ORCONDITION = { condition: CONDITION; or: OR; condition1: CONDITION; };
type OR = "or";
type EQCONDITION = { expression: EXPRESSION; eq: EQ; expression1: EXPRESSION; };
type EQ = "=";

type EXPRESSION = string | number | COLUMNEXPR;
type COLUMNEXPR = { columnName: COLUMNNAME; };

type TABLENAME = string;
type COLUMNNAME = string;
```

Figure 20 Type for outcomes of database query dialogs based on grammar in Figure 17.

```
{ query:
  { select : {
      selectExprs: [{ columnName: "name" }, { columnName: "address" }]
    },
    from: { tableNames: ["Customers"] },
    where: {
      condition: {
        condition: {
          expression: { columnName: "country" },
          eq: "=",
          expression1 : "US"
        },
        and: "and",
        condition1 : {
          expression: { columnName: "zipcode" },
          eq: "=",
          expression1 : 55555
        }
      }
    }
  }
}
```

Figure 21 Outcome of database query dialog in Figure 19.

7 Related Work

We found little prior work on using grammars or types to specify dialogs. Bridge beautifully codifies the essence of polite conversation into a grammar [4], but in addition, requires a separate task model to specify a complete chat-bot, whereas we use a grammar to specify the chat-bot itself. GF is a framework for describing grammars of natural languages [20], but focuses on the NLU and NLG components of the architecture in Figure 15, whereas we focus on the dialog interpreter. Bringert uses GF for dialogs [5], but requires the developer to specify multiple grammars and use dependent types, whereas our programming model uses a single grammar and is simpler. Finally, Denecke and Weibel propose a type system for dialogs that constrains individual slots in a frame [8], whereas our types specify the entire dialog flow and validate its outcome.

McTear wrote a survey about spoken dialog technology [19] and the text book by Jurafsky and Martin contains a chapter about dialog and conversational agents [13]. While neither of them focuses on programming models for dialogs, they describe different approaches, two of which imply simple programming models. First, the finite-state approach scripts the dialog at a low level, making it powerful but cumbersome [12, 23]. Second, the frame-based approach focuses on the form-filling pattern [3, 18]. One way to view our vision for a grammar-based dialog programming model is as a generalization of frames.

Other work on programming models for natural language includes natural-language interfaces to databases [1]; CNL (controlled natural language) [2, 9, 16]; and synthesis from natural language [10, 15]. Dialog can be viewed as an alternative that is less controlled than CNL but has a broader scope than synthesis.

8 Conclusion

This paper took a programming-language audience on a tour of the exciting trend of natural-language dialogs between humans and computers. We collected a catalog of relevant flow patterns and articulated the coherence-flexibility dilemma, which we believe to be the key to building good conversational agents. This paper also briefly described our vision of using grammars as a programming model for dialogs.

References

1 Ion Androutsopoulos, Graeme D. Ritchie, and Peter Thanisch. Natural language interfaces to databases – an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

2 Matthew Arnold, David Grove, Benjamin Herta, Michael Hind, Martin Hirzel, Arun Iyengar, Louis Mandel, V.A. Saraswat, Avraham Shinnar, Jérôme Siméon, Mikio Takeuchi, Olivier Tardieu, and Wei Zhang. META: Middleware for events, transactions, and analytics. *IBM R&D*, 60(2–3):15:1–15:10, 2016.

3 Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry Thompson, and Terry Winograd. GUS, a frame-driven dialog system. *Artificial Intelligence*, 8(2):155–173, 1977.

4 Derek Bridge. Towards conversational recommender systems: A dialogue grammar approach. In *European Conference on Case Based Reasoning (ECCBR) Workshops*, pages 9–22, 2002.

5 Björn Bringert. Rapid development of dialogue systems by grammar compilation. In *Workshop on Discourse and Dialogue (SIGdial)*, pages 223–226, 2007.

6 Herbert H. Clark and Susan E. Brennan. Grounding in communication. *Perspectives on socially shared cognition*, 13:127–149, 1991.

7 Herbert H. Clark and Edward F. Schaefer. Contributing to discourse. *Cognitive Science*, 13(2):259–294, 1989.

8 Matthias Denecke and Alex Waibel. Dialogue strategies guiding users to their communicative goals. In *European Conference on Speech Communication and Technology (EuroSpeech)*, pages 1339–1342, 1997.

9 Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for knowledge representation. *Reasoning Web*, pages 104–124, 2008.

10 Sumit Gulwani and Mark Marron. NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *International Conference on Management of Data (SIGMOD)*, pages 803–814, 2014.

11 Stephen Holland. Talents in the left brain, 2001. Retrieved Jan., 2017. URL: http://hiddentalents.org/brain/113-left.html.

12 IBM. Watson Conversation service. Retrieved Jan., 2017. URL: https://www.ibm.com/watson/developercloud/conversation.html.

13 Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, second edition, 2009.

14 Ron Kaplan. Beyond the GUI: It's time for a conversational user interface. *Wired*, 2013. URL: https://www.wired.com/2013/03/conversational-user-interface/.

15 Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. Learning to transform natural to formal languages. In *Conference on Artificial Intelligence (AAAI)*, pages 1062–1068, 2005.

16 Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, 2014.

17 Chia-Wei Liu, Ryan Lowe, Iulian V. Serban, Michael Noseworthy, Laurent Charlin, and Joelle Pineau. How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation, 2016. URL: http://arxiv.org/abs/1603.08023v1.

18 Bruce Lucas. VoiceXML for web-based distributed conversational applications. *Communications of the ACM (CACM)*, 43(9):53–57, 2000.

19 Michael F. McTear. Spoken dialogue technology: Enabling the conversational interface. *ACM Computing Surveys (CSUR)*, 34(1):90–169, 2002.

20 Aarne Ranta. Grammatical Framework: A type-theoretical grammar formalism. *Journal of Functional Programming (JFP)*, 14(2):145–189, 2004.

21 Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 360–384, 2014.

22 Marilyn A. Walker, Diane J. Litman, Candace A. Kamm, and Alicia Abella. PARADISE: A framework for evaluating spoken dialogue agents. In *Association for Computational Linguistics (ACL)*, pages 271–280, 1997.

23 Jason D. Williams, Nobal B. Niraula, Pradeep Dasigi, Aparna Lakshmiratan, Carlos Garcia Jurado Suarez, Mouni Reddy, and Geoff Zweig. Rapidly scaling dialog systems with interactive learning. In *International Workshop on Spoken Dialog Systems (IWSDS)*, pages 1–13, 2015.

Leveraging Sequential Computation for Programming Efficient and Reliable Distributed Systems

Ivan Kuraj[1] and Armando Solar-Lezama[2]

1 **MIT CSAIL, Cambridge, MA, USA**
2 **MIT CSAIL, Cambridge, MA, USA**

Abstract

While sequential programs represent a simple and natural form for expressing functionality, corresponding distributed implementations get considerably more complex. We examine the possibility of using the sequential computation model for programming distributed systems and requirements for making that possible. The benefits of such an approach include easier specification and reasoning about behaviors in the system, as well as a possibility to directly reuse existing techniques for checking correctness and optimization of sequential programs to produce efficient and reliable distributed implementations.

1998 ACM Subject Classification D.1.3 [Concurrent Programming] Distributed Programming, D.3.4 [Processors] Code Generation, Compilers, Optimization, F.1.1 Models of Computation

Keywords and phrases distributed systems, sequential computation, verification

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.7

1 Introduction

The sequential model of computation is often the most natural way to think about programming. Sequential programs represent computations which are performed by evaluating expressions in a predefined order. By contrast, programming distributed systems is much more challenging because programmers have to worry about controlling concurrent computations and consistency of their results, as well as a number of additional aspects, such as communication between nodes and managing data across the system.

Programming models for distributed systems need to provide means to deal with this additional complexity [5, 2]. Many modern programming models and languages for developing distributed systems allow expressing behaviors of individual nodes of the system in the sequential model, while using specialized abstractions and language constructs for specification of aspects such as communication [25, 13, 12, 7, 5]. For example, flexible and general models that allow controlling communication, expose it and require implementing it directly at a low level of abstraction, and force programmers to split behaviors into distinct program units with separate message sends and handlers [15, 26, 22]. The underlying programming models usually make trade-offs between the expressiveness and support for certain distributed aspects [5, 3, 11]. Finding an optimal trade-off, however, is difficult because models that are too low-level increase the complexity for the programmer, but models that are too high-level take away control from the programmer and risk producing code that does not match the programmer's expectations.

The paper presents a new programming model that aims to simplify the development of event-driven distributed programs. The key novelty of our programming model is that it

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 7; pp. 7:1–7:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

decomposes the problem of developing such applications into three distinct steps. In the first step, the programmer defines a data model and a collection of sequential routines that define the behavior of the system and are effectively treated as transactions. The programmer can reason about the semantics of each such transaction sequentially, without worrying about concurrency or data distribution. In the second step, the programmer uses type annotations to describe how the data is distributed across a collection of computational nodes. Based on these type annotations, the system determines where to execute the defined transactions and derives the necessary communication and synchronization in order to guarantee the sequential semantics of their executions. Finally, in the third step, the programmer defines a set of logical triggers, which dictate when the transactions should execute. The triggers can launch a transaction based on the global state of the distributed system.

We argue that the separation of concerns afforded by our approach leads to a programming model that is not just simple and expressive for developing distributed programs, but also simplifies reasoning about their behaviors, and enables various optimizations to produce *efficient implementations*. For example, the paper illustrates the potential for applying existing techniques for *verification and optimization of sequential programs* in this model. Moreover, the model effectively allows adding and changing both behaviors and distributed aspects without the need to change the code for existing behaviors.

Although demonstrated through a few running examples, the paper focuses on the ideas and a few key characterizations that are needed to define the new programming model. This paper does not present a fully-developed language that is general and applicable for producing efficient implementations for a wide range of distributed systems. Moreover, the paper focuses on the characterization and semantics of the programming model in terms of distribution of data and behaviors, and tying such behaviors to external stimuli. In turn, multiple aspects of realistic distributed systems, which include security and failure-tolerance, were omitted, making the model applicable only to certain scenarios of distributed systems where nodes operate in reliable and trusted environments.

2 From Sequential to Distributed Programs

2.1 Overview of the Approach

We illustrate the ideas behind the approach by implementing the functionality of a simple distributed application for managing storage and supplies of a central warehouse with multiple independent stores. We focus on the functionality of ordering an item from the warehouse (which possibly resides on a remote location) and updating the store's inventory accordingly. The developers start by specifying this behavior as a sequential program, from the perspective of only one store and the warehouse, ignoring distributed aspects of the system. In the subsequent step, the developers transforms the sequential program into a distributed implementation by specifying how the sequential behavior is instantiated for every store in the system, and distributed across all stores and the warehouse.

The developers start by defining the data used by the program. Given the chosen starting point of a sequential program with a single store, the developers declare a map from items (identified by `String`) to their quantities and a single variable that reflects the quota for items, which belongs to a particular store, but will later be instantiated at each store in the final distributed implementation[1]:

```
var quantities: Map[String, Nat]; var quota: Nat
```

[1] We use Scala-like syntax, with variables declared with `var`.

Afterwards, the developers implement the function for ordering some quantity of a particular item:

```
def order(item: String, quant: Nat): Boolean =
  if (quantities(item) >= quant && quota >= quant) {     // check if order eligible
    quantities(item) = quantities(item) - quant;        // update (inventory) state
    quota = quota - quant;
    true                                                        // order successful
  } else false
```

which, given the item and quantity to order, performs a simple check: if the given quantity is available in the warehouse and the store has sufficient quota left, it updates the corresponding variables and returns `true`, otherwise returns `false`. For simplicity, we assume each item takes exactly one unit of the available quota.

An important insight behind the new approach is that, modulo distribution, this simple function faithfully reflects the behaviour of item ordering. The functionality does not depend on how the data and computation are distributed; adding distributed aspects effectively provides a different view and instantiation of the functionality. Note that unlike some distributed algorithms that inherently require programming low-level aspects like communication (e.g. next actions depend on the received messages at particular nodes during execution), functionality in our example can fully be captured by ignoring distributed aspects [20]. Even though distributed implementations differ from “pure” behaviors expressed with sequential computation, given the necessary information about distributed aspects, they can be used as specifications of behaviors that can be automatically transformed into final distributed implementations.

To that end, since the sequential computation model is not sufficient to characterize such distributed applications, we identify components of specification that can, when coupled with sequential programs, completely characterize distributed implementations:

- allocation of data (used in the function) to different nodes in the system
- specifying how is the behavior (defined by the function) invoked in the system
- consistency of data and behaviors (in the presence of concurrency) in the system

Provided this additional information, developers can capture the desired behaviours in the system and allow the compiler to produce the appropriate distributed implementation. The compiler, which produces the final low-level implementation, can then decide to allocate computation and communication in a way such that the given specification – of data allocation, consistency and triggering of behaviors – is satisfied. Note that low-level details of the resulting implementation, such as communication and computation allocation, can still be decided and implemented in potentially multiple different ways by the compiler. We consider possibilities for providing and satisfying these specifications in the subsequent sections.

2.2 Location-dependent Types

We propose specifying data (and computation) allocation through types. To that end, we allow declaring nodes that participate in the system and enrich the type system to specify location of data or computation represented by the given expression. For any expression, besides associating standard type to it (e.g. in simply typed lambda calculus), we associate an additional label that designates the node the expression resides on or is computed at.

For our running example, the developers declare two sets of nodes, *Server* and *Client*, where the *Server* is a singleton set (which represents the single, centralized, warehouse). Node types are effectively instances of a higher-level type; they serve as labels that allow distinguishing between nodes. Developers can then annotate declarations and expressions to specify the intended data allocation in the final implementation: having type `T`, with `T`$_{\diamond node}$ the developer designates allocation of the expression of the given type to *node*, which belongs to some node type. Then, developers assign `quantities` to the (single) *Server*, and `quota`, the arguments, and the result of the function `order` to the *Client* nodes:

`var quantities: Map[String,Nat]`$_{\diamond Server}$`;` $\forall client : Client$`, var quota: Nat`$_{\diamond client}$
$\forall client : Client$`, def order(item: String`$_{\diamond client}$`, quant: Nat`$_{\diamond client}$`): Boolean`$_{\diamond client}$` = ...`

To designate that `order` might be invoked by any store, developers write $\forall$ to quantify over the set of store (*Client*) nodes (similarly to classical type-dependent systems). Note that the definition of `order` is omitted; it remains the same as before.

It is interesting to consider how the location information given in types affects the resulting implementation. Specifically, the program states that both arguments, as well as the result of the function, are allocated to the client node. However, `quantities` – variable used in the function – is allocated to the server, thus communication between the nodes is inevitable. To produce an implementation that type-checks (performs allocation as specified), the compiler has to update quantities on the server and return the result back to the client. Thus, the intended implementation where a client node (i.e. a store) sends arguments to the server (i.e. the warehouse) and awaits a response would typecheck successfully and can be produced as the resulting implementation.

Given the previous definition, since the state update has to happen on the server, the following has to typecheck:

`(quantities(item) = quantities(item) - quant): Unit`$_{\diamond Server}$

meaning the expression is evaluated at the *Server*. However, value of the assignment can be computed also on the client store (if e.g. forced by a typing annotation):

`(quantities(item) - quant): Nat`$_{\diamond client}$

which would produce a correct, but less efficient implementation, since it would incur additional two-way communication between the client and the server (for `quant` and the result). Effectively, location-dependent types allow developers to dictate allocation of data and computation within the system. Moreover, they can be used to prevent certain communication (e.g. for security reasons), where unwanted implementations, which could lead to potential data leaks, would not typecheck.

2.3 Triggering Behavior with Logical Formulas

In order to capture how are behaviours in the system invoked, we propose defining triggers as logical formulas, which can talk about events occurring across the system, as well as data, i.e. state, located at different nodes in the system. Whenever the condition defined by the trigger becomes true, the associated behavior is invoked. By combining events that can be bound to arbitrary internal and external actions (or stimuli) in the system, with formulas, we gain flexibility for specifying various, possibly reactive, distributed systems.

In our running example, we would like to trigger `order` as a consequence of an explicit user action (e.g. user interaction). We assume to have event $EvOrder_{Client}(quant : Nat)$ that can occur at any store in the system and is parametrized by a value for quantity (populated at the time of the instantiation of the event). In our current prototype, events are declared with a special construct in the language, while compiler generates function calls for each declared event (with arguments that match parameters of the event), which, when called, “fire” the associated event in the system.

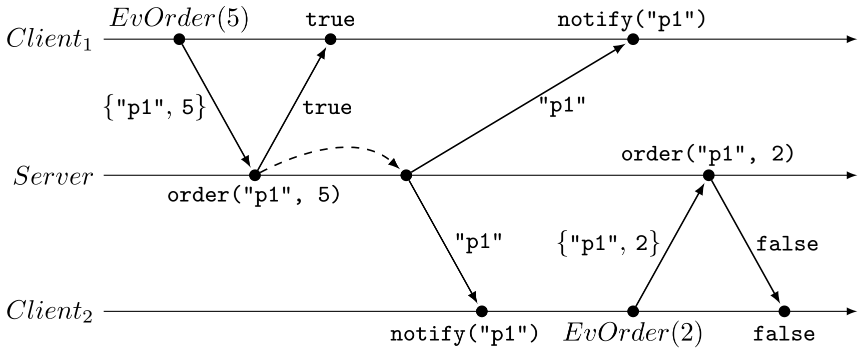


Figure 1 Distributed execution of two instances of `order` and one of `notify` behavior.

After declaring a variable for the selected item at every store, the developers can specify that `order` gets triggered whenever *EvOrder* fires at any client node (i.e. store):

```
∀client : Client, var selected: String◇client          // designates selected item
∃c : Client. EvOrder_c(quant) → order(selected◇c, quant)
```

where `order` is called with `selected` variable located on the client and the `quant` parameter from the event. (With `e`$_{\diamond n}$ we specify allocation of expression `e` to node n without specifying the full type.) The event expression on the left of the arrow can be thought of as pattern matching: variable `quant` captures the parameter value carried by the event.

Our programming model allows using logical formulas to specify triggering of behaviors when a certain condition becomes true in the system. If we consider a new functionality of notifying stores when the item they have selected gets out of stock, the following logical formula can specify such trigger. (We omit the function definition, which only updates the store-local state.)

```
def notify(item: String): String◇client
∃client : Client, item : String. selected◇client = item ∧ quantities(item) = 0 → notify(item)
```

The behavior returns a notification string (that is saved at the store client) and gets invoked whenever a store has selected an item (by changing `selected`) with quantity 0. (Note that a shorter condition with $quantities(selected_{\diamond client}) = 0$ can be written as well.) The semantics of triggering dictates that behaviors are invoked only once, whenever the condition changes from false to true in the system. Note that the produced implementation needs to incur communication between stores and the warehouse to check if the given condition became true, due to the $\exists$ quantifier (at least when the selected item at some store changes or quantity for some item becomes 0). To guarantee this, the compiler emits additional checks that check the condition whenever it might become true; more specifically, after every change to `quantities` on the server and `selected` on the clients.

2.4 Semantics and Consistency of Resulting Implementations

In order to characterize possible valid resulting distributed implementations, the model should constrain their behavior to match the behavior defined with sequential program, as well as the constraints of distributed aspects. Effectively, a valid implementation should project the given sequential programs onto the distributed system, respecting specified allocation and triggering conditions.

A possible execution of a distributed implementation of behaviors defined by `order` and `notify` is illustrated in Figure 1. The system includes one *Server* and two *Client* nodes.

We assume the initial value of `selected` is `"p1"` at both store clients and its quantity is `5` at the warehouse. The dots on the timelines designate the points in time of either firing of a trigger or evaluation of the given expression at the given node. Note that *EvOrder* events fire (after being invoked programatically) on the store client nodes, while the dashed line designates that the execution of `order` triggers the `notify` behavior; all three invoke the behavior defined by the appropriate function. Labels on the edges denote values that are being communicated between nodes. Note that behaviors are split, by the compiler, into multiple executions on potentially multiple nodes in the system.

An important aspect for enabling natural and convenient reasoning about (possibly concurrent) executions of behaviors is guaranteeing consistent execution. We propose allowing reasoning about end-effects of executions as if behaviors specified with sequential programs executed atomically, in the same order observed anywhere in the system (alike guaranteeing linearizable executions [14]). In Figure 1, the system executes the behaviors in a consistent way, relative to the linear order of triggering of each of the behaviors; more specifically, fist invoking `order` on $Client_1$ which causes `notify` to be invoked afterwards, followed by another `order` invoked on $Client_2$.

Even though providing such strong guarantees in the distributed setting might be costly (atomicity requirement might require effectively locking all nodes participating in the behavior beforehand [4]), we demonstrate the possibility for avoiding such overheads by analyzing the defined behaviors and their possible concurrent executions. We analyze three different cases of the resulting implementation, going from the variant that uses the most pessimistic mechanisms to variants that leverage the specifics of the behaviors to relax the used mechanisms arriving at a more efficient implementation that achieves the same results in terms of the intended semantics and strong consistency guarantees.

Let us consider the running example, with a single server for the warehouse and multiple store clients, with a small addition. In addition to the presented operations `order` and `notify`, for the purpose of examining negative effects of reordering of behaviors observed at store clients, we assume an additional operation `notifyAvailable`, which is similar to `notify`, but notifies the store that the selected item became available (its quantity became positive):

$$\exists \mathit{client} : \mathit{Client},\ \mathit{item} : \mathit{String}.\ \mathit{selected}_{\diamond \mathit{client}} = \mathit{item} \ \land\ \mathit{quantities}(\mathit{item}) > 0 \ \rightarrow \ \texttt{notifyAvailable(item)}$$

Note that, as shown in Figure 1, since the state is allocated according to specified location-dependent types, behaviors consist of code execution on both types of nodes, together with message sending and handling. Consequently, this allows inconsistent execution orders in which executions of `notify` and `notifyAvailable` are observed in different order on the server and clients. To guarantee strong consistency, the compiler needs to emit distributed implementations that prevent such inconsistent executions. We discuss different mechanisms the compiler might choose to use in this case, depending on the analysis of the semantics of the involved behaviors:

Distributed Locking. A pessimistic method for ensuring strong consistency can be achieved by using distributed locking. Commonly used in transactional processing, distributed locking tries to arrange a particular set of nodes to agree on a particular transaction, avoiding inference from other transactions, effectively locking those nodes for exclusive rights of the transaction [4]. Although achieving strong consistency (through strong serializability, where the order of transactions is defined by the acquisition of locks), algorithms for achieving such locking are prohibitively expensive and often unusable in practice.

In our example, this method can straightforwardly be applied to all the nodes in the system, such that behavior in the system can be invoked only after "acquiring" the global lock. This approach clearly ensures strong consistency, albeit at a prohibitive cost of allowing execution of only one behavior (either that of `order` or notifications) at any point in time across the whole system.

Central Point of Serialization. A simple observation in our example, where behaviors overlap at the single warehouse server, reveals the possibility to achieve more efficient, but also strongly consistent implementation. More specifically, it is sufficient to rely on the warehouse server to enforce ordering between executions of behaviors that might interfere, at the point they are executed on the server. A common mechanism for achieving this is to simply assign an index to messages that correspond to behaviors at the central point of serialization (in our case, the server), so that all nodes in the system can order messages, and thus corresponding executions [2, 4].

In our example, since the functionality of ordering and notifications depend only on the state that is located on the server, the server assigns an index to messages that carry the resulting values (e.g. a Boolean value for `order`) so that clients deliver (and end) behaviors in the same order as on the server. Guaranteeing the same order of observing behaviors is sufficient to guarantee linearizability, where the effects are the same as if behaviors were executed in a serial manner. Note that, e.g. when a notification is issued for an out-of-stock item (`notify`), it is issued after the `order` that caused it, so that store clients always receives confirmations of their orders and corresponding out-of-stock messages in the right order (with the need to store messages that are received out-of-order to deliver them later).

Removing Redundant Executions. A further observation is that even though ordering of behaviors is sufficient, it is not necessary to execute all parts of behaviors in certain cases; some parts of executions can simply be ignored, while preserving the semantics.

Behaviors for notifications gets invoked whenever quantity of an item becomes 0 or gets increased from 0. If an item quantity becomes 0 and then gets increased, it is incorrect to observe the two different associated notifications in a different order on any of the store clients. (This clearly cannot happen if either of the two previously presented mechanisms is used.) Interestingly, for any set of notification behaviors that is executed, since both `notify` and `notifyAvailable` just mutate per-store state (perform destructive updates), the store clients can simply execute (i.e. observe) only the latest one, ignoring all the previous ones. Therefore, a simple optimization of the previous implementation is to always execute the latest notification behavior on the store clients, regardless of their order determined on the server and discard any out-of-order notifications. This does not violate the semantics and guarantees linearizable executions, while achieving better performance in cases of concurrent notification behaviors. Note that the compiler can perform this optimization in any such case of destructive updates (without side-effects).

A key insight behind our proposal is that the compiler, after analyzing defined behaviors and their concurrent executions, can discover that not only the most pessimistic implementation is possible, but also two further optimizations, arriving at a more efficient implementation that satisfies the semantics and consistency requirements of the given program. In the worst case, if the compiler cannot discover any optimizations, the emitted implementation can use the most pessimistic mechanism. We leave further concerns of handling semantics and consistency concerns, as well as possible optimizations the compiler can perform in the general case open, while assuming such strong guarantees in the rest of the paper. We did not fully explore the extent to which such a program analysis can detect and perform optimizations to arrive at efficient implementations for a wider range of distributed systems. It would be interesting to examine the possibility of employing various existing lower-level distributed algorithms and systems in the produced implementations.

2.5 Defining Dynamic Structure of Distributed Systems

Some distributed systems dynamically maintain structure which conceptually corresponds to a (sequentially defined) datastructure. In such cases, changes in the structure of the system reflect modifications of the corresponding datastructure. To demonstrate flexibility of the model in such cases, we incorporate a notion of a mapping between a datastructure and the structure of the desired distributed system.

Let's assume we want the structure of our system to correspond to a search tree (which is not uncommon, e.g. in large-scale computing [23]). Having declared nodes that store data in the system with $Node$ and client nodes with $Client$, the developers can designate the mapping between the defined binary search tree abstract datatype to nodes with $\leftrightarrow_\sigma$[2]:

```
$Node \leftrightarrow_\sigma BST$, trait BST
case object Leaf extends BST; case class Inner(l: BST, v: Int, r: BST) extends BST
```

which associates every `Leaf` and `Inner` instance to a node (of type $Node$) in the system. Afterwards, the developers can refer to a node associated with an expression `e: BST` with $\sigma(e)$. The programming model creates one mapping for every designated datastructure; it's purpose is to provide implicit associations between instances of the datastructure and labels that denote location. Therefore, location of tree instances `tree = Inner(l, v, r)` and `l` are, $\sigma(tree)$ and $\sigma(l)$, respectively (where, by default, the model treats them as different physical nodes as well).

Next, the developers implement insertion of a new element into the tree, where each key is assigned to a separate (newly created) node, with the following function:

```
$\exists c : Client,\ n : Node.\ EvInsert_{(c,n)}$(key) $\rightarrow$
def insert(tree: BST$_{\diamond n}$, key: Int$_{\diamond c}$): Inner = tree match {
  case Leaf => Inner(Leaf, key, Leaf): Inner$_{\sigma(tree)}$
  case Inner(l, v, r) => if (v < key) Inner(l, v, insert(r, key))
    else if (v > key) Inner(insert(l, key): Inner$_{\diamond\sigma(l)}$, v, r)
    else Inner(l, v, r) }
```

where the event $EvInsert$ represents an action on some client node c that targets a data node n. (Where the most common case for n is the node that represents the root of the tree.) Due to the declared mapping between nodes and trees, when a new tree node is created, a new data node in the distributed system is bound to it. Effectively, this code implements a distributed version of the tree, where each tree node is located on a separate physical node.

For illustration, the developers have annotated two expressions in the function, both of which if typechecked, should execute on nodes defined with σ. For example, insertion into the left sub-tree executes on, and creates, data node $\sigma(l)$, where $\sigma(l)$ is different from $\sigma(tree)$, thus insertion into sub-trees executes across different nodes in the system. Note that a valid implementation, which matches the datatype, needs to (know how to) create new tree nodes and assign appropriate values to them (i.e. initialize their state). (Interestingly, for an implementation closer to realistic scenarios, data nodes can be mapped to elements that can be easily created and migrated between different physical nodes, such as actors in the actor model [1].)

[2] Again, we use Scala syntax for defining abstract datatypes as class hierarchies.

3 Verifying Distributed as Sequential Programs

One of the insights behind our approach is that by allowing developers to specify distributed systems with sequential programs we can eliminate much of the complexity that stems from handling distributed aspects, and as a consequence, decrease the amount of programming errors, as well. Moreover, we will demonstrate that such an approach allows incorporating existing techniques for analyzing and verifying correctness of sequential programs into development of distributed applications.

3.1 Checking Correctness of Application Logic

In addition to simpler reasoning about the behavior of the system, alleviating the concurrency, communication, and other low-level details enables direct application of techniques for checking correctness of sequential programs. Here, we demonstrate that we can easily check functionality of the system with a standard verification technique, solely by the virtue of relying on sequential programs for specifying behavior.

Even though our running example is simple, we can imagine verifying the property that no order can be made if the given item is out of stock, regardless of the quantity. To check this property, we formalize it with the following logical formula:

$$\forall quant, item.\ quantities(item) = 0 \wedge (res = order(item, quant)) \rightarrow \ res = \bot$$

which can easily be translated into a verification condition and checked with an off-the-shelf SMT solver. After encoding this condition as an SMT instance, we verified it in less than half of a second with the CVC4 SMT solver.

Verifying the condition as a low-level distributed implementation would need to take into account the introduced distributed aspects and would become considerably more complex. This example hints that by separating functionality from specifying distributed aspects, we can leverage existing verification tools for sequential programs and effectively translate all the obtained guarantees to the resulting distributed implementations.

3.2 Checking Correctness of Concurrent Behaviors

The fact that we can rely on behaviors faithfully translated into strongly consistent executions of a distributed system affects the extent to which the system can be tested and checked. As one of the possible techniques that offer potential for scalability, we will consider model-checking concurrently executing behaviors and demonstrate an immediate gain in scalability, relative to model-checking low-level distributed implementations.

Let's add functionality of item transfer to our warehouse application: stores can transfer specified items (for simplicity, we allow only a single transfer of a predefined quantity) to other stores, while those items should be ordered at some point later from the warehouse (to account the transfer). Transfers effectively transfer quotas between stores, so the store that received a transfer can use it for orders, while the store that made the transfer needs to settle it with the warehouse (i.e. to decrease its quota accordingly). We omit some definitions: transfer is tracked with `tStat`, while the code for `order` now uses and updates a transfer, if any transfer at the store exists. The developers implement this functionality with:

$$\begin{array}{ll}\forall client : Client\texttt{, var tStat: String}_{\diamond client} & \texttt{// track a transfer}\\ \exists c1, c2 : Client.\ EvTransfer(c1, c2)(\texttt{item})\ \wedge c1 \neq c2\ \rightarrow \texttt{transfer(item, tStat}_{\diamond c1}\texttt{, tStat}_{\diamond c2}\texttt{)} & \end{array}$$

where they ensure that transferring an item can occur between two stores (here, located at client nodes `c1` to `c2`), as long as the two nodes differ. The condition should prevent the item from being transferred back to the original store and avoid chain of transfers without accounting the transfer with the warehouse (i.e. artificially inflating the quotas).

The developers can then formulate the correctness condition as an LTL formula, where the transfer status is appropriately encoded with predicates *pending* (transfer sent from a store), *received*, and *settled* (the pending transfer gets settled):

$$\forall \mathit{item}.\ \mathbf{G}(\mathit{received}(\mathit{item}) \to \mathbf{X}(\mathit{pending}(\mathit{item})\ \mathbf{U}\ \mathit{settled}(\mathit{item})))\,.$$

The formula states that for all items *item*, it is always (globally) true that, if *item* is *received*, starting from the next step, the transfer status of *item* will be *pending* until it is *settled*. Note that such a formula is sufficient since we assume only one transfer is possible. Model-checking this formula can reveal that the condition in the trigger is not sufficient: a store can get it's own item transferred back, simply by giving it and receiving it back, and use it without the necessary accounting.

By checking behaviors as transactional executions of given sequential programs (which is sound due to strong consistency guarantees), we can discover this bug in considerably less iterations than when checking low-level implementations, due to the combinatorial explosion of the search space caused by intertwined low-level steps, including message sends and receives.

4 Program Transformations as Optimization

Having functionality expressed with sequential programs enables applying program analysis not just for checking functional correctness, but also for optimizations; many semantic-preserving transformations that apply to sequential programs can be reused in order to generate more efficient distributed implementations.

4.1 Optimizing with Data Allocation

Data allocation greatly influences possible resulting implementations and their performance. In many cases, by invoking behaviors only when needed the compiler can optimize away much of the communication in the system, while preserving the specified functionality.

In our running example, when implementing `order`, instead of sending data to perform the check on the warehouse server, the compiler can generate an implementation with:

`(quota >= quant): Boolean`$_{\diamond \mathit{client}}$

which checks the given condition on the store client and thus avoids incurring unnecessary communication in case the condition is false (i.e. the store does not have sufficient quota to make the order and the order fails immediately).

Note that in this case as well, this optimization can be discovered and performed by the compiler automatically, without any intervention from the developers. At this point, the compiler only considers optimizations that decrease the amount of communication in the system, while in many cases other optimization metrics could be used as well. (The compiler can choose implementations that make different trade-offs; the implementation might incur less communication, but transfer more data overall.) The programming model offers possibilities for extending the compiler to consider different optimization metrics.

4.2 Removing Unnecessary Triggers

Conditions, which invoke behaviors, might trigger at any point and place in the system; in the general case, the compiler needs to insert code that checks the condition at many places in the implementation, pessimistically. However, often, there are much fewer places where conditions can actually trigger. In addition to only checking condition at places at which the variables mentioned in the formula of the condition might change, if proved that the condition cannot become true regardless of the value of variables, the condition checking at that place can be eliminated altogether. This optimization becomes more significant in cases where to check the condition, communication between nodes needs to be incurred.

In the running example, lets consider the distributed warehouse with previously defined behaviors, including notifications for an item becoming available for ordering (`notifyAvailable`, as presented before). In addition, developers add functionality for adding a certain quantity of an item to the warehouse (refilling the warehouse) as `addItem` (which is similar to `order`; we will omit the definition for brevity and assume it can be invoked at store clients similarly):

$\exists c : \mathit{Client}.\ \mathit{EvAdd}_c(\mathit{quant}) \to$ `\mathit{addItem}(\mathit{selected}`$_{\diamond c}$`, \mathit{quant})`

Having all these sequential definitions, in the worst case, the resulting implementation would need to check conditions for the two notifications (`notify` and `notifyAvailable`) both at the place where the item quantity gets decreased (in `order`) and increased (in `addItem`), since at those places item quantities change (and notifications might potentially need to be invoked). However, only two checks are (provably) needed: for the case of out-of-stock notifications, they surely cannot be triggered when item quantity is increased (in `addItem`). The compiler can guarantee this by checking the satisfiability of the following implication:

$$\exists \mathit{item}.\ \neg(\mathit{quantities}(\mathit{item}) = 0) \land (\mathit{res} = \mathit{quantities}(\mathit{item}) + 1) \to \mathit{res} = 0$$

which states that there exists an item, for which if the quantity was not zero (due to triggering only conditions that become true), after incrementing the item's quantity, the quantity can become 0. This logical formula is clearly not satisfiable. Therefore, an optimized implementation can completely omit checking the condition and the notification functionality in that case. In the produced implementation, this halves the total number of invocations of the functionality for both notifications, on average.

4.3 Inferring and Generating Contexts

Some distributed applications behave in specific ways depending on the current context: most notably, some behaviors might be enabled only under a specific context. In our programming model, such contexts can be specified implicitly in triggering conditions. However, one possible optimization that compiler can perform is to infer more general contexts from the specified triggering conditions, and maintain and propagate them across the system to optimize certain behavior executions, e.g. to avoid unnecessary communication.

As an illustration of the idea, in our running example, if developers change the definition of `order` and add an additional expression to the triggering condition, such that the functionality depends on the warehouse being non-empty (since otherwise the order will fail), as:

$\exists c : \mathit{Client}.\ \mathit{EvOrder}_c($`quant`$) \land (\exists \mathit{item}.\ \mathit{quantities}(\mathit{item}) > 0) \to$ `order(selected`$_{\diamond c}$`, quant)`

and add other functions that depend on the same condition or the negation of it (i.e. that the store is empty, $\forall \mathit{item}.\ \mathit{quantities}(\mathit{item}) = 0$), the program analysis can infer this as a context. If so, the compiler can then produce an implementation that propagates the state of the warehouse as the context, within messages for other behaviors, and prevent unnecessary executions of further `order` requests.

Effectively, given the specifications are satisfied, program analysis can abstract the resulting system as a state machine. In addition to being more efficient, due to prohibiting unnecessary executions depending on the context, the relation between behaviors and identified contexts can make reasoning and verifying the system easier.

5 Related Work

Many approaches presented in prior work focus on using sequential computation to some extent while introducing additional abstractions, such as remote procedure calls, reactive values, and conflict-free replicated data type, for handling distributed aspects of the system [7, 17, 25, 9, 24]. A related line of research includes programming platforms based on writing sequential programs that aim at abstracting away infrastructure concerns to allow focusing on the application logic [3, 18]. An overview of different programming models and the influence of the sequential model on programming distributed systems is given in [5, 2]. In general, even though these models abstract away some of the complexity, due to the close match between the program and the final distributed implementation, expressing certain complex behaviors requires low-level reasoning and careful structuring of the program [28, 26].

Our approach is aligned with the idea of using high-level specifications of distributed aspects and offloading the search for low-level implementations to the compiler. Some approaches lift the abstraction of specifying behaviors by using similar mechanisms to the ones employed by our approach, including logical formulas (used for triggering in our approach) in the form of event guards and await statements, and the concept of location, which allows automatic data distribution according to specifying computations [20, 16, 10]. Prior work discusses the importance of preserving semantics of sequential computation and its effects on possible optimizations, as well as the potential role for programming distributed systems [21, 19]. In the similar spirit, this work tries to motivate lifting the level of abstraction by demonstrating potential gains in simplicity and performance. Moreover, it provides a different perspective on formalization of sequential computation and specifications to allow additional means for ensuring correctness and efficiency of the resulting implementation.

While our approach focuses on implementing behaviors which can be conceptually expressed as sequential programs, it lacks expressiveness for programming distributed algorithms that inherently require dealing with aspects like processes and messages, and require control of low-level concerns [20, 26]. While re-implementing such algorithms is rarely needed, they often cannot be used directly via an external library (e.g. if modifications to some of its internals are needed); our approach aims at utilizing different existing algorithms as means to an end whenever necessary, even in cases their code needs to be customized for specific needs of the intended distributed application.

Our approach shares some of the high-level goals with the following lines of research on programming distributed systems:

Tierless Programming Models. Similar in spirit of avoiding the complexity and breaking the underlying programing model, tierless programming models focus on simplifying specification of aspects that cut across different tiers and unify them into a single model (and traditionally, focus on web development) [8, 27, 7]. Although these models simplify some of the aspects considered in this work, including communication, storage and interaction, their focus is to remove the complexity that arises due to handling different tiers of the system, rather than on preserving the semantics and structure of sequential computation within the same tier. Note that tierless models usually adopt existing mechanisms and constructs, such as client-server architecture and RPC for communication [7].

Actor-Based Programming Models. Despite being flexible and providing clean abstractions for programming distributed event-driven systems that can easily be mapped to actual physical systems, actor models suffer from being close to the low-level implementations, where the structure of the system and behaviors need to match closely with the declared programs, making them complex and hard to reason about [1, 15, 13, 26]. Interestingly, actor-based programming frameworks represent a good fit for a low-level model that can be leveraged in the final emitted implementations [19, 26].

Partitioned Global Address Space Partitioned global address space (PGAS) models aim to provide a simple programming model, and consequently allow better performance, for parallel programs by unifying the support for data and task parallelism, and abstracting the data model through a global address space [6, 10]. The concept of a "place" in these models allows allocating computations and data across the global address space, at a level that can be closer to the intended (sequential) behavior. Although places allow assigning a cost model to data accesses (based on the topology), automatic data distribution is usually restricted to partitioning of regular and dense data structures such as arrays; some PGAS languages require explicit distribution of data objects to remain expressive for irregular and sparse structures [10]. Nodes in our model are similar to places in PGAS in that they contain running computations, which in turn might be spread across multiple different nodes. However, our model does not rely on specific patterns of data distribution and parallelism; it analyzes defined behaviors to emit event-driven implementations that need to satisfy consistency guarantees, and appropriately allocate both computation and data.

6 Concluding Remarks and Vision

The sequential model of computation provides a natural way for expressing computation. However, the sequential model alone is not sufficient for programming distributed systems. As such, it is either heavily ignored, or to large extent complicated, in modern programming models and languages for distributed systems, due to the need to accommodate distributed aspects such as data allocation and communication.

This paper explores fully reusing sequential computation model for expressing behavior, while characterizing intended distributed systems with orthogonal specifications. With separation of concerns of expressing behavior and specifying distributed aspects of the system, by writing orthogonal constraints, we can achieve development of distributed systems without breaking the simplicity of writing and reasoning about sequential programs. We have shown an approach to specifying data and computation allocation through enhancing the type system and defining behavior invocations with logical formulas. We motivated the new approach by demonstrating potential benefits in the development process, not just in terms of simplicity in writing programs, but also checking their correctness and applying semantic-preserving optimizations for emitting efficient distributed implementations.

A number of challenges remains for completely characterizing the programming model and transforming it into a programming language expressive for development of realistic distributed systems. We only briefly discussed the strong semantics and consistency guarantees that the model should provide as an interface for developers, while demonstrating an approach that can emit efficient implementations in certain scenarios. Achieving strong guarantees, together with efficiency, in the general case, remains an open problem, for which a solution would potentially require combining multiple techniques and results from the domain of programming languages and distributed computing. As hinted in the paper, providing a high-level interface for specifying behaviors through sequential programs opens up possibilities for many lower-level design choices in the final implementation; one interesting venue to explore represents not just more flexible data allocation, but also data sharing and replication, and the needed mechanisms the compiler would need to utilize.

References

1 Gul Agha. Actors: a Model of Concurrent Computation in Distributed Systems. *MIT Press*, 1986.

2 Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 1989.

3 Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. Cloud-native, event-based programming for mobile applications. In *MOBILESoft*, 2016.

4 Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 1981.

5 Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and Distribution in Object-oriented Programming. *ACM Computing Surveys*, 1998.

6 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, Vivek Sarkar, 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 *OOPSLA*, 2005.

7 Adam Chlipala. Ur/Web: A simple model for programming the Web. In *POPL*, 2015.

8 Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *FMCO*. 2007.

9 Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *PLDI*, 2013.

10 Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, 2015.

11 Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *OOPSLA*, 2014.

12 Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with High-Level Programming Languages. *ESOP*, 2001.

13 Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 2009.

14 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 1990.

15 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, 1973.

16 K. R. Jayaram and Patrick Eugster. Program analysis for event-based distributed systems. In *DEBS*, 2011.

17 JMacroRPC – reactive client/server web programming. URL: `http://hackage.haskell.org/package/jmacro-rpc`.

18 Emre Kiciman, Benjamin Livshits, Madanlal Musuvathi, and Kevin C. Webb. Fluxo: a system for internet service programming by non-expert developers. In *SoCC*, 2010.

19 Ivan Kuraj and Daniel Jackson. Exploring the role of sequential computation in distributed systems: motivating a programming paradigm shift. In *Onward!*, 2016.

20 Yanhong A. Liu, Scott D. Stoller, Bo Lin, Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *OOPSLA*, 2012.

21 Daniel Marino, Todd Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. The Silently Shifting Semicolon. *SNAPL*, 2015.

22 M. Felleisen Matthews, J., R. B. Findler, P. T. Graunke, S. Krishnamurthi. Automatically Restructuring Programs for the Web. In *ASE*, 2003.

23 Mehul Nalin Vora. Hadoop-HBase for large-scale data. In *ICCSNT*, 2011.

24 Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *PPDP*, 2015.

25 Meteor – pure javascript web framework. URL: `http://meteor.com`.

26 Aleksandar Prokopec and Martin Odersky. Isolates, Channels, and Event Streams for Composable Distributed Programming. *Onward!*, 2015.

27 Manuel Serrano and Gérard Berry. Multitier programming in Hop. *Communications of the ACM*, 2012.

28 Andrew Stuart Tanenbaum and Robbert van Renesse. A critique of the remote procedure call paradigm. Technical report, Vrije Universiteit, 1987.

Intermittent Computing: Challenges and Opportunities*

Brandon Lucia[1], Vignesh Balaji[2], Alexei Colin[3], Kiwan Maeng[4], and Emily Ruppel[5]

1 Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA
2 Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA
3 Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA
4 Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA
5 Carnegie Mellon University, Department of ECE, Pittsburgh, PA, USA

Abstract

The maturation of energy-harvesting technology and ultra-low-power computer systems has led to the advent of intermittently-powered, batteryless devices that operate entirely using energy extracted from their environment. Intermittently operating devices present a rich vein of programming languages research challenges and the purpose of this paper is to illustrate these challenges to the PL research community. To provide depth, this paper includes a survey of the hardware and software design space of intermittent computing platforms. On the foundation of these research challenges and the state of the art in intermittent hardware and software, this paper describes several future PL research directions, emphasizing a connection between intermittence, distributed computing, energy-aware programming and compilation, and approximate computing. We illustrate these connections with a discussion of our ongoing work on programming for intermittence, and on building and simulating intermittent distributed systems.

1998 ACM Subject Classification C.0 Hardware/Software Interfaces, D.4.5 Reliability

Keywords and phrases Intermittent computing, Energy-harvesting devices

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.8

1 Introduction

Recent years have seen a shift toward increasingly small and low-power computing devices across a variety of application domains, including IoT devices [16], wearable, implantable, and ingestible medical sensors [31, 19], infrastructure monitors [28], and small satellites [46, 2]. Advances in energy-harvesting technology [34, 26, 18, 27] have enabled applications that run entirely using energy harvested from their environment without the restriction of tethered power or maintenance requirements of a battery. These devices harvest and buffer energy as it is available and operate when sufficient energy is banked. Operation in these devices is *intermittent* because energy is not always available to harvest and, even when energy is available, buffering enough energy to do a useful amount of work takes time. The hardware of an intermittently operating device can include general purpose computing components, such as a CPU or microcontroller (MCU), an array of sensors, and one or more radios for communication. Typical devices contain *volatile* memory that loses its state on a power

* This work was funded by National Science Foundation grant CNS-1526342 and a gift from Disney Research Pittsburgh.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 8; pp. 8:1–8:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

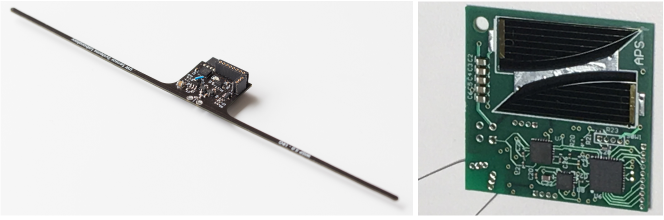


Figure 1 Two energy-harvesting devices. RF-powered WISP Platform [34] (left) and our solar-powered EDBsat single-board satellite (right).

failure, such as SRAM and DRAM, and *non-volatile* memory that retains its state on a power failure, such as Flash and FRAM [42]. Figure 1 shows two energy-harvesting platforms.

Programmers of today's intermittently operating devices use a typical, C-like embedded programming abstraction despite a number of important differences between the intermittent execution model and a typical embedded execution model. In particular, software running on an intermittently operating device executes until energy is depleted and the device browns out. When energy is again available, software resumes execution from some point in the history of its execution, i.e., the beginning of `main()` or a checkpoint [33, 17, 24, 3, 4, 23, 9, 43]. The key distinction between a conventional execution and intermittent execution is that a conventionally executing program is assumed to run to completion but an intermittent execution must *span* power failures. To tolerate power failures that occur hundreds of times per second, multiple layers of the system require an intermittence-aware design, including languages, runtimes, and application logic.

This paper provides a survey of current research challenges in intermittent computing and a vision for future intermittence research in the PL and systems community. To achieve that goal, Section 2 describes several PL and systems challenges brought about by intermittent computing. Section 3 describes the design space of intermittent computing devices, focusing on hardware and software characteristics that are likely to affect future research. A goal of this work is to show how intermittent computing brings together other areas of PL and systems research, including, distributed computing and concurrency, energy-aware programming and compilation, and approximate computing. Section 4 describes several programming languages research directions that address intermittence. This paper is intended to inspire and equip PL researchers to begin using and researching intermittent computing systems.

2 Intermittent Computing Challenges

Intermittent operation is an impediment to programming today's intermittently operating devices. The difficulty stems from the fact that an intermittent execution proceeds in bursts when energy is available and includes periods of inactivity when energy is not available. This succession of active and inactive periods is illustrated in Figure 2. Intermittent execution makes control-flow unpredictable, compromises an application's forward progress, leaves memory inconsistent, leaves a device inconsistent with its environment, and complicates device-to-device communication. We discuss these problems briefly and cite work exploring them in depth.

Control-flow. To an executing program, resuming after a power failure is a discontinuity in control-flow that is not explicitly expressed in source code. Programmers of intermittent devices must deal with implicit control flows to potentially unpredictable points in an

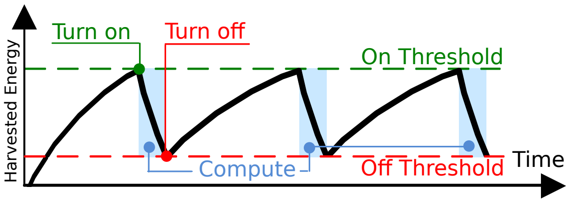


Figure 2 Intermittent execution. An intermittently-powered device executes its program in bursts as energy is available.

execution's history, such as a recent checkpoint [33, 24, 17, 23, 43] or the beginning of a task [6, 9].

Some platforms (like the WISP [34]), always begin executing with the same quantum of energy available and (in effect) do not recharge during execution[1]. If a restarted computation cannot successfully reach a checkpoint or complete a computational task using the start-time quantum of energy, then the system will unsuccessfully attempt to execute the same span of code repeatedly, preventing the program from making meaningful progress. This "Sysiphean" computation problem [33, 6] is particularly problematic in energy-starved environments. Guaranteeing forward progress in intermittent execution models is an important, unsolved challenge, especially for systems with explicit, statically-demarcated checkpoints and tasks.

Data consistency. Recent work [23] demonstrated that a naive combination of checkpointing and direct access to non-volatile memory in an intermittent device [33, 24, 17] can lead to memory inconsistencies. The key problem is that volatile state, such as the device's registers, stack, and global variables, are erased or rolled back to a previous state (e.g., a checkpoint) when the power fails. In contrast, the byte-addressable, non-volatile storage retains its values and those values may be inconsistent with the rolled-back volatile state. Keeping the contents of both types of memory correct requires careful, expert-level programming or system support [23, 9, 43] to ensure that non-volatile values are kept consistent with frequently erased or reverted volatile values. Due to the limited supply of energy, the time [23, 43, 33, 17] and space [9] cost of managing memory is a key factor that determines the resources available to the application.

Environmental consistency. Like other embedded systems, intermittently operating devices receive inputs from the outside world via sensors. Sensed data become stale and unusable if they are buffered across a long time period without harvestable energy. Sensor accesses intended to be atomic with one another may be split by a power failure, causing their resultant data to be inconsistent with the device's real environment. Prior work on system support for intermittent task atomicity [9, 23] avoids this problem by letting the programmer define tasks containing I/O that should re-execute atomically. Other work [11, 15] explicitly tracks time to avoid staleness issues.

Concurrency. Sensors, peripheral devices, and collections of MCUs may all operate concurrently as a single, intermittent device. As is common in embedded systems, concurrency with sensors is largely interrupt-driven. For example, an MCU may request data from a

[1] The recharge rate is non-zero, but negligible compared to the energy discharged during an execution period.

sensor, and a sensor may buffer and reply with data. Similarly, two MCUs may exchange and compute on data in parallel. Concurrency control in such scenarios is complicated by intermittent interruptions. If control-flow in one or more concurrent execution threads is re-directed to an earlier point by an intermittent power failure, how should the system manage the visibility in each thread of values produced by both threads? We are unaware of existing work that specifically addresses concurrency and intermittence together. Most existing intermittence research [9, 23, 43, 3, 4] assumes a single control thread and does not define the behavior of operations that are concurrent with intermittent control threads.

Compounding the state management problem, the timing, precision, and frequency of concurrent components are influenced by the availability of buffered and harvestable energy. Energy-dependent concurrency control becomes especially complex in a device with *federated energy storage* [14]. In a federated system, components charge and discharge their own storage elements independently. As a result, each component becomes an *intermittent resource* available at different times, depending on its energy supply and capacity. The software must synchronize access to the intermittent resources, not only in the relative logical time, but also in real physical time.

Distributed intermittent devices. Distributed collections of intermittently operating devices must interact with one another via radio. Most work has focused on physical-layer mechanisms to enable devices to communicate [22, 5]. We observe several reasons why coordinating distributed intermittent devices is difficult, beyond the issues at the physical layer. The cost of communicating is high: a fixed-length period of communication costs an order of magnitude more energy than a similar period of computation [20, 12]. Deciding when to incur the high cost of communication, and how much data to transmit or receive is a delicate trade-off of energy for precision or functionality. Synchronizing a collection of intermittently operating devices is an unsolved problem and a communication between unsynchronized, intermittent end-points is only successful if both are coincidentally operating for a long enough time, at the same time. A distributed intermittent system must gracefully allow communication to fail very frequently.

3 The Intermittent System Design Space

The challenges in Section 2 are a consequence of the hardware and software design of the energy-harvesting device. Exploring the design space is necessary to understand why programming intermittent devices is challenging and to inform future PL research on intermittent systems. The design space of intermittent devices is rich with inter-dependent hardware and software components that dictate behavior and applicability. Our discussion focuses on three design parameters: (1) energy harvesting and storage; (2) memory and execution models; and (3) software development toolchain.

3.1 Energy Harvesting and Storage.

The behavior of a program running on an energy-harvesting device depends on a number of factors: its energy-harvesting modality, energy storage mechanism, and power-on/power-off behavior.

Energy Harvesting. Energy harvesters vary widely from device to device. Solar panels deliver power proportional to their illuminated area. Solar harvesters with a compact form factor (cm^2) typically generate tens of μW to tens of mW of power. A device powered by RF energy depends on the availability of radio waves in a specific frequency range. Harvestable RF power varies from nW (ambient sources [22]) to μW (RFID-readers and power transmitters [34, 37, 30]). Mechanical harvesters range from nW-scale buttons [27] and sliders [18] to multi-Watt self-powered knobs [44].

In the simplest design the harvester output is connected directly to the load (i.e. MCU, sensors). This design is only appropriate if the harvester's current output matches the load's current draw (e.g., ~1 mA for a 4 MHz MSP430) and its voltage output is acceptable to the load (e.g., 1.8-3.3 V). In such a design, the duration of an intermittent execution interval equals the duration of the incoming energy burst. This design is rarely applicable, because the harvester rarely matches the current and voltage of the load. Instead, the load is usually *decoupled* from the harvester by an energy buffer, e.g. a capacitor. Hardware or software controls charging and discharging of the storage element. As a result, intermittent execution intervals are regularly periodic even if input energy is erratic.

Energy Storage. The energy buffering mechanism affects system and software behavior. Power systems that decouple the load and harvester operate in repeated charge-discharge cycles. First, the device accumulates energy, while consuming a negligible amount. With sufficient energy stored, the device begins to operate until the energy is depleted. The energy storage *capacity*, fixed at design time, determines the maximum amount of computation that is possible without a power failure.

The energy storage mechanism is a key design parameter because it dictates a device's physical size. Designers may be *volumetrically* constrained by an application (e.g., in-body devices [19]), limiting energy capacity and capability. Capacitors are cheap and small but not energy-dense. Super-capacitors are an order of magnitude more dense, but moderately larger and more costly. An energy harvester can also charge a small battery and, unlike a capacitor that appreciably leaks energy, the battery will leak slowly, permitting operation over long periods without harvestable energy. Batteries, however, have drawbacks. Conventional batteries (e.g., coin-cells, AA) are heavy and fragile. Thin-film batteries are light, but inapplicable in some harsh environments; e.g., suffering permanent failures in low-temperatures space applications [46]. Batteries wear-out, reducing efficiency and requiring replacement, which can be labor intensive or impossible in adversarial environments. Battery chemistry makes assessing a battery's remaining charge difficult. Voltage is a poor indicator of a battery's stored energy because capacity varies with wear, temperature, and workload. In contrast, a capacitor's voltage reflects its energy content, allowing hardware or software to read the voltage and react to energy events, such as a full charge or an impending power failure [33, 3, 8].

Energy Distribution. A device's pattern of intermittent execution activity depends on when energy accumulates and when it is consumed. Charge/discharge behavior can be implicit in the hardware, or controlled explicitly by hardware or software logic. Absent energy-distribution logic, a device will operate whenever its energy buffer's voltage is within operating range. However, relying on implicit on/off behavior is impractical because it leads to *thrashing*: the storage element never has time to accumulate a significant amount of energy before being drained. Instead, explicit on/off logic accumulates charge without consuming energy up to a threshold energy level. With a capacitor as the storage medium, the energy threshold level translates to a threshold capacitor voltage.

Two quantitative design parameters that lead to qualitative differences in system behavior are the turn-on and turn-off voltage thresholds. In some devices (e.g., WISP5 [45], Powercast [30]) the turn-on threshold is fixed in hardware to the maximum operating voltage. Setting the turn-on threshold to the maximum voltage makes the device turn on with maximum energy stored. Other systems (e.g., WISP4 [34]) set the turn-on threshold to the minimum operating voltage. Setting the turn-on threshold to the minimum voltage allows software to control when the device starts operating. The software may put the processor to sleep and periodically check the accumulated energy until the desired level is reached. With this design, the system can spend only as much time charging as necessary for a particular task. Symmetrically, the turn-off threshold may be fixed in hardware or managed by software. By default, the turn-off threshold is the minimum operating voltage of the device, but a deliberate design may turn off the device at a higher voltage. None of the above designs is unconditionally superior to all others. Threshold settings qualitatively change the turn on/turn off behavior and determine the intermittent execution intervals experienced by the software.

Systems whose load consists of multiple components with separate power rails (e.g., discrete sensor or radio ICs, multiple processors), open a design choice of *federating* [14] the energy storage into multiple isolated banks. In contrast to a shared energy buffer, a federation of per-component buffers de-couples unrelated hardware components letting each fail independently. Federated energy buffers do not necessarily charge in synchrony: one component may accumulate sufficient energy to turn on at a time that is different from and unpredictable to other components. Software on a federated platform faces the inter-component concurrency challenge described in Section 2.

3.2 Memory system and execution model

The effect of a power failure on a system and the system's resumption behavior follows from the memory system and the mechanism for preserving progress in the execution model.

Memory system. The most general model of a device's hardware includes both volatile memory (e.g., SRAM and DRAM) and non-volatile memory (e.g., Flash, FRAM). On some architectures [43] all main memory is non-volatile, leaving MCU-internal state (e.g., registers) volatile. At the extreme of the design space are architectures where all memory and internal processor state (including registers and microarchitectural structures) is non-volatile [21]. Converting volatile structures to non-volatile may eliminate some of the memory inconsistency issues described in Section 2. However, fully non-volatile architectures and main memories have two drawbacks. First, efficiency suffers, because the relatively low-latency, low-energy volatile memory accesses become relatively high-latency, high-energy non-volatile memory accesses. We measured and compared the energy cost of a volatile SRAM access to a non-volatile FRAM access on a TI MSP430FR5969 MCU and found that the FRAM access consumed 2-3x more energy on average. SRAM, with an access latency around 10ns is faster than today's FRAM, which has latency around 50-80ns [39]; however, with the often low clock frequencies of low-power MCUs (around 8MHz), SRAM and FRAM accesses take a single cycle [42]. Furthermore, a fully non-volatile architecture is at best a partial solution to the problem of preserving progress across power failures, because some state is fundamentally not "latchable" and must be re-initialized by executing code. For example, a MEMS sensor must perform an initialization routine before it can be sampled.

Looking forward, it is likely that intermittently operating device designs will include deeper, more complex memory hierarchies with a mixture of cache layers and non-volatility. We anticipate that it will be important to adapt techniques for managing non-volatility [47, 7, 29] to work in the energy, time, and memory constrained intermittent environment.

In particular, in the intermittent execution model, the recovery path is not exceptional but common and must be efficient, in contrast to traditional applications of non-volatile memory on servers or workstations.

Execution Model. The precise execution model of an intermittent device depends on how software and hardware preserve progress and program state. Most intermittent systems run "bare metal" programs, bypassing any operating system support to avoid unnecessary time or energy cost. In typical "bare-metal“ embedded systems, without system support for intermittent operation, a power failure erases volatile values and retains non-volatile ones. Checkpoint-based models [43, 33, 17, 24] preserve register, stack, and global variable values, including the program counter, and restore them after a power failure. As Section 2 discusses, checkpoints alone leave memory inconsistent, necessitating multi-versioning models [23, 9, 43] that also preserve and restore parts of non-volatile memory.

Without system support, after a power failure control flows to the program's entry point (i.e., `main()`). In checkpointing models [33, 17, 24, 43] execution resumes from a compiler-inserted or dynamically-decided checkpoint. In a task-based model [23, 9], the programmer explicitly deconstructs the program into tasks that execute atomically (and idempotently). After a power failure, execution resumes from the beginning of the most recently executed, statically-demarcated task boundary. Alternatively, some systems propose to stop the execution when power failure is deemed to be imminent and save a checkpoint then [3, 41, 4]. Without a progress latching mechanism, the application is limited to short, uninterruptible "one-shot" tasks [6].

Models with statically defined tasks require some extra programmer effort, compared to dynamic checkpoints. The advantage of a static task system is that the programmer has more control over which regions of the code are atomic and idempotent. Control over atomicity and idempotence is often important in code with application level requirements on I/O operations (e.g., a temperature and pressure sensor must be read atomically, without an intervening delay due to a power failure).

A system's state and progress preservation strategy, as well as the way the programmer expresses atomicity and idempotence constraints originate the control flow, data consistency, and environmental consistency challenges described in Section 2.

3.3 Development Environment

The effect of the power system on the behavior of software on intermittent devices complicates its development, testing, and debugging. Tools designed for continuously-powered systems do not help find bugs that manifest only under particular power failure timings or test across energy environments. Consequently, recent work proposed targeted tools for monitoring, debugging, and profiling [8], energy tracing [13], and transferring code onto intermittent devices [40, 1].

Our work on EDB [8], the Energy-interference-free Debugger, provided the first support for passively monitoring and interactively debugging intermittently-operating devices with assertions, breakpoints, and watchpoints. Debuggers available before EDB require the device to be powered continuously, making it difficult to observe, diagnose, and fix system behavior that only manifests when running on harvested energy. Working from this motivation, EDB uses a combination of hardware support and a package of co-designed software libraries to provide support for important debugging tasks during intermittent executions on energy-harvesting devices. EDB's key source of novelty is to avoid "energy-interference", which is any exchange of energy between the debugger and the target that could perturb the intermittent

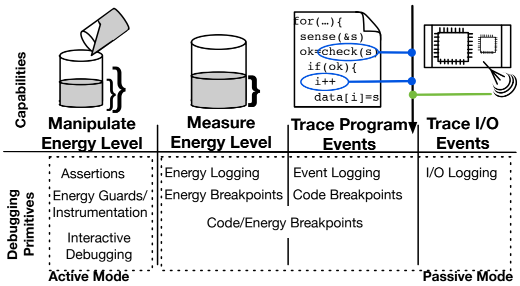


Figure 3 EDB's capabilities and features [8].

execution, changing its behavior. EDB supports passive monitoring tasks, such as tracing the device's energy level, tracing manually inserted code markers, and tracing I/O operations (such as RFID Rx/Tx). EDB also supports "active" tasks, including interactive, breakpoint debugging, high-energy-cost instrumentation, and invasive data invariant checking. The key to supporting energy-hungry "active" tasks is to *compensate* for energy consumed. Before an active task, EDB checks the device's energy level. After completing an energy-hungry active task, EDB restores the device's energy level to its level before the debugging task. With its support for passive and active debugging and tracing, EDB is the first debugger to bring necessary, basic debugging functionality to the intermittent computing domain. EDB is available for release at `http://intermittent.systems` and Figure 3 (reproduced from [8]) shows an overview of EDB's main capabilities.

Prior to EDB, Sympathy [32] addressed the challenges of debugging networks of sensor nodes, although Sympathy did not address intermittent operation. Ekho [13] addressed the lack of tools for measuring and reproducing energy conditions that vary over time. Energy availability at a given time can be represented by a current-voltage (I-V) curve. Ekho records a time-series of I-V curves in the field and replays them on-demand in the lab for reproducing issues and examining the behavior across energy environments. To simplify deployment, recent work [40, 1] developed a mechanism to reliably and efficiently transfer code (or other data) to a device using the RFID protocol, while the device is intermittently-powered.

3.4 Programming Support

Few real programming language design efforts have targeted intermittent and energy-harvesting devices. Eon [36] was the first language for an energy-harvesting system. Eon did not explicitly target intermittence, but instead tried to gracefully degrade application behavior with scarce energy. Eon gives a task a priority and tries to execute high priority tasks more often, subject to energy constraints.

Our work on Chain [9] is the first language designed explicitly to deal with intermittence, through a task-based control-flow abstraction and a channel-based abstraction for non-volatile memory that maintains consistency via static multi-versioning. The key idea in Chain is to decompose the program into a collection of tasks, which are annotated functions, and to explicitly describe the flow of execution from one task to the next. Chain guarantees that, even in the presence of power failures, tasks execute atomically. Tasks can exchange data consistently using *channels*, which are Chain's abstraction of non-volatile memory. A task may only ever read from or write to a normal channel, but not both.

The "channel access exclusion" property of Chain's channels ensure that regardless of the presence or timing of power failures, a task's inputs are always available (in its input channels)

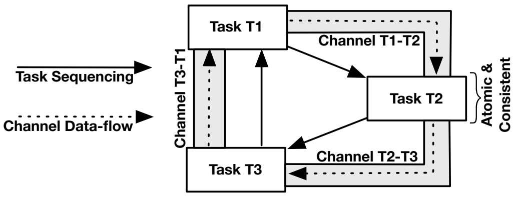


Figure 4 A schematic of a Chain program [9]. The program has three tasks that execute in sequence and pass data to one another via channels.

and its outputs always have a place (in its output channels). Statically multi-versioning data in channels allows a Chain implementation to arbitrarily re-start a task from its entry point with a consistent memory state. Idempotently re-executing a task until an execution attempt eventually completes makes the effects of a Chain task atomic, when a Chain task finally completes. Moreover, Chain eliminates the need to save and restore any volatile state because all volatile variables are required to be task-local, and initialized inside a task. Chain's unique memory abstraction, task-based control-flow, and freedom from costly checkpointing mechanisms leads to a substantial performance improvement, compared to typical volatile data checkpointing systems [33], and even non-volatile data versioning mechanisms [23]. Figure 4 (reproduced from [9]) shows a schematic view of a Chain program. A Chain reference implementation is available for researchers at `http://intermittent.systems`, including support libraries and example code to help get started building Chain applications for the WISP [34] or other intermittent devices.

The development of languages, debuggers, program analyses, and testing tools, for intermittent systems is an area of PL research open for contributions from the community. The impact of this research is widespread use of battery-free, devices across a variety of application domains.

4 Future Research Opportunities in Intermittent Computing

Intermittent computing is a promising, emerging PL research area. Next we outline our work at the intersection of energy-awareness, distributed computing, and approximation in intermittent systems.

4.1 Programming Intermittent Systems

Despite building momentum, existing approaches to programming intermittent devices have several key drawbacks: (1) increased programmer effort [23, 9] to define tasks; (2) no programmer guidance or optimization for sizing tasks [23, 9, 43]; (3) run time [43, 23] and memory [9] overheads; (4) unsound inference of application-level properties (e.g., I/O atomicity) [43]; (5) assumptions about memory volatility [43]. These limitations of prior work motivate further study. Our ongoing work aims to address the above challenges with new programming abstractions that minimize overheads, reduce programmer burden, while retaining programmer control over atomicity.

We are developing a new task-based programming model, based on Chain [9], that fundamentally departs from Chain's static multi-versioning approach. Chain creates a copy of each variable for each pair of tasks that communicate through that variable, which introduces time and space overhead as well as programmer burden. Our new efforts avoid multi-versioning using novel compiler analyses, dynamic multi-versioning, and a simple, efficient commit mechanism to keep data consistent. The key insight in our new work is that it is possible to *privatize* a copy of data to a task, allowing safe access to copies that can be stored in non-volatile memory, or in energy-efficient volatile buffers. Our initial experiments with applications from Chain [9] including compressive sensor logging and data filtering suggest that eliminating Chain's static versioning and channel management overheads yields up to 4x decrease in memory consumption and a 1.5x-7x performance improvement.

Energy-aware programming and compilation. With our new, task-based programming model efforts, we are building energy-aware compiler support [10] to help the programmer express tasks that are optimized to the underlying hardware. Assuming the common, "execute with maximum charge" hardware model [34] described in Section 3, our compiler statistically assesses whether a task's energy cost exceeds the maximum charge level of the device. Such a task would never complete and the compiler can automatically sub-divide the task, or guide the programmer in sub-dividing the task. Our work represents only a point in the energy-aware programming and compilation design space; intermittent systems warrant further exploration in this area.

Approximate execution models. Approximate execution models offer an alternative approach to handling power failure. In task-based systems (e.g., Chain[9]), after a power failure, previously executed instructions are re-executed. Re-execution burns time and energy in order to complete the task and produce a result. In an approximate execution model, accuracy can be traded off instead of spending time and energy on re-execution, by *abandoning* the interrupted task. Then, the challenge is to decompose the application into tasks and prioritize the tasks such that the completion of any subset of tasks produces a meaningful (approximate) result. For example, in an approximate motion detector, decomposed into tasks spatially, only some regions of the image would be searched under poor energy conditions. An alternative approximate execution model might reduce the cost of multi-versioning state by accepting inconsistency in some of the data values.

4.2 Distributed Intermittent Systems

Building distributed systems of intermittent devices enables new battery-less applications, e.g., sensing and actuation systems, computer vision [25], and swarms of tiny satellites [46, 2]. Realizing this vision demands that the PL community develop programming and system foundations for distributed, intermittent systems. The difficulty of specifying a correct, efficient distributed, intermittent system is compounded by the absence of development tools, specification languages, memory abstractions, and execution models. Our ongoing work focuses on intermittent distributed shared memory abstractions and simulator-based developer support.

Distributed, intermittent shared memory. We are building the first energy- and intermittence-aware, distributed shared-memory system. The key challenge, noted in Section 2, is that a pair of intermittent devices can only interact when both are active. Our intermittent distributed shared memory (iDSM) has a flat address space, with data spanned and replicated across the nodes in a system (similar to continuously-powered DSMs) [38]. Our iDSM's main contribution is an energy- and intermittence-aware memory consistency mechanism.

Maintaining iDSM consistency is difficult because both nodes involved in a request for data are rarely simultaneously powered. We address the problem by tracking request success and failure and adapting nodes' memory request behavior based on the likelihood of a request's success. If a node's request for another node's copy of a shared page frequently fails, we throttle the rate of requests between those two nodes. Instead, when either node makes a request, it prioritizes a different node with a higher historical success rate. This communication policy is energy-aware and affects memory consistency. The energy-awareness stems from the energy environment's influence over nodes' communication success rate. The policy determines memory consistency because preferentially non-communicating nodes will share updates less often, leaving data inconsistent for longer. Space- and time-dependent energy-availability requires the system to distribute data replicas to avoid "stranding" data on inaccessible nodes. iDSM research will benefit from PL contributions on new data consistency and replication policies, latency-tolerant synchronization mechanisms, and domain-specific language support for constraining how intermittent nodes interact.

Approximate, distributed, intermittent systems. Intermittent, distributed systems can leverage approximate memory consistency to improve performance and ensure progress. Assuming an iDSM with mutex locks, approximate locks with timeout-based release behavior may help prevent deadlocks when a node holding a lock fails. The cost of deadlock-freedom is the need to handle the effects of broken atomicity and potential inconsistency, which can lead to errors or a crash. Such a synchronization mechanism might integrate with type support [35] to ensure that critical program values are never corrupted, even at a cost in performance or progress.

Simulating distributed, intermittent systems. We built a flexible simulation framework for distributed, intermittent systems to help study the performance impact of energy-awareness and approximation on our iDSM without the high engineering cost of a real hardware setup. Our simulator consumes logged power traces (similar to Ekho [13]) to accurately model intermittent power cycling in a simulated collection of distributed nodes. Our inter-node communication model is flexible and currently models ambient backscatter broadcasts within a small network [22]. The simulator models the iDSM memory space and private, per-node scratchpad memory spaces, both of which are accessible through a simulator-defined interface. A simulated node queues local and iDSM operations and attempts to dequeue and execute operations on each reboot. iDSM operations traverse the network to the owner of requested data, succeeding only when the requester and data owner are powered simultaneously. Our simulator provides key insights into the communication and consistency characteristics of our iDSM.

5 Conclusion

Intermittent, energy-harvesting computing devices promise important, future applications, and a variety of future PL and computer systems research challenges. This paper provided a survey of the challenges and the design space of intermittent devices, framing a vision for future PL research into intermittent computing.

Acknowledgments. We thank the anonymous reviewers for their feedback and suggestions to improve our manuscript.

References

1 Henko Aantjes, Amjad Y. Majid, Przemysław Pawełczak, Jethro Tan, Aaron Parks, and Joshua R. Smith. Fast Downstream to Many (Computational) RFIDs. In *IEEE INFOCOM*

2017 – The 36th Annual IEEE International Conference on Computer Communications, May 2017.

2 Justin A. Atchison and Mason Peck. A millimeter-scale lorentz propelled spacecraft. In *AIAA Guidance, Navigation and Control Conference*, August 2007. doi:10.2514/6.2007-6847.

3 Domenico Balsama, Alex Weddell, Geoff Merrettt, Bashir Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded System Letters*, 7(1):15–18, March 2015. doi:10.1109/LES.2014.2371494.

4 D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980, 2016. doi:10.1109/TCAD.2016.2547919.

5 Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. BackFi: High throughput wifi backscatter. In *SIGCOMM'15*, pages 283–296, October 2015. doi:10.1145/2785956.2787490.

6 Michael Buettner, Ben Greenstein, and David Wetherall. Dewdrop: An energy-aware task scheduler for computational RFID. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2011.

7 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 *16th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, March 2015.

8 Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In *21st ACM Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 577–589, April 2016.

9 Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 514–530, October 2016.

10 Alexei Colin, Preeti Murthy, and Brandon Lucia. Cleancut: Static task boundary placement for intermittent programs. In *Workshop on Hilariously Low-Power Computing*, April 2016.

11 Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. TARDiS: A branch-and-merge approach to weak consistency. In *International Conference on Management of Data*, June 2016. doi:10.1145/2882903.2882951.

12 G. de Meulenaer, F. Gosset, F. X. Standaert, and O. Pereira. On the energy cost of communication and cryptography in wireless sensor networks. In *2008 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, pages 580–585, Oct 2008. doi:10.1109/WiMob.2008.16.

13 Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: realistic and repeatable experimentation for tiny energy-harvesting sensors. In *12th ACM Conference on Embedded Networked Sensor Systems (SenSys'14)*, pages 330–331, November 2014. doi:10.1145/2668332.2668382.

14 Josiah Hester, Lanny Sitanayah, and Jacob Sorber. Demo: A hardware platform for separating energy concerns in tiny, intermittently-powered sensors. In *13th ACM Conference on Embedded Networked Sensor Systems (SenSys'15)*, pages 447–448, November 2015. doi:10.1145/2809695.2817847.

15 Josiah Hester, Kevin Storer, Jacob Sorber, and Lanny Sitanayah. Towards a language and runtime for intermittently powered devices. In *Workshop on Hilariously Low-Power Computing*, April 2016.

16 International Telecommunication Union. Overview of the internet of things. `http://handle.itu.int/11.1002/1000/11559`, June 2012.

17 H. Jayakumar, A. Raha, and V. Raghunathan. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*, January 2014. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6733152`.

18 Mustafa Karagozler, Ivan Poupyrev, Gary Fedder, and Yuri Suzuki. Paper Generators: Harvesting energy from touching rubbing and sliding. In *ACM Symposium on User Interface Software and Technology (UIST)*, October 2013. `doi:10.1145/2501988.2502054`.

19 Yoonmyung Lee, Gyouho Kim, Suyoung Bang, Yejoong Kim, Inhee Lee, Prabal Dutta, Dennis Sylvester, and David Blaauw. A modular 1mm3 die-stacked sensing platform with optical communications and multi-modal energy harvesting. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 402–403, February 2012.

20 Ting Liu, Christopher Sadler, Pei Zhang, and Margaret Martonosi. ZebraNet. In *2nd Intl. Conference on Mobile Systems, Applications and Services (MobiSys'04)*, pages 256–269, June 2004. `doi:10.1145/990064.990095`.

21 Ting Liu, Christopher Sadler, Pei Zhang, and Margaret Martonosi. An energy-efficient nonvolatile microprocessor considering software-hardware interaction for energy harvesting applications. In *Intl. Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, April 2016. `doi:10.1109/VLSI-DAT.2016.7482577`.

22 Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua Smith. Ambient backscatter: wireless communication out of thin air. In *SIGCOMM'13*, pages 39–50, October 2013. `doi:10.1145/2534169.2486015`.

23 Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 575–585, June 2015.

24 A. Mirhoseini, E. M. Songhori, and F. Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *IEEE Pervasive Computing and Communication Conference (PerCom)*, March 2013. URL: `http://aceslab.org/sites/default/files/Idetic.pdf`.

25 Saman Naderiparizi, Zerina Kapetanovic, and Joshua R. Smith. Wispcam: An rf-powered smart camera for machine vision applications. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, ENSsys'16, pages 19–22, 2016. `doi:10.1145/2996884.2996888`.

26 Joseph Paradiso. Systems for human-powered mobile computing. In *DAC*, July 2006. `doi:10.1145/1146909.1147074`.

27 Joseph Paradiso and Mark Feldmeier. A compact, wireless, self-powered pushbutton controller. In *Proceedings of the 3rd International Conference on Ubiquitous Computing (UbiComp'01)*, pages 299–304, September 2001.

28 Gyuhae Park, Tajana Rosing, Michael Todd, Charles Farrar, and William Hodgkiss. Energy harvesting for structural health monitoring sensor networks. *ASCE Journal of Infrastructure Systems*, 14(1):64–79, March 2008. `doi:10.1061/(ASCE)1076-0342(2008)14:1(64)#sthash.ULLx9D2h.dpuf`.

29 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *ISCA*, June 2014.

30 Powercast Co. Development Kits – Wireless Power Solutions. `http://www.powercastco.com/products/development-kits/`. Visited July 30, 2014.

31 Proteus Digital Health. Proteus Discover. `http://proteus.com`, 2016.

32 Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd International*

Conference on Embedded Networked Sensor Systems, SenSys'05, pages 255–267, New York, NY, USA, 2005. ACM. `doi:10.1145/1098918.1098946`.

33 Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ASPLOS*, March 2011. URL: `https://spqr.eecs.umich.edu/papers/ransford-mementos-asplos11.pdf`.

34 Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. on Instrumentation and Measurement*, 57(11):2608–2615, November 2008.

35 Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, 2011. `doi:10.1145/1993498.1993518`.

36 Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys'07, pages 161–174, 2007. `doi:10.1145/1322263.1322279`.

37 Tolga Soyata, lucian Copeland, and Wendi Heinzelman. Rf energy harvesting for embedded systems: A survey of tradeoffs and methodology. *IEEE Circuits and Systems Magazine*, 16(1):22–57, February 2015. `doi:http://dx.doi.org/10.1109/MCAS.2015.2510198`.

38 Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM'01, pages 149–160, 2001. `doi:10.1145/383059.383071`.

39 D. Takashima, S. Shuto, I. Kunishima, H. Takenaka, Y. Oowaki, and S. Tanaka. A sub-40 ns random-access chain fram architecture with a 768 cell-plate-line drive. In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 102–103, February 1999. `doi:http://dx.doi./org/10.1109/ISSCC.1999.759147`.

40 J. Tan, P. Pawełczak, A. Parks, and J. R. Smith. Wisent: Robust downstream communication and storage for computational rfids. In *IEEE INFOCOM 2016 – 35th Annual IEEE Int'l Conf. on Computer Communications*, pages 1–9, April 2016.

41 Texas Instruments Inc. Intelligent system state restoration after power failure with compute through power loss utility. `http://www.ti.com/lit/ug/tidu885/tidu885.pdf`, April 2015.

42 TI Inc. Overview for MSP430FRxx FRAM. `http://ti.com/wolverine`, 2014. Visited July 28, 2014.

43 Joel Van Der Woude and Mathew Hicks. Intermittent computation without hardware support or programmer intervention. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 17–32, November 2016.

44 Nicolas Villar and Steve Hodges. The Peppermill: A human-powered user interface device. In *Conference on Tangible, Embedded, and Embodied Interaction (TEI)*, January 2010. `doi:10.1145/1709886.1709893`.

45 WISP. `http://wisp5.wikispaces.com/`, 2016.

46 Zac Manchester. KickSat: a tiny open-sourced spacecraft. `http://kicksat.github.io`, 2016.

47 Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, December 2013. URL: `http://www.cse.psu.edu/~juz138/files/150-zhao.pdf`.

Uncanny Valleys in Declarative Language Design

Mark S. Miller[1], Daniel von Dincklage[2], Vuk Ercegovac[3], and Brian Chin[4]

1 Google Inc., Mountain View, CA, USA
erights@google.com

2 Google Inc., Mountain View, CA, USA
danielvd@google.com

3 Google Inc., Mountain View, CA, USA
vuke@google.com

4 Google Inc., Mountain View, CA, USA
brianchin@google.com

Abstract

When people write programs in conventional programming languages, they over-specify how to solve the problem they have in mind. Over-specification prevents the language's implementation from making many optimization decisions, leaving programmers with this burden. In more declarative languages, programmers over-specify less, enabling the implementation to make more choices for them. As these decisions improve, programmers shift more attention from implementation to their real problems. This process easily overshoots. When under-specified programs almost always work well enough, programmers rarely need to think about implementation details. As their understanding of implementation choices atrophies, the controls provided so they can override these decisions become obscure.

Our declarative language project, Yedalog, is in the midst of this dilemma. The improvements in question make our users more productive, so we cannot simply retreat back towards over-specification. To proceed forward instead, we must meet some of the expectations we prematurely provoked, and our implementation's behavior must help users learn expectations more aligned with our intended semantics.

These are general issues. Discussing their concrete manifestation in Yedalog should help other declarative systems that come to face these issues.

1998 ACM Subject Classification D.3.2 Constraint and Logic Languages

Keywords and phrases Declarative logic programming language

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.9

1 Background

Kowalski famously observed [4] that "Algorithm = Logic + Control". Declarative languages enable users to ask logical questions. The "logic" of a declarative program is a description of what a correct answer looks like. The "control" explains how to compute answers that satisfy that description. Often these two components are not separate parts of a declarative program but distinct ways of reading a program. For example, the *declarative reading* of a Haskell program considers Haskell functions to be the mathematical functions they seem. The *operational reading* sees these functions as code explaining how to compute results from input arguments. In a declarative language, the computed results must be within the declarative reading's description.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 9; pp. 9:1–9:12
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Listing 1 A trivial Horn-clause rule.

```
aa(X,Z) :- bb(X,Y), cc(Y,Z).
```

In declarative language design, there is an inescapable tradeoff between expressiveness and automation. *General purpose* declarative languages such as Haskell and Prolog are very expressive but with limited automation – incrementally more complex questions can be asked for incrementally more effort; but their users are responsible for controlling the direction of execution. *Special purpose* languages like Datalog and SQL are highly automated but with limited expressiveness – many questions cannot be asked, but of those that can, users can leave operational concerns to the implementation.[1]

As general purpose languages improve their automation, and as special purpose languages improve their expressiveness, the gap between them will narrow but not close. These improvements create dilemmas. For which questions should users let the language figure out how to compute answers? When should users still make operational choices, and how should they express them? User uncertainty about their remaining operational responsibility is the uncanny valley of concern to this paper.

Yedalog [2] is a general-purpose Datalog-like language for scalable exploratory data analysis – for asking questions of large semi-structured data sets. Yedalog has been used in production for several years by several teams. Nevertheless, the Yedalog implementation makes many control decisions that general purpose languages normally leave to their users. To reduce user uncertainty about their remaining operational responsibility, we designed and documented an informal operational model. But people learn by experience more than explanation. From their experience using Yedalog – seeing which of their programs work or do not – our users learned a different model. The model they learned assumes some forms of automation beyond any we had planned to support.

As we change Yedalog to meet these unanticipated expectations, we must better anticipate what models users will learn from Yedalog's new behavior. These models must provide users with better clarity about how to use Yedalog well. New user expectations will in turn affect what further changes we make to Yedalog. This feedback loop shapes our trajectory through the design space. How should we steer it and where will it lead?

1.1 Logic Programming

To locate Yedalog in the declarative language design space, we rapidly zoom in from declarative languages to logic programming languages, to Horn-clause logic programming languages, to Datalog-like languages, and finally to Yedalog.

Among declarative languages, Kowalski's distinction is especially crisp for logic programming languages. Logic programs consist of *facts*, *rules*, and *queries*. Queries produce *answers*. In the operational reading, facts are data, rules are procedures, queries are calls binding input parameters, producing answers binding output parameters. In the declarative reading,

[1] "Declarative" is sometimes used to mean what this paper calls "highly automated" – the ability of users to avoid operational concerns. In this paper, "declarative" means only the ability of users to be confident that what is computed corresponds to what they have logically described. In our terms, Haskell and Prolog are declarative. Datalog and SQL are both declarative and highly automated. All these languages, as well as Yedalog, include unsound escape hatches that compromise this confidence. But so long as these escape hatches are explicit and visibly absent from most programs, we still consider these to be declarative languages.

Listing 2 Length of shortest path.

```
path(X,Z) min= edge(X,Z).
path(X,Z) min= path(X,Y) + edge(Y,Z).
```

facts are propositions assumed to be true, rules express how some propositions imply other propositions, queries are parameterized hypotheses, and answers are their parameterizations, to be proved from those facts and rules. Execution is the search for such proofs.

How search proceeds through this search space matters. The Horn-clause logic programming languages – Prolog, Concurrent Prolog [8], Datalog – have essentially the same abstract syntax (e.g. Listing 1) with the same logical meaning, but make wildly different operational choices. To the programmer trying to get practical work done, these languages feel vastly different from each other.

Programming in Prolog resembles conventional call-return programming augmented with backtracking search, where the programmer must write the program according to the precise order execution should proceed. *When `aa` is called, call `bb` and then `cc`.* Programming in Concurrent Prolog resembles actors exchanging asynchronous messages. *When `aa`'s inputs are ready, run `bb` and `cc` in parallel, communicating on their shared variable `Y`.*

Programming in Datalog resembles database query languages like SQL, where facts are data-tables, rules create views, and queries are queries. *Join `bb` and `cc` to produce `aa`.* Datalog implementations have all the freedom of query planning that databases enjoy:

- `bb` might run first, generating `Y` values for `cc` to test
- `cc` might run first, generating `Y` values for `bb` to test
- `bb` and `cc` might both generate answers intersected on `Y`

This paper freely mixes logic and database terminology. A relational table is also a disjunction of facts. `bb(X,Y),cc(Y,Z)` is a conjunction and also a join. The multiple rules of `path` in Listing 2 are a disjunction and also a relational union. Yedalog, like many Datalog-like languages, supports aggregation. `path` expresses shortest path as a `min` aggregation over the disjunction of all path lengths. When lengths are known to be non-negative, some systems will implement it using Dijkstra's algorithm [3, 7].

1.2 Yedalog's Goals

Yedalog's focus is scalable data exploration. As our users spend less attention on how things execute, they spend more attention on asking questions and interpreting answers. We thus aim for the following goals[2]. These goals conflict, requiring us to make tradeoffs and compromises based on our sense of the costs and benefits.

Query planning freedom. Programs should not accidentally over-specify the implementation. The programmers' natural way of asking questions, crafted without attention to operational details, should leave the Yedalog implementation with enough freedom to make good choices.

Good optimization decisions. As Yedalog makes better implementation choices, users do not need to.

[2] In addition, to be more quickly understood by our audience, Yedalog has a more C-like surface syntax than we show in this paper.

Listing 3 Inferring orders and modes.

```
# front has modes (in,in) and (out,in)
# reverse has modes (in,out) and (out,in)

back(E,L) :- front(E,R), reverse(R,L).
```

Query planning compatibility. In order to preserve query planning freedom, the program's observable behavior should be *compatible enough* under all decisions the implementation is allowed to make.

Usable operational controls. Automatic planning will sometimes be inadequate. We must provide users the tools they need to cope, such as operational knobs for overriding default decisions.

Section 2 explains why "Query planning freedom" needs a different understanding of "Query planning compatibility" than we expected, and how to provide it. Section 3 shows how to handle errors without violating query planning compatibility. Section 4 discusses uncanny valleys in design spaces. Section 5 concludes.

2 Inferring execution orders

In pure Datalog, predicates represent concrete data or computed views of data. These data-oriented predicates can be materialized as finite relational tables. By contrast, most computational predicates express an infinite relation among their parameters. Integer addition embodies an infinite set of triples. Factorial embodies an infinite set of pairs. To better support general-purpose use, Yedalog programs can freely mix data-oriented and computational predicates.

Each Yedalog predicate has a set of *modes*. Each mode says, for each parameter, whether the parameter is *input* or *output*. Finite data predicates support an all-out mode, where all parameters are output parameters. Infinite predicates can only support modes containing at least one input parameter. For an input parameter, the caller must provide concrete data.

An output parameter is strictly more general: It can output data for the caller to use, or use data provided by the caller as input, by comparison or indexing. For example, the `edge` predicate of Listing 2 would normally be an all-out finite table indexed (at least) on its first column. Without input `edge` will enumerate all edges. With a first node as input, `edge` will lookup and efficiently enumerate only those edges emerging from that node.

Yedalog combines mode inference with mode-based reordering. The `back` predicate of Listing 3 says that `E` is at the back of `L` if it is at the front of `L` reversed. The `front` and `reverse` predicates represent infinite relations since there are infinitely many possible lists. They support the modes stated in Listing 3. Since `front` demands a binding for `R` and `reverse` can provide one, the only feasible orders run `reverse`'s `(out,in)` mode first, generating `R` bindings for either mode of `front` to use. From the feasible orders of `back`'s bodies, we can infer the possible modes of `back`: `(in,in)` and `(out,in)`.

We avoid explosive search by inferring only the minimal set of most general modes. The modes supported by `reverse` are already its minimal set, since neither `(in,out)` nor `(out,in)` is more general than the other. For `back`, we infer only `(out,in)` since it is strictly more general than `(in,in)`.

We provide our users knobs to make operational decisions. Some are subtle: Our `+` and `-` operators are irreversible, leading users naturally to force the right choice between

Listing 4 Factorials: The bad, bad, worse, and ugly.

```
factA(0) = 1.
factA(M+1) = (M+1) * factA(M).

factB(0) = 1.
factB(N) = N * factB(N-1).

factC(0) = 1.
factC(N) = N * factC(N-1) :- N >= 1.

factD(0) = 1.
factD(N) = (N >= 1 && N * factD(N-1)).
```

top-down and bottom-up when it makes a difference, as in the following example. Others are explicit: Conjunction order is unspecified by default, but we provide an `&&` operator to force left-to-right order. When should it be used? Forcing order destroys choices an implementation could have used well. Not forcing can occasionally leave programs incorrect. How do users decide?

Listing 4 shows four versions of recursive factorial that are all declaratively correct. `factA` executes bottom-up, starting at 0, and reliably fails to terminate as it enumerates larger numbers[3]. `factB` executes top-down, starting with the requested argument, and reliably fails to terminate as it enumerates ever smaller negative numbers.

By “bottom-up” we mean computing forward from known facts, like the factorial of zero, to implied facts like the factorial of one, until reaching the query. By “top-down” we mean computing backward from the initial query like the factorial of 7, to subgoals like the factorial of 6, until reaching known facts like the factorial of zero[4].

As an informal experiment, we asked our users to write the standard introductory recursive factorial function. 80% submitted variations of `factC`. On the current Yedalog implementation, `factC` always happens to execute correctly and pass any possible tests. However, Yedalog is free to make either the recursive call first or the `(N >= 1)` test first. Had `factC` recurred first, it would not have terminated. Instead, it would speculatively enumerate ever smaller negative numbers before the `(N >= 1)` test that would disqualify these speculations. By contrast, `factD` is the correct Yedalog program no one wrote, which uses `&&` to avoid this hazard. Although we have documented these issues well, *none* of our survey responses even mentioned this ordering issue as a possible concern.

We documented an operational model in which `factC` might not terminate. Our users understood a model in which `factC` always works, which is a more accurate model of what our implementation does. Which is more right? As we change Yedalog, which of these models do we start with? These questions led us to better understand query planning freedom and query planning compatibility.

[3] Although pure Datalog programs always terminate, Datalog programs with arithmetic may not.

[4] Yodalog actually implements `factB` by magic sets [1]. Magic sets is often described as bottom-up because it reuses the machinery of bottom-up execution. But since magic sets mostly work backward from queries to facts, in this paper we do not distinguish between top-down and magic sets, using “top-down” for both.

2.1 How much freedom to plan badly?

No matter what operational model language designers document, *the operational model implementors implement must support those user expectations that implementors dare not break.* At the same time, to enable implementors the freedom to choose among more good plans, the operational model must also allow them to choose among more bad plans. Query planning compatibility helps us navigate the conflict.

Under all allowed implementation choices, `factA` and `factB` never work and `factD` always works, which upholds query planning compatibility. But saying that `factC` may or may not terminate denies reality. The fact that many patterns like `factC` always execute well today means that we dare not break them. Put this way, query planning compatibility is not so much a goal to achieve as a way to understand what query planning freedom is already lost.

Of course, we are not concerned about breaking `factC` itself. No one wrote this particular program until we asked. Our users wrote this specific program because of expectations they learned from some larger category of programs. They form these categories by generalizing over many concrete experiences. How they generalize depends on what they find intuitive. From an HCI (human computer interaction) perspective none of this is surprising; but programming languages raise the stakes. Expectations people learn from interactive use they adjust and relearn under continued use. By contrast, widespread programmer expectations get baked into large numbers of programs.

Of two observably different outcomes X and Y, we say Y is *compatible enough* with X when the expectations users learn from X do not deter implementors from causing Y. “Compatible enough” is thus always a judgement call, weighing the costs of breaking X expectations *vs.* the benefits of Y. This applies to performance as well as correctness. As we change Yedalog’s implementation to make better choices in general, we might make some previously-efficient programs somewhat slower, but we dare not impose prohibitive costs on patterns already in widespread use.

“Compatible enough” is directional: No non-malicious user minds if a previously non-terminating program starts to work or a previously expensive program become cheaper. Due to the same directionality, implementors should be aware that each improvement is also a potential commitment, cutting off their freedom to make other choices. At every stage, we should rationalize our commitments back into our language design, in order to shape what freedom usefully remains [9].

The next section explain such an improvement and evaluates it by these criteria.

2.2 Unrolling multi-recursion

Since `factC` already works in the implementation, how can we change our model so that `factC` must work in all implementations? For `factC` itself we can do so trivially. Yedalog’s stratification analysis already distinguishes potentially recursive calls from statically non-recursive calls. If we require recursive calls to be scheduled as late as feasible – which a good planner would do anyway – then `factC` becomes correct.

Any intuitive category that includes `factC` also includes `fibA` from Listing 5. However, `fibA` is multi-recursive. It makes more than one potentially recursive call. They cannot both go last. As far as the implementation knows, either conjunct, if consistently run first, might never terminate even if the other conjunct would have caused an early failure.

Computation within each conjunct of a conjunction is *speculative* – only relevant when none of the other conjuncts fails. Speculative execution in hardware works because speculation checks cannot be indefinitely postponed. We could get a similar effect by specifying

Listing 5 Unrolling multi-recursion.

```
fibA(0) = 0.
fibA(1) = 1.
fibA(N) = fibA(N-1) + fibA(N-2) :- N >= 2.

fibB(0) = 0.
fibB(1) = 1.
fibB(N) = (N >= 2 && X == fibC(N-1) && Y == fibC(N-2) && X+Y).

fibC(0) = 0.
fibC(1) = 1.
fibC(N) = (N >= 2 && Y == fibB(N-2) && X == fibB(N-1) && X+Y).
```

fairness among conjuncts, so speculation checks cannot starve. We have found a cheap approximation of fairness: Unroll a multi-recursive predicate like `fibA` into mutually multi-recursive predicates like `fibB` and `fibC`, where we rotate among the possible orders of which recursive call comes first.

The unrolled implementation does have a real performance cost: `fibA` has one memo table, reducing the naively exponential costs to linear. The unrolled form has two memo tables, doubling the number of misses we pay for. Only multi-recursive predicates pay this cost, which is linear only in the width of the multi-recursion. Wide multi-recursion is rare, so these costs are minor.

This unrolling will not cause previously-working programs to become non-terminating. It will cause some previously non-terminating programs to become correct, which sounds good. However, such “improvements” can do more harm than good. If an intuitive general category of code reliably does not terminate today, like the categories containing `factA` or `factB`, then we would muddy the waters with an “improvement” that allows some programs in such a category to work under some implementations, unless it requires all programs in that category to work on all implementations. As far as we can tell, this unrolling technique does not muddy the waters. Every general category that previously had reliably failed will continue to reliably fail.

This unrolling technique implements only a static approximation of fairness. This raises some interesting issues.

- How do we specify the approximation of fairness that this unrolling implements? We do not want to specify the unrolling technique itself because we want to preserve the freedom to achieve the same benefit by other means.
- For what programs is this approximation inadequate? We expect the accidental occurrence of such programs to be exceedingly rare, which would make these occurrences that much more uncanny when they do occur. No matter what we specify, we should expect users to learn expectations that only true fairness could implement.
- Can we close this remaining gap – implement true fairness for those rare cases – without significant cost to other programs?

Despite these open issues, for our purposes this unrolling technique is good enough. Other projects with different tradeoffs may judge these same issues differently.

Listing 6 Making failure noisy.

```
fibE(N) = fibA(N);
fibE(N) = raise(‘Must not be negative: $N‘) :- N < 0.

qq(M,N) = fibA(M) + fibE(N);
```

3 Errors as noisy failures

In real programs, deployed in production and interacting with a wide variety of systems, many things can go wrong. Say a filename is misspelled. The parts of the program that would process the contents of the file are, declaratively, queries about the contents of a file with that name. Since there is no file with that name, these queries have no answers, i.e, they fail. Failure is normally silent, but a surprising failure that violates programmer expectations needs to alert the programmer, so that the likely problem can be fixed.

The `fibA` predicate of Listing 5 fails silently on negative input. The `fibE` predicate in Listing 6 acts just like `fibA` except that, on negative input, its `raise` expression fails and reports an error. To account for this, we extend our operational model to say that a query has some number of answers and reports some number of errors. A query that has no answers, fails. A query that reports no errors is silent. On negative input `fibE` produces a noisy failure. Errors have no declarative significance, so `fibA` and `fibE` have the same logical meaning. But `fibE` also produces diagnostic information. To route this diagnostic information appropriately, we must determine how errors propagate through Yedalog's constructs. Our error design has the following goals:

Suppress error storms. In a sharded computation, such as a large map-reduce job spread out over many machines, one underlying problem might trigger a massive number of errors, although most contain no new information.

Preserve at least one diagnostic. To suppress error storms, we discard tremendous numbers of errors. But we must not discard all of them. Few things are more frustrating than a program that silently behaves badly.

Do not make non-erroneous execution significantly slower. Errors are for exceptional cases we hope happen rarely. If support for occasional errors slows down the common case, we have made a bad tradeoff.

Do not make erroneous execution explosively slower. Although we allow error handling to be expensive, this is not a blank check.

Our error design should also respect the following general efficiency goals:

Stop conjunctions early on failure. The body of `qq` in Listing 6 calls both `fibA` and `fibE` in a conjunction. Whichever goal runs first might fail, rendering the other irrelevant. Efficient execution should be able to skip such irrelevant code.

Allow disjunctions to stop early on saturation. A disjunction *saturates* when no further disjuncts could change the outcome. The efficiency of Dijkstra's algorithm requires `path` to stop examining alternative paths that cannot further reduce the minimum. It is not realistic to require such optimizations in general, but we must allow them.

To suppress error storms while preserving at least one diagnostic, we allow errors to be consolidated. When a query reports at least one error, all errors but one may be discarded, preserving the property that it reports at least one error. Different plans may result in different errors being discarded. Does this violate query planning compatibility?

Listing 7 Holding speculative errors.

```
   (try {
       dd(X)
    } catch (E) {
       (ee(X),ff(X),gg(X)) && raise(E)
    }) && (ee(X),ff(X),gg(X))
```

Listing 8 Folding the holding of speculative errors.

```
   (try {
       dd(X)
    } catch (E) {
       eTail(X) && raise(E)
    }) && eTail(X)

   eTail(X) = (try {
       ee(X)
    } catch (E) {
       fTail(X) && raise(E)
    }) && fTail(X)).
etc...
```

By convention, Yedalog errors contain only diagnostic information meant for human interpretation. Users may learn to expect specific errors, but usually they fix the indicated problem rather than write programs that depend on errors to occur. We have not encountered Yedalog programs that depend on the content of the errors that were reported. We thus consider reporting at least one error under one plan to be compatible enough with reporting at least one error, any error, under another plan.

3.1 Errors in conjuncts

The goal "Stop conjunctions early on failure", taken literally, conflicts with query planning compatibility. To resolve the conflict, we must split silent failures from noisy failures. Say Yedalog's static analysis conservatively assumes that either `fibA` or `fibE` may fail and that either may report an error. Yedalog certainly has the freedom to run this conjunction in either order. If `fibA` runs first and fails, stopping the conjunction early, the error that `fibE` might have reported is not noticed. The conjunction as a whole would produce a silent failure. On the other hand, if `fibE` runs first and produces a noisy failure, stopping the conjunction early, then the silent failure `fibA` might have produced would not be noticed. If `fibE`'s error propagates anyway, then the conjunction as a whole produces a noisy failure.

This violates query planning compatibility. For a conjunction, the difference between silent and noisy failure is too surprising. This violation is not just a problem in theory. We became aware of the issue when correct programs started reporting errors that "could not happen", confusing everyone. This reinforces our sense that conjuncts are seen as speculative, and conjunctive failure as a failed speculation. *What happens in a failed speculation stays in a failed speculation.*

The efficiency motivation for "Stop conjunctions early on failure" applies only to silent failure. Since our efficiency goals require silent failures to stop early, query planning compatibility demands that noisy failures cannot. After `fibE` reports an error, Yedalog must treat this error as speculative, with `fibA` as the speculation check. We must execute `fibA` just enough to determine if it would have any answers, if it had run first. We do not care what the answers are or if it would have produced any more answers. If `fibA` produces any answers, then the conjunction as a whole can fail reporting `fibE`'s errors. If `fibA` fails silently, the conjunction must as well. This "unnecessary" execution of `fibA` may be expensive, but not explosively so. It only happens when the implementation was allowed to run `fibA` first and pay those costs.

What about conjunctions with data dependencies, such as `dd(X),ee(X),ff(X),gg(X)`, where the named predicates are out-moded? Each may be used to generate `X` values or to test them. Under normal conditions, whichever executes first would generate and the rest would test. But any may also report errors.

To ensure that speculative errors only propagate once the speculation commits, the compiler could generate code approximately like that in Listing 7, where each of the remaining three-way conjunctions must be similarly expanded. To avoid an exponential expansion, we first fold each remaining conjunction into a separate predicate as shown in Listing 8. This has no explosive costs. But it is too expensive for the completely non-erroneous case – the outermost `&&` chain. Instead, we will leave this one chain fully unfolded.

3.2 Errors in disjuncts

A disjunction saturates when no further disjuncts would change the outcome. Any disjunction under a negation immediately saturates on the first answer. This answer establishes that the disjunction succeeds, allowing the negation to immediately fail, not caring what the answer is or if there are any more. We can realize some of these optimization opportunities more easily than others, so we allow disjunctions to stop early on saturation without requiring them to do so. We wish to preserve the query planning freedom to realize more of these opportunities over time.

Allowing disjuncts to stop early on saturation, by duality, should have the same conflict between efficiency, query planning compatibility, and preserving diagnostics. The dual solution would be to hold the contributions from a noisy disjunct – both its answers and at least one error – to see if the remaining disjunction would saturate silently. If it does, the noisy disjunct could have been skipped under other possible plans.

Returning to the shortest path example of Listing 2, say that the `edge` predicate, when asked for the length of a certain edge, answers and reports an error. If this edge lies on the shortest path, and if no path without this edge is tied for shortest, then the search could not have saturated without asking about this edge. Otherwise, depending on the algorithm used and the non-deterministic order in which edges were examined, a possible plan might not ask about this edge, not notice the error it would report, saturate, and silently answer with the minimal path. For the same graph and the same program, another possible plan would ask about this edge, notice the error, take its length into account, and proceed until saturating to the same answer. If it propagates this error, then a program that was silently succeeding might start reporting errors even when run on the same data.

Were we to apply the same standard of query planning compatibility that we applied to conjunctions above, and to apply the dual solution, we would postpone consideration of this edge until everything else settles down, giving us a candidate non-erroneous shortest path length. We would then contribute back in the postponed edge length and wait for the algorithm to settle again. If it settles on a shorter length, then we propagate both this error and the shorter path length. Otherwise we would silently report the unchanged candidate path length.

Should we bother with this extra bookkeeping, to avoid this observable difference of outcomes? The duality hides an important psychological difference: Disjuncts are not speculative. The success of one disjunct does not trigger an expectation that the other disjuncts "could not happen" but merely that they "might not happen". We have not found errors from unnecessary disjuncts to cause confusion in practice. Thus, we hold disjuncts to a lower standard of compatibility than we require of conjuncts.

4 Discussion

The original uncanny valley [5], in the context of robotics and computer graphics, predicted how a pattern of confused expectations leads to a feeling of creepiness. Their valley is a transient dip in affinity along a trajectory of progress towards lifelike human portrayals. Before the valley lies the pleasantness of a cute toy. After the valley are portrayals so lifelike they continue to amaze. To progress to that achievement, one must journey through the valley, where portrayals are good enough to provoke perceptual expectations that they then disappoint. We use this as a loose metaphor; our concern is not creepiness.

In declarative language design, we start at two pleasantly stable points in the design space. The first is occupied by expressive general purpose languages, in which users can ask any question but have full responsibility for figuring out how to compute an answer. These users discharge their responsibility without confusion by programming in terms of clear operational models. These programs over-specify, foreclosing on many optimization opportunities, wasting both human attention and computational resources. The second stable point is occupied by highly automated special purpose languages whose users do not need any operational model, leaving implementations free to use a wide range of fancy optimizations that need not be explained, in order to answer a limited range of questions.

From these two stable points, we see in the distance the promise of a third: A general purpose language in which users can ask many questions without operational concern, understand when they do need to make operational decisions, and understand how to express them. To find this third point we entered the valley, where operational controls are needed so rarely that they are expected even less. Despite this mismatch, our users are already much more productive, so we proceed.

Other languages are on similar journeys. Dyna [3] in particular entered this valley ahead of us and helped us find our footing. Software engineering has many uncanny valleys. A vivid example outside of language design is refactoring IDEs.

Refactoring IDEs were first invented and used for Smalltalk, a dynamically typed language. Without static types, automated refactorings have many false hits, so refactoring interactions always involve the programmer reviewing each decision. Programmers learn by doing. From the experience using these tools, programmers rapidly learn that they need to carefully decide whether to approve or reject each individual change.

Refactoring IDEs for Java use its static types to make many decisions reliably. For example, when changing the order of a function's parameters, the IDE can correctly identify exactly the call sites of this function, with no false hits and (in the absence of reflection) no false misses. Nevertheless, when it reorders argument expressions at these call sites it still might break the program – these argument expressions might now perform their side effects in the wrong order. However, this happens so rarely that most programmers never experience it. Programmers learn by doing. From these experiences, programmers learn to assume these refactorings are correct and not to bother reviewing each individual call site [6].

Should we make these refactorings less reliable, so programmers stop learning that they are more reliable than they are? Hardly. Rather, this example illustrates that we would have knowingly proceeded into this valley anyway because the benefits are worth it, and that retreat is not an attractive option. The only way out is through.

5 Conclusions

General purpose declarative languages, at first, leave many operational decisions to their programmers; but may absorb more operational responsibility over time. Declarative languages that absorb all this responsibility start special purpose; but may become more general over time. These paths lead to a dilemma, where these systems have gotten good enough that users perceive them, and use them, as more than they are. Expectations outrun reality. This problem is also an opportunity, to use the feedback between implementation behavior and user expectations to help shape both to be more aligned and, together, more effective.

Acknowledgements. For feedback and suggestions, we thank Yedalog's users, Peter Ludemann, Terry Stanley, Kevin Reid, and anonymous SNAPL referees.

References

1 Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15. ACM, 1985.

2 Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S. Miller, Franz Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring knowledge at scale. In *Proc. of the 1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *LIPIcs – Leibniz International Proceedings in Informatics*, pages 63–78. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015. `doi:10.4230/LIPIcs.SNAPL.2015.63`.

3 Jason Eisner and Nathaniel W. Filardo. Dyna: Extending Datalog for Modern AI (full version), 2011. URL: `https://www.cs.jhu.edu/~jason/papers/eisner+filardo.datalog11-long.pdf`.

4 Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.

5 Masahiro Mori, Karl F. MacDorman, and Norri Kageki. The uncanny valley [from the field]. *IEEE Robotics & Automation Magazine*, 19(2):98–100, 2012 original 1970 Energy.

6 Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program metamorphosis. In *European Conference on Object-Oriented Programming*, pages 394–418. Springer, 2009.

7 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.

8 Ehud Y. Shapiro. *Concurrent Prolog: Collected Papers*. MIT press, 1987.

9 Allen Wirfs-Brock. Programming language standardization: Patterns for participation. In *5th Asian Conference on Pattern Languages of Programs*.

Programming Language Tools and Techniques for 3D Printing

Chandrakana Nandi[1], Anat Caspi[2], Dan Grossman[3], and Zachary Tatlock[4]

1 **Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA**
`cnandi@cs.washington.edu`

2 **Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA**
`caspian@cs.washington.edu`

3 **Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA**
`djg@cs.washington.edu`

4 **Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA**
`ztatlock@cs.washington.edu`

Abstract

We propose a research agenda to investigate programming language techniques for improving affordable, end-user desktop manufacturing processes such as 3D printing. Our goal is to adapt programming languages tools and extend the decades of research in industrial, high-end CAD/CAM in order to help make affordable desktop manufacturing processes more accurate, fast, reliable, and accessible to end-users. We focus on three major areas where 3D printing can benefit from programming language tools: design synthesis, optimizing compilation, and runtime monitoring. We present preliminary results on synthesizing editable CAD models from difficult-to-edit surface meshes, discuss potential new compilation strategies, and propose runtime monitoring techniques. We conclude by discussing additional near-future directions we intend to pursue.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.3.4 Processors, I.2.2 Automatic Synthesis

Keywords and phrases 3D printing, rapid prototyping, desktop manufacturing, compilers, verification, synthesis

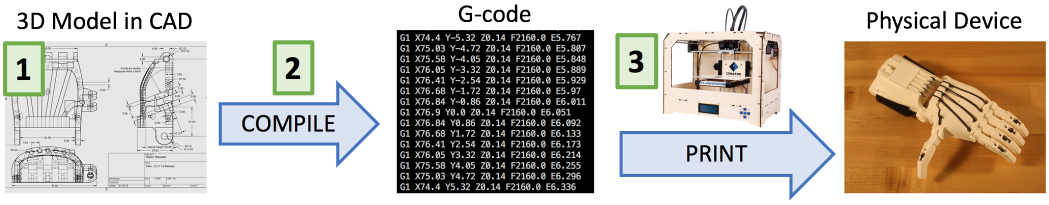
Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.10

1 Introduction

Affordable desktop-class 3D printers, laser cutters, and Computer Numerical Control (CNC) mills will soon be available to millions of people [7]. The potential social benefits of broad, end-user access to these technologies have been much hyped, but the current reality is that desktop-class hardware and tools are often significantly less accurate, fast, and reliable than their industrial counterparts. In industry, these processes take place on expensive, high-end machines managed by trained experts. While desktop-class hardware will continue to improve, we believe that without key software improvements, democratized manufacturing practice by end-users on affordable hardware is bound to fall short of its ambitious promise.

Many programming language techniques can be adapted to address analogous problems in desktop manufacturing – developing Computer Aided Design (CAD) models and editing

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 10; pp. 10:1–10:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



```
G1 X74.4 Y-5.32 Z0.14 F2160.0 E5.767
G1 X75.03 Y-4.72 Z0.14 F2160.0 E5.807
G1 X75.58 Y-4.05 Z0.14 F2160.0 E5.848
G1 X76.05 Y-3.32 Z0.14 F2160.0 E5.889
G1 X76.41 Y-2.54 Z0.14 F2160.0 E5.929
G1 X76.68 Y-1.72 Z0.14 F2160.0 E5.97
G1 X76.84 Y-0.86 Z0.14 F2160.0 E6.011
G1 X76.9 Y0.0 Z0.14 F2160.0 E6.051
G1 X76.84 Y0.86 Z0.14 F2160.0 E6.092
G1 X76.68 Y1.72 Z0.14 F2160.0 E6.133
G1 X76.41 Y2.54 Z0.14 F2160.0 E6.173
G1 X76.05 Y3.32 Z0.14 F2160.0 E6.214
G1 X75.58 Y4.05 Z0.14 F2160.0 E6.255
G1 X75.03 Y4.72 Z0.14 F2160.0 E6.296
G1 X74.4 Y5.32 Z0.14 F2160.0 E6.336
```

Figure 1 *The 3D Printing Development Cycle.* To 3D print a device: (1) An engineer first designs a 3D model using standard CAD tools (e.g., SolidWorks [28]). (2) This model is compiled into a sequence of low-level *G-code* commands that corresponds to basic actions the printer can take (move the print head, start/stop extrusion, lower the build plate, etc.). (3) The printer directly executes the G-code, producing a physical object.

existing objects are analogous to synthesis; generating accurate, efficient tool paths (paths the print head of a 3D printer follows) from a CAD model is analogous to optimizing compilation; automatically tracking operations to ensure safety and halt before bogus operations is analogous to runtime monitoring. Much as RAID software [17] enabled cheap, unreliable storage hardware to compete with expensive, reliable alternatives, we believe that programming language tools can significantly improve the state of the art in desktop manufacturing.[1] In the remainder of this paper we focus on one type of desktop manufacturing, 3D printing, but we believe the proposed research can be generalized to other processes like laser cutting and milling.

We propose a three-part research agenda to begin tackling these challenges for affordable, end-user desktop manufacturing:

- In Section 3, we discuss program synthesis techniques to generate easy-to-edit CAD models from difficult-to-edit surface meshes, and eventually to enable optimization and refactoring of complex CAD models.
- In Section 4, we discuss compiler techniques to improve printer performance via parallelization, and eventually to automatically account for observed errors in output prints.
- In Section 5, we discuss runtime monitoring for 3D printers to aid debugging, automatically detect printing errors, and eventually fix errors on the fly.

2 Background on 3D printing

3D printers come in a wide variety of designs, from lithography-based resin printers to inkjet-based powder printers. The challenges and techniques described in this paper apply to many of these designs, but to make the discussion more concrete, we focus on "cartesian fused filament fabrication" (FFF) printers (Figure 2(a)), the most common and affordable type of printer. Figure 1 depicts the typical workflow for using such a device:

1. **Design.** Users first design their model using CAD tools. There is a diverse array of available CAD tools including freely available options such as OpenScad [16] or SketchUp [26] and proprietary packages like Rhinoceros [21] and SolidWorks [28] which can cost thousands of dollars. In this paper, we focus on OpenSCAD since it conveniently represents CAD models as programs and is widely used on design sharing websites such as Thingiverse [32]. Figure 3 shows two CAD programs in OpenSCAD. OpenSCAD provides

[1] Some gap between industrial and desktop manufacturing will always remain. Two ton CNC mills are inherently more rigid and stable than ten kilogram mini-mills after all.

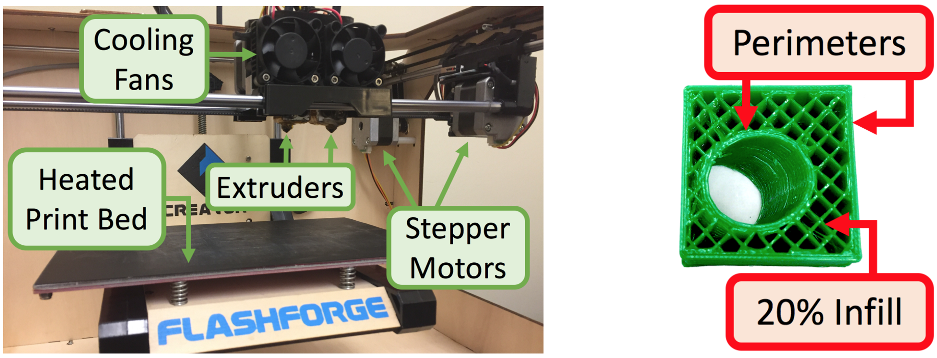


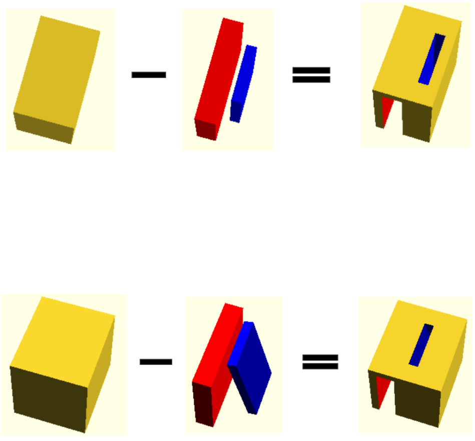
Figure 2 (a) Major printer components. (b) Perimeter and infill cross-section.

various primitive 3D structures (e.g., `cube`), transformations (e.g., `translate`) and combinators (e.g., `difference`), that can be used together to create complex 3D models. While OpenSCAD is programmatic, many CAD tools, such as Rhino [21], are GUI based. Irrespective of the interface, the models designed using these tools are declarative in nature, i.e., they only describe the 3D structure and parameters of a model, not how to manufacture it.

2. **Compilation.** Compilation happens in two phases: (A) A CAD model is translated to an intermediate representation, typically in STereoLithography (STL) format [8]. This represents the surface mesh in the form of polygons in a 3D coordinate system. (B) The STL is “sliced” to obtain *G-code*. The G-code is a sequence of imperative commands that control extrusion, movement and temperature. The slicer determines the *tool path* which refers to the path the print head should follow while printing. A typical slicing strategy is discussed in Section 2.1 in more detail.
3. **Print.** The printer runs firmware that interprets the G-code and sends low-level hardware control signals to motors, heating elements, and cooling fans. An extruder melts print material and pushes it through a nozzle to build up the part layer-by-layer starting with the first layer directly on the build plate. There are many physical phenomena involved in this step that affect the print quality – the inertia on the print head, thermal expansion of the print material, the temperature and humidity of the environment, etc.
4. **Iterate.** Finally, there is an implicit fourth step which is to repeat the above steps until the 3D object comes out as expected.

2.1 Baseline Slicing

At a high level, common slicers [25, 23, 3, 27] take a 3D surface geometry in STL and divide it into a sequence of 2D slices parallel to the xy-plane at regular intervals of height h (typically $h \approx 0.1$mm). Thus the i^{th} slice represents the perimeters of the object to be printed at height $i \times h$. To generate G-code, the slicer computes tool paths to trace the perimeters at each height and fill the space between perimeters with a regular pattern at some user-specified density (see Figure 2(b). Within these layers, the slicer inserts G-codes to start and stop extrusion during movements along perimeters and over fill areas. The slicer then inserts additional G-code between the instructions for each layer to increment the printer’s z axis by h. Finally, the slicer inserts an initial preamble to set fan speeds as well as build plate and extruder temperatures to appropriate values for the material being printed. Throughout



```
w = 33; d = 20; h = 30;
difference() {
  cube([w, d, h]);
  translate([-1, 3, 3])
    cube([w + 2, 7, h + 1]);
  translate([w/2 - 19/2, 13, -1])
    cube([19, 3.5, h + 2]);
}
```

```
w = 33; d = 28; h = 30;
difference() {
  cube([w, d, h]);
  translate([-1, 3, 3])
    cube([w + 2, 7, h + 1]);
  translate([w/2 - 19/2, 13, -1])
    rotate([-15, 0, 0])
      cube([19, 3.5, h + 6]);
}
```

Figure 3 Renderings of a tea scoop holder with the original CAD program and the modified CAD program with the wall thickness and angle of the holder changed (changes are underlined).

the slicing process, the compiler performs optimizations to minimize the travel time of the print head.

2.2 Challenges in 3D printing

CAD/CAM and related computer-aided manufacturing are some of the oldest areas of computer science [30]. However, work in this space is often targeted at an industrial setting where accurate, fast (and therefore expensive) equipment is operated by highly motivated experts. The advent of affordable, desktop-class 3D printers for end-users gives rise to new challenges that we believe programming language techniques can help address. While improving hardware trends will inevitably ease some challenges with 3D printing, we believe that new software techniques will be essential for narrowing the gap between what's possible on affordable hardware and industrial practice. Toward that end, we propose initially focusing on three challenge areas:

Design. CAD tools have been widely used for decades, but still present users with a steep learning curve – even if one can clearly describe the desired model in plain English, it is not obvious what buttons to click and menus to navigate in the CAD tool to actually make that model from scratch. One possibility is to customize existing CAD models to meet new requirements. Unfortunately, most of the models shared in large online repositories like Thingiverse [32] are not the CAD models – they contain only the surface mesh in the form of STL which is difficult to edit successfully since much of the high-level information about the design (e.g. structural constraints) has been compiled away.

Performance. 3D printing is a slow process – it can take more than a day to print a large complex model. It is also generally unclear when and where the process can be made faster – going too fast in regions with fine detail can ruin a print because the material may not have time to cool sufficiently before the next layer.

Popular slicers such as Simplify3D [23], Cura [3], and ReplicatorG [20] cannot generate G-code that takes advantage of multiple print heads simultaneously, and thus in practice most printers with multiple print heads use only one. As discussed later, exploiting such latent parallelism significantly complicates the slicing strategy.

Furthermore, 3D printing typically involves manual inspection and tweaking. Users must often repeat the process several times to get the print they expected. Each iteration requires manually editing the CAD model, slicer parameters, or the printer settings. Ideally, users could avoid such manual fixes if slicers were able to automatically compensate for errors between iterations.

Reliability. 3D printing depends on the type of printer, the material being printed, and environmental conditions such as temperature. Even with perfect designs that have been correctly sliced, some problems that arise during printing can only be noticed *while* printing. It would be ideal if whenever an error occurs, we could halt the print to avoid wasting time and material and then work backwards to identify which command in the G-code led to the failure. Ideally, this information could even be used to repair errors automatically on the fly.

3 Synthesis

For many users, designing a part from scratch is challenging due to CAD's steep learning curve. They avoid this challenge by downloading, slicing, and printing parts shared as STL files in online repositories like Thingiverse [32]. Some users scan parts they wish to print using 3D scanners which also produce STL-like representations. These approaches are sufficient when the part is standalone and fits the user's needs. However, it is insufficient when the user wants to combine or modify parts. This is because many modifications are difficult in surface geometry representations like STL. STL tools like Blender [2] or AutoDesk's MeshMixer [1] can easily scale and rotate a design, but cannot effectively modify parts where some aspects depend on others (e.g., a gear whose tooth count depends on its radius). Even when CAD programs are made available, they can still be difficult to edit as end users often do not parameterize their designs or incorporate the structural constraints that make expert-written models easy to modify.

Past research in computer graphics and animation has focused on obtaining higher level representations from low level polygon meshes. For example, Krishnamurthy et al. [10] have shown how to fit smooth surfaces to irregular polygon meshes using B-splines and displacement maps. While smoothing can convert dense polygon meshes to aesthetically pleasing and more user-friendly representations, the output of these manipulations contains limited structural information about the model (e.g., if the model is a gear, what are the dimensions, orientations, and angles of its teeth?). Having this information is particularly important for desktop class 3D printing where users might want to individually customize functional parts by changing the relationships between its subcomponents (by varying the design parameters). On the other hand, in graphics and animation, aesthetics and performance are of key importance. The main difference between the idea we present here and prior work on fitting surfaces to polygon meshes is that we are primarily interested in recovering underlying structural information from polygon meshes and presenting it in the form of editable CAD models so that modifications and manipulations of subparts becomes straightforward. With this motivation, we propose *synthesizing* well-engineered and easy-to-edit CAD models from surface geometry representations like STL.

For example, consider the model of a tea scoop holder in Figure 3. In order to make the tea scoop fit better, we wanted to change the angle of the holder which required increasing the thickness of the base so that the holder would not cut through the walls due to the

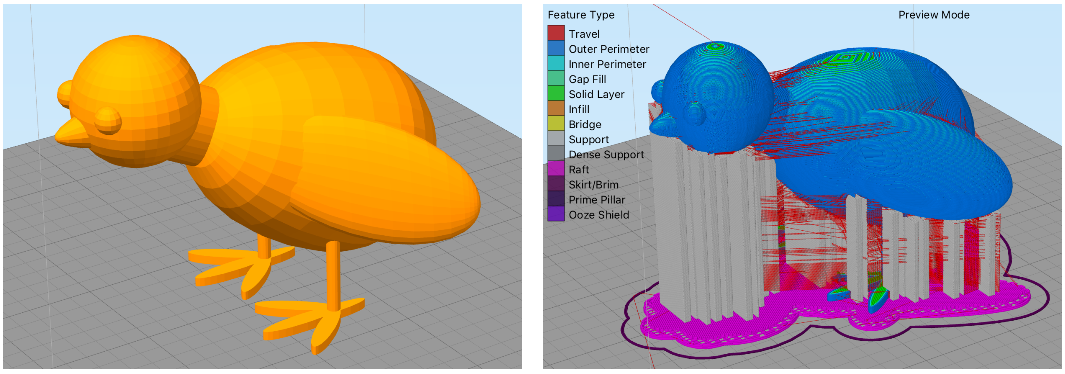


Figure 4 (a) CAD model of a chicken. (b) Tool path produced by slicer.

rotation. As Figure 3 shows, this change was very easy to make in the CAD model (changes underlined). In general, such changes are difficult to make by editing the STL surface mesh because some parts of the object remain unchanged while others are scaled and yet others are independently rotated.

As another example, consider the model of a chicken in Figure 4. The legs of this model are too thin and hence broke easily during printing. We wanted to make them thicker while keeping the rest of the model at the same scale and ensuring that the chicken still balances stably on its feet. The most convenient way to do this is to simply increase the radius of the leg cylinders in CAD. However, the CAD model for the chicken was not available – we only had access to the surface mesh in the form of an STL file. By reverse engineering the CAD from the surface mesh, we were able to easily thicken the legs and successfully print the model.

We have designed and implemented an early prototype synthesis algorithm (Algorithm 1) that achieves some of the goals above. This is essentially a form of decompilation: given an STL file S, find a simple CAD model which, when rendered, yields S. The algorithm is based on the principle that every CAD model can be synthesized by either subtracting one part from another part or unioning two parts together. Like many early program synthesis projects, this algorithm is a combinatorial search that is intractable for models with more than a dozen parts. However, in our problem domain, that is often plenty – OpenSCAD for example has only 4 types of primitive solid objects that can be combined to build various complex models. Figure 3 (column 2) shows example outputs of our algorithm.

Future directions

We believe that the intersection of CAD modeling and program synthesis is ripe with interesting problems. As one concrete example, our prototype synthesis algorithm tends to produce overly verbose CAD programs for highly symmetric parts since the algorithm's search omits looping constructs. More generally, we believe research should explore synthesis techniques for minimizing CAD models, similar to copy paste detection [11], and also for superoptimizing G-code, similar to techniques used in highly parallel low-power architectures [18].

We also propose further synthesis of high level models from surface meshes, but for more constrained targets than full CAD models. In particular, "peeling" based modeling where an object is approximated by composing interlocking flat sheets. Such designs have the advantage of being printable as only flat sheets, which is faster and packs tighter than traditional solid printing designs. Another natural generalization of this approach is exploring

Algorithm 1 Synthesis algorithm for generating CAD models

```
procedure SEARCH(model)
    if empty(model) then return [ Empty() ]
    else
        candidates = [ ]
        for b in PRIMITIVEBOUNDS(model) do
            diff = SUBTRACT (b, model)
            for c in SEARCH (diff) do
                candidates.append(Diff(b,c))
        for m1, m2 in SPLIT (model) do
            cs1 = SEARCH (m1)
            cs2 = SEARCH (m2)
            for c1 in cs1 do
                for c2 in cs2 do
                    candidates.append(Union(c1, c2))
        return candidates
```

how designs can be synthesized to take advantage of flexible filaments, e.g., by generating origami-inspired hinged designs.

4 Compilers for 3D Printing

3D printing seeks to efficiently compile an abstract object description to an actual, physical object. As described in Section 2, this compilation is typically composed of three stages: (1) CAD to STL, (2) STL to G-code, and (3) G-code to low-level hardware control signals. Just as traditional compiler research often focuses on middle-ends, here we focus on stage (2), also known as the *slicer*. The slicer is an ideal target as it typically has the greatest impact on print time and quality and also translates between standard languages independent of front-end CAD details and back-end printer firmware details.

In addition to the core compilation strategy presented in Section 2, slicers provide additional important features, as shown in Figure 4 (right). These include inserting support structures under part overhangs beyond some threshold angle d (typically $d \approx 45°$) and inserting a “raft,” a thick set of initial layers to improve part adhesion to the print bed. These additions are often essential for successfully printing complex parts and we hope to explore their design space in future work. However, we propose that initial PL research in this area should begin by focusing on the core compilation challenges of performance, accuracy, and correctness.

Parallelization

Many desktop class printers have multiple extruders which, in principle, should enable parallelism during the printing process. In practice, these extruders are only used one at a time to support features such as multi-color prints or using dissimilar raft and support materials. Exploiting the latent parallelism of multiple print heads requires extending the slicing algorithm to partition the tool paths within each layer to sets of paths for each head. This is challenging because the print heads are in a fixed orientation relative to one another (typically mounted linearly along the printer’s upper gantry). Thus, all extruders move together at fixed offsets from one another. Correctly generating G-code to manage the timing of all the coordinated movements presents a significant compilation challenge. Recently, some researchers have started focusing on parallelizing 3D printing for specially-built industrial printers [19], but the techniques used are proprietary and it is still unclear how they can be applied to help end-users operating desktop class printers.

Our goal is to explore how classic compiler techniques such as peephole optimizations can be applied to achieve parallelism. We suggest building directly upon the simple baseline slicing strategy without making additional assumptions about printer hardware, since, as mentioned above, an important objective is maintaining accessibility for end users. As a first step, we will develop a G-code analysis to identify situations where a secondary extruder will entirely traverse an extrusion path parallel to one that the primary extruder would eventually extrude anyway. In these scenarios, we can keep the G-code produced by the traditional slicing algorithm and merely tweak it to both (1) enable the secondary extruder as it traverses the parallel future path, (2) remove the G-code for the primary extruder following the subsequent path, and (3) patch up movements to connect the G-code before and after the removed path.

Future Directions

Analogous to the bad old days of early compilers, users must occasionally *manually* tweak the generated G-code to fix some print errors. This can be due to misbehavior of the printer hardware (e.g., certain movements may cause a stepper motor to "skip" a position, especially at high speeds, and require a small G-code tweak to mitigate) or bugs in the slicer (e.g., failure to retract the filament before a long movement, leading to smearing). We propose that future slicer research investigate improving accuracy by incorporating error from earlier trials into subsequent re-slicings. For example, if a generated part is 0.3mm too narrow in the x direction due to printer hardware inaccuracy, the slicer could automatically insert “padding” in x movements to compensate. We also hope to explore formalizing both STL and G-code in order to reason formally about the correctness of slicing algorithms. Such a formal foundation will also enable implementing more sophisticated slicing algorithms with confidence by proving them equivalent to simpler strategies.

5 Runtime Monitoring

Desktop-class 3D printers are currently affordable because they use inexpensive stepper motors, basic extruder designs, and lightweight frames. While these economical choices are precisely what make the technology broadly available, they also lead to unreliability and error. Even with a perfect CAD design and error-free slicing, prints can still fail due to motors skipping steps, nozzles clogging, and environmental variations in temperature and humidity. Future hardware improvements may mitigate some of these concerns, but even experts operating high-end equipment still must often iteratively refine their process to get the best results. Debugging failures and making performance tweaks is difficult because the operation of the printer is opaque as it interprets tens of thousands of lines of G-code to generate a part. In this section we propose video-based runtime monitoring techniques to help debugging, detect printing errors, and address failures on the fly. These techniques take inspiration from traditional programming language runtimes which aid debugging and ensure safety by providing facilities to handle exceptions, prevent errors like division by zero or array out of bounds accesses, and dynamic type checking.

Record. Resemble? Respond!

Sitthi-Amorn et al. [24] have shown the use of depth cameras in runtime monitoring to repair height errors *on the fly* in their 3D printing platform, MultiFab. They use image processing and 3D scanning to identify pixels with varying depths and add pixel-wise corrective layers to the printing process. We propose extending these techniques to validate and repair other properties (e.g., dimensional accuracy) while also keeping the hardware requirements affordable and performance overhead low.

A first step in aiding low cost print failure debugging is to log operations using commodity cameras to record prints and tag each frame with the currently executing G-code instruction. The logs can help users identify where the G-code may need to be tweaked to address poor printer performance. These logs can also help printer firmware developers tune and debug the low-level control code that translates G-code operations into carefully timed motor commands.

With this simple foundation laid, the next natural step would be to develop analyses which compare G-code programs and printing video streams to ensure that execution correctly matches expected behavior. Such analyses can be used to abort print jobs early or selectively disable printing in independent regions where something has gone wrong. This can be useful when a long-running job printing multiple copies of a complex part goes wrong for just one of the copies. Currently, the printer blindly continues executing G-code, oblivious to the small localized failure. This often causes cascading errors as subsequent extrusions over the failed area do not adhere correctly and are dragged over to interfere with the printing of other copies which, independent of the initial failure, would have otherwise successfully printed. If instead a runtime monitor could detect that an execution is no longer faithfully simulating the behavior specified by the input G-code program, printing could be halted early for failed parts, allowing other parts to successfully finish printing and to avoid wasted material.

Future Directions

A major challenge with video-based runtime monitoring for 3D printers is that responses must be carried out quickly in order to be effective, but printers typically only contain cheap microcontrollers for executing firmware. Future research should explore hybrid analysis techniques where partial evaluation of a video-based analysis is carried out at slicing time, before the first G-code instruction for a part is ever sent to the printer. Video-based analyses should also be investigated to enable coordination of printers with other manufacturing processes, e.g., a robotic arm. Such coordination could enable more sophisticated multi-process desktop manufacturing, e.g., by enabling a pick-and-place machine to embed magnets or metal fixtures within a part as it is being printed.

6 Related work

Several projects have explored new analyses of 3D models, slicing techniques, and user interfaces to help mitigate current limitations in 3D printing. These results appear across a diverse array of venues, from graphics to HCI, and many focus on industrial settings or specialized hardware which future economies of scale or hardware improvements may make broadly accessible. The programming languages community has only recently started looking into these problems, e.g., in OpenFab [34], a framework for programmatically specifying material and texture with the help of a domain specific language. Below we highlight some noteworthy and inspirational examples from other communities attacking 3D printing challenges.

The strength of a 3D printed part is non-uniform due to stronger adhesion within a layer than across layers. Umetani et al. developed a static analysis of CAD models to determine optimal printing orientations for maximizing mechanical strength [33]. Galjaard et al. explored optimizing CAD models to minimize material use while maintaining key strength performance properties [6]. Teibrich et al. [31] introduced a patching technique to repair already printed objects to avoid printing again from scratch, thereby saving material. Delfs et al. [4] developed a tool that can optimize the orientation of a part during 3D printing in order to make the surface smoother.

In terms of speeding up early prints, Mueller et al.'s work on WirePrint [13] and faBrickator [15] provide creative examples of how non-uniform height slicing and hybrid build approaches (in this case using Lego$^{\text{TM}}$) can radically reduce turnaround time when developing prototypes. Mueller et al. [14] have also introduced laser cutting based techniques for rapid prototyping using folding and stretching of an object instead of cutting joints.

Stava et al. [29] proposed a technique based on structural analysis that automatically detects and fixes structural problems in models. Zhou et al. [35] proposes another structural analysis for 3D printable objects that uses material and geometric properties. FlatFitFab [12] is an interactive interface that allows users to specify functional parts and provides real-time simulations that visualize stress. Dumas et al. [5] recently proposed a texture synthesis algorithm that takes a surface mesh and an example pattern as inputs and generates a texture.

New 3D printing applications are also constantly emerging, particularly within medical contexts such as tissue and organ fabrication; customized prosthetics and implants; and drug manufacturing, dosage forms, delivery, and discovery [9, 22].

7 Conclusion

In this paper, we proposed an early research agenda for using programming language techniques to help make affordable, desktop-class manufacturing processes (such as 3D printing) more accurate, fast, and accessible to end-users. Even as the available hardware improves, we believe there will continue to be opportunities for software to narrow the gap between expensive, high-end processes and the widely available, democratized means of production. Here, we discussed three major domains where 3D printing in particular can benefit from such research – applying program synthesis techniques to improve the design process, applying compiler techniques to speed up and improve prints, and applying runtime monitoring approaches to ease debugging. We are eager to further explore these particular lines of work and look forward to seeing how the PL community can help address these challenges more broadly.

Acknowledgements. We thank Michael D. Ernst for sharing interesting CAD and STL models with us. Doug Woos and John Toman were very helpful in providing feedback on earlier drafts of the paper. We are especially grateful to our shepherd Jonathan Ragan-Kelley and the anonymous reviewers for pointing us to interesting related working and guiding several key improvements to the paper. Finally, we thank the members of the UW PLSE lab for their camaraderie and putting up with many hours of noisy 3D printing.

References

1 Autodesk. Meshmixer. `http://www.meshmixer.com/`.

2 blender. Creative freedom starts here. `https://www.blender.org/`.

3 Cura Software. https://ultimaker.com/en/products/cura-software.

4 P. Delfs, M. T̈ows, and H.-J. Schmid. Optimized build orientation of additive manufactured parts for improved surface quality and build time. *Additive Manufacturing*, 12, Part B:314–320, 2016. Special Issue on Modeling & Simulation for Additive Manufacturing. doi:10.1016/j.addma.2016.06.003.

5 Jérémie Dumas, An Lu, Sylvain Lefebvre, Jun Wu, and Christian Dick. By-example synthesis of structurally sound patterns. *ACM Trans. Graph.*, 34(4):137:1–137:12, July 2015. doi:10.1145/2766984.

6 Salomé Galjaard, Sander Hofman, and Shibo Ren. New opportunities to optimize structural designs in metal by using additive manufacturing. In Philippe Block, Jan Knippers, Niloy J. Mitra, and Wenping Wang, editors, *Advances in Architectural Geometry 2014*, pages 79–93. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-11418-7_6.

7 Gartner Forecast: 3D Printers, Worldwide, 2015. https://www.gartner.com/doc/3132417.

8 T. Grimm. *User's Guide to Rapid Prototyping*. Society of Manufacturing Engineers, 2004.

9 G. T. Klein, Y. Lu, and M. Y. Wang. 3D Printing and Neurosurgery – Ready for Prime Time? *World Neurosurgery*, 80(3):233–235, 9 2013.

10 Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH'96, pages 313–324, New York, NY, USA, 1996. ACM. doi:10.1145/237170.237270.

11 Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation – Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251254.1251274.

12 James McCrae, Nobuyuki Umetani, and Karan Singh. Flatfitfab: Interactive modeling with planar sections. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST'14, pages 13–22, New York, NY, USA, 2014. ACM. doi:10.1145/2642918.2647388.

13 Stefanie Mueller, Sangha Im, Serafima Gurevich, Alexander Teibrich, Lisa Pfisterer, François Guimbretière, and Patrick Baudisch. WirePrint: 3D Printed Previews for Fast Prototyping. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST'14, pages 273–280, New York, NY, USA, 2014. ACM. doi:10.1145/2642918.2647359.

14 Stefanie Mueller, Bastian Kruck, and Patrick Baudisch. LaserOrigami: Laser-cutting 3D Objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'13, pages 2585–2592, New York, NY, USA, 2013. ACM. doi:10.1145/2470654.2481358.

15 Stefanie Mueller, Tobias Mohr, Kerstin Guenther, Johannes Frohnhofen, and Patrick Baudisch. faBrickation: Fast 3D Printing of Functional Objects by Integrating Construction Kit Building Blocks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'14, pages 3827–3834, New York, NY, USA, 2014. ACM. doi:10.1145/2556288.2557005.

16 OpenSCAD. http://www.openscad.org/.

17 David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD'88, pages 109–116, New York, NY, USA, 1988. ACM. doi:10.1145/50202.50214.

18 Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. *SIGPLAN Not.*, 51(4):297–310, March 2016. `doi:10.1145/2954679.2872387`.

19 Project Escher. `http://projectescher.com/`.

20 ReplicatorG lowering the barrier to 3D printing. `http://replicat.org/`.

21 Rhinoceros. `https://www.rhino3d.com/`.

22 C. Schubert, M. C. van Langeveld, and L. A. Donoso. Innovations in 3D Printing: a 3D Overview from Optics to Organs. *British Journal of Ophthalmology*, 98(2):159–161, 2014.

23 SIMPLIFY3D. `https://www.simplify3d.com/`.

24 Pitchaya Sitthi-Amorn, Javier E. Ramos, Yuwang Wangy, Joyce Kwan, Justin Lan, Wenshou Wang, and Wojciech Matusik. MultiFab: A Machine Vision Assisted Platform for Multi-material 3D Printing. *ACM Trans. Graph.*, 34(4):129:1–129:11, July 2015. `doi:10.1145/2766962`.

25 Skeinforge. `http://reprap.org/wiki/Skeinforge`.

26 SketchUp. `http://www.sketchup.com/`.

27 Slic3r. `http://slic3r.org/`.

28 Solidworks. `http://www.solidworks.com/`.

29 Ondrej Stava, Juraj Vanek, Bedrich Benes, Nathan Carr, and Radomír Měch. Stress Relief: Improving Structural Strength of 3D Printable Objects. *ACM Trans. Graph.*, 31(4):48:1–48:11, July 2012. `doi:10.1145/2185520.2185544`.

30 Ivan E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop*, DAC'64, pages 6.329–6.346, New York, NY, USA, 1964. ACM. `doi:10.1145/800265.810742`.

31 Alexander Teibrich, Stefanie Mueller, François Guimbretière, Robert Kovacs, Stefan Neubert, and Patrick Baudisch. Patching physical objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software &#38; Technology*, UIST'15, pages 83–91, New York, NY, USA, 2015. ACM. `doi:10.1145/2807442.2807467`.

32 Thingiverse. `http://www.thingiverse.com/`.

33 Nobuyuki Umetani and Ryan Schmidt. Cross-sectional Structural Analysis for 3D Printing Optimization. In *SIGGRAPH Asia 2013 Technical Briefs*, SA'13, pages 5:1–5:4, New York, NY, USA, 2013. ACM. `doi:10.1145/2542355.2542361`.

34 Kiril Vidimče, Szu-Po Wang, Jonathan Ragan-Kelley, and Wojciech Matusik. OpenFab: A Programmable Pipeline for Multi-material Fabrication. *ACM Trans. Graph.*, 32(4):136:1–136:12, July 2013. `doi:10.1145/2461912.2461993`.

35 Qingnan Zhou, Julian Panetta, and Denis Zorin. Worst-case structural analysis. *ACM Trans. Graph.*, 32(4):137:1–137:12, July 2013. `doi:10.1145/2461912.2461967`.

Toward Semantic Foundations for Program Editors*

Cyrus Omar[1], Ian Voysey[2], Michael Hilton[3], Joshua Sunshine[4], Claire Le Goues[5], Jonathan Aldrich[6], and Matthew A. Hammer[7]

1 **Carnegie Mellon University, Pittsburgh, PA, USA**
`iev@cs.cmu.edu`

2 **Carnegie Mellon University, Pittsburgh, PA, USA**
`iev@cs.cmu.edu`

3 **Oregon State University, Corvallis, OR, USA**
`hiltonm@eecs.oregonstate.edu`

4 **Carnegie Mellon University, Pittsburgh, PA, USA**
`sunshine@cs.cmu.edu`

5 **Carnegie Mellon University, Pittsburgh, PA, USA**
`clegoues@cs.cmu.edu`

6 **Carnegie Mellon University, Pittsburgh, PA, USA**
`aldrich@cs.cmu.edu`

7 **University of Colorado Boulder, Boulder, CO, USA**
`matthew.hammer@colorado.edu`

Abstract

Programming language definitions assign formal meaning to complete programs. Programmers, however, spend a substantial amount of time interacting with *incomplete* programs – programs with holes, type inconsistencies and binding inconsistencies – using tools like program editors and live programming environments (which interleave editing and evaluation). Semanticists have done comparatively little to formally characterize (1) the static and dynamic semantics of incomplete programs; (2) the actions available to programmers as they edit and inspect incomplete programs; and (3) the behavior of editor services that suggest likely edit actions to the programmer.

This paper serves as a vision statement for a research program that seeks to develop these "missing" semantic foundations. Our hope is that these contributions, which will take the form of a series of simple formal calculi equipped with a tractable metatheory, will guide the design of a variety of current and future interactive programming tools, much as various lambda calculi have guided modern language designs. Our own research will apply these principles in the design of Hazel, an experimental *live lab notebook* programming environment designed for data science tasks. We plan to co-design the Hazel language with the editor so that we can explore concepts such as edit-time semantic conflict resolution mechanisms and mechanisms that allow library providers to install library-specific editor services.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, D.2.6 Programming Environments

Keywords and phrases program editors; type systems; live programming; program prediction

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.11

* This work is supported in part through a gift from Mozilla; by NSF under grant numbers CCF-1619282, 1553741 and 1439957; by AFRL and DARPA under agreement #FA8750-16-2-0042; and by the NSA under lablet contract #H98230-14-C-0140. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Mozilla, NSF, AFRL, DARPA or NSA.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 11; pp. 11:1–11:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Language-aware program editors (like Eclipse or Emacs, with the appropriate extensions installed [13]) offer programmers a number of useful editor services. Simple examples include (1) syntax highlighting, (2) type inspection, (3) navigation to variable binding sites, and (4) refactoring services. More sophisticated editors provide context-aware code and action suggestions to the programmer (using various code completion, program synthesis and program repair techniques). Many editors also offer *live programming* [26, 5] services, e.g. by displaying the run-time value of an expression directly within the editor as the program runs.

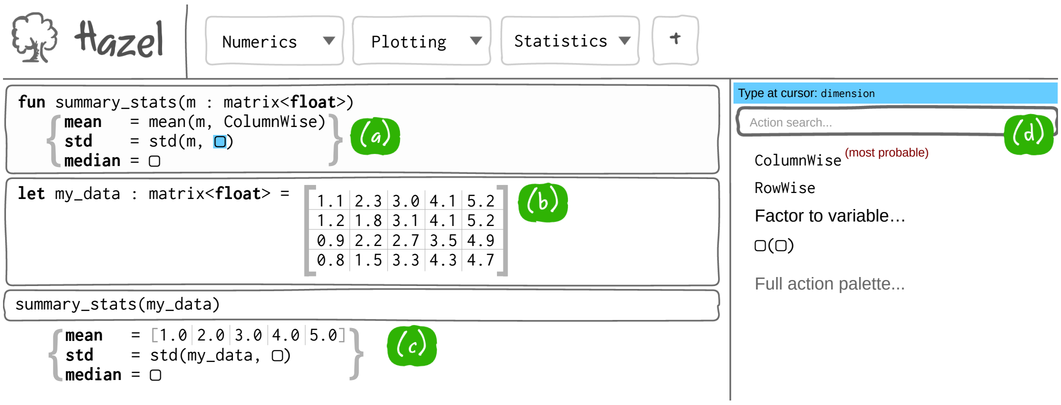
When these editor services encounter *complete programs* – programs that are well-formed and semantically meaningful (i.e. assigned meaning) according to the definition of the language in use – they can rely on a variety of well-understood reasoning principles and program manipulation techniques. For example, a syntax highlighter for well-formed programs can be generated automatically from a context-free grammar [47] and the remaining editor services enumerated above can follow the language's type and binding structure as specified by a standard static semantics. Live programming services can additionally follow the language's dynamic semantics.

The problem, of course, is that many of the edit states encountered by a program editor do not correspond to complete programs. For example, the programmer may be in the midst of a transient edit, or the programmer may have introduced a type error somewhere in the program. Standard language definitions are silent about incomplete programs, so in these situations, simple program editors disable various editor services until the program is again in a complete state. In other words, useful editor services become unavailable when the programmer needs them most! More advanced editors attempt to continue to provide editor services during these incomplete states by using various *ad hoc* and poorly understood heuristics that rely on idiosyncratic internal representations of incomplete programs.

This paper advocates for a research program that seeks to understand both incomplete programs, and the editor services that interact with them, as semantically rich mathematical objects. This research program will broaden the scope of the "programming language theory" (PLT) tradition, which has made significant advances by treating complete programs, programming languages and logics as semantically rich mathematical objects.

In following the PLT tradition, we intend to start by developing a series of minimal calculi that build upon well-understood typed lambda calculi to capture the essential character of incomplete programs and various editor services of interest. Editor designers will be able to apply the insights gained from studying these calculi (together with insights gained from the study of human factors and other topics) to design more sophisticated program editors. Some of these editors will evolve directly from editors already in use today. In parallel with these efforts, we plan to design a "clean-slate" programming environment, Hazel, based directly on these first principles. This will allow researchers to explore the frontier of what is possible when one considers languages and editors within a common theoretical framework. Such a clean-slate design will also likely prove useful in certain educational settings, and even some day evolve into a practical tool.

Figure 1 shows a mockup of the Hazel user interface, which is loosely modeled after the widely adopted IPython / Jupyter lab notebook interface [36]. This figure will serve as our running example throughout the remainder of the paper. Each section below briefly summarizes a fundamental problem that we must confront as we seek to develop a semantic foundation for advanced program editors. For each problem, we discuss existing approaches, including those advanced by our own recent research, and suggest a number of promising future research directions that we hope that the community will pursue.



```
fun summary_stats(m : matrix<float>)
   { mean   = mean(m, ColumnWise) }
   { std    = std(m, □)           }
   { median = □                   }

let my_data : matrix<float> =  1.1 2.3 3.0 4.1 5.2
                               1.2 1.8 3.1 4.1 5.2
                               0.9 2.2 2.7 3.5 4.9
                               0.8 1.5 3.3 4.3 4.7

summary_stats(my_data)
   { mean   = [1.0 2.0 3.0 4.0 5.0] }
   { std    = std(my_data, □)       }
   { median = □                     }
```

Type at cursor: dimension

Action search...

ColumnWise (most probable)

RowWise

Factor to variable...

□(□)

Full action palette...

Figure 1 A mockup of Hazel.

2 Problem 1: Syntactically Malformed Edit States

Textual program editors frequently encounter edit states that are not well-formed with respect to the textual syntax of complete programs. For example, consider a programmer constructing a call to a function `std`:

```
std(m,
```

There is a syntax error, so editor services that require a syntactically complete program must be disabled. This is unsatisfying.

Sophisticated editors like Eclipse, and editor generators like Spoofax [20], use *error recovery* heuristics that silently insert tokens so that the editor-internal representation is well-formed [1, 7, 14, 19]. These heuristics are typically provided manually by the grammar designer, though certain heuristics can be generated semi-automatically by tools that are given a description of the scoping conventions of the language or of secondary notational conventions (e.g. whitespace) [19, 12]. Error recovery heuristics require guessing at the programmer's intent, so they are fundamentally *ad hoc* and can confuse the programmer [19].

A more systematic alternative approach, and the approach that we plan to explore with Hazel, is to build a *structure editor* – a program editor where every edit state maps onto a syntax tree, with *holes* representing leaves of the tree that have not yet been constructed. This representation choice sidesteps the problem of syntactically malformed edit states. Notice that in Figure 1, the program fragment in cell **(a)** contains holes, appearing as squares. This design also permits non-textual *projections* of expressions, e.g. the 2D projection of a matrix value in cell **(b)**. We will return to the topic of non-textual projections below.

Structure editors have a long history. For example, the Cornell Program Synthesizer was developed in the early 1980s [45]. Although text-based syntax continues to predominate, there remains significant interest in structure editors today, particularly in practice. For example, Scratch is a structure editor that has achieved success as a tool for teaching children how to program [40]. `mbeddr` is an editor for a C-like language [51], built using the commercially supported MPS structure editor workbench [50]. TouchDevelop is an editor for an object-oriented language [46]. Lamdu [24] and Unison [8] are open source structure editors for functional languages similar to Haskell. Most work on structure editors has focused on the user interfaces that they present. This is important work – presenting a fluid user interface involving higher-level edit actions is a non-trivial problem, and some aspects of this problem remain open even after many years of research. There is reason to be optimistic, however,

$$
\begin{array}{llll}
\mathsf{HTyp}: & \dot{\tau} & ::= & \dot{\tau} \to \dot{\tau} \mid \texttt{num} \mid \llparenthesis\rrparenthesis \\
\mathsf{HExp}: & \dot{e} & ::= & x \mid \lambda x.\dot{e} \mid \dot{e}(\dot{e}) \mid \underline{n} \mid \dot{e} + \dot{e} \mid (\dot{e} : \dot{\tau}) \mid \llparenthesis\rrparenthesis \mid \llparenthesis\dot{e}\rrparenthesis
\end{array}
$$

Figure 2 Syntax of H-types and H-expressions in the Hazelnut calculus [34].

with recent studies suggesting that programmers experienced with a modern keyboard-driven structure editor (e.g. `mbeddr`) can be highly productive [3, 52].

Researchers have also explored various "hybrid" approaches, which incorporate holes into an otherwise textual program editor. These hybrid approaches are appealing in part because tools for interacting with text, like regular expressions and various differencing techniques used by version control systems, are already well-developed. For example, recent work on *syntactic placeholders* envisions a textual program editor where edit actions cause textual placeholders (a.k.a. holes) of various sorts to appear, rather than leaving the program transiently malformed [2]. This "approximates" the experience of a structure editor in common usage, while allowing the programmer to perform arbitrary text edits when necessary. Some programming systems, e.g. recent iterations of the Glasgow Haskell Compiler (GHC) [42] and the Agda proof assistant [32], support a workflow where the programmer places holes manually at locations in the program that remain under construction. Another hybrid approach would be to perform error recovery by attempting to insert holes into the internal representation used by the program editor, without including them in the surface syntax exposed to programmers. If "pure" structure editing proves too rigid as we design Hazel, we will explore hybrid approaches.

3 Problem 2: Statically Meaningless Edit States

No matter how an editor confronts syntactically malformed edit states, it must also confront edit states that are syntactically well-formed but statically meaningless. For example, the following value member definition (assuming an ML-like language) has a type inconsistency:

```
val x : float = std(m, ColumnWise)
```

because `std` has type `matrix(float) * dimension -> vec(float)`, but the type annotation on `x` is `float`, rather than `vec(float)`. This leaves the entire surrounding program formally meaningless according to a standard static semantics.

In the presence of syntactic holes, the problem of reasoning statically about incomplete programs becomes even more interesting. Consider the incomplete expression `std(m, □)` from cell **(a)** in Figure 1. Although it is intuitively apparent that the type of this expression, after hole instantiation, could only be `vec(float)` (the return type of `std`), and that the hole must be instantiated with values of type `dimension`, the static semantics of complete expressions is again silent about these matters.

Various heuristic approaches are implemented in Eclipse and other sophisticated tools, but the formal character of these heuristics are obscure, buried deep within their implementations. What is needed is a clear static semantics for incomplete programs, i.e. programs that contain holes (in both expressions and types), type inconsistencies, binding inconsistencies (i.e. unbound variables), and other static problems. Such a static semantics is necessary for Hazel to be able to provide type inspection services. For example, in the right column of Figure 1, Hazel is informing the programmer that the expression at the cursor, highlighted in blue in cell **(a)**, must be of type `dimension`). Similarly, Hazel must be able to assign the incomplete function `summary_stats` an incomplete function type for it to be able to understand subsequent applications of `summary_stats`. Here, the function body has been filled out enough to be able to assign the function the following incomplete function type:

```
matrix(float) -> { mean :  vec(float), std :  vec(float), median : □ }
```

We have investigated a subset of this problem in recent work [34] by defining a static semantics for a simply typed lambda calculus (with a single base type, `num`, for simplicity) extended with holes and type inconsistencies (but no binding inconsistencies). Figure 2 defines the syntactic objects of this calculus – *H-types*, $\dot{\tau}$, are types with holes $\llparenthesis\rrparenthesis$, and *H-expressions*, $\dot{e}$, are expressions with holes $\llparenthesis\rrparenthesis$, and marked type inconsistencies, $\llparenthesis\dot{e}\rrparenthesis$. We call marked type inconsistencies *non-empty holes*, because they mark portions of the syntax tree that remain incomplete and behave semantically much like empty holes. Types and expressions that contain no holes are *complete types* and *complete expressions*, respectively.

We will not reproduce further details here. Instead, let us simply note some interesting connections with other work.

First, type holes behave much like unknown types, ?, from Siek and Taha's pioneering work on gradual typing [41]. This discovery is quite encouraging, given that gradual typing is also motivated by a desire to make sense of one class of "incomplete program" – programs that have not been fully annotated with types.

Empty expression holes have also been studied formally, e.g. as the *metavariables* of contextual modal type theory (CMTT) [31]. In particular, expression holes can have types and are surrounded by contexts, just as metavariables in CMTT are associated with types and contexts. This begins to clarify the logical meaning of a typing derivation in Hazelnut – it conveys well-typedness relative to an (implicit) modal context that extracts each expression hole's type and context. The modal context must be emptied – i.e. the expression holes must be instantiated with expressions of the proper type in the proper context – before the expression can be considered complete. This relates to the notion of modal necessity in contextual modal logic.

For interactive proof assistants that support a tactic model based directly on hole filling, the connection to CMTT and similar systems is quite salient. For example, Beluga [37] is based on dependent CMTT and aspects of Idris' editor support [4] are based on a similar system – McBride's OLEG [25]. As we will discuss in Sec. 5, our notion of a program editor supports actions beyond hole filling.

There are a number of future research directions that are worth exploring.

Binding inconsistencies. In the simple calculus developed so far, all variables must be bound before they are used, including those in holes. We plan extend Hazelnut to support reasoning when a variable is mentioned without having been bound (as is a common workflow). Dagenais and Hendren also studied how to reason statically about programs with binding errors using a constraint system, focusing on Java programs whose imports are not completely known [11]. They neither considered programs with holes or other type inconsistencies, nor did they formally specify their technique. However, they provide a useful starting point.

Expressiveness. The simple calculus discussed above is only as expressive as the typed lambda calculus with numbers. We must scale up the semantics to handle other modern language features. Our plan is to focus initially on functional language constructs (so that Hazel can be used to teach courses that are today taught using Standard ML, OCaml or Haskell). This will include recursive and polymorphic functions, recursive types, and labeled product (record) and sum types. We also propose to investigate ML-style structural pattern matching. All of these will require defining new sorts of holes and static inconsistencies, including: (1) non-empty holes at the type level, to handle kind inconsistencies; (2) holes in label position; and (3) holes and type inconsistencies in patterns.

Automation. Although we plan to explore some of these language extensions "manually," extending our existing mechanized metatheory, we ultimately plan to *automatically* generate a statics for incomplete terms from a standard statics for complete terms, annotated perhaps with additional information. There is some precedent for this in recent work on the Gradualizer, which is capable of producing a gradual type system from a standard type system with lightweight annotations that communicate the intended polarities of certain constructs [10]. However, although it provides a good starting point, gradual type systems only consider the problem of holes in types. Our plan is to build upon existing proof automation techniques, e.g. Agda's reflection [48] (in part because our present mechanization effort is in Agda).

4 Problem 3: Dynamically Meaningless Edit States

Modern programming tools are increasingly moving beyond simple "batch" programming models by incorporating *live programming* features that interleave editing and evaluation [44, 43, 26]. These tools provide programmers with rapid feedback about the dynamic behavior of the program they are editing, or selected portions thereof [27]. Examples include *lab notebooks*, e.g. the popular IPython/Jupyter [36], which allow the programmer to interactively edit and evaluate program fragments organized into a sequence of cells (an extension of the read-eval-print loop (REPL)); spreadsheets; live graphics programming environments, e.g. SuperGlue [26], Sketch-n-Sketch [9] and the tools demonstrated by Bret Victor in his lectures [49]; the TouchDevelop live UI framework [5]; and live visual and auditory dataflow languages [6]. In the words of Burckhardt et al. [5], live programming environments "capture the imagination of today's programmers and promise to narrow the temporal and perceptive gap between program development and code execution".

Our proposed design for Hazel combines aspects of several of these designs to form a *live lab notebook interface*. It will use the edit state of each cell to continuously update the output value displayed for that cell and subsequent cells that depend on it. Uniquely, rather than providing meaningful feedback about the dynamic behavior only once a cell becomes complete, Hazel will provide meaningful feedback also about the dynamic behavior of incomplete cells (and thereby further tighten Burckhardt's "perceptive gap").

For example, in cell **(c)** of Figure 1, the programmer applies the incomplete function `summary_stats` to the matrix `my_data`, and the editor is still able to display a result. The value of the column-wise mean is fully determined, because evaluation does not encounter any holes, whereas the standard deviation and median computations cannot be fully evaluated. Notice, however, that the standard deviation computation does communicate the substitution of the applied argument, `my_data`, for the variable `m`.[1]

To realize this functionality, we need a dynamic semantics for incomplete programs that builds upon our proposed static semantics. There is some precedent for this: research in gradual typing considers the dynamic semantics of programs with holes in types, and our proposed static semantics for incomplete programs borrows technical machinery from

[1] To avoid exposing the internals of imported library functions, evaluation does not step into functions, like `std`, that have been imported from external libraries indicated by the row at the top of Figure 1 (unless specifically requested, not shown).

theoretical work on gradual typing [41]. However, we need a dynamic semantics for incomplete programs that also have expression holes (and in the future, other sorts of holes).

Research on CMTT has not yet considered the problem of evaluating expressions under a non-empty metavariable context. Normally, this would violate the classical notion of Progress – evaluation can neither proceed, nor has it produced a value. We conjecture that this is resolved by (1) positively characterizing *indeterminate* evaluation states, those where a hole blocks progress at all locations within the expression, and (2) defining a notion of Indeterminate Progress that allows for evaluation to stop at an indeterminate evaluation state. By gradualizing CMTT and defining these notions, we believe we can achieve the basic functionality described above.

There are several more applications that we aim to explore after developing these initial foundations. For example, it would be useful for the programmer to be able to select a hole that appears in an indeterminate state and be taken to its original location. There, they should be able to inspect the *value* of a subexpression under the cursor in the environment of the selected hole (rather than just its type). Again, CMTT's closures provide a theoretical starting point for this debugger service.

It would also be useful to be able to continue evaluation where it left off after making an edit to the program that corresponds to hole instantiation. This would require proving a commutativity property regarding hole instantiation. Fortunately, initial research on commutativity properties for holes has been conducted for CMTT, which will serve as a starting point for this work [31]. There are likely to be interesting new theoretical questions (and, likely, some limitations) that arise if one adds non-termination and memory effects.

Relatedly, IPython/Jupyter [36] support a feature whereby numeric variable(s) in cells can be marked as being "interactive", which causes the user interface to display a slider. As the slider value changes, the value of the cell is recomputed. It would be useful to be able to use the mechanisms just proposed to incrementalize parts of this recomputation.

5 Problem 4: A Calculus of Edit Actions

The previous sections considered the structure and meaning of intermediate edit states. However, to understand the act of *editing* itself, we need *a calculus of edit actions* that governs transitions between these edit states.

In a structure editor, the ideal would be for every possible edit state to be both statically and dynamically meaningful according to the semantics proposed in the previous two sections. This corresponds formally to proving a metatheorem about the action semantics: when the initial edit state is semantically meaningful, the edit state that results from performing an action is as well. In a textual or hybrid setting, these structured edit actions would need to be supplemented by lower-level text edit actions that may not maintain this invariant. In addition to this crucial metatheorem, which we call *sensibility*, there are a number of other metatheorems of interest that establish the expressive power of the action semantics, e.g. that every well-typed term can be constructed by some sequence of edit actions.

In our recent work on Hazelnut, we have developed an action calculus for the minimal calculus of H-types and H-expressions described in Section 3 [34]. We have mechanically proven the sensibility invariant, as well as expressivity metatheorems, using the Agda proof assistant. What remains is to investigate *action composition principles*. For example, it would be worthwhile to investigate the notion of an *action macro*, whereby functional programs could themselves be lifted to the level of actions to compute non-trivial compound actions. Such compound actions would give a uniform description of transformations ranging from the simple – like "move the cursor to the next hole to the right" – to quite complex whole program refactorings, while remaining subject to the core semantics. Using proof automation, it should be possible to prove that an action macro implements derived action logic that is admissible with respect to the core semantics. This would eliminate the possibility of "edit-time" errors. This is closely related to work on tactic languages in proof assistants, e.g. the Mtac typed macro language for Coq [53], differing again in that the action language involves notions other than hole filling.

6 Problem 5: Meaningful Suggestion Generation and Ranking

The simplest edit actions will be bound to keyboard shortcuts. However, Hazel will also provide suggestions to help the programmer edit incomplete programs by providing a *suggestion palette*, marked **(d)** in Figure 1. This palette will suggest semantically relevant code snippets when the cursor is on an empty hole. It will also suggest other relevant edit actions, including high-level edit actions implemented by imported action macros (e.g. the refactoring action in Figure 1). When the cursor is on a non-empty hole, indicating a static error, it will suggest bug fixes. We plan to also consider bugs that do not correspond to static errors, including those identified explicitly by the programmer, and those related to assertion failures or exceptions encountered when using the live programming features of Hazel. In these situations, we plan to build on existing automated fault localization techniques [18, 38, 39].

Note that features like these are not themselves novel. Many editors provide contextually relevant suggestions. Indeed, suggestion generation is closely related to several major research areas: code completion [30, 17], program synthesis [15], and program repair [22, 28, 23, 21].

The problems that such existing systems encounter is exactly the problem we have been discussing throughout this proposal: when attempting to integrate these features into an editor, it is difficult to reason about malformed or meaningless edit states. Many of these systems therefore fall back onto tokenized representations of programs [17]. Because Hazel will maintain the invariant that every edit state is a syntactically and semantically meaningful formal structure, we can develop a more principled solution to the problem of generating meaningful suggestions. In particular, we will be able to *prove* that every action suggestion generated for a particular edit state is meaningful for that edit state.

In addition to investigating the problem of populating the suggestion palette with semantically valid actions, we will consider the problem of evaluating the statistical likelihood of the suggestions. This requires developing a statistical model over actions. We will prove that this statistical model is a proper probability distribution (e.g. that it "integrates" to 1), and that it assigns zero probability to semantically invalid actions. We will also develop techniques for estimating the parameters of these distributions from a corpus of code or a corpus of edit actions. Collectively, we refer to these contributions as a *statistical action suggestion semantics.*

Ultimately, we envision this work as being the foundation for an *intelligent programmer's assistant* that is able to integrate semantic information gathered from the incomplete program with statistics gathered from other programs and interactions that the system has observed to do much of the "tedious" labor of programming, without hiding the generated code from the programmer (as is the case with fully automated program synthesis techniques).

7 Language-Editor Co-Design

In designing Hazel, we are intentionally blurring the line between the programming language and the program editor. This opens up a number of interesting research directions in language-editor co-design. For example, it may be possible to recast "tricky" language mechanisms, like function overloading, type classes [16], implicit values, and unqualified imports, as editor mechanisms. Because we will be treating programming as a structured conversation between the programmer and the programming environment, the editor can simply ask the programmer to resolve ambiguities when they arise. The programmer's choice is then stored unambiguously in the underlying syntax tree.

Another important research direction lies in exploring how types can be used to control the presentation of expressions in the editor. In the textual setting, we have developed *type-specific languages* (TSLs) [33]. It should be possible to define an analagous notion of *type-specific projections* (TSPs) in the setting of a structure editor. For example, the matrix projection shown in Figure 1 need not be built in to Hazel. Instead, the `Numerics` library provider will be able to introduce this logic. In particular, TSPs will define not only derived visual forms, but also derived edit actions (e.g. "add new column" for the example just given.) It should be possible to switch between multiple projections (including purely textual projections) while editing code and interacting with values. This line of research is also related to our work on *active code completion*, which investigated type-specific code generation interfaces in a textual program editor (Eclipse) [35].

Another interesting direction is that of semantic, interactive documentation. In particular, in Hazel, references to program structures that appear in documentation will be treated in the same way as other references and be subject to renaming and other operations. Documentation will also be capable of containing expressions of arbitrary types (e.g. of the `Image` or `Link` type). Together with the type-specific projection mechanism mentioned above, we hope that this will allow Hazel to function not only as a structured programming environment, but also as a structured document authoring environment! By understanding hyperlinks as variable references (in, perhaps, a different modality [29]), we may be able to blur the line between a module and a webpage.

8 Conclusion

To summarize, there are a number of interesting semantic questions that come up in the design of program editors. We advocate a research program that studies these problems using mathematical tools previously used to study programming languages and complete programs. This work will both demystify the design of program editors and open up the doors for a number of advanced editor services. Ultimately, we envision an intelligent programmer's assistant that combines a deep semantic understanding of incomplete programs with a broad statistical understanding of common idioms to help humans author both programs and documents (as one and the same sort of artifact.)

Acknowledgments. We thank the SNAPL 2017 reviewers and our paper shepherd Nate Foster for the thoughtful comments and suggestions.

References

1 Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.*, 1(4):305–312, 1972. doi:10.1137/0201022.

2 Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In *Software Language Engineering (SLE)*, 2016. doi:10.1145/2997364.2997374.

3 Dimitar Asenov and Peter Müller. Envision: A fast and flexible visual code editor with fluid interactions (Overview). In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014.

4 Edwin Brady. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

5 Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in UI programming. In *PLDI*, 2013. doi:10.1145/2491956.2462170.

6 Margaret M. Burnett, John W. Atwood Jr., and Zachary T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *IEEE Symposium on Visual Languages*, 1998.

7 Philippe Charles. *A practical method for constructing efficient LALR (K) parsers with automatic error recovery.* PhD thesis, New York University, 1991.

8 Paul Chiusano. Unison. http://www.unisonweb.org/. Accessed: 2016-04-25.

9 Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. In *PLDI*, 2016.

10 Matteo Cimini and Jeremy G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *POPL*, 2016.

11 Barthélémy Dagenais and Laurie J. Hendren. Enabling static analysis for partial Java programs. In *OOPSLA*, 2008.

12 Maartje de Jonge, Emma Nilsson-Nyman, Lennart C. L. Kats, and Eelco Visser. Natural and flexible error recovery for generated parsers. In *Software Language Engineering (SLE)*, 2009.

13 Erich Gamma and Kent Beck. *Contributing to Eclipse: principles, patterns, and plug-ins.* Addison-Wesley Professional, 2004.

14 Susan L. Graham, Charles B. Haley, and William N. Joy. Practical lr error recovery. *ACM SIGPLAN Notices*, 14(8), 1979. doi:10.1145/872732.806967.

15 Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP'10, pages 13–24, New York, NY, USA, 2010. ACM. doi:10.1145/1836089.1836091.

16 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

17 Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Software Engineering (ICSE)*, pages 837–847, 2012.

18 James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE)*, pages 467–477, Orlando, FL, USA, 2002. doi:10.1145/581339.581397.

19 Lennart C. L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, and Eelco Visser. Providing rapid feedback in generated modular language environments: adding error recovery to scannerless generalized-LR parsing. In *OOPSLA*, 2009. doi:10.1145/1640089.1640122.

20 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, 2010.

21 Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *International Conference On Automated Software Engineering (ASE)*, 2015.

22 Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38:54–72, 2012. doi:10.1109/TSE.2011.104.

23 Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *POPL*, 2016. doi:10.1145/2837614.2837617.

24 Eyal Lotem and Yair Chuchem. Project Lamdu. http://www.lamdu.org/. Accessed: 2016-04-08.

25 Conor McBride. *Dependently typed functional programs and their proofs.* PhD thesis, University of Edinburgh. College of Science and Engineering. School of Informatics., 2000.

26 Sean McDirmid. Living it up with a live programming language. In *OOPSLA*, 2007.

27 Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, 2013.

28 Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, 2016.

29 Tom Murphy VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *International Symposium on Trustworthy Global Computing*, pages 108–123. Springer, 2007.

30 Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Improving IDE Recommendations by Considering Global Implications of Existing Recommendations. In *New Ideas and Emerging Results Track at the 34th International Conference on Software Engineering (ICSE)*, 2012.

31 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3), 2008.

32 Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

33 Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP*, 2014.

34 Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew Hammer. Hazelnut: A bidirectionally typed structure editor calculus. In *POPL*, 2017. URL: https://arxiv.org/abs/1607.04180.

35 Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active code completion. In *International Conference on Software Engineering (ICSE)*, 2012.

36 Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. URL: http://ipython.org.

37 Brigitte Pientka. Beluga: Programming with dependent types, contextual data, and contexts. In *International Symposium on Functional and Logic Programming (FLOPS)*, 2010. doi:10.1007/978-3-642-12251-4_1.

38 Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, 2013.

39 Manos Renieris and Steven Reiss. Fault localization with nearest neighbor queries. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2003.

40 Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, November 2009. doi:10.1145/1592761.1592779.

41 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

42 Simon Peyton Jones, Sean Leather and Thijs Alkemade. Typed holes in GHC. `https://wiki.haskell.org/GHC/Typed_holes`.

43 Steven L. Tanimoto. VIVA: A visual language for image processing. *J. Vis. Lang. Comput.*, 1(2):127–139, 1990. URL: `http://dx.doi.org/10.1016/S1045-926X(05)80012-6`, `doi:10.1016/S1045-926X(05)80012-6`.

44 Steven L. Tanimoto. A perspective on the evolution of live programming. In *1st International Workshop on Live Programming, (LIVE)*, 2013.

45 Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A Syntax-directed Programming Environment. *Commun. ACM*, 24(9):563–573, 1981. `doi:10.1145/358746.358755`.

46 Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. TouchDevelop: Programming Cloud-connected Mobile Devices via Touchscreen. In *SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2011. `doi:10.1145/2048237.2048245`.

47 Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF meta-environment: A component-based language development environment. In *International Conference on Compiler Construction (CC)*, 2001.

48 Paul Van Der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, 2012.

49 Bret Victor. Inventing on principle. Invited talk, Canadian University Software Engineering Conference (CUSEC), 2012.

50 Markus Voelter. Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 383–430. Springer, 2011.

51 Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. Mbeddr: An Extensible C-based Programming Language and IDE for Embedded Systems. In *SPLASH*, 2012. `doi:10.1145/2384716.2384767`.

52 Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards User-Friendly Projectional Editors. In *Software Language Engineering (SLE)*, 2014. `doi:10.1007/978-3-319-11245-9_3`.

53 Beta Ziliani, Derek Dreyer, Neelakantan R Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming*, 25:e12, 2015.

Linking Types for Multi-Language Software: Have Your Cake and Eat It Too[*]

Daniel Patterson[1] and Amal Ahmed[2]

1 Northeastern University, Boston, MA, USA
dbp@ccs.neu.edu

2 Northeastern University, Boston, MA, USA
amal@ccs.neu.edu

Abstract

Software developers compose systems from components written in many different languages. A business-logic component may be written in Java or OCaml, a resource-intensive component in C or Rust, and a high-assurance component in Coq. In this multi-language world, program execution sends values from one linguistic context to another. This boundary-crossing exposes values to contexts with unforeseen behavior – that is, behavior that could not arise in the source language of the value. For example, a Rust function may end up being applied in an ML context that violates the memory usage policy enforced by Rust's type system. This leads to the question of how developers ought to reason about code in such a multi-language world where behavior inexpressible in one language is easily realized in another.

This paper proposes the novel idea of *linking types* to address the problem of reasoning about single-language components in a multi-lingual setting. Specifically, linking types allow programmers to annotate where in a program they can link with components inexpressible in their unadulterated language. This enables developers to reason about (behavioral) equality using only their own language and the annotations, even though their code may be linked with code written in a language with more expressive power.

NOTE: This paper will be much easier to follow if viewed/printed in color.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Linking, program reasoning, equivalence, expressive power of languages, fully abstract compilation

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.12

1 Reasoning in a Multi-Language World

When building large-scale software systems, programmers should be able to use the best language for each part of the system. Using the "best language" means the language that makes it is easiest for a programmer to *reason* about the behavior of that part of the system. Moreover, programmers should be able to reason *only* in that language when working on that component. That might be Rust for a high-performance component, a terminating domain-specific language for a protocol parser, or a general-purpose scripting language for UI code. In some development shops, domain-specific languages are used in various parts of systems to better separate the logic of particular problems from the plumbing of general-purpose programming. But it's a myth that programmers can reason in a single

[*] This research is supported in part by the NSF (grants CCF-1453796 and CCF-1422133) and a Google Faculty Research Award.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 12; pp. 12:1–12:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

language when dealing with multi-language software. Even if a high-assurance component is written in Coq, the programmer must reason about extraction, compilation, and any linking that happens at the machine-level. An ML component in a multi-language system has contexts that may include features that don't exist in ML. This is a problem for programmers, because as they evolve complex systems, much time is spent refactoring – that is, making changes to components that should result in equivalent behavior. Programmers reason about that equivalence by thinking about possible program contexts within which the original and refactored components could be run, though usually they only think about contexts written in their own language. But if they have linked with another language, the additional contexts from that language also need to be taken into account. Equivalence in all contexts, or *contextual equivalence*, is therefore central to programmer reasoning. Unfortunately, programmers cannot rely upon contextual equivalence of their own language. Instead, since languages interact after having been compiled to a common target, the contextual equivalence that programmers must rely upon is that of the compilation target, which may have little to do with their source language.

For programmers writing components in safe languages like OCaml, the situation is made worse by the fact that the common target is likely a low-level unsafe language like assembly which permits direct access to memory and the call-stack. An object-code linker will verify symbols, but little more. This means that whenever an OCaml programmer links with C code via the FFI, they have to contend with the fact that the C code they write can easily disrupt the equivalences they rely on when reasoning about their OCaml code. Rather than being able to rely upon tooling, the user of a C library must reason carefully about how the C code will interact, at the assembly level, with their OCaml abstractions. For example, an OCaml function that is polymorphic in its arguments could have these arguments inspected by C code it linked with, violating parametricity. On the other side, the C programmer attempting to write a library that can be linked with OCaml must keep all the invariants of OCaml in mind and attempt not to violate any of them. This is difficult and requires reasoning not only about how the OCaml and C languages work but also how they are compiled to assembly, because it is at the assembly level that they interact.

Since programmers use a language for its features and linguistic abstractions, we would like programmers to be able to reason using contextual equivalence for that language, even in the presence of target-level linking. A *fully abstract* compiler enables exactly this reasoning: it guarantees that if two components are contextually equivalent at the source their compiled versions are contextually equivalent at the target. However, this guarantee comes at a steep cost: a fully abstract compiler must disallow linking with components whose behavior is inexpressible in the compiler's source language. But often that extra behavior or control is exactly why the programmer is linking with a component written in another "more expressive" language. An example of additional *behavior* is a non-concurrent language linking with a thread implementation written in C. An example of additional *control* is an unrestricted language linking with a concurrent data structure written in Rust, where linear types ensure data-race freedom.

There are two ways in which a programming language $\mathcal{A}$ can be *more expressive* than another language $\mathcal{B}$, where, following Felleisen [12], we assume both languages have been translated to a common substrate (for us, compiled to a common target), such that $\mathcal{A}$ contexts can be wrapped around $\mathcal{B}$ program fragments:

1. $\mathcal{A}$ has features unavailable in $\mathcal{B}$ that can be used to create contexts that can distinguish components that are contextually equivalent in $\mathcal{B}$. We say that language $\mathcal{A}$ is *positively* more expressive than language $\mathcal{B}$, since the (larger) set of $\mathcal{A}$ contexts have more power to distinguish. For instance, $\mathcal{A}$ may have references or first-class control while $\mathcal{B}$ does not.

2. $\mathcal{A}$ has rich type-system features unavailable in $\mathcal{B}$ that can be used to rule out contexts that, at less precise types, were able to distinguish inequivalent $\mathcal{B}$ components. We say that language $\mathcal{A}$ is *negatively* more expressive than language $\mathcal{B}$, since type restrictions on $\mathcal{A}$ contexts result in a (smaller) set of well-typed $\mathcal{A}$ contexts that have less power to distinguish. For instance, $\mathcal{A}$ may have linear types or polymorphism while $\mathcal{B}$ does not.[1]

The greater expressivity of programming languages explored by Felleisen [12] is what we call *positive* expressivity. As far as we are aware, the notion of *negative* expressivity, presented in this *dual* way, has not appeared in the literature.

Linking with code from more expressive languages affects not just *programmer* reasoning, but also the notion of equivalence used by *compiler writers* to justify correct optimizations. While there has been a lot of recent work on verified compilers, most assume no linking (e.g., [19, 20, 22, 29, 32, 17]), or linking only with code compiled from the same source language [4, 5, 15, 23, 16].

One approach that does support cross-language linking is Compositional Compcert [30], which nonetheless only allows linking with components that satisfy CompCert's memory model. Another approach is the multi-language style of verified compilers by Perconti and Ahmed [28], which allows linking with arbitrary target code that may be compiled from another source language $\mathcal{R}$. This approach, which embeds both the source $\mathcal{S}$ and target $\mathcal{T}$ into a single multi-language $\mathcal{ST}$, means that compiler optimizations can be justified in terms of $\mathcal{ST}$ contextual equivalence. However, as a tool for programmer reasoning, this comes at a significant cost, as the programmer needs to understand the full $\mathcal{ST}$ language and the compiler from $\mathcal{R}$ to $\mathcal{T}$. Moreover, the design of the multi-language fixes what linking should and should not be permitted, a decision that affects the notion of contextual equivalence used to reason about every component written in the source language.

We contend that compiler writers should not get to decide what linking is allowed, and indeed, we don't think they want to. Currently compiler writers are forced to either ignore linking or make such arbitrary decisions because existing source-language specifications are incomplete with respect to linking. Instead, this should be a part of the language specification and exposed to the programmer so that she can make fine-grained decisions about linking, which leads to fine-grained control over what contexts she must consider when reasoning about a particular component. Every compiler should then be fully abstract, which means it preserves the equivalences chosen by the programmer.

We advocate extending source-language specifications with *linking types*, which minimally enrich source-language types and allow programmers to optionally annotate where in their programs they can link with components that would not be expressible in their unadulterated source language. As a specification mechanism, types are familiar, and naturally allow us to change equivalences locally. They fulfill our desire to allow the programmer fine-grained

[1] Example 1: $\mathcal{A}$ has **linear types**.
Consider $\mathcal{B}$ components of type $\texttt{unit} \to \texttt{int}$:

c1 increments a counter on each call and returns it
c2 increments a counter on first call and returns it

And $\mathcal{B}$ distinguishing context:

$$\lambda \texttt{c}.\, \texttt{c}\,(); \texttt{c}\,() \quad : (\texttt{unit} \to \texttt{int}) \to \texttt{int}$$

With $\mathcal{A}$ contexts of type $(\texttt{unit} \to \texttt{int})^{\texttt{L}} \to \texttt{int}$, c1 and c2 are contextually equivalent, since each must be called exactly once.

Example 2: $\mathcal{A}$ has **polymorphism**.
Consider $\mathcal{B}$ components of type $\texttt{unit} \to \texttt{int}$:

$\texttt{p1} = \lambda \texttt{x}.\, 0$
$\texttt{p1} = \lambda \texttt{x}.\, 1$

And $\mathcal{B}$ distinguishing context:

$$\lambda \texttt{f}.\, \texttt{if } \texttt{f}()\texttt{=0} \texttt{ then diverge else } () \quad : (\texttt{unit} \to \texttt{int}) \to \texttt{unit}$$

With $\mathcal{A}$ contexts of type $\forall \alpha.(\texttt{unit} \to \alpha) \to \texttt{unit}$, p1 and p2 are contextually equivalent, since their return values may not be inspected.

control, as they appear on individual terms of the language. A linking-types extension will also often introduce new terms (and operational semantics) intended solely for reasoning about the additional contexts introduced through linking. These new terms are a representative abstraction of potentially complex new behavior from another language that the programmer wants to link with. (This is analogous to how Gu *et al.* [13] lift potentially complex behavior in a lower abstraction layer into a simpler representation in a higher layer.) Now if the programmer reasons about contexts including those terms, she will have considered the behavior of all contexts that her component may be linked with after compilation.

We envision that language designers will provide many different linking-types extensions for their source languages. Programmers can then opt to use zero or more of these extensions, depending on their linking needs.

2 Linking Types, Formally

To formally present the basic idea of linking types, we consider a setting with two simple source languages – see Figure 1 (top) – and show how to design linking types that mediate different interactions between them. Our source languages are λ, the simply typed lambda calculus with integer base types, and λ^{ref}, which extends λ with ML-like mutable references. We want type-preserving, fully abstract compilers from these source languages to a common target language. That target should have a rich enough type system so that the compiler's type translation can ensure full abstraction by using types to rule out linking with target contexts whose behavior is inexpressible in the source. Here we illustrate the idea with a fairly high-level target language $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$ – see Figure 1 (bottom) – that includes mutable references and exceptions and has a modal type system that can distinguish pure computations from those that either use references or raise exceptions.[2] We include exceptions in the target as a representative of the extra control flow often present in low-level targets (e.g., direct jumps). An impure target computation $\mathtt{E}^{\bullet}_{\tau_{\mathtt{exc}}}\,\tau$ (pronounced "impure exception-raising tau computation") may access the heap while computing a value of type τ or raising an exception of type $\tau_{\mathtt{exc}}$. In contrast, a pure computation $\mathtt{E}^{\circ}_{0}\,\tau$ (pronounced "pure tau computation") may not access the heap, and cannot raise exceptions as the exception type is the void (uninhabited) type $\mathtt{0}$.

Consider the scenario where the programmer writes code in λ and wants to link with code written in λ^{ref}. Assume this linking happens after both λ and λ^{ref} have been compiled using *fully abstract* compilers to $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$. We illustrate this with concrete example programs $\mathbf{e_1}$ and $\mathbf{e_2}$ which are equivalent in λ. Now consider the context $\mathsf{C}^{\mathsf{ref}}$ which implements a simple counter using a reference cell. The λ compiler, since it is fully abstract, would have to disallow linking with $\mathsf{C}^{\mathsf{ref}}$ since it can distinguish $\mathbf{e_1}$ from $\mathbf{e_2}$. More generally, in order to rule out this class of equivalence-disrupting contexts, the fully abstract compiler would have to prevent linking with any code that has externally visible effects.[3] This can be accomplished by a type-directed compiler that sends all λ arrows $\tau_1 \rightarrow \tau_2$ to pure functions $\tau_1' \rightarrow \mathtt{E}^{\circ}_{0}\,\tau_2'$, where τ_1' and τ_2' are the translations of types τ_1 and τ_2. This would rule out linking with contexts with heap effects like $\mathsf{C}^{\mathsf{ref}}$. But in this case, the programmer wants to link these together and is willing to lose some equivalences in order to do so.

[2] We use a modal type system here, but any type-and-effect system would suffice.

[3] For simplicity, the type system we show here doesn't support effect masking, so we rule out linking with all effectful code. More realistic target languages, e.g., based on Koka [18], would support linking with code without externally visible effects.

$$
\begin{array}{llll}
\lambda & \tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \tau \to \tau \\
 & \mathbf{e} & ::= & () \mid \mathbf{n} \mid \mathbf{x} \mid \lambda \mathbf{x} : \tau .\, \mathbf{e} \mid \mathbf{e}\,\mathbf{e} \\
 & & & \mathbf{e} + \mathbf{e} \mid \mathbf{e} * \mathbf{e} \mid \mathbf{e} - \mathbf{e} \\
 & \mathbf{v} & ::= & () \mid \mathbf{n} \mid \lambda \mathbf{x} : \tau .\, \mathbf{e}
\end{array}
\qquad
\begin{array}{llll}
\lambda^{\mathsf{ref}} & \tau & ::= & \ldots \mid \mathsf{ref}\, \tau \\
 & \mathsf{e} & ::= & \ldots \mid \mathsf{ref}\, \mathsf{e} \mid \mathsf{e} := \mathsf{e} \mid !\mathsf{e} \\
 & \mathsf{v} & ::= & \ldots \mid \ell
\end{array}
$$

$$
\begin{array}{llll}
\lambda^{\mathtt{ref}}_{\mathtt{exc}} & \tau & ::= & \mathtt{0} \mid \mathtt{unit} \mid \mathtt{int} \mid \mathtt{ref}\, \tau \mid \tau \to \mathtt{E}^{\epsilon}_{\tau_{\mathtt{exc}}}\, \tau \\
 & \epsilon & ::= & \bullet \mid \circ \\
 & \mathtt{e} & ::= & () \mid \mathtt{n} \mid \mathtt{x} \mid \lambda \mathtt{x} : \tau .\, \mathtt{e} \mid \mathtt{e}\,\mathtt{e} \mid \mathtt{e} + \mathtt{e} \mid \mathtt{e} * \mathtt{e} \mid \mathtt{e} - \mathtt{e} \mid \mathtt{throw}\, \mathtt{e} \\
 & & & \mathtt{catch}\, \mathtt{e}\, \mathtt{with}\, \mathtt{val}\, \mathtt{x} \Rightarrow \mathtt{e} ;\, \mathtt{exc}\, \mathtt{y} \Rightarrow \mathtt{e} \mid \mathtt{ref}\, \mathtt{e} \mid \mathtt{e} := \mathtt{e} \mid !\mathtt{e} \\
 & \mathtt{v} & ::= & () \mid \mathtt{n} \mid \lambda \mathtt{x} : \tau .\, \mathtt{e} \mid \ell
\end{array}
$$

$$
\boxed{\Gamma \vdash \mathtt{v} : \tau}
\qquad
\frac{}{\Gamma \vdash () : \mathtt{unit}}
\qquad
\frac{\Gamma, \mathtt{x} \colon \tau \vdash \mathtt{e} \colon \mathtt{E}^{\rho}_{\tau_{\mathtt{exn}}}\, \tau'}{\Gamma \vdash \lambda\, \mathtt{x} \colon \tau . \mathtt{e} \colon \tau \to \mathtt{E}^{\rho}_{\tau_{\mathtt{exn}}}\, \tau'}
$$

$$
\boxed{\Gamma \vdash \mathtt{e} \colon \mathtt{E}^{\epsilon}_{\tau_{\mathtt{exn}}}\, \tau}
\qquad
\frac{\Gamma \vdash \mathtt{v} \colon \tau}{\Gamma \vdash \mathtt{v} \colon \mathtt{E}^{\circ}_{0}\, \tau}
\qquad
\frac{\Gamma \vdash \mathtt{e}_1 \colon \mathtt{E}^{\rho_1}_{\tau_{\mathtt{exn}}} (\tau \to \mathtt{E}^{\rho_3}_{\tau_{\mathtt{exn}}}\, \tau') \qquad \Gamma \vdash \mathtt{e}_2 \colon \mathtt{E}^{\rho_2}_{\tau_{\mathtt{exn}}}\, \tau}{\Gamma \vdash \mathtt{e}_1\, \mathtt{e}_2 \colon \mathtt{E}^{\rho_1 \vee \rho_2 \vee \rho_3}_{\tau_{\mathtt{exn}}}\, \tau'}
\qquad
\frac{\Gamma \vdash \mathtt{e} \colon \mathtt{E}^{\rho}_{\tau_{\mathtt{exn}}}\, \tau \qquad \vdash \tau}{\Gamma \vdash \mathtt{ref}\, \mathtt{e} \colon \mathtt{E}^{\bullet}_{\tau_{\mathtt{exn}}}\, \mathtt{ref} \tau}
$$

$$
\frac{\Gamma \vdash \mathtt{e}_1 \colon \mathtt{E}^{\rho_1}_{\tau_{\mathtt{exn}}}\, \mathtt{ref}\, \tau \qquad \Gamma \vdash \mathtt{e}_1 \colon \mathtt{E}^{\rho_2}_{\tau_{\mathtt{exn}}}\, \tau}{\Gamma \vdash \mathtt{e}_1 := \mathtt{e}_2 \colon \mathtt{E}^{\bullet}_{\tau_{\mathtt{exn}}}\, \mathtt{unit}}
\qquad
\frac{\Gamma \vdash \mathtt{e} \colon \mathtt{E}^{\rho}_{\tau_{\mathtt{exn}}}\, \mathtt{ref}\, \tau}{\Gamma \vdash !\mathtt{e}_1 \colon \mathtt{E}^{\bullet}_{\tau_{\mathtt{exn}}}\, \tau}
$$

$$
\frac{\Gamma \vdash \mathtt{e} \colon \mathtt{E}^{\rho}_{\tau_{\mathtt{exn}}}\, \tau \qquad \Gamma, \mathtt{x} \colon \tau \vdash \mathtt{e}_2 \colon \mathtt{E}^{\rho_2}_{\tau'_{\mathtt{exn}}}\, \tau' \qquad \Gamma, \mathtt{y} \colon \tau_{\mathtt{exn}} \vdash \mathtt{e}_1 \colon \mathtt{E}^{\rho_1}_{\tau'_{\mathtt{exn}}}\, \tau'}{\Gamma \vdash \mathtt{catch}\, \mathtt{e}\, \mathtt{with}\, \mathtt{val}\, \mathtt{x} \Rightarrow \mathtt{e}_1 ;\, \mathtt{exc}\, \mathtt{y} \Rightarrow \mathtt{e}_2 \colon \mathtt{E}^{\rho_1 \vee \rho_2}_{\tau'_{\mathtt{exn}}}\, \tau'}
\qquad
\frac{\Gamma \vdash \mathtt{e} \colon \tau_{\mathtt{exn}} \qquad \vdash \tau}{\Gamma \vdash \mathtt{throw}\, \mathtt{e} \colon \mathtt{E}^{\circ}_{\tau_{\mathtt{exn}}}\, \tau}
$$

Figure 1 λ and λ^{ref} syntax (top), $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$ syntax and selected static semantics (bottom).

$$
\begin{array}{l}
\mathbf{e_1} = \lambda \mathbf{c}.\, \mathbf{c}() \\
\mathbf{e_2} = \lambda \mathbf{c}.\, \mathbf{c}(); \mathbf{c}() \\
\\
\forall \mathbf{C}^{\lambda} . \mathbf{C}^{\lambda}[\mathbf{e_1}] \approx_{\lambda} \mathbf{C}^{\lambda}[\mathbf{e_2}]
\end{array}
\qquad
\begin{array}{lll}
\mathsf{C}^{\mathsf{ref}} & = & \mathsf{let}\, \mathsf{x} = \mathsf{ref}\, 0\, \mathsf{in} \\
 & & \mathsf{let}\, \mathsf{c}'\, () = \mathsf{x} := !\mathsf{x} + 1; !\mathsf{x}\, \mathsf{in}\, [\cdot]\mathsf{c}' \\
\mathsf{C}^{\mathsf{ref}}[\mathbf{e_1}] \Downarrow 1 & & \\
\mathsf{C}^{\mathsf{ref}}[\mathbf{e_2}] \Downarrow 2 & &
\end{array}
$$

To enable the above linking, we present a linking-types extension for λ that includes both an extended language λ^{κ} and functions κ^{+} and κ^{-} that relate types of λ and λ^{κ}. The λ^{κ} type system includes reference types and tracks heap effects. We need to track heap effects to be able to reason about the interaction between the pure λ code and impure λ^{ref} code that it will be linked with. This extension is shown on the left in Figure 2. The parts of λ^{κ} that extend λ are typeset in **magenta**, whereas terms that originated in λ are **orange**. λ^{κ} types τ include base types **unit** and **int**, reference types $\mathbf{ref}\,\tau$, and a computation type $\mathbf{R}^{\epsilon}\, \tau$, analogous to the target computation type $\mathtt{E}^{\epsilon}_{\tau_{\mathtt{exn}}}\, \tau$, but without tracking exception effects. λ^{κ} terms $\mathbf{e}$ include terms from λ, as well as terms for allocating, reading, and updating references.

With this extension, we annotate $\mathbf{e_1}$ and $\mathbf{e_2}$ with a linking type that specifies that the input can be heap-effecting: $\lambda \mathbf{c}.\, \mathbf{c}() \not\approx^{ctx}_{\lambda^{\kappa}} \lambda \mathbf{c}.\, \mathbf{c}(); \mathbf{c}() : (\mathbf{unit} \to \mathbf{R}^{\bullet}\, \mathbf{int}) \to \mathbf{R}^{\bullet}\, \mathbf{int}$. At this type, $\mathbf{e_1}$ and $\mathbf{e_2}$ are no longer contextually equivalent and, further, can be linked with the counter library.

Without the above annotation, the compiler would translate the type of $\lambda \mathbf{c}.\, \mathbf{c}()$ or $\lambda \mathbf{c}.\, \mathbf{c}(); \mathbf{c}()$ from the λ type $\mathbf{unit} \to \mathbf{int}$ to the $\lambda^{\mathtt{ref}}_{\mathtt{exn}}$ type $\mathtt{unit} \to \mathtt{E}^{\circ}_{0}\, \mathtt{int}$, and the type expected by the counter from the λ^{ref} type $\mathsf{unit} \to \mathsf{int}$ to the $\lambda^{\mathtt{ref}}_{\mathtt{exn}}$ type $\mathtt{unit} \to \mathtt{E}^{\bullet}_{0}\, \mathtt{int}$. Since these are not the same, an error would be reported: that $\mathbf{unit} \to \mathbf{int}$ is not compatible with $\mathsf{unit} \to \mathsf{int}$.

$$
\begin{array}{llll}
\lambda^{\kappa} & \tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\ \tau \mid \tau \to \mathbf{R}^{\epsilon}\ \tau \\
 & \mathbf{e} & ::= & () \mid \mathbf{n} \mid \mathbf{x} \mid \lambda \mathbf{x} : \tau.\ \mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid \mathbf{e} + \mathbf{e} \\
 & & & \mid \mathbf{e} * \mathbf{e} \mid \mathbf{e} - \mathbf{e} \mid \mathbf{ref}\ \mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid !\mathbf{e} \\
 & \mathbf{v} & ::= & () \mid \mathbf{n} \mid \lambda \mathbf{x} : \tau.\ \mathbf{e} \mid \ell \\
 & \epsilon & ::= & \bullet \mid \circ
\end{array}
\qquad
\begin{array}{llll}
\lambda^{\mathsf{ref}\kappa} & \tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\ \tau \mid \tau \to \mathbf{R}^{\epsilon}\ \tau \\
 & \mathbf{e} & ::= & () \mid \mathsf{n} \mid \mathsf{x} \mid \lambda \mathsf{x} : \tau.\ \mathbf{e} \mid \mathbf{e}\ \mathbf{e} \mid \mathbf{e} + \mathbf{e} \\
 & & & \mid \mathbf{e} * \mathbf{e} \mid \mathbf{e} - \mathbf{e} \mid \mathsf{ref}\ \mathsf{e} \mid \mathsf{e} := \mathsf{e} \mid !\mathsf{e} \\
 & \mathbf{v} & ::= & () \mid \mathsf{n} \mid \lambda \mathsf{x} : \tau.\ \mathbf{e} \mid \ell \\
 & \epsilon & ::= & \bullet \mid \circ
\end{array}
$$

$$
\begin{array}{lcl}
\kappa^{+}(\mathbf{unit}) & = & \mathbf{unit} \\
\kappa^{+}(\mathbf{int}) & = & \mathbf{int} \\
\kappa^{+}(\tau_1 \to \tau_2) & = & \kappa^{+}(\tau_1) \to \mathbf{R}^{\circ}\ \kappa^{+}(\tau_2) \\
\\
\kappa^{-}(\mathbf{unit}) & = & \mathbf{unit} \\
\kappa^{-}(\mathbf{int}) & = & \mathbf{int} \\
\kappa^{-}(\mathbf{ref}\ \tau) & = & \kappa^{-}(\tau) \\
\kappa^{-}(\tau_1 \to \mathbf{R}^{\epsilon}\ \tau_2) & = & \kappa^{-}(\tau_1) \to \kappa^{-}(\tau_2)
\end{array}
\qquad
\begin{array}{lcl}
\kappa^{+}(\mathsf{unit}) & = & \mathbf{unit} \\
\kappa^{+}(\mathsf{int}) & = & \mathbf{int} \\
\kappa^{+}(\mathsf{ref}\ \tau) & = & \mathbf{ref}\ \kappa^{+}(\tau) \\
\kappa^{+}(\tau_1 \to \tau_2) & = & \kappa^{+}(\tau_1) \to \mathbf{R}^{\bullet}\ \kappa^{+}(\tau_2) \\
\kappa^{-}(\mathbf{unit}) & = & \mathsf{unit} \\
\kappa^{-}(\mathbf{int}) & = & \mathsf{int} \\
\kappa^{-}(\mathbf{ref}\ \tau) & = & \mathsf{ref}\ \kappa^{-}(\tau) \\
\kappa^{-}(\tau_1 \to \mathbf{R}^{\epsilon}\ \tau_2) & = & \kappa^{-}(\tau_1) \to \kappa^{-}(\tau_2)
\end{array}
$$

Figure 2 Linking-types extension of λ and λ^{ref}.

This error matches our intuition – that an arrow means something fundamentally different in a pure language and one that has heap effects. For advanced users, the compiler could explain the type translations that gave rise to that incompatibility. By contrast, with the type annotation $\mathbf{unit} \to \mathbf{R}^{\bullet}\ \mathbf{int}$ both types translate to the same $\lambda^{\mathtt{ref}}_{\mathtt{exn}}$ type $\mathtt{unit} \to \mathrm{E}^{\bullet}_{0}\mathtt{int}$ and thus no error will be raised.

With the linking-types-extended language, note that the additional *terms* are intended only for reasoning, so that programmers can understand the kind of behavior that they are linking with; they should not show up in code written by the programmer. If we allowed programmers to use these terms in their code, we would be changing the programming language itself, whereas linking types should only allow a programmer to change equivalences of their existing language. Our focus is *linking*, not general language extension. The last part of the linking-types extension is the pair of functions κ^{+}, for embedding λ types in λ^{κ}, and κ^{-} for projecting λ^{κ} types to λ types. We will discuss the properties that κ^{+} and κ^{-} must satisfy below.

Also shown in Figure 2 is a linking-types extension of λ^{ref} that allows λ^{ref} to distinguish program fragments that are free of heap effects and can then safely be passed to linked λ code. This results in essentially the same extended language λ^{κ}; the only changes are the arrow and reference cases of κ^{+} and κ^{-} and in terms that should be written by programmers.

We can now develop fully abstract compilers from λ^{κ} and $\lambda^{\mathsf{ref}\kappa}$ – rather than λ and λ^{ref} – to $\lambda^{\mathtt{ref}}_{\mathtt{exc}}$ using the following type translation to ensure full abstraction:

$$
\begin{array}{ll}
\langle\!\langle \mathbf{unit} \rangle\!\rangle & = \mathtt{unit} \\
\langle\!\langle \mathbf{int} \rangle\!\rangle & = \mathtt{int} \\
\langle\!\langle \mathbf{ref}\ \tau \rangle\!\rangle & = \mathtt{ref}\ \langle\!\langle \tau \rangle\!\rangle \\
\langle\!\langle \tau_1 \to \mathbf{R}^{\epsilon}\ \tau_2 \rangle\!\rangle & = \langle\!\langle \tau_1 \rangle\!\rangle \to \mathrm{E}^{\epsilon}_{0}\ \langle\!\langle \tau_2 \rangle\!\rangle
\end{array}
$$

2.1 Properties of Linking Types

For any source language $\lambda_{\mathtt{src}}$, an extended language $\lambda^{\kappa}_{\mathtt{src}}$ paired with κ^{+} and κ^{-} is a linking-types extension if the following properties hold:

- $\lambda_{\mathtt{src}}$ terms are a subset of $\lambda^{\kappa}_{\mathtt{src}}$ terms.
- $\lambda_{\mathtt{src}}$ type τ embeds into a $\lambda^{\kappa}_{\mathtt{src}}$ type by $\kappa^{+}(\tau)$.

- $\lambda^{\kappa}_{\texttt{src}}$ type τ^{κ} projects to a $\lambda_{\texttt{src}}$ type by $\kappa^{-}(\tau^{\kappa})$.
- For any $\lambda_{\texttt{src}}$ type τ, $\kappa^{-}(\kappa^{+}(\tau)) = \tau$.
- κ^{+} preserves and reflects equivalence:
  $\forall \texttt{e}_1, \texttt{e}_2 \in \lambda_{\texttt{src}}.\ \texttt{e}_1 \approx^{ctx}_{\lambda_{\texttt{src}}} \texttt{e}_2 : \tau \iff \texttt{e}_1 \approx^{ctx}_{\lambda^{\kappa}_{\texttt{src}}} \texttt{e}_2 : \kappa^{+}(\tau)$.
- $\forall \texttt{e}, \tau.\ \texttt{e} : \tau \implies \texttt{e} : \kappa^{-}(\tau)$ when $\texttt{e}$ only contains $\lambda_{\texttt{src}}$ terms.
- A compiler for $\lambda^{\kappa}_{\texttt{src}}$ should be fully abstract, but it need only compile terms from $\lambda_{\texttt{src}}$.

Reasoning about contextual equivalence means reasoning about the equivalence classes that contain programs. Thus we can understand the effect of linking types, and of the properties that guide them, by studying how the extensions affect equivalence classes. In Figure 3, we present three programs (A, B, and C) valid in both λ and λ^{ref}. At the type $(\mathbf{int} \to \mathbf{int}) \to \mathbf{int}$, all three programs are equivalent in λ, which we illustrate by putting $\texttt{A}, \texttt{B}, \texttt{C}$ in a single equivalence box. In λ, all functions terminate, which means that calling the argument f zero, one, or two times before discarding the result is equivalent. However, in λ^{ref}, A, B, and C are all in different equivalence classes, since f may increment a counter, which means a context could detect the number of times it was called.

The top of the diagram shows equivalence classes for $\lambda^{\kappa}/\lambda^{\mathsf{ref}\kappa}$. Here we can see how equivalences can be changed by annotating these functions with different linking types. Note that equivalence is only defined at a given type, so we only consider when all three functions have been given the same linking type.

At the type $(\mathbf{int} \to \mathbf{R}^{\circ}\mathbf{int}) \to \mathbf{R}^{\circ}\mathbf{int}$ these programs are all equivalent since this linking type requires that f be pure. At the type $(\mathbf{int} \to \mathbf{R}^{\bullet}\mathbf{int}) \to \mathbf{R}^{\bullet}\mathbf{int}$ all three programs are in different equivalence classes, because the linking type allows f to be impure, which could be used by a context to distinguish the programs. At the type $(\mathbf{int} \to \mathbf{R}^{\circ}\mathbf{int}) \to \mathbf{R}^{\bullet}\mathbf{int}$ all three programs are again equivalent. While the type allows the body to be impure, since the argument f is pure, no difference can be detected. The last linking type $(\mathbf{int} \to \mathbf{R}^{\bullet}\mathbf{int}) \to \mathbf{R}^{\circ}\mathbf{int}$ is a type that can only be assigned to the program A, because if the argument f is impure but the result is pure the program could not have called f.

We can see here that κ^{+} is the "default" embedding, which has the important property that it preserves equivalence classes from the original language. Notice that κ^{+} for λ and λ^{ref} both do this, and send the respective source $(\texttt{int} \to \texttt{int}) \to \texttt{int}$ to different types.

3 Additional Applications of Linking Types

This section contains examples of languages we would like to be able to link with λ^{ref} but which contain features that require we either rule out such linking or give up on programmers being able to reason in their source language (without linking types). We consider idealized languages here – and indeed, we believe that programmers would benefit from smaller, more special-purpose languages in a software project – but the ideas carry through to full languages with different expressivity.

3.1 Linearity in Libraries

Substructural type systems are particularly useful for modeling resources and for reasoning about where a resource must be used or when consuming a resource should render it unusable to others. Simple examples include network sockets and file handles, where opening creates the resource, reading consumes the resource and possibly creates a new one, and closing consumes the resource. An ML programmer may want to use libraries written in a linear or affine language (such as Rust) to ensure safe resource handling. But if the language with

$$\begin{array}{ll}
\texttt{program}\,\mathtt{A} & \lambda\mathtt{f} : \mathtt{int} \to \mathtt{int}.\, \mathtt{1} \\
\texttt{program}\,\mathtt{B} & \lambda\mathtt{f} : \mathtt{int} \to \mathtt{int}.\, \mathtt{f}\ \mathtt{0};\ \mathtt{1} \\
\texttt{program}\,\mathtt{C} & \lambda\mathtt{f} : \mathtt{int} \to \mathtt{int}.\, \mathtt{f}\ \mathtt{0};\ \mathtt{f}\ \mathtt{0};\ \mathtt{1}
\end{array}$$

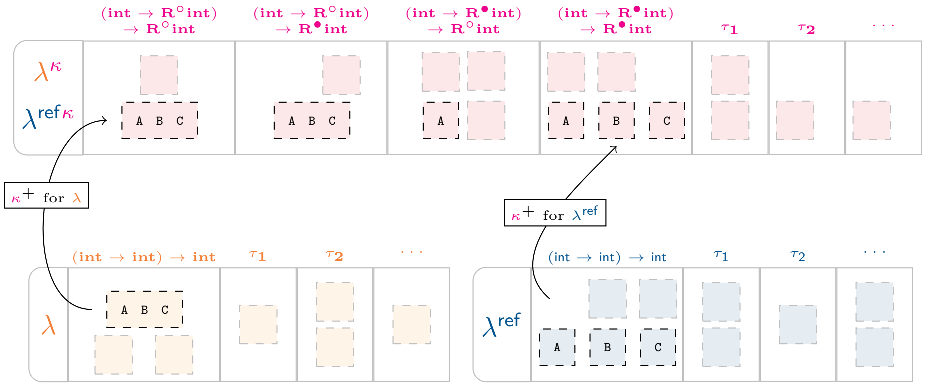


Figure 3 Equivalence classes when giving different linking types to programs.

linear or affine types allows values to cross the linking boundary, ML needs to respect the linear or affine invariants to ensure soundness. For instance, if an ML component passes a value as affine to a Rust component but retains a pointer to the value and later tries to use it after it was consumed (in Rust), it violates the affine invariant that every resource may be used at most once, making the program crash. Similarly, if an ML component never consumes a linear value, it violates the linear invariant that every resouce must be used exactly once, resulting in a resource leak.

A fully abstract compiler would prevent linking in the above scenarios since two components that are equivalent in a linear/affine language can easily be distinguished by a context that does not respect linear/affine invariants. For instance, an affine function that consumes its affine input and one that does not are equivalent if the context cannot later try to consume the same input.

We can use linking types to give non-linear languages access to libraries with linear APIs. Specifically, we would extend the types of our non-linear source language λ^{ref} as follows:

$$\begin{array}{lcl}
\phi & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\,\tau \mid \tau \to \tau \\
\tau & ::= & \phi \mid \phi^{\mathbf{L}}
\end{array}
\qquad
\begin{array}{lcl}
\kappa^{-}(\phi) & = & \kappa^{\mathbf{L}-}(\phi) \\
\kappa^{-}(\phi^{\mathbf{L}}) & = & \kappa^{\mathbf{L}-}(\phi)
\end{array}$$

$$\begin{array}{lcl}
\kappa^{+}(\mathsf{unit}) & = & \mathbf{unit} \\
\kappa^{+}(\mathsf{int}) & = & \mathbf{int} \\
\kappa^{+}(\mathsf{ref}\,\tau) & = & \mathbf{ref}\,\kappa^{+}(\tau) \\
\kappa^{+}(\tau_1 \to \tau_2) & = & \kappa^{+}(\tau_1) \!\to \kappa^{+}(\tau_2)
\end{array}
\qquad
\begin{array}{lcl}
\kappa^{\mathbf{L}-}(\mathbf{unit}) & = & \mathsf{unit} \\
\kappa^{\mathbf{L}-}(\mathbf{int}) & = & \mathsf{int} \\
\kappa^{\mathbf{L}-}(\mathbf{ref}\,\tau) & = & \mathsf{ref}\,\kappa^{-}(\tau) \\
\kappa^{\mathbf{L}-}(\tau_1 \to \tau_2) & = & \kappa^{-}(\tau_1) \!\to\! \kappa^{-}(\tau_2)
\end{array}$$

Note that the target of compilation would either need to support linear types or enforce linearity at runtime – e.g., via contracts à la Tov and Pucella [31].

3.2 Terminating Protocol Parsers

For certain programming tasks, every program should terminate – for instance, HTTP protocol parsing should never end up in an infinite loop. A programmer could implement such tasks using a special-purpose language in which divergence is impossible. We still, however, need to link such terminating languages with general-purpose languages – while the protocol parser should always terminate, the server where it lives better not!

A fully abstract compiler would have to prevent such linking, since two components that are equivalent in a terminating language can easily be distinguished by a context with nontermination. For instance, a function that calls its argument and discards the result, and one that ignores its argument are equivalent if the context provides only terminating functions as arguments, but not if the context provides a function that diverges when called.

We can use linking types to allow terminating and nonterminating languages to interact. Concretely, we can extend the types of our nonterminating language λ^{ref} with a terminating function type, written $\tau \rightarrow \tau\downharpoonleft$. The extension is as follows, but we elide cases of κ^{+} and κ^{-} that are the same as in Figure 2:

$$
\begin{array}{lcl}
\tau & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\,\tau \mid \tau \rightarrow \tau\downharpoonleft \mid \tau \rightarrow \tau \\
\kappa^{+}(\tau_1 \rightarrow \tau_2) & = & \kappa^{+}(\tau_1) \rightarrow \kappa^{+}(\tau_2)
\end{array}
\qquad
\begin{array}{lcl}
\kappa^{-}(\boldsymbol{\tau_1} \rightarrow \boldsymbol{\tau_2}\downharpoonleft) & = & \kappa^{-}(\boldsymbol{\tau_1}) \rightarrow \kappa^{-}(\boldsymbol{\tau_2}) \\
\kappa^{-}(\boldsymbol{\tau_1} \rightarrow \boldsymbol{\tau_2}) & = & \kappa^{-}(\boldsymbol{\tau_1}) \rightarrow \kappa^{-}(\boldsymbol{\tau_2})
\end{array}
$$

The typing rules (elided) would likely need to rely on some syntactic termination check for functions ascribed the terminating arrow type. We could also imagine making the terminating arrow rely on a runtime timeout. The latter would require a new application typing rule to reflect that sometimes applying a terminating function might return a nonce value indicating that computation was cut off, and our language would need to be trivially extended with sum types to handle that possibility in programs.

3.3 Surfacing Cost of Computation

Some security vulnerabilities rely on the fact that the cost of a computation may be discernable (e.g., by observing time, or CPU or memory consumption). To prove the absence of such vulnerabilities, we could remove the mechanism of observation – but this is likely impossible, since even if we remove timing from our language, if the program communicates over the network timing can happen on other systems. A more promising strategy is to introduce the notion of cost (time or space) into the model and then prove that various branches are indistinguishable in that model (see, e.g. [6], [14], [9]). Nonetheless, one would not want to have to write non-security-sensitive parts of programs in one of these cost-aware languages. This motivates a linking-types extension of a non-cost-aware language – in this case again our idealized λ^{ref} – with a notion of computations with cost. As before, we only show differences from Figure 2:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\tau ::= \mathbf{unit} \mid \mathbf{int} \mid \mathbf{ref}\,\tau \mid \tau \rightarrow \mathbf{C}^{\bullet}\tau \mid \tau \rightarrow \mathbf{C}^{\mathbf{N}}\tau} \\
\kappa^{+}(\tau_1 \rightarrow \tau_2) & = & \kappa^{+}(\tau_1) \rightarrow \mathbf{C}^{\bullet}\kappa^{+}(\tau_2)
\end{array}
\qquad
\begin{array}{lcl}
\kappa^{-}(\boldsymbol{\tau_1} \rightarrow \mathbf{C}^{\bullet}\boldsymbol{\tau_2}) & = & \kappa^{-}(\boldsymbol{\tau_1}) \rightarrow \kappa^{-}(\boldsymbol{\tau_2}) \\
\kappa^{-}(\boldsymbol{\tau_1} \rightarrow \mathbf{C}^{\mathbf{N}}\boldsymbol{\tau_2}) & = & \kappa^{-}(\boldsymbol{\tau_1}) \rightarrow \kappa^{-}(\boldsymbol{\tau_2})
\end{array}
$$

The type system is modal – computations $\mathbf{C}^{\mathbf{N}}\tau$ have a known cost $\mathbf{N}$, and computations $\mathbf{C}^{\bullet}\tau$ have an unknown cost. Fully abstract compilation from a cost-aware language and the above extended language would only allow known-cost computations to be passed to the cost-aware language. As before, this relies upon the target language supporting a type system that is at least as expressive, such that it can safely separate the known-cost and unknown-cost modalities.

This application of linking types echos the work by D'Silva *et al.* [11], which discusses enriching the model in which properties are investigated to encompass side channels like timing. While their work investigates machine models, ours relies upon type systems in the language where linking takes place. Further, D'Silva *et al.* envision programmers would *opt in* to security properties via annotations that would change how the compiler treated a piece of code, whereas we envision that the compiler would preserve source equivalences by default and programmers would have to *opt out* of the default fully abstract compilation by using linking types. We believe that our approach can be used with other side channels as well, provided sufficient mechanisms exist to distinguish computations that might reveal information from those that cannot.

3.4 Gradual Typing

As we have already shown, linking types are useful when linking more precisely and less precisely typed languages. Taken to an extreme, we can add linking types to a un(i)typed language to facilitate sound linking with a statically typed language. We can do this by starting with a language with a single type, the dynamic type, and then constructing an extension that adds further types. A typed target language would then allow code compiled from a different, typed, source language to be linked with this gradually typed language. A fully abstract compiler for the extended language would have to make use of run-time checks at the boundaries between typed and untyped code, analogous to sound gradual typing.

4 Bringing Linking Types to Your Language

To understand linking types and the way they interact with existing languages, we consider an example of how a language designer would incorporate them and discuss their usefulness and viability (à la Cardelli [8]).

Day 1: Fully abstract compiler. As a first step, the language designer implements and proves fully abstract a type-directed compiler for her language $\mathcal{A}$. To make it more concrete, you can consider $\mathcal{A}$ to be the language λ from earlier in the paper, but this scenario is general – you could equally consider $\mathcal{A}$ to be a language like OCaml. The compiler targets a typed low-level intermediate language $\mathcal{L}$, using an appropriate type translation to guarantee that equivalences are preserved. This, concretely, could be a target like $\lambda^{\texttt{ref}}_{\texttt{exn}}$, but could also be a richly-typed version of LLVM. All linking should occur in $\mathcal{L}$, which means the subsequent passes, to LLVM, assembly, or another target, need not be fully abstract.

Discussion

- Full abstraction is a key part of linking types, as it is required to preserve the equivalences that programmers rely upon for reasoning.
- The representative terms added to the linking-types-extended language are used in the proof of full abstraction, which essentially requires showing that target contexts can be back-translated to equivalent source contexts.
- While we use static types in our target to ensure full abstraction – and gain tooling benefits from it (explored in Day 3) – we can also use dynamic checks when appropriate (e.g. [25, 10]).
- We are currently designing a language like $\mathcal{L}$, which we expect to be similar to a much more richly typed version of LLVM, such that types could be erased and existing LLVM code-generation infrastructure could be used (as discussed by Ahmed at SNAPL'15 [1]).

Day 2: Linking with more expressive code. $\mathcal{A}$ programmers are happy using the above compiler since they can reason in terms of $\mathcal{A}$ semantics, even when using libraries directly implemented in $\mathcal{L}$ or compiled from other languages. But, soon the language designer's users ask to link their code with a $\mathcal{B}$ language library with features in $\mathcal{L}$ but not in $\mathcal{A}$, something that the fully abstract compiler currently prevents. In the example used earlier in the paper,

$\mathcal{B}$ would be λ^{ref}, and the additional feature would be mutable references, but again, this is a general process that could apply to other features.

The compiler writer introduces a linking-types extension to capture the inexpressible features for her $\mathcal{A}$ programmers. She implements a type checker for the fully elaborated linking types and extends her fully abstract compiler to handle the extended $\mathcal{A}^{\kappa}$ types.

Discussion

- While the linking types will in general be a new type system, no impact is seen on type inference, because linking types are never inferred: first the program will have source types inferred, and then all source types will be lifted to the linking types, using the programmer-specified annotations where present and the default κ^{+} embedding where annotations are absent.

Day 3: When can components in two languages be linked? Happily able to link with other languages, the $\mathcal{A}$ programmer uses the tooling associated with the $\mathcal{A}$ and $\mathcal{B}$ compilers to determine when a $\mathcal{B}$ component can be used at a linking point. The tool uses the compiler to translate the $\mathcal{B}$ component's type $\tau_{\mathcal{B}}$ to an $\mathcal{L}$ type $\tau_{\mathcal{L}}$ and then attempts to back-translate $\tau_{\mathcal{L}}$ to an $\mathcal{A}$ type $\tau_{\mathcal{A}}$ by inverting the $\mathcal{A}$ compiler's type translation. Should this succeed, the component can be used at the type $\tau_{\mathcal{A}}$. This functionality allows the programmer to easily work on components in both $\mathcal{A}$ and $\mathcal{B}$ at once while getting *cross-language type errors* if the interfaces do not match. In the example used earlier in the paper, such a type error showed up when trying to link the λ^{ref} counter library with the λ client that had not been annotated.

Discussion

- This functionality depends critically on the type-directed nature of our compilers and the presence of types in the low-level intermediate language $\mathcal{L}$, where the types become the *medium* through which we can provide useful static feedback to the programmer.
- While linking these components together relies upon shared calling conventions, this is true of any linking. Currently, cross-language linking often relies upon C calling conventions.

Day 4: Backwards compatibility for programmers. At the same time, another programmer continues to use $\mathcal{A}$, unaware of the $\mathcal{A}^{\kappa}$ extension introduced in Day 2, since linking types are *optional* annotations. At lunch, she learns about linking types and realizes that the $\mathcal{C}$ language she uses could benefit from the $\mathcal{L}$ linking ecosystem. She asks the compiler writer for a $\mathcal{C}$-to-$\mathcal{L}$ compiler.

Discussion

- Linking types are entirely opt-in – a programmer can use a language that has been extended with them and benefit from the compiler tool-chain without knowing anything about them. Only when she wants to link with code that could violate her source-level reasoning does she need to deal with linking types.
- FFIs are usually considered “advanced material” in language documentation primarily due to the difficulty of using them safely. Since linking types enable safe cross-language linking, we hope that linking-type FFIs will not be considered such an advanced topic.

Day 5: Backwards compatibility for language designers. Never a dull day for the compiler writer: she starts implementing a fully abstract compiler from $\mathcal{C}$ to $\mathcal{L}$, but realizes that $\mathcal{L}$ is not rich enough to capture the properties needed. She extends $\mathcal{L}$ to $\mathcal{L}^{*}$, and proves fully abstract the translation from $\mathcal{L}$ to $\mathcal{L}^{*}$. Since full abstraction proofs compose, this means that she immediately has a fully abstract compiler from $\mathcal{A}^{\kappa}$ to $\mathcal{L}^*$. She then implements a fully abstract compiler from $\mathcal{C}$ to $\mathcal{L}^*$. Programmers can then link $\mathcal{A}^{\kappa}$ components and $\mathcal{C}$ components provided that the former do not use $\mathcal{C}$ features that cannot be expressed in $\mathcal{A}^{\kappa}$. Luckily for the compiler writer, the proofs mean that the behavior of $\mathcal{A}^{\kappa}$, even in the presence of linking, was fully specified before and remains so. Hence, $\mathcal{A}$ and $\mathcal{A}^{\kappa}$ programmers need not even know about the change from $\mathcal{L}$ to $\mathcal{L}^*$.

Discussion

- While implementing fully abstract compilers is nontrivial, the linking-types strategy permits a gradual evolution, not requiring redundant re-implementation and re-proof whenever changes to the target language are made.
- More generally, the proofs of full abstraction mean that the compiler and the target are irrelevant for programmers – behavior is entirely specified at the level of the (possibly extended by linking types) source.

5 Research Plan and Challenges

We are currently studying the use of linking types to facilitate building multi-language programs that may consist of components from the following: an idealized ML (essentially System F with references); a simple linear language; a language with first-class control; and a terminating language. We plan to develop a richly typed target based on Levy's call-by-push-value (CBPV) [21] that can support fully abstract compilation from our linking-types-extended languages. Zdancewic (personal communication on Vellvm2) has recently demonstrated a machine equivalence between a variant of CBPV and an LLVM-like SSA-based IR so this provides a path from our current intended target to a richly typed LLVM.

One critical aspect of such a type system is that it should be able to identify when a component is free of a given effect, even though the component may use that effect internally. For instance, a component that throws exceptions internally but handles them all should be assigned an exception-free type. We expect to draw inspiration from the effect-masking in the Koka language [18], where mutable references that never escape do not cause a computation to be marked as effectful.

Realizing such a multi-language programming platform involves a number of challenges. First, implementing fully abstract compilers is nontrivial, though there has been significant recent progress by both our group and others that we expect to draw upon [2, 3, 10, 7, 24, 25]. Second, low-level languages such as LLVM and assembly are typically non-compositional which makes it hard to support high-level compositional reasoning. In recent work, we have designed a compositional typed assembly language that we think offers a blueprint for designing other low-level typed IRs [26, 27]. Finally, we have only begun investigating how to combine different linking-types extensions. The linking-types extensions we are considering are based on type-and-effect systems, so we believe we can create a lattice of these extensions analogous to an effect lattice.

6 Conclusion

Large software systems are written using combinations of many languages. But while some languages provide powerful tools for reasoning *in* the language, none support reasoning *across* multiple languages. Indeed, the abstractions that languages purport to present do not actually cohere because they do not allow the programmer to reason solely about the code she writes. Instead, the programmer is forced to think about the details of particular compilers and low-level implementations, and to reason about the target code that her compiler generates.

With *linking types*, we propose that language designers incorporate linking into their language designs and provide programmers a means to specify linking with behavior and types inexpressible in their language. There are many challenges in how to design linking types, depending on what features exist in the languages, but only through accepting this challenge can we reach what has long been promised – an ecosystem of languages, each suited to a particular task yet stitched together seamlessly into a single large software project.

Acknowledgements. The authors are grateful to Matthias Felleisen for valuable discussion and feedback on linking types.

References

1 Amal Ahmed. Verified Compilers for a Multi-Language World. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15–31, 2015. doi:10.4230/LIPIcs.SNAPL.2015.15.

2 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.

3 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.

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

5 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.

6 Guy E. Blelloch and Robert Harper. Cache and I/O efficient functional algorithms. In *ACM Symposium on Principles of Programming Languages (POPL), Rome, Italy*, pages 39–50, January 2013.

7 William J. Bowman and Amal Ahmed. Noninterference for free. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*, September 2015.

8 Luca Cardelli. Program fragments, linking, and modularization. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, pages 266–277, January 1997.

9 Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *ACM Symposium on Principles of Programming Languages (POPL), Paris, France*, January 2017.

10 Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, 2016.

11 Vijay D'Silva, Mathias Payer, and Dawn Song. The correctness-security gap in compiler optmization. In *Language-theoretic Security IEEE Security and Privacy Workshop (LangSec)*, 2015.

12 Matthias Felleisen. On the expressive power of programming languages. In *Science of*

Computer Programming, pages 134–151. Springer-Verlag, 1990.

13 Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 595–608, January 2015.

14 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

15 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.

16 Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, Florida*, pages 178–190. ACM, 2016.

17 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.

18 Daan Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming, Grenoble, France*, April 2014.

19 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.

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

21 Paul Blain Levy. *Call-by-Push-Value.* Ph. D. dissertation, Queen Mary, University of London, London, UK, March 2001.

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

23 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. In *International Conference on Functional Programming (ICFP), Vancouver, British Columbia, Canada*, August 2015.

24 Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *International Conference on Functional Programming (ICFP), Nara, Japan*, September 2016.

25 Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 37(2):6:1–6:50, April 2015.

26 Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably mixing a functional language with assembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain*, June 2017. To appear. Available at `http://www.ccs.neu.edu/home/amal/papers/funtal.pdf`.

27 Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. FunTAL: Reasonably mixing a functional language with assembly (technical appendix). Available at `http://www.ccs.neu.edu/home/amal/papers/funtal-tr.pdf`, April 2017.

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

29 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.

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

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

32 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.

Teaching Programming Languages by Experimental and Adversarial Thinking*

Justin Pombrio[1], Shriram Krishnamurthi[2], and Kathi Fisler[3]

1 Computer Science, Brown University, Providence, RI, USA
2 Computer Science, Brown University, Providence, RI, USA
sk@cs.brown.edu
3 Computer Science, WPI, Worcester, MA, USA

Abstract

We present a new approach to teaching programming language courses. Its essence is to view programming language learning as a natural science activity, where students probe languages experimentally to understand both the normal and extreme behaviors of their features. This has natural parallels to the "security mindset" of computer security, with languages taking the place of servers and other systems. The approach is modular (with minimal dependencies), incremental (it can be introduced slowly into existing classes), interoperable (it does not need to push out other, existing methods), and complementary (since it introduces a new mode of thinking).

1998 ACM Subject Classification D.3.3 Language Constructs and Features, K.3.2 Computer and Information Science Education

Keywords and phrases mystery languages, interpreters, paradigms, education

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.13

1 An Enumeration of Principles

What do we want students to get out of a programming languages course? There are many things we might hope for, some of which include:

- The ability to distinguish syntax from semantics: to understand that a syntax is not "deterministic", in that one syntax can have many different meanings, and these differences of meaning can trip up the unwary programmer.
- The ability to look past superficial labels (such as "object-orientation") to understand features in depth, including the profound variations between their different realizations. (For instance, even with classes and inheritance, single-, multiple-, and mixin-inheritance result in very different languages and corresponding programming styles).
- The ability to discern good design from bad. This is, of course, a deeply personal matter of judgment: one person's beauty is another's monkey-patch. A good instructor presumably does not only impose their own world-view but also helps students understand the design trade-offs between different approaches, and also provides some historical background that helps students appreciate how current designs came to be.
- The ability to learn new programming languages. Especially in an era where we see a kind of "Cambrian explosion" of languages from both academia and industry, it is essential for students to have skills to pick up new languages quickly and accurately.

* This work was partially supported by the US NSF.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 13; pp. 13:1–13:9
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

These are some of the principles *we*, the authors, want students to learn, and we constantly re-design our courses to help students get closer to learning these ideas. This paper describes our latest and most radical re-design to get at the heart of this learning.

In this document, we will use the term *programming language* with a certain meaning: with an emphasis on *programming*. That is, we don't mean a formal calculus. We mean an actual system that one can run and whose programs can actually do something of interest to people other than pure programming-language researchers.

2 How We Teach and Learn About Languages

How do courses address these principles? Most undergraduate ("bachelor's") courses that focus on language principles – rather than implementation methods – seem to fall into two major schools. One approach subdivides languages by *paradigm* (though some authors have argued that the very concept of a paradigm is flawed [7]), and teaches students about variations within the paradigm as well as differences between paradigms. Some of the most widely-used programming languages books (like that by Sebesta [9]) are organized along these lines. Another view (formerly embraced by the present authors) is that one learns best by "teaching" a language to the patient but unforgiving computer, in the form of *definitional interpreters* [8]. This approach is also widely taught [1, 4, 6, 5].

We are unconvinced that either approach addresses our four principles particularly well. Though some of these can be addressed, it would be at a significant cost. For instance, most interpreter-based courses rarely implement multiple semantics for one syntactic feature (usually excepting function application or, in special cases, garbage collection [2]). They *could*, but even a high-level implementation can take a lot of time, and simply producing an interpreter does not by itself force reflection on programming *in* the implemented languages.

In contrast, it is worth pausing to ask the following question:

> What do *you*, dear reader, do when confronted with a new programming language?

For instance, consider one of the major recent languages: industrial ones like TypeScript, Flow, Hack, Swift, or Dart; or more academic ones like Agda, Idris, Liquid Haskell, or Rosette. Imagine you needed to learn one or more of them. Would you break it down by paradigms alone? Do you imagine writing a definitional interpreter? Or would you dive in and start writing programs to explore the space of the language? Would you look up a few examples, try them out, then embark on exploring the state space – like an adversarial thinker does – by asking, "Hmm, I wonder what happens if I tweak *this* to say *that* instead?"

3 Languages as Natural Science Artifacts

The reality of any programming language is that it is a complex ecosystem of parts that interlock in interesting ways, most of them at best poorly specified. They are rarely accompanied by a formal semantics. Even when a semantics exists, it may not cover the whole language, including the parts that programmers need to "get things done". Much of the semantics may lie instead in natural language prose documentation. Some of it may only be expressed in the language implementation's test suites.

Of course, even when a language is completely formally described, and when a programmer can read and comprehend the semantics, this is of only so much use. There is a large gap between the semantics itself and its *consequences*: for instance, the difference between eager and lazy evaluation strategies can be summed up completely in a few characters of the right semantics specification system, but the consequences of these differences – the different data structures they enable, the different proof methodologies they require, the different time/space reasoning they demand, and so on – are vast and still being explored.

Therefore, a programming language is less a purely mathematical object and more like an object found in nature. In addition to any formal interfaces it may present, we should – and do – also view it as a target for experimentation. We subject it to a variety of tests. Most of all, we follow a loose version of the scientific method: we form a hypothesis about the language's behavior; we construct tests (example programs and their expected outputs) consistent with the hypothesis; if the tests pass, we reinforce the hypothesis, otherwise we find it falsified; we use the falsification to generate a new hypothesis, which results in new tests. When we have exhausted our ability (or energy) to falsify, we accept our hypothesis as a tentative truth about the language, and use it to construct actual programs (whose behavior may – after painful debugging sessions – again falsify our understanding).

4 Mystery Languages for Teaching

To bring this mindset into *teaching* programming languages, we have started to experiment with what we call *mystery languages*. (This perhaps unfortunately abbreviates to "ML", though the allusion to machine learning is not unintentional; better suggestions welcome.) A mystery "language" is one syntax with multiple semantics that explore the design space for a feature. The student's task is to tell apart the differences and categorize them.

4.1 What They Are

Each mystery language exercise follows this template:

- A specification of a syntax. Each syntactic term is accompanied by an intentionally vague textual description of its semantics. (The reason for intentional vagueness will soon be made clear.)
- An editor for writing programs in that syntax.
- A parser for that syntax.
- Implementations of *multiple different behaviors for the same syntax* – we call these the *variants*. Through experiments, students will have to figure out how these differ.
- A display area to show the output of each variant.

There are two reasons for vagueness. First, the description must encompass all the variants. Second, it mirrors the vagueness sometimes found in real-world language documentation, where the meaning is clear to the author but can be misleading to someone coming from a different mental model of the same feature.

As a simple example, suppose a language introduces a notation like `a[i]`, and the description says that this is an array dereference. When `i` is within bounds, all variants might behave identically. But when `i` is out of bounds, one variant might signal an error (like Java does), another might return a special undefined value (like JavaScript does), and the third might return the content of some other array (like C sometimes does).

In practice, the mystery language variants are much weightier than this, and primarily concentrate on "normal" rather than error behavior. For instance, they present different semantics for argument evaluation, for structure mutation, for inheritance, and so on. Space precludes us from discussing them in detail, but readers may find it instructive to see (and even play with!) several concrete examples. Look for the "ML:" assignments on:

`http://cs.brown.edu/courses/cs173/2016/assignments.html`

This idea also *scales down*: in the very first week, with just numerals and a few binary operations, students can already explore the different semantics given to numbers (all floating point à la JavaScript, a mix of integer and floating point as found in most mainstream languages, bignums and rationals à la Scheme – as well as behaviors for division by zero, ranging from an error to a value representing infinity to a value representing undefinedness).

Certainly, discriminating variants is not new. For decades, texts have asked for or shown pithy examples that illustrate, for instance, the difference between call-by-name and call-by-value. What we have done is taken this idea to the extreme to explore its consequences: We examine dozens of variants; we actually implement them, so they don't just live on paper; and we put them in a consistent syntactic (Section 4.2) and execution framework so that students can focus on just what is changing (the semantic variants) without the distraction of changing syntax, execution environment, etc. These quantitative changes add up to a qualitative difference, enabling a new kind of pedagogy.

4.2 Assignment Prompts

Because a language has many parts, the assignment statement directs students to the parts on which they should focus their exploration. We have chosen – though this is by no means necessary – to grow a single language (syntax) incrementally, so each new assignment introduces some new syntactic features. Naturally, the focus is primarily on that new feature. However, many new features interact with any or all the other (previous) features of the language, so in studying that new feature, the student must explore its interactions too.

Given a set of variants of a mystery language, a student's task is to somehow figure out how they differ – this is where they engage in *adversarial* behavior – and then explain these differences (the *science*). Students must thus submit their solutions in two parts:

Classifier. A small set (usually no more than five) of programs, called *discriminators*, in the common syntax. Each discriminator must produce different output on at least two variants, and between the entire classifier, all variants must be distinguished.

Theory. A prose explanation of what they think the underlying behavior is. This is where the scientific method kicks in: they must formulate a theory and defend it using their classifier. Therefore, a good solution does not just turn in the first classifier found, but rather goes through several classification-theorizing (i.e., concrete-abstract) iterations until, ultimately, the examples are able to support the provided theory.

It can be tempting to produce the smallest classifier possible – e.g., by combining several discriminators into one. The theory acts as a counterweight against this. To provide a clear description, it makes sense to keep different explanatory programs separate.

4.3 Rules for Language Variations

Because students are being given black-box implementations with essentially undefined behavior, they have no way of knowing how complex or perverse a language variant might be. We therefore adopt the following rule. Every variant is tagged as one of the following:

Core. A Core variant has behavior (for the feature of interest) that was or is found in some widely-used, mainstream language. This does not mean the behavior could not be considered perverse; it simply means that at some point in time, many programmers were exposed to it. (This can, for instance, include dynamic scoping, or functions without proper recursion.)

Advanced. Advanced variants are similar to Core ones but are, in our judgment, either uncommon or especially difficult to find. For instance, an Advanced variant might mimic the vagaries of "lifting" variable declarations in scope in JavaScript, or might implement call-by-copy-result on function calls.

Prank. Prank variants are where we have fun. They might, for instance, turn some identifiers into Roman numerals, introduce laziness into some positions (in a language with mutation), change case-sensitivity, or alter evaluation order. (Sometimes, real languages have subtle implementation errors that result in bizarre behavior; these would also be good candidates for pranks.)

Importantly, we inform students that they: are expected to classify all Cores; should try to classify as many Advanced variants as they can; but, to get a good grade, are *not* expected to classify *any* Prank variants. This way, students with time or motivation can explore the Pranks, but most students can avoid them and prevent the assignments from turning into frustrating and bottomless time-sinks exploring arbitrary perversity.

4.4 Linking to Lectures

There are several ways in which the mystery language homeworks can interact with the lecture schedule. One approach we found especially productive was to have one or more lectures on each mystery language assignment immediately after it was turned in, while their work (and struggles) were fresh in students' heads. The lecture consists of:

- Live classification using discriminators provided by students in the class.
- A revision of these programs into a more "canonical" classifier, if necessary.
- A description of the expected underlying theory, with ties to the classifier.
- A discussion of the broader perspective surrounding this theory: one part historical (which languages did or do this) and one part design (why they did or do so). It is valuable to present these as non-judgmentally as possible: they all had advocates with good reasons at some point. Reconstructing their thinking, and showing why it is no longer relevant, is often much more instructive than simply dismissing it out of hand. (Features like dynamic scope, or COBOL's approach to "recursion", or multiple inheritance, might all fall in this camp.) In particular, one hopes this will help students recognize these arguments if they or their colleagues make them in a language that they subsequently design!

The lecture is also the ideal time to introduce the relevant vocabulary: e.g., "static scope", "aliasing", etc. These terms, which may have seemed rather abstract and perhaps irrelevant beforehand, now have an urgent value, because they precisely capture the concepts students both observed in their discriminators and struggled to describe in their theories.

4.5 Grading Criteria

One part of the grading is trivial: telling whether the classifier properly classifies. Indeed, the beauty of this part is that it is essentially self-grading: at the time of submission, students already know how well they did (and can seek help from course staff right away).

Grading the theory, of course, requires knowledge and judgment. The graders need to know what the "true" differences are, but should also be aware that there may be more than one accurate way of describing those differences. They then assess how well a student's theory predicts the variants' actual behavior, and how well articulated it is. Of course, when all discussion of the feature is deferred to after the assignment is due, students fundamentally lack the vocabulary to describe what they see. Grading then measures how well they were able to articulate the *idea* behind such concepts even though they lacked a crisp term for it. This is not easy to grade, but ease of grading should not always be the primary criterion.

5 Experience

We have now experimented with variations of this approach in two courses at two different universities, with preliminary success. At WPI it was used as a small component of the course; at Brown, it was the primary structure of the whole course. Here, we report on data from Brown. We believe a detailed statistical analysis would suggest false precision; instead, we offer a broad-strokes summary.

The course at Brown is primarily taken by juniors and seniors (3rd and 4th year college students), but also some sophomores (2nd year), master's students, and PhD students. The students have very diverse backgrounds: some have seen primarily Java and some Python; some have seen Racket, OCaml, and Java; some have seen Pyret; many have seen low-level programming in C; and so on. This class had about 70 students turn in each assignment (though some were done in pairs).

On almost every assignment, all but 2–3 students successfully classified all the languages. The exceptions were: missing parameters being treated as undefined values (à la JavaScript); COBOL-style non-re-entrant function calls; call-by-copy-result; shallow-copy on calls for structures; and mixing laziness with state. For these, about 10% of the class failed to fully classify the languages (but still distinguished most variants). An assignment with several different semantics of field access (corresponding to Python, JavaScript, R, etc.) saw the biggest variation, but this is because only two were tagged Core, one was Advanced, and the other two Prank. (Still, $\frac{1}{3}$ of the class classified them all.)

In terms of their theories, until about half-way through, as many as half the students got grades that indicated notable weaknesses (but better than nothing). As students improved, we altered our grading scale to provide more refined information. In the second half, about 60% obtained an A grade, 35% obtained a B, and a handful got C's and A+'s (superlative).

How long did the work take? About 95% spent under five hours on each assignment (self-reported), with half or more (depending on the assignment) spending under two hours. (For simpler homeworks about a quarter reported spending under an hour, but these durations vanished on the more difficult assignments.) Every assignment had about 5% spend 5–10 hours, and virtually nobody ever reported spending over ten hours.

Students were also surveyed at the end of the semester. They reported that the mystery language assignments:

- Gave them tools to confront a new language:
  50% strongly positive, 42% somewhat positive
- Helped them separate syntax from behavior:
  62% strongly positive, 20% somewhat positive
- Helped them learn about possible behaviors of the features they studied:
  62% strongly positive, 29% somewhat positive
- Helped them learn to judge between different behaviors:
  48% strongly positive, 48% somewhat positive

In short, there were very few negative sentiments, and the assignments met the course's learning goals overwhelmingly well.

Students were also asked whether the assignments were frustrating and whether they were fun. Thankfully, students did not view these as contradictory. Approximately: 19% found them very frustrating, 48% somewhat frustrating, 29% not very frustrating, 1% not frustrating at all. Simultaneously: 29% found them very much fun, 48% somewhat fun, 14% not much fun, and 1% not fun at all. (That one student intensely disliked almost all aspects of the course, across the board.)

6 Perspective

Having discussed the specifics of these assignments in considerable detail, it is now worth revisiting the claims made in the abstract and introduction and seeing how well the assignments measure up.

First, we should justify the paper's title. Our approach is clearly experiment-centric, and we believe that it also has notable parallels with the view taken by people who break into systems. The mindset that suggests combining disparate and dissociated elements to explore their outcomes is essentially at the heart of mystery languages, too.

What our results (Section 5) suggest is that the vast majority of students who opted to take this new format of course are able to engage in this behavior. These are not students who were selected for the "security mindset" (most had not taken the hacking-oriented security course; the course had similar enrollments and drop rates as previous years; etc.). There were also no discernible biases in outcomes. For instance, women did not perform worse than men; proportionally, they actually did better. (This cannot be explained by grader bias, because all grading was anonymized.) Furthermore, the majority of the class found it constructive to engage in this activity, and even found fun in the frustration.

Student performance and the survey results show that this approach was very successful at helping students separate syntax from semantics, enabled them to explore in some depth the features they confronted, and gave them a framework for exploring language designs.

As far as exploring new languages, the final course project asked them to choose the "object-oriented" features of one of JavaScript, Ruby, Python, or R, and (with no scaffolding from us) write up a descriptive report accompanied by illustrative examples "highlighting any behavior that seems unconventional or peculiar", and discussing their consequences for type system design. This work was graded by a team mostly excluding this paper's authors. Students averaged 80% with a standard deviation of 22% – a reasonable outcome considering that this was the first time all semester they were asked to perform such a task.

In addition, mystery languages appear to have the following virtues:

Modularity. Mystery languages can be used with minimal curricular dependencies. The WPI course was and remained mostly implementation-based, but was able to incorporate a small quantity of (a variant of) mystery languages. The Brown course (which had previously been entirely implementation-based) switched mostly to this, but also had a few implementation assignments in parallel. Mystery languages can also be injected into a paradigms or other style of class, being used to concretely illustrate certain points.

Incrementality. As the WPI experience shows, one does not need to change the structure of their course entirely. A few select features can be explored using mystery languages, after which the course can either grow their use or not. In particular, instructors can give them a try without having to change their classes wholesale. Furthermore, our data show that a few hours are sufficient to complete a mystery language assignment, making the base footprint very light.

Interoperability. Mystery languages interface well with other approaches. In a paradigms approach, for instance, they can be used to make certain salient issues concrete, and to let students explore them through programs rather than just on paper. In an interpreter approach, they can be used to introduce and make real a feature before students write a definitional interpreter for it – and also give them a ground-truth against which to test their interpreter.

Complementarity. Mystery languages introduce a new way of thinking about languages. These don't displace talking about paradigms or exploring the fine structure of languages through individual expressions in an interpreter, but rather complement it. While in principle similar points could be made through, say, writing different interpreters for one syntax, the burden of building several languages feels much greater than that of writing a few test programs – and, to many students, may also be less fun, since it involves more "building" and less "breaking".

Therefore, we believe it would be profitable for other instructors to also consider incorporating this approach. Indeed, we would ideally hope to build a large communal repository of mystery languages so that instructors can pick-and-choose based on taste, interests, and also the need to avoid plagiarism.[1]

7 Challenges

That's the good news. There are also many challenges induced by this approach that we need to explore over time.

- None of our claims have controls. Perhaps students would meet the same learning objectives just as effectively through other teaching models. Perhaps we are naturally biased in the selection of questions and of metrics. Perhaps this approach does not work at other universities. We feel a larger community evaluation effort is needed to obtain definitive answers to some of these questions.
- It is tremendously difficult to build and maintain all these languages. This semester alone, we essentially built 36 different ones![2] Furthermore, we built them in JavaScript for student convenience (and you, too, can try them – right now – from the URL given above, as did many course alumni). Unfortunately, JavaScript is hardly a natural medium for building programming language implementations, much less variations between them. A language laboratory like Racket [3] would have been a much better choice in principle. However, the choice of language must be balanced against deployment issues – especially early in the life cycle of this approach, when errors are sure to creep in and will necessitate rapid and painless re-deployment of fixes.
- It can be difficult for students to formulate a theory about black-box languages, and to express it without suitable terminology at their command. (In particular, we do not yet have good means to help students who are utterly stuck.) It would have been constructive for them to somehow redo their work after the accompanying lectures.
- It is difficult to have one syntax apply across all variants. This can lead to slightly perverse choices: e.g., for field access, we used the syntax `o["x"]` for all languages, even those where the field name is first-order – where `o.x` would have been much more natural. (The first-order languages imposed syntactic restrictions on what can appear inside brackets, which otherwise appears to be an expression position.)
- We have had time to explore only a small space of language features. Many more, such as types, concurrently, and advanced control, remain untouched.

Nevertheless, we view these as challenges, not as insurmountable obstacles.

Acknowledgements. We are deeply grateful to Anjali Pal, Sahakait Benyasut, Sara Hartse, William Kochanski, and Alex St. Laurent, the Brown course undergraduate TA staff, for

[1] The danger of plagiarism is in part why we do not describe the languages in more detail in this paper: there are only so many non-prank variants one can create, after all!

[2] Some feature interactions due to implementation shortcuts unfortunately resulted in stranger language semantics than even we expected – and which students caught...

enthusiastically taking the plunge with this new approach. We especially thank William for helping us formulate the written portion of submissions in terms of the scientific method, which provided much-needed guidance and clarity. We also thank Eli Rosenthal for a useful discussion that helped trigger this approach. We are grateful to Matthias Felleisen, Eli Barzilay, and Preston Tunnell Wilson for useful conversations and feedback.

References

1 Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

2 Gregory H. Cooper, Arjun Guha, Shriram Krishnamurthi, Jay McCarthy, and Robert Bruce Findler. Teaching garbage collection without implementing compilers or interpreters. In *ACM Technical Symposium on Computer Science Education*, 2013.

3 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket manifesto. In *Summit on Advances in Programming Languages*, 2015.

4 Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press.

5 Samuel Kamin. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley.

6 Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*.

7 Shriram Krishnamurthi. Teaching programming languages in a post-Linnaean age. In *SIGPLAN Workshop on Undergraduate Programming Language Curricula*, 2008. Position paper.

8 John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(3), 1960.

9 Robert Sebesta. *Concepts of Programming Languages*. Addison-Wesley, fifth edition.

Let's Fix OpenGL

Adrian Sampson

Cornell University, Ithaca, NY, USA
asampson@cs.cornell.edu

Abstract

From windowing systems to virtual reality, real-time graphics code is ubiquitous. Programming models for constructing graphics software, however, have largely escaped the attention of programming languages researchers. This essay introduces the programming model of OpenGL, a ubiquitous API for real-time graphics applications, for a language-oriented audience. It highlights six broad problems with the programming model and connects them to traditions in PL research. The issues range from classic pitfalls, where established thinking can apply, to new open problems, where novel research is needed.

1998 ACM Subject Classification D.3.m Miscellaneous

Keywords and phrases language design, real-time graphics, OpenGL, GPUs, heterogeneity

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.14

1 Throwing Some Shader

Nearly every consumer computing device on the market, from smartwatch to workstation, comes with a graphics processing unit (GPU). Any software that renders graphics in real time must exploit a GPU for reasonable performance, which entails programming to one of the mainstream APIs that graphics cards support. GPU vendors have settled on two common GPU interfaces, OpenGL [42] and Direct3D [31], so graphics software almost exclusively builds on one of these two APIs.

OpenGL and Direct3D may have been created as hardware abstractions, but they do double duty as programming models. The two APIs use a common structure consisting of two components: a full-fledged programming language for writing programs that run on the GPU, and a set of C functions for communicating between the CPU and an attached GPU. Both components contend with a vast array of classic problems in programming languages: abstraction and reuse; the need to avoid obscure run-time errors; expressiveness without sacrificing performance; and so on. The APIs, however, have largely avoided adopting the answers that programming languages research has developed to these problems – even basic, conventional wisdom in our community has escaped the design of graphics APIs.

This essay introduces OpenGL and its pitfalls for the PL-minded reader and advocates for more research that applies language ideas to this underserved domain. It enumerates six language problems that OpenGL programmers face and proposes possible directions for solving them. Some problems invite straightforward applications of established traditions in PL research, and others are open problems without clear solutions. Despite its difficulties, GPU-accelerated graphics programming is enormously popular – it underlies a $90 billion global video game industry, for example [32] – so research that addresses its shortcomings has potential for real-world impact.

Graphics programming also represents the tip of the spear for *heterogeneous programming*, the general problem of orchestrating separate, specialized hardware units in a single program. As the capabilities of traditional CPUs stagnate, software will need to exploit increasingly

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 14; pp. 14:1–14:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
// Vertex shader:
in vec4 position;
in float dist;
out vec4 fragPos;
void main() {
  fragPos = position;
  gl_Position = position + dist;
}
```

```
// Fragment shader:
in vec4 fragPos;
void main() {
  gl_FragColor = abs(fragPos);
}
```

Listing 1 A GLSL shader pair.

exotic hardware to continue making advances [38]. Ensembles of oddball hardware beyond the GPU, from FPGAs to fixed-function units, will only increase the need for heterogeneous programming models with the same set of challenges as OpenGL.

2 Graphics Programming with OpenGL

This section dissects a tiny OpenGL program.[1] While this essay does not illustrate Direct3D directly, the programming model there is similar and exhibits similar pitfalls.

2.1 Shader Programs

The soul of a real-time graphics application is its *shader programs*. A shader is a short program that runs on the GPU as part of the rendering pipeline to define the shape and appearance of objects in the scene. There are several kinds of shaders, but the two most common are the *vertex shader* and the *fragment shader*, which respectively compute the position of each vertex in 3D space and the color of each pixel on an object's surface. In OpenGL, shaders are written in the special-purpose GLSL programming language, which is a variant of C. Direct3D has its own shader language, HLSL, which is a similar but incompatible C variant.

Listing 1 shows a vertex and fragment shader in GLSL. Each shader consists of a `main` function and some global definitions. The global definitions use `in` and `out` qualifiers to mark variables that represent the shader's inputs and outputs. In this vertex shader, for example, a `position` vector and a `dist` scalar both come from the CPU. This shader assigns the magic `gl_Position` variable to this parameter – this is the vertex shader's output. The `position` value is only available at the first stage – in the vertex shader – so more work is required to pass it along to the fragment stage. This shader pair declares a second variable, `fragPos`, in both programs to hold the `position` value from the vertex stage and make it available in the fragment stage. Finally, the fragment shader uses `fragPos` as an input to compute its output: the `gl_FragColor` magic variable.

2.2 Shaders are Strings

GLSL code only runs on the GPU. *Host code* on the CPU uses a traditional general-purpose language – usually C or C++. To draw an object in a 3D scene, the host code needs to compile the GLSL source code to the GPU's internal instruction set, send its parameters, and invoke it. Each GPU vendor uses a different internal representation for shaders, so GLSL

[1] Details are omitted here for focus. Full source code is online: `http://adriansampson.net/doc/tinygl/`.

```
// Embed shader source code in string literals.
static const char *vertex_shader = "in vec4 position; ...";
static const char *fragment_shader = "in vec4 fragPos; ...";

// Compile the vertex shader.
GLuint vshader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vshader, 1, &vertex_shader, 0);

// Compile the fragment shader.
GLuint fshader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fshader, 1, &fragment_shader, 0);

// Link the pair together.
GLuint program = glCreateProgram();
glAttachShader(program, vshader);
glAttachShader(program, fshader);
glLinkProgram(program);
```

Listing 2 Compiling a shader pair.

source is the *lingua franca* that provides compatibility: every GPU driver includes its own GLSL compiler.

Listing 2 shows the C boilerplate for compiling and linking a vertex/fragment shader pair. Here, the GLSL source code is embedded in the executable using a string literal; it is also common to use `fopen` to load the source from a text file when the program starts up. Later, to draw an object in a frame, the host code uses the `program` reference to tell the GPU which linked shader pair to use when drawing an object.

2.3 CPU–GPU Coordination

To supply the shaders' inputs, the host code looks up *location* handles for each `in` variable in the GLSL code. There are two main options: the shader code can mark each variable with a fixed index, or the host code can look the variables up by name. Listing 3 shows the latter, which manifests as a series of `glGet*Location` calls.

Our example shaders use two kinds of input variables: `position` is a *vertex attribute*, meaning that it takes a different value for every invocation of the vertex shader's `main` function; and `dist` is a *uniform*, so it remains constant across the object's vertices. For attributes, the program needs to allocate a *buffer* representing the GPU's memory region for the variable.

Finally, to draw each frame, the program selects the compiled shader pair with a call to `glUseProgram(program)`. Then, to provide a value for the `position` attribute, it executes `glBufferSubData` to copy data from the host memory – i.e., a plain C array – to the GPU-side buffer. For the uniform, the render loop uses a `glUniform*` call to set the variable.

3 Problems & Potential Solutions

Even this abridged example should raise some language-design alarms in the mind of a PL researcher. The problems start, but do not end, with the ordinary infelicities of any aging C API design: hidden state, minimal static safety checks, and so on. This section enumerates six obstacles in graphics programming. The first problems are classic pitfalls with established answers in the PL literature on language extensibility, static safety, and

```
// Setup code:

// Look up shader variable locations.
GLuint loc_position = glGetAttribLocation(program, "position");
GLuint loc_dist = glGetUniformLocation(program, "dist");

// Allocate a buffer for the attribute.
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glVertexAttribPointer(loc_position, size, GL_FLOAT, GL_FALSE, 0, 0);

// ...
// In the render loop:

glUseProgram(program);

// Copy the vertex positions into the buffer.
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);

// Set the uniform variable.
glUniform1f(loc_dist, 4.0);
```

Listing 3 Communicating with the shaders.

metaprogramming. Here, real-time graphics represents a new application domain for existing lines of research. The final three issues expose new open problems that pertain specifically to graphics: rate-oriented language formalisms, type systems for linear algebra, and defining "correctness" for visual systems.

3.1 Shader Languages are Subsets of Supersets of C

GLSL and its Direct3D equivalent, HLSL, are for the most part plain, everyday imperative programming languages. They have variable declarations, `if` conditions, `for` loops, function calls, and global mutable state – just like any ordinary imperative language. Shader languages, however, are unique, one-off reinventions that *resemble* C without being clean extensions. The distinction makes life more difficult for programmers, who need to carefully keep track of the subtle differences between GLSL and "real" C. In C, for example, the name of the type declared by `struct t {...}` is `struct t`; in GLSL as in C++, the name is just `t`. In C++, variable declarations can appear inside an `if` condition; in GLSL as in C, they cannot. These myriad incompatibilities make it difficult to move code between the CPU and GPU.

The need for *ad hoc* language extension also complicates compiler implementations: current compilers either need to reinvent a complete C-like parser and compiler from scratch [24] or hack an existing frontend such as GCC or Clang. Both approaches are error prone: recent work by Donaldson et al. [11, 12] has revealed crashing bugs in a staggering array of vendor-supplied GLSL compilers. Apple's new Metal shading language [1] is based on the C++14 standard, but even it relies on a custom Clang fork with informal restrictions on certain features, such as subclassing and recursive calls.

Potential solutions. Shader languages' needs are not distinct enough from ordinary imperative programming languages to warrant ground-up domain-specific designs. They should

should instead be implemented as extensions to general-purpose programming languages. There is a rich literature on language extensibility [34, 23, 37] that could let implementations add shader-specific functionality, such as vector operations, to ordinary languages. The potential productivity benefit is large, especially for host languages other than C: programmers could opt for more diverse languages without needing to context-switch to C-like syntax and semantics to write shader code.

Some existing work implements *embedded DSLs* for generating GLSL code [41, 6, 3, 13, 29, 28, 5, 30, 2, 26], which is a useful first step. But this embedded approach yields code that looks much different from the host language. It also typically requires that host programs generate GPU code on the fly, at run time. A language-extension approach could match the syntax and semantics of the host language without incurring the cost of dynamic code generation.

3.2 Loose CPU-to-GPU and Stage-to-Stage Coupling

In OpenGL, interactions between the CPU and GPU are *stringly typed:* the `glGet*Location` calls in Listing 3 look up variables in the shader programs by their names. Communication between shaders is similarly brittle: the two separate programs in Listing 1 need to agree on a name for `fragPos`, which must not conflict with the CPU-to-GPU name `position`. Even though both C and GLSL are statically typed languages, neither compiler can statically check their naming agreement. Shader source code is only compiled after the host code begins executing, and the program might arbitrarily pair vertex and fragment shaders together.

The lack of static semantics comes with all the same productivity pitfalls as programming in a dynamic language like JavaScript. Typos in variable names are not reported until run time; type errors are similarly deferred; refactoring tools in IDEs are hobbled; and static compilers must conservatively eschew optimizations. Regardless of opinions on static typing, programmers tend to agree that at least an unsound, optional *lint*-like static checker can be helpful – but OpenGL programmers do not enjoy even that basic luxury.

Potential solutions. Language research should endeavor to clean up the abstractions between shader code and CPU code. At a bare minimum, CPU–GPU and inter-stage communication must be made type safe. In the near term, researchers should explore backwards-compatible approaches to giving static semantics to complete C++/GLSL hybrid programs. A program analysis could ingest the OpenGL API calls in the host code and the variable declarations in the shader code to check that they align and to propagate type information between the two languages. A sound analysis that rules out any possible CPU–GPU disagreement may be too difficult to achieve, but even a best-effort checker could help avoid unnecessary run-time failures.

In the long term, more research should unify CPU–GPU programming in a single language that spans the CPU and all GPU stages. Communicating a value from the vertex stage to the fragment stage should introduce no more syntactic or cognitive overhead than defining and referencing a variable. Instead of relying on the programmer to divide the complete computation into stages, the compiler should take responsibility for splitting CPU host code from GPU shader code.

In a hypothetical unified programming model, the primary question is how much to rely on compiler automation and where to use explicit programmer control. A binding-time analysis [33], for example, could automatically determine the earliest possible stage for each computation, but earlier is not always better: running code once on the CPU and communicating it to the GPU can be more costly than running the same code redundantly on the GPU. Two recent languages from the graphics community, Spark [16] and Spire [21], propose to use a type system instead. Type annotations let the programmer control where and when each expression in a unified program is executed. I am currently exploring a similarly explicit design where a multi-stage programming language [45] models the GPU's pipeline stages.

3.3 Massive Metaprogramming

Performance is a first-order concern in real-time graphics, so programmers need to avoid all unnecessary overhead in shader programs. To avoid the overhead that comes with generality, applications typically generate many specialized variants of more general shader programs called *übershaders* [19]. An übershader for metal materials, for example, might combine many parameterized effects to support different settings for color, shininess, damage, rust, texture, and so on. Übershaders are convenient for artists and other non-programmers, who can tweak parameters to design a specific effect without writing any code. But these monolithic designs pay a performance penalty for their generality: pervasive parameters incur CPU–GPU communication overheads and add costly branching to the shader code.

To avoid these overheads, some implementations recover efficient shader code by generating *specialized* shader programs that "bake in" each set of parameters and strip out unneeded functionality. Shader specialization occurs at a massive scale: modern video games can generate hundreds of thousands of shader variants [22, 21]. The only tool OpenGL offers for shader specialization, however, is the C preprocessor with its familiar `#define` and `#ifdef` directives. Unhygienic token-stream rewriting may not be so bad for small-scale metaprogramming, but it does not scale to large-scale shader specialization. Graphics programmers resort to developing *ad hoc* toolchains to stitch together snippets of GLSL code into whole shader programs [49].

Potential solutions. The urgent need for programmable specialization of general shader programs is an opportunity for metaprogramming research. Graphics programmers should be able to write and reuse libraries of tactics for manipulating shader code for efficiency. Both run-time and compile-time metaprogramming can be useful: while it is less common in current practice, dynamic specialization could eliminate some shader overhead that is out of reach for static techniques.

Metaprogramming techniques from the programming languages community are up to the task. They can enforce safe program generation [45], allow composition of compile-time macros from multiple, independent libraries [15], and even incorporate dynamic profiling data [8]. Shader specialization represents an opportunity to stretch the scalability of this classic work. Where most work on metaprogramming focuses on implementing language extensions or generating a single target program, shader specialization requires the system to synthesize thousands of variants and choose between them at run time. The massive scale creates new challenges: programmers may need new mechanisms to *limit* specialization, for example, to stay within practical limits.

3.4 Informal Semantics for Multiple Execution Rates

Each stage in a GPU's graphics pipeline runs at a different rate. Interactions between the rates have subtle implications for the semantics of complete, multi-shader programs. The fragment shader, for example, runs many times for every execution of the vertex shader: it interpolates the pixels between adjacent vertices on a surface. The values passed between the vertex and fragment stage are also interpolated. The `fragPos` variable in Listing 1 is exactly equal to `position` in the vertex shader, but it takes on interpolated values in the fragment shader. Therefore, an expression involving `fragPos` has subtly different semantics depending on which stage it appears in. The story gets more complicated with other shader types: *geometry shaders*, for example, operate on multiple adjacent vertices simultaneously.

The OpenGL standard defines the meaning of each stage individually. It does not attempt a general theory for the semantics of arbitrary shader rates and their interaction. If future generations of GPUs introduce new programmable stages to the graphics pipeline, each new rate will need a new *ad hoc* definition. Some work defines the semantics of *general-purpose* GPU programming models such as CUDA [18, 20, 27], but these simpler GP-GPU languages do not have multi-rate execution or fixed-function interpolation logic. Programmers are left with only informal descriptions of the semantics of interacting systems of shader programs.

Potential solutions. Language research should develop a core calculus for massively parallel, multi-rate programs. Programs in a hypothetical λ_{GPU}-calculus would describe how and when state from one stage becomes visible to a set of parallel invocations in another stage. The new multi-rate semantics may resemble an existing multi-stage semantics [45, 14] where control flows linearly through a series of nested stages. Graphics-specific phenomena such as inter-stage interpolation should also be made explicit in this calculus. In λ_{GPU}, researchers could not only formalize the semantics of real, mainstream GPUs but also explore the space of alternative GPU designs to inform future hardware development.

3.5 Latent Types for Linear Algebra

Graphics code – both inside shaders and in host code – consists mainly of vector and matrix operations. Points in space are floating-point vectors (called `vec3` or `vec4` in GLSL) and transformations between vector spaces are represented as 4×4 matrices (the `mat4` type). Every realistic system of shaders needs to juggle a handful of common vector spaces: typically, a *model* space, where vectors are relative to a specific object's position; *world* space, which all objects share; *camera* space, relative to the camera's perspective; and *projection* space, relative to the 2D canvas where the scene will be drawn.

Shader code is correspondingly littered with duplicate variables that represent the same vector in different spaces. For example, most programs pass model, view, and projection matrices to their shaders, each of which can transform from one vector space to the next. Shaders then create camera-space and world-space versions of input vectors and use them in computations. For example, *lighting models* for simulating reflections typically start by computing the angle of light, which involves subtracting the light source position vector from the model's position vector:

```
in mat4 model, view, projection;
in vec4 position;  // in object space
in vec4 light_position;  // in world space
void main() {
  vec4 position_camera = view * model * position;
  vec4 position_world = model * position;
  // ...
  vec4 light_direction = light_position - position_world;
}
```

The subtraction `light_position - position_world` happens in world space. The result would be meaningless if the program instead used `position_camera`: the spaces do not match. There are clearly legal and illegal ways to combine matrices and vectors, but the shader language offers no help with enforcing these rules: programmers resort to naming conventions and boilerplate to keep things straight.

Potential solutions. The vector-space problem in graphics code is an opportunity for type system research. A linear-algebra type system could take inspiration from type systems for units of measure: the type of a vector value would tag it with a vector space. The corresponding transformation matrix would be marked with a pair of vector spaces: the space it translates *from* and the one it translates *to*. For example, a vector `v` might have type `vec4<A>` to indicate that it is in space A, and a matrix `m` of type `mat4<A, B>` would translate from vector space A to B. Using these two argument types, the type system can give the multiplication expression `m * v` the type `vec4<B>`. It is an error to multiply `v` by a different matrix of type `mat4<C, D>` where $C \neq A$ because the result has no meaningful vector space. This hypothetical type system could automate the process of tagging vectors and checking their correspondence. Because a vector space type is defined by a transformation matrix value, such a linear-algebra type system may benefit from exploiting a dependent type system [44].

Beyond basic checking, the type system could help *synthesize* the appropriate transformations rather than relying on the programmer to write the boilerplate. For example, a new expression form `v in B` could automatically find the right matrix to multiply by `v` to produce a B-space vector. This implicit approach would avoid the need for a convoluted naming scheme to distinguish `position` vs. `position_camera` vs. `position_world`. Synthesizing transformations automatically would also enable new optimizations: a tool could avoid redundant computation and communication by separating vector-space transformations from the main program text. For example, an expression `(v1 * v2) in B` can be computed by first transforming both vectors into space B and then multiplying them; equivalently, the program might multiply the vectors in some other space and then transform the product. These diverging possibilities form a search space for synthesis.

3.6 Visual Correctness and Quality Trade-Offs

While learning to program my first few shaders, I implemented the textbook Phong lighting model [36], a "hello world" of shader programming. In my first implementation, I made a mistake I cautioned against in the previous section: I used the wrong vector when converting between vector spaces. This single-token bug got lost amid the conversion boilerplate. The result, depicted in Figure 1a, looked ugly: the reflections were too intense and failed to light the entire object. It was not bad enough, however, to raise suspicion – I assumed that the simplistic lighting algorithm itself was to blame. According to my version control logs, the bug stayed in place for *nine months* before I found and fixed it (Figure 1b). The problem was that the result, while incorrect, was plausible enough that it was not *clearly* incorrect.

Testing and verification tools only work when programmers are willing to specify correctness, and specifying correctness is particularly difficult in graphics. The human visual system's tolerance to error makes it challenging to define *correctness* for rendering systems. Is a bug really a bug if most humans do not notice anything wrong with a scene? How do you write a unit test for "visual correctness"? Based on conversations with graphics programmers, testing seems to be very rare: developers instead make incremental changes and spot-check them manually to deem the output acceptable.

Beyond bugs, graphics programmers also *intentionally* compromise visual quality in return for efficiency. Real-time graphics animations are not perfect recreations of the real

(a) Buggy. **(b)** Correct (probably).

Figure 1 Output from a buggy and corrected implementation of the Phong lighting model. The difference is obvious now but was hard to detect without a ground-truth comparison.

world; it is more important that they maintain a high frame rate than for every object to look as realistic as a ray-traced reference image. Applications can even dynamically switch between multiple *levels of detail* for the same object depending on its salience in a given scene [22]. It is typically up to the programmer to manually select and implement quality-compromising optimizations, although some recent graphics work has proposed to automate the process [22, 48, 43, 35].

Potential solutions. Controlling output quality is the central challenge in *approximate computing* research [39, 9, 7, 40]. Researchers should treat graphics programming as an instance of approximate computing: the same set of statistical quality controls could apply.

Software engineering research should seek to understand how graphics programmers currently reason about correctness. What *ad hoc* processes have developers invented to cope with a world where perfect correctness is unachievable and bugs are in the eye of the beholder? With this baseline understanding, languages research can build tools to improve existing modify-and-check workflows. Recent work on live coding [17], for example, could help shorten the cognitive distance from source code modifications to visual feedback. More radical tools could seek to alleviate the need for manual output inspection – for example, by incorporating crowdsourced opinions [4].

4 Postscript

Like any outmoded but entrenched programming model, OpenGL remains universal despite its flaws. Many content designers avoid interacting with graphics APIs directly by building on monolithic *game engines* such as Unity [46] or Unreal [47], which sacrifice flexibility in exchange for abstraction. And real-world programmers can be wary of new language tools from academia, so adoption will be slow for research on graphics programming – even for proposals that unambiguously improve on the status quo.

However, 2017 is a particularly fertile moment for new ideas in real-time graphics programming. The standards body that specifies OpenGL recently published the largest change yet to its recommendations: Vulkan [25] is a ground-up redesign. Vulkan is a response to industry demands for a *lower-level* API than OpenGL [10], which hides too many performance knobs that software needs to tune. While OpenGL played a dual role as a hardware abstraction layer and a programming layer and arguably failed at both, Vulkan promises to abandon the pretense of being programmable: it is designed solely as a system abstraction. This shift has the potential to create an ecosystem of new, high-level programming tools that build on top of Vulkan and finally dislodge OpenGL's monopoly on graphics programming. The iron is hot, and programming languages research should strike.

Acknowledgments. Conversations with Yong He, Kayvon Fatahalian, and Tim Foley introduced me to real-time graphics programming and its infelicities. Their patient explanations pointed me in this direction. Todd Mytkowicz and Kathryn McKinley endured my early floundering with language design questions while I visited Microsoft Research. The anonymous SNAPL reviewers and Pat Hanrahan were exceptionally insightful with their suggestions for framing.

References

1 Apple. Metal shading language specification, version 1.2. `https://developer.apple.com/metal/metal-shading-language-specification.pdf`.

2 Chad Austin and Dirk Reiners. Renaissance: A functional shading language. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2005.

3 Baggers. Varjo: Lisp to GLSL language translator. `https://github.com/cbaggers/varjo`.

4 Daniel W. Barowy, Charlie Curtsinger, Emery D. Berger, and Andrew McGregor. AutoMan: A platform for integrating human-based and digital computation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012.

5 Tobias Bexelius. GPipe. `http://hackage.haskell.org/package/GPipe`.

6 Kovas Boguta. Gamma. `https://github.com/kovasb/gamma`.

7 Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

8 William J. Bowman, Swaha Miller, Vincent St-Amour, and R. Kent Dybvig. Profile-guided meta-programming. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2015.

9 Michael Carbin, Sasa Misailovic, and Martin Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

10 Advanced Micro Devices. Mantle programming guide and API reference 1.0. `https://www.amd.com/Documents/Mantle-Programming-Guide-and-API-Reference.pdf`.

11 Alastair F. Donaldson. Crashes, hangs and crazy images by adding zero. Medium, November 2016. `https://medium.com/@afd_icl/689d15ce922b`.

12 Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Workshop on Metamorphic Testing (MET)*, 2016.

13 Conal Elliott. Programming graphics processors functionally. In *Haskell Workshop*, 2004.

14 Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. Automatically splitting a two-stage lambda calculus. In *European Symposium on Programming (ESOP)*, 2016.

15 Matthew Flatt. Composable and compilable macros: You want it when? In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2002.

16 Tim Foley and Pat Hanrahan. Spark: Modular, composable shaders for graphics hardware. In *SIGGRAPH*, 2011.

17 Mark Guzdial. Trip report on Dagstuhl seminar on live coding, September 2013. Blog@CACM. URL: `http://cacm.acm.org/blogs/blog-cacm/168153`.

18 Axel Habermaier. The model of computation of CUDA and its formal semantics. Master's thesis, Institut für Informatik, Universität Augsburg, 2011.

19 Shawn Hargreaves. Generating shaders from HLSL fragments. In *ShaderX3: Advanced Rendering with DirectX and OpenGL*. 2004.

20 Chris Hathhorn, Michela Becchi, William L. Harrison, and Adam M. Procter. Formal semantics of heterogeneous CUDA-C: a modular approach with applications. In *Conference on Systems Software Verification (SSV)*, 2012.

21 Yong He, Tim Foley, and Kayvon Fatahalian. A system for rapid exploration of shader optimization choices. In *SIGGRAPH*, 2016.

22 Yong He, Tim Foley, Natalya Tatarchuk, and Kayvon Fatahalian. A system for rapid, automatic shader level-of-detail. In *SIGGRAPH Asia*, 2015.

23 Görel Hedin and Eva Magnusson. JastAdd: An aspect-oriented compiler construction system. *Science of Computer Programming*, 47:37–58, 2003.

24 Khronos Group. glslang. `https://github.com/KhronosGroup/glslang`.

25 Khronos Vulkan registry. `https://www.khronos.org/registry/vulkan/`.

26 LambdaCube 3D. `http://lambdacube3d.com`.

27 Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic verification and test generation for GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

28 Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. In *SIGGRAPH*, 2004.

29 Michael McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2002.

30 Sean McDirmid. Two lightweight DSLs for rich UI programming. `http://research.microsoft.com/pubs/191794/ldsl09.pdf`.

31 Microsoft. Direct3D. `https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466.aspx`.

32 Newzoo. Top 100 countries by 2015 game revenues, 2015. `https://newzoo.com/insights/articles/newzoos-top-100-countries-by-2015-game-revenues/`.

33 F. Nielson and R. H. Nielson. Automatic binding time analysis for a typed λ-calculus. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 1988.

34 Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction (CC)*, 2003.

35 Fabio Pellacini. User-configurable automatic shader simplification. In *SIGGRAPH*, 2005.

36 Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

37 Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2013.

38 Adrian Sampson, James Bornholt, and Luis Ceze. Hardware–software co-design: Not just a cliché. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

39 Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.

40 Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

41 Carlos Scheidegger. Lux: the DSEL for WebGL graphics. `http://cscheid.github.io/lux/`.

42 Mark Segal and Kurt Akeley. The OpenGL 4.5 graphics system: A specification. `https://www.opengl.org/registry/doc/glspec45.core.pdf`.

43 Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. In *SIGGRAPH Asia*, 2011.

44 Chris Stucchio. Type-safe vector addition with dependent types, December 2014. `https://www.chrisstucchio.com/blog/2014/type_safe_vector_addition_with_dependent_types.html`.

45 Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 1997.

46 Unity game engine. `https://unity3d.com`.

47 Unreal game engine. `https://www.unrealengine.com/`.

48 Rui Wang, Xianjin Yang, Yazhen Yuan, Wei Chen, Kavita Bala, and Hujun Bao. Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics*, 33(6), November 2014.

49 Steven Wittens. ShaderGraph: Functional GLSL linker. `https://github.com/unconed/shadergraph`.

Search for Program Structure[*]

Gabriel Scherer

PRL Group, Northeastern University, Boston, MA, USA
gabriel.scherer@gmail.com

Abstract

The community of programming language research loves the Curry-Howard correspondence between proofs and programs. Cut-elimination as computation, theorems for free, `call/cc` as excluded middle, dependently typed languages as proof assistants, etc.

Yet we have, for all these years, missed an obvious observation: "the structure of *programs* corresponds to the structure of proof *search*". For pure programs and intuitionistic logic, more is known about the latter than the former. We think we know what programs are, but logicians know *better*!

To motivate the study of proof search for program structure, we retrace recent research on applying *focusing* to study the canonical structure of simply-typed λ-terms. We then motivate the open problem of extending canonical forms to support richer type systems, such as polymorphism, by discussing a few enticing applications of more canonical program representations.

1998 ACM Subject Classification F.3.3.4 Type Structure, F.4.1.2 Lambda Calculus and Related Systems, F.4.1.8 Proof Theory

Keywords and phrases programs, proofs, focusing, canonicity

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.15

1 Introduction

1.1 Canonical representations for arithmetic expressions in one variable

To introduce the idea of *canonical representation* of programs, let us start with an example in a simpler domain, simple arithmetic expressions over one fixed variable x.

$$\text{expression} \qquad a, b \quad ::= \quad n \in \mathbb{N} \mid a + b \mid a - b \mid a \times b \mid x$$

Suppose we are interested in the *meaning* of expressions, where the meaning of a is taken to be the evaluation function that takes in input the value of the variable x and returns the value of a with this parameter choice. You may think of inputs and outputs as natural numbers – or elements of a given ring.

Our presentation of expressions above has some nice properties, in particular it makes it obvious that the set of expressions is closed over addition and multiplication; it is easy to compose expressions from smaller expressions. But it also admits pairs of expressions that are syntactically distinct but have the same meaning, such as $2 + 2$ and 4, or $(a + b) \times c$ and $a \times c + b \times c$. We call these pairs *redundancies*.

There exists another common representation of these expressions: by applying simplification rules that preserve meaning, we can turn such expression a into a *polynomial* such as, for example, $2x^3 + 4x - 1$, which can be described formally as either 0 or a non-empty sum

[*] This research was supported in part by the National Science Foundation (grant CCF-1422133).

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 15; pp. 15:1–15:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$\sum_{0 \leqslant k \leqslant d} c_k x^k$ of *monomials* of the form $c_k x^k$ with $c_k \in \mathbb{Z}$ and $c_d \neq 0$, for a natural number d called the *degree* of the polynomial.

We see our arithmetic expressions and the polynomials as two different *representations* of the same sort of objects – their meanings as evaluation functions. Polynomials, as a syntactic representation, have less redundancies than our expressions: $2+2$ and 4 are distinct expressions, but they are represented by the same polynomial $4x^0$. When a representation has less redundancies than another, we say that it is *more canonical.* In fact, one can easily show that polynomials have no redundancies at all: if two polynomials are syntactically distinct, they have distinct meanings. When a representation has this property, we say that it is *canonical.*

A more canonical representation has more structure: its definition encodes more of the meaning of the objects being represented. A canonical representation is very rigid, a lot of information about the object is apparent in its syntax, which often makes it easier to operate on it. For example, it is non-obvious whether an arithmetic expression a is constant (for all values of x), while this question can be easily decided for polynomials – it is constant when it is 0 or its degree is 0. In general, polynomials are much more convenient to manipulate for virtually any application, and they are the ubiquitous choice of representation when studying these objects.

1.2 Canonical representations of simply-typed λ-terms and its applications.

In this article, we discuss the design and use-cases of canonical representations of programs expressed in typed λ-calculi, which are more complex than arithmetical expression: finding a good notion of canonical representation can already be challenging.

Because canonical representations reveal so much, in their structure, of the meaning or *identity* of the programs, we should expect them to play a central role in answering many questions about programs. However, until recently theoretical difficulties hampered such practical applications.

In Section 2 (Theory), we present a recent brand of work that brought a much better understanding of canonical forms in the simply-typed case, inspired by ideas from *logic* and *proof theory*, in particular *focusing* and *saturation.* This inspiration is an instance of the Curry-Howard correspondence between proofs and programs that is different in nature from the its previous use-cases that are familiar to functional programmers, and closer to some formal approaches to logic programming. We then discuss why extending these representations outside the simply-typed systems – to type systems with polymorphism, closer to realistic programming languages or proof assistants – is difficult and interesting future work.

In Section 3 (Practice), we discuss potential applications of canonical representations of typed programs.

- Full abstraction, discussed in Section 3.1. Full abstraction, as a formal property of a translation from a source to a target language, requires that the translation preserves the equivalence between source programs. We discuss how having canonical representations on the source gives surprisingly strong full abstraction results; rather than a practical application, this gives new ways to think about full abstraction results and their consequences, by giving a new family of fully-abstract translations that behave in a fairly different ways from those obtained through more typical proof techniques.
- Equivalence checking, discussed in Section 3.2. Being able to mechanically check for equivalence opens the door to many interesting practical applications, such as automati-

cally verifying that a "refactoring" change indeed preserved the meaning of the program as intended – a somewhat tedious task that human reviewers currently have to perform – and solving some ambiguity situations or conflicts arising from a "diamond inheritance" situation. Because we know that for general programming language the question of equivalence fatally becomes undecidable, we must be careful to consider applications where "time out" is an acceptable answer.

- Program synthesis, studied in Section 3.3. Type-directed program synthesis algorithms try to enumerate all expressions of a certain type until they find one that satisfies certain user-provided conditions, such as input-output pairs or unit tests. More canonical representations eliminate redundancies – two syntactically distinct expressions of the same program behavior – so using them would seem very beneficial for program synthesis, by reducing the search space of program expressions to search. In fact, as we will show, existing work on program synthesis used some simplfications presented as heuristics, that are all instances of the simplifications leading to focusing-based more canonical representations.

  On the other hand, canonical representations may also require additional book-keeping, and to a certain point decrease raw efficiency of term enumeration. We propose studying these situations where de-normalization is desirable, starting from canonical representations, rather than studying partial heuristics starting from naive program representations.

2 Theory

2.1 Functional and logic programming

We have lived for too long in a world where logical justifications for programming language research were separated in two disconnected areas:

- *Functional programming*, whose computational behavior is given by term *reduction*, corresponding in logic to *cut-elimination* or in general elimination of detours in proofs.
- *Logic programming*, whose computational behavior is given by *proof search*.

It is natural for functional programmers to think about *complete* proofs that correspond to the program terms they are familiar with. They study relations between proofs or between programs, typically by a *reduction* relation that expresses computation, and an *equivalence* relation that expresses indistinguishability.

Curry-Howard then transfers intuition back and form between *program*-formers with a computational interpretation and *proof*-formers with an interpretation as a reasoning principle. Some success stories include relating control operators to classical reasoning principles [12], functional-reactive or event-driven programming to linear temporal logic [13, 23], session types to linear logic propositions [3], and building proof assistants with convenient computational principles out of advanced dependent type theories (Agda, Coq, LF, PRL...).

Logic programming considers *partial proofs* – derivations with some unfilled subgoals left to prove – to describe the state of a computation that evolves by proof search [18], either stopping when a complete proof is reached or enumeration all possible completions. *Proposition*-formers embed certain search principles that give rise to new computational behavior, possibly bound to the choice of well-designed search strategies. Besides proving a rich generalization of Prolog that is pleasing to proof theorists and type theorists alike, success stories include using linear logic to specify concurrent and distributed systems [16, 26] or to study interactive fiction gameplays and interactive storytelling [17].

We encourage functional programmers to start caring about proof search as well. When logicians propose a new strategy for proof search in a given logic, it avoids traversing redundant derivations of the same proof. In terms of programming, it removes redundant expressions of the same program. This lets functional programmers better understand the structure and identity of programs, by suggesting more canonical representations.

2.2 Identity in logic and programming

We make a careful distinction between a *program*, that is a specific behavior that the user has in mind when she implements it in a programming language, and its *expression* as a specific term in a given system of *representation* – defined by a syntax, typing rules, etc. A program can have arbitrarily many expressions; we say that two expressions are *equivalent* if they represent the same program. For example, refactoring is the process of moving from one expression of a program to another, that is judged more amenable to further modification.

In comparison to the arithmetic expressions of Section 1.1, programs (program behaviors) correspond to evaluation functions, and expressions correspond to terms, either in the syntax of simple arithmetic expressions – one choice of representation – or as polynomials – another choice.

▶ **Definition 1** (Canonicity). A representation T is *more canonical* than a representation S if they represent the same programs, and there is a mapping $\lfloor \_ \rfloor : T \to S$ such that the equivalence classes of T (the maximal sets of expressions of the same program) are mapped by $\lfloor \_ \rfloor$ to subsets of the equivalence classes of S. For example, β-normal λ-terms are more canonical than arbitrary λ-terms. A representation T is *canonical* if each equivalence class is a singleton – all equivalent representations are syntactically equal. For example, if you take a reasonable definition λ-calculus with integers, and restrict yourself to closed terms of type integer, then β-normal forms are canonical.

For programmers there is usually an extremely clear definition of the behavior of an expression, and therefore of what it means for two expressions to be equivalent. It may depend on what they decided to observe – they may, for example, ignore or pay attention to time and memory consumption – but is non-controversial as soon as the observable are agreed upon.

Logicians also make a distinctions between the expression of a formal proof and the mathematical proof it represents, and may consider that two derivations, two formal proofs are "morally the same". But how to define this distinction is not at all obvious: ordinary users of proofs do not consider their *behavior*, they are merely satisfied that they exist at all and could not care less about their formal identity.

How to explain, then, the following state of affairs? The identity of proofs has been studied for a long time by logicians, who proposed many different means of representing a mathematical proof, meant to better capture its identity, to strive at canonicity: natural deduction, sequent calculus, hypersequents / deep inference, focused sequents, tableaux, connection matrices, proof nets, etc. On the other hand, λ-terms reign as the ubiquitous representations of functional programs and, besides normalization of closed programs, the question of choosing alternative means of expression is rarely considered.

▶ Remark. We propose a tentative explanation – an excuse: canonicity is bad for programming. The choice of a highly non-canonical representation of programs allows programmers rich means of expressions of the same program. Two distinct expressions of the same program may correspond to stylistic difference, or an expression of intent in the way the program being worked on will evolve over time. Non-canonicity may be essential for flexibility, clarity and modularity. This is similar to the fact that human mathematicians do, in fact, use products $P \times Q$ of arbitrary polynomials P, Q when it is what they want to express, despite this not within the grammar of polynomials themselves – the product is only a simple arithmetic expression, which is simpler to *produce*. Yet, just as polynomials, if we want to write tools to *consume* programs, better understanding canonical representations can be essential.

2.3 Focusing for canonicity of simply-typed effectful λ-calculus

Focusing is based on the logical notion of *invertibility*. An inference rule is *invertible* when applying it during proof search cannot get you stuck: its premises are provable if and only if its conclusion is provable. Consider the logical rules corresponding to construction of functions, pairs and sums:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2} \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2}\, i \in \{1,2\}$$

The rule for functions and pairs are invertible, but the rule for sum is not: using it means making a choice, and it is possible to get into a dead-end – if you try to prove A_1 but only A_2 was provable. On the side of destructors, the opposite is true: deciding to apply a function may get you in a dead-end if you don't know how to build an argument for it, while eliminating a sum merely considers two possible outcomes, losing no information.

Focusing, originally introduced for linear logic in Andreoli [1], is a series of conditions on proofs: some proofs are valid *focused proofs*, and other are not. First, if invertible rules can never get you in a dead-end, you can make them mandatory: focusing forces them to be applied eagerly, as much as possible. Second, once you can only apply non-invertible rules, the user has to select a formula (of the context or the goal), and focused imposes that apply non-invertible rules be applied to it as long as possible. For example, if the goal is $(X_1 + X_2) + (Y_1 + Y_2)$, focusing demands not only that they make a (non-invertible) choice for the head sum connective, but also choose between one of the X_i (or Y_i) by doing a nested sum introduction. As another example, if a function variable $f : X \to Y \to Z$ is in the context, selecting it means that one has to pass all arguments at once, passing both a X and a Y argument – stopping after applying X to work on another formula would be an invalid focused proof. Intuitively, using this function is only useful to get the final result Z, so it is always possible to either never apply it (if Z is not needed), or delay the application of X until Y is also available.

Zeilberger [31] proposed to consider focused proofs as a *syntax* for a programming language, guided by the observation that focusing could justify the dual semantic restrictions on polymorphism in effectful call-by-value and call-by-name languages – this works explains the value restriction for strict intersection types and the context restriction for lazy union types appearing in Dunfield and Pfenning [8], and the explanation was extended to second-order universal and existential polymorphism in Munch-Maccagnoni [19].

The idea of invertibility is one way to understand why adding sums make equivalence harder. We say that a type connective is *positive* if its right introduction rule is non-invertible, and *negative* otherwise: $(\to, \times, 1)$ are negatives, and $(+, 0)$ are positive. It is easy to decide equivalence of the simply-typed λ-calculus with only connectives of one polarity: we previously remarked that it is easy to define canonical forms in the negative fragment $\mathsf{STLC}(\to, \times, 1)$, but it is equally easy in the positive fragment $\mathsf{STLC}(+,0)$. It is only when both polarities are mixed that things become difficult.

A key result of Zeilberger [32, Separation Theorem, 4.3.14] is that a focusing-based presentation of the simply-typed λ-calculus is *canonical* in the *effectful* setting where we assume that function calls may perform side-effects – at least using the specific reduction strategy studied in CBPV [15]. Two syntactically distinct effectful focused program expressions are observationally distinct – the canonicity proof relies on two distinct error effects, to distinguish evaluation order, and an integer counter to detect repeated evaluation. The fact that any λ-term can be given a focused form comes from a *completeness* theorem, the analogous of completeness of focusing as a subset of logical proofs. However, this syntax is not canonical anymore if we consider the stronger equivalences of pure functional programming, where duplicated or discarded computations cannot be observed.

Let us write P, Q for positive types, types that start with a positive head connective, and N, M for negative types, that start with a negative head connective. In a λ-term in focused form with types of both polarities – see a complete description in Scherer [28, Chapter 10] – a non-invertible phases can be of two forms which we shall now define. It can start with a *positive neutral* $(p : P)$, which is a sequence of non-invertible constructors of the form σ_i _ for positive types, they commit to a sequence of risky choice to build the value to return. Or it can start with a *negative neutral* $n[y : N]$, that performs a series of function applications $n\ p$ or pair projections $\pi_i\ n$ to a variable $y : N$ from the context. Such a negative neutral term, if it is of the type of an atomic goal $(n : X)$, is returned unchanged. If it is of some positive type P, it is bound to a variable name to be decomposed decomposed by the following invertible phase: $\texttt{let}\ (x : P) = n[y : N]\ \texttt{in}\ \dots$. One way to think of these two choices is that positive neutrals $(p : P)$ build a part of the output value, while negative neutrals $n[y : N]$ observe a part of the input environment.

2.4 Saturation for local canonicity of simply-typed pure λ-calculus

Focusing enforces a form of *backward* search structure on canonical program representations: the invertible structure of terms is purely determined by the types of the goal and the context. To obtain canonical forms for pure terms, Scherer and Rémy [30] had to use intuitions from proof search again, by forcing the non-invertible part of terms to have the structure of a *forward* search: at the end of each invertible phase, try to *saturate* the context by making as much observations $\texttt{let}\ (x : P) = n[y : N]\ \texttt{in}\ \dots$ as possible, returning a value $(x : P)$ only when introducing new observations does not teach us anything new. We call terms that follow this discipline *saturated terms* – for a more detailed explanation of saturation, see Scherer [28, Chapter 11].

There is no redundancy among saturated terms, but the completeness theorem is weaker than for focusing alone. To have the saturation process terminate, one has to fix in advance a finite set of possible observations for each context. For any such choice, only finitely many λ-terms can be given a saturated normal form – and conversely, for any finite set of λ-terms there exists a saturation strategy that allows to represent them as saturated normal form. This is enough for practical applications – testing whether a type has a unique inhabitant, testing whether two terms are equivalent (two is finitely many) – but still a technical limitation.

3 Practice

These results on the simply-typed λ-calculus are very encouraging, but they do not provide canonical forms in presence of *parametric polymorphism*, which gives even stronger equational principles. In other words, canonical forms for practically used programming languages are still out of reach.

We argue that building them would give us a much better understanding of what our programs *are*, and open the door for important applications.

1. Easy ways to build fully abstract translations.
2. Program synthesis.
3. Equivalence procedures.

3.1 Full abstraction

A translation from one program representation to another, a compiler for example, is said to be *fully abstract*[1] if two programs are observationally equivalent if and only if their translations are observationally equivalent.

Full abstraction is a strong property that gives a lot of information of a compilation process: it tells us that any equational reasoning at the source level remain valid at the target level. One can think of it as a usability property (knowing the source language is enough) or a security property (the compiler protects programs from being separated by attacks at the target level)

As a result of its strength, it is a difficult property to establish : many translations that we intuitively believe are fully-abstract have only been proved so very recently, or are not known to be. For example, fully-abstract translations from the simply-typed λ-calculus to the untyped λ-calculus or from the simply-typed λ-calculus (with recursive types) to System F were only established in 2016 [7, 21], and type-erasing fully-abstract translations from System F are an open problem. Yet we can, with apparent ease, provide you with very strong results.

▶ **Theorem 2** (Full abstraction for free!). *Take any finite subset S of the pure simply-typed λ-calculus with functions, pairs, sums and units. Take any system T in which any λ-term can be embedded: the untyped λ-calculus, System F, impure System F with references and* `call/cc`*, the Calculus of Constructions... There is a fully-abstract translation from S to T.*

Proof. For any pure λ-term e in S, let us define $\lfloor s \rfloor$ its canonical form as defined in Scherer [28], seen as a λ-term. We assumed that the λ-calculus embeds into T. We translate s into the embedding of $\lfloor s \rfloor$.

By definition of canonicity, if two terms s, t are observationally equivalent, then $\lfloor s \rfloor$ and $\lfloor t \rfloor$ are syntactically equal, and in particular their embedding in T is the *same* program. In other words, this translation is fully abstract. ◀

This result is a bit puzzling[2]. For example, we are not requiring the type $A \to B$ of pure functions in the pure simply-typed λ-calculus to be translated in T into a type of pure functions: the embedding into ML with non-termination and references, for example, would return a term at the type of ML impure functions. Yet the translation is fully-abstract.

Full-abstraction proofs are delicate because they are often built on the ability to *back-translate* contexts of the target language into the source language – completely or approximately. The proof technique established in Devriese, Patrignani, and Piessens [7] might be able to back-translate contexts of the calculus of (inductive) construction, but it sounds like a daunting amount of work. Instead, we paid the cost of coming up with canonical

[1] Full abstraction was originally introduced as a quality criterion for denotational semantics instead of syntactic program translation. We mention this in Section 4.2 (Game semantics).

[2] Note that it is not an immediate consequence of strong normalization for the simply-typed λ-calculus, otherwise it would extend to System F as well.

representations in the first place; once you have them for your source language, you get fully-abstract translations for free!

3.2 Equivalence procedures

Canonical representations also give procedures to decide equivalence for a programming language: to check whether two representations are equivalent, one can check whether their canonical forms are syntactically equal. Conversely, it is often – but not always – the case that equivalence procedures can be read back as a way to describe canonical forms for a language.

We know surprisingly little about deciding program equivalence. Equivalence in presence of non-empty sum types has remained an advanced, somewhat active research topic since it was proved decidable in Ghani [11], and equivalence in presence of the empty type (zero-ary sums) was decided only recently [29]. We know that observational equivalence for System F is undecidable, but while the (equally undecidable) problem of type inference has been actively studied since at least Pfenning [24], we know of no research on equivalence algorithms for polymorphic calculi.[3] Similarly, an impressive amount of effort has been invested in automatically checking that an implementation respects a specification (despite the undecidability barriers), but almost none went into automatic checking of implementation equivalence – as if this question jumped straight from automata to process calculi, without ever touching functional programs.

We hope that studying proof search and canonical representations in presence of polymorphism will stimulate work on semi-decidable equivalence algorithms for powerful type systems. The applications of equivalence procedures are everywhere, and we will mention some of them:

1. Robust equivalence checking opens the door to automatic verification of refactoring code transformations. Refactoring edits are typically designed to be easily reviewable by human programmers, so they should be within reach of an equivalence algorithm, even if it is very incomplete. Note that this very quickly involves the tricky equivalence of sums: moving a condition test earlier or later in a program is an instance of β-expansion on booleans (a sum type).
2. ML module systems have generative and applicative functors, the later being characterized by the fact that two independent applications of the same functor to the same parameter return compatible modules. This means that two independent libraries that made the same instantiation choice are compatible, instead of having to be modified to depend on a common functor applications. However, equality of module parameters, "static equivalence", is implemented in a very syntactic and restrictive way, essentially requiring that both independent libraries refer to a shared definition of the *parameter*. To liberate independent libraries, we should let each define their parameter, and check that they are *equivalent*.
3. The question of equivalence is also underlying the problem of *coherence* of implicit elaboration mechanisms such as type-classes and implicits. To remain predictable, language designs try to guarantee that each implicit elaboration problem has a unique solution across the user code – either by imposing somewhat ad-hoc priority/ordering restrictions or by imposing non-modular conditions, such as imposing all implicit instances

[3] One notable exception is the work of Bernardy, Jansson, and Classen [2] on polymorphic testing, which is secretly about program equivalence.

to be declared at the toplevel. If we had reliable equivalence checking, we could for example allow local declarations under the condition that local instances remain coherent with outer instances – checking equivalence for all pairs of elaboration chains for the same instance – or automatically determine which global instances need to be locally disabled to preserve coherence.

4. In general equivalence checking is the proper design tool to handle "diamond inheritance" situations. If a declaration inherits from two classes exporting the same method definition, existing languages either impose an arbitrary choice, or fail, or require a renaming; renaming is correct, but cumbersome in the common case where both definitions are equivalent – sometimes in non-trivial ways. This is common for example when modeling mathematical hierarchies: a finite monoid is both a monoid and a finite set, which both exports (non-conflicting) definitions of a carrier set.

3.3 Program synthesis

In Scherer and Rémy [30] we used canonical forms for the simply-typed λ-calculus to answer the following question: when it is the case that a given type is has a *unique* inhabitant modulo program equivalence? This unicity situation is a perfect opportunity for program synthesis, as the code that programmer would have in mind is completely determined by the type information flowing from the context. We presented some interesting examples (unique inhabitants at non-trivial types of `lens` operators), but of course unicity rarely happens in common programs today, only in highly polymorphic library functions, or code written with rich dependent types [27].

More practical-minded work on type-directed program synthesis, such as the recent works of Osera and Zdancewic [22], Frankle, Osera, Walker and Zdancewic [10], and Polikarpova, Kuraj and Solar-Lezama [25], proceeds by enumerating arbitrary many programs at a given type, looking for one that satisfies a user-provided specification – expressed as a set of input-output examples, or a refinement type. Even if the specifications can be woven inside the search procedure to reduce the search space, there is still a combinatorial explosion in the number of synthesized AST nodes that makes scaling to rich programs extremely challenging.

By reducing the redundancy among the programs of a given size, canonical forms provide order-of-magnitude[4] reduction in search space. In fact, many of the experimentally-motivated space reduction heuristics proposed in Osera and Zdancewic [22] are subsumed by focusing.

We propose three ways in which the study of canonical forms could interact and enrich future research on type-directed functional program synthesis:

1. We believe that instead of starting from a λ-term enumeration and carving out to reduce the search space, synthesis algorithms could be usefully expressed as directly from the search procedure for canonical forms – if they are known for the type system at hand – or the most-canonical representation known.
2. On the other hand, sometimes strong canonicity imposes some bookkeeping that can actually degrade search performance – Frankle, Osera, Walker and Zdancewic [10] reports that sometimes they willingly *avoid* η-expansion to reduce term size. Syntactic descriptions of "weakly" focused programs that do not impose full deep inversion has been proposed [20], but we need to build empirical and theoretical understanding of *when* and *why* de-canonicalization is useful for synthesis.

[4] The order-of-magnitude claim comes from Frankle, Osera, Walker and Zdancewic [10].

3. Finally, synthesis algorithms today mix techniques that are easily explained by the *proof search*-inspired point of view we promote in the present article, and others, such as the *predicate interpolation* used in Polikarpova, Kuraj and Solar-Lezama [25], that are not justified in these terms. It would be interesting to find justifications for them in purely logical terms – as logicians have historically have been able to do for various aspects of logic programming, such as forward vs. backward-search [5] or magic sets [4].

4 Advanced topics

4.1 Full abstraction seen differently

In Theorem 2 (Full abstraction for free!) we demonstrated that knowing a canonical representation for a source language lets us build fully-abstract translation to many different target languages with almost no assumption of those targets.

To be clear, we do not currently expect this approach to scale to the realistic languages we are interested to get fully-abstract compilers and translations for, because of the decidability barrier for canonical representations. Getting a *more canonical* representation is useful for the applications mentioned so far, but this approach to full abstraction requires fully canonical representations, and may therefore remain restricted to idealized languages. Nonetheless, in the rest of this section, we explore what we can learn about full abstraction from this extreme setting.

This full-abstraction argument is interesting because it gives a concrete example of the general idea that full-abstraction results from a source to a target language need not be built on back-translation techniques. Most of the known full-abstraction results rely on the ability to back-translate some parts of target terms and contexts – for example those whose type is the translation, in the target type system, of a source type – to the point where practitioners would sometimes conflate full-abstraction with the ability to back-translate. We demonstrate that knowing more about the source equivalence lets you work less on the relation between the source and target languages.

Full abstraction is also often seen as a security property, and considering our proof technique in this light is interesting. If we consider distinguishing two terms as an "attack", the statement of full abstraction says that the translation protects the source terms of any additional distinguishing power (attacks) in the target language; this may be achieved by having the compiler insert some sort of "protective wrapper" around the translated terms, to disable the more low-level (malicious?) features of the target language – see Fournet, Swamy, Chen, Dagand, Strub and Livshits [9] for example.

In our canonicity-based full abstraction result, there is no protective layer (neither static or dynamic) around the terms. Instead of disabling the observation features of the target language, we make sure that each source term will behave as all others equivalent terms under any additional observation from the target.

For example, consider the type $(X \to Y) \to X \to Y \to Y$ in the pure simply-typed λ-calculus and its two inhabitants $\lambda f\, x\, y.\, f\ x$ and $\lambda f\, x\, y.\, y$. To get a fully-abstract translation into a target calculus with side-effects such as input-output, one would typically wrap the two terms into a protective barrier that prevents using side-effects to tell if the f argument is used or not – in this example, the two terms are not equivalent, so they will remain distinguishable, but the protective barrier would be used in any case. This protection could be static, using a type of pure functions in the target language to forbid effects in f, or dynamic, using some kind of IO mocking/reification to block external observers from seeing writes. In our translation, no such protection is used; instead, saturation transforms both terms, in the source language (before translation), into terms of the form $\lambda f\, x\, y.\, \texttt{let}\ y' = f\ x\ \texttt{in}\ \ldots$, where the $\ldots$ are y' for the first program and y for the second. No protective wrapping is necessary anymore: bad f-observing side-effects will make the same observation on both normal forms.

Finally, one may wonder about the efficiency impact of the normalization into a canonical form – in particular the saturation. At the source type $(X \to Y) \to X \to Y \to Y$ in the example above, we turned $\lambda f\, x\, y.\, y$ into $\lambda f\, x\, y.\, \texttt{let}\ y' = f\ x\ \texttt{in}\ y$, introducing an apparently useless computation. In the general case, many such observations of the context may be introduced by saturation – how much depends on the finite set of terms that delimits the observable horizon. However, notice that this price needs only be paid if the function type $X \to Y$ is translated into a type of impure functions in the target. If it is translated to a pure function type, then the unused binding $\texttt{let}\ y' = f\ x$ can be removed after embedding in the target language. In general, saturation may introduce many additional observations, but only the ones that are required for full abstraction to hold need to be preserved; simple type-directed target simplification strategies may be able to convert back to target programs much closer to what non-canonicity-based fully-abstract compiler would produce.

4.2 Game semantics

Some readers may wonder about a relation between *canonicity* of syntactic representations that we propose to discover using *focusing*, and the *full abstraction* property of some denotational semantics obtained through *game semantics*. The comparison is as follows.

Game semantics provide a way to describe programs as *games* or, more precisely, *strategies* (as mathematical objects in a denotational semantics, but they could be given a concrete syntax), that allow to express an extremely broad scope of computational behaviors. In particular, two distinct such objects are distinct behaviors (there is no redundancy), but in general a lot of those objects do *not* correspond to interpretations any program of the language you are starting with: they may exhibit various kind of non-determinism, state, demand sensitivity, all sorts of weird things. Then you progressively carve out subsets of those objects that have nice behavioral properties (for example, innocent strategies), ruling out the strangest behaviors. If you do this enough, you can manage to remove all the non-standard stuff, and the remaining objects are exactly the programs of your language: you have a fully-abstract model.

The path to canonical forms goes in the opposite direction. We start with a syntax that, obviously, contains only expressions of the programs of our language, but a lot of these expressions may describe the same behavior. Then we carve out redundant expressions by proposing more and more canonical representations. If we do this enough, we can manage to remove all the redundancy, and the remaining expressions are exactly the programs of our language: we have a canonical representation.

In the best case, a language is so well-understood that both processes have reached completion: we have canonical representations and fully abstract denotations of it that are in one-to-one correspondence. This has been worked out for propositional linear logic in Laurent [14].

5 Conclusion

Syntax is a powerful tool to study and manipulate program behavior, and we emphasize the problem of finding *more canonical* syntactic representations of our program, that have less redundancy and are thus closer to a canonical description of their behavior.

We claim that this research programme can take heavy inspiration from proof-theoretic results and methods to study the structure of *proof search*, extending a new arm of the Curry-Howard correspondence.

Recent results have already been obtained for simply-typed systems, giving precise descriptions of normal forms for both effectful and pure λ-calculi. The extension to parametric polymorphism, and other advanced type systems (dependent types...) is an open problem that you should consider working on!

Powerful type systems, with their stronger equational reasoning principles, should not only make programming more expressive and safer, they should make it *easier*. We need better programming tools for that to happen, and hopefully more canonical representations are one way to make this happen.

Acknowledgments. Amal Ahmed led an extremely stimulating discussion on the nature and limitation of the full abstraction results provided by factorization through canonical forms in Theorem 2, which we tried to summarize in Section 4.1 (Full abstraction seen differently).

References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

2 Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *ESOP*, 2010.

3 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, 2010.

4 Kaustuv Chaudhuri. Magically constraining the inverse method using dynamic polarity assignment. In *LPAR*, 2010. URL: `https://hal.inria.fr/inria-T00535948`.

5 Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *J. Autom. Reasoning*, 40(2-3), 2008.

6 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP*, 2000.

7 Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully abstract compilation by approximate back-translation. In *POPL*, 2016.

8 Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *POPL*, January 2004.

9 Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *POPL*, 2013. URL: `http://research.microsoft.com/pubs/176601/js-Tstar.pdf`.

10 Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, 2016.

11 Neil Ghani. Beta-Eta Equality for Coproducts. In *TLCA*, 1995.

12 Timothy G. Griffin. A formulae-as-type notion of control. In *POPL*, 1989.

13 Alan Jeffrey. Ltl types frp: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *PLPV*, 2012.

14 Olivier Laurent. Syntax vs. semantics: a polarized approach. *Theoretical Computer Science*, 343(1–2):177–206, 2005.

15 Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *TLCA*, 1999.

16 Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *PPDP*, 2005.

17 Chris Martens. Ceptre: A language for modeling generative interactive systems. In *AIIDE*, 2015.

18 Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 1991.

19 Guillaume Munch-Maccagnoni. Focalisation and Classical Realisability. In *CSL*, 2009.

20 Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot, 2013.

21 Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *ICFP*, 2016.

22 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

23 Jennifer Paykin, Neelakantan R. Krishnaswami, and Steve Zdancewic. The essence of event-driven programming. In *draft*, 2016.

24 Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP*, 1988.

25 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.

26 Anders Schack-Nielsen and Carsten Schürmann. Celf – A logical framework for deductive and concurrent systems (system description). In *IJCAR*, 2008.

27 Gabriel Scherer. Mining opportunities for unique inhabitants in dependent programs. In *DTP Workshop*, 2013.

28 Gabriel Scherer. *Which types have a unique inhabitant?* PhD thesis, Université Paris-Diderot, 2016.

29 Gabriel Scherer. Deciding equivalence with sums and the empty type. In *POPL*, 2017.

30 Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? In *ICFP*, 2015.

31 Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 2008.

32 Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

A Appendix: Noam's history of focusing

Noam Zeilberger's work contains, expressed very clearly, the result that focusing gives a canonical term language for effectful programs – Zeilberger [32, Separation Theorem, 4.3.14]. Interestingly, this aspect is not emphasized in the general work, not is the connection to logic programming (despite the direct connection to Frank Pfenning's work), and the formulation in terms of a Curry-Howard correspondence between term canonicity and proof search is not to be found in his thesis – it seems to be a contribution of the present article.

We tried to guess how Noam Zeilberger ended up working with focusing and its applications to functional programming problems. We assumed that familiarity with focusing came first from the literature on logic programming, and applications to the simply-typed λ-calculus came only later. We asked him, and it turns out we were completely wrong. Most of his reply, lightly edited, is quoted below:

> The original reason I started thinking about evaluation order and pattern-matching was actually to understand intersection and union types. I was intrigued by the paper on "Tridirectional typechecking" by Joshua Dunfield and Frank [8], where they showed that the union-elimination rule requires an "evaluation context restriction" in the presence of effects, dual to the value restriction on interesection-introduction which Rowan Davies and Frank had found was necessary for CBV+effects ("Intersection types and computational effects", [6]). I was not

completely satisfied with the treatment of subtyping in these papers, though. Rowan and Frank's paper explained that the subtyping distributivity law

$$(A \to B) \wedge (A \to C) \leqslant A \to (B \wedge C) \tag{1}$$

from classical intersection type systems is unsound for CBV+effects, while Joshua and Frank's paper showed that the dual rule for unions

$$(A \to C) \wedge (B \to C) \leqslant (A \vee B) \to C \tag{2}$$

is unsound for CBN+effects (they didn't mention this explicitly, but it can be deduced from their examples showing the necessity of an evaluation context restriction on union-elimination). The solution in both papers was to simply drop these principles (1) and (2), but it's a fact that (1) is *sound* for CBN even in the presence of effects, while (2) is sound for CBV+effects. What I noticed (which really is kind of folklore) was that it's possible to reduce these questions of subtyping to typing by taking an "identity coercion" interpretation of subtyping. For example, it is possible to "derive" principle (1) by typing the eta-expansion of a function variable:

$$f : (A \to B) \wedge (A \to C) \vdash \texttt{fn x => f(x)} : A \to (B \wedge C)$$

The point is that the corresponding typing derivation goes away if we place a value restriction on intersection-introduction, because the subexpression `f(x)` is not a value. My hypothesis was that it should be derive *all* of the subtyping laws which are sound for a given evaluation order by typing the appropriate identity coercion, but this places constraints on the design of the type system, that the typing rules be "sufficiently complete". In particular, subtyping rules such as

$$\begin{array}{ll} (A * B) \wedge (A * C) \leqslant A * (B \wedge C) & \quad (A + B) \wedge (A + C) \leqslant A + (B \wedge C) \\ A * (B \vee C) \leqslant (A * B) \vee (A * C) & \quad A + (B \vee C) \leqslant (A + B) \vee (A + C) \end{array}$$

seemed to require inversion principles based on pattern-matching to be built into the typing rules. It took a few more steps to get to focusing. Here's a followup mail I wrote to Robert Harper on June 1 2006:

> I think I understand what you were getting at earlier – would you find it more acceptable to speak of call-by-value vs call-by-name *connectives*? Or maybe strict vs lazy connectives? Part of what this work is about is that we have to treat (for example) strict product and lazy product as different logical connectives, with different introduction and elimination rules. You can pattern-match against values of "strict type", and pattern-match against *covalues* of "lazy type". If you are familiar with the idea of focusing, strict (aka "positive") connectives are synchronous on the right and asynchronous on the left, whereas lazy (aka "negative") connectives are synchronous on the left and asynchronous on the right.
>
> [...] It may not be necessary to segregate distinct "call-by-value" and "call-by-name" type systems as I did in the version of the paper you have, and instead just take their union as a system mixing strict and lazy connectives – I haven't worked that out.

AP: Artificial Programming

Rishabh Singh[1] and Pushmeet Kohli[2]

1 **Cognition Group, Microsoft Research, Redmond, WA, USA**
`risin@Microsoft.com`

2 **Cognition Group, Microsoft Research, Redmond, WA, USA**
`pkohli@microsoft.com`

Abstract

The ability to automatically discover a program consistent with a given user intent (specification) is the holy grail of Computer Science. While significant progress has been made on the so-called problem of Program Synthesis, a number of challenges remain; particularly for the case of synthesizing richer and larger programs. This is in large part due to the difficulty of search over the space of programs. In this paper, we argue that the above-mentioned challenge can be tackled by learning synthesizers automatically from a large amount of training data. We present a first step in this direction by describing our novel synthesis approach based on two neural architectures for tackling the two key challenges of *Learning to understand partial input-output specifications* and *Learning to search programs*. The first neural architecture called the *Spec Encoder* computes a continuous representation of the specification, whereas the second neural architecture called the *Program Generator* incrementally constructs programs in a hypothesis space that is conditioned by the specification vector. The key idea of the approach is to train these architectures using a large set of (ϕ, P) pairs, where P denotes a program sampled from the DSL L and ϕ denotes the corresponding specification satisfied by P. We demonstrate the effectiveness of our approach on two preliminary instantiations. The first instantiation, called Neural FlashFill [29], corresponds to the domain of string manipulation programs similar to that of FlashFill [13, 14]. The second domain considers string transformation programs consisting of composition of API functions. We show that a neural system is able to perform quite well in learning a large majority of programs from few input-output examples. We believe this new approach will not only dramatically expand the applicability and effectiveness of Program Synthesis, but also would lead to the coming together of the Program Synthesis and Machine Learning research disciplines.

1998 ACM Subject Classification D.1.2 [Programming Techniques] Automatic Programming, I.2.2 [Artificial Intelligence] Program Synthesis, I.5.1 [Artificial Intelligence]: Neural Nets

Keywords and phrases Neural Program Synthesis, Neural Programming, Neural FlashFill

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.16

1 Introduction

The impact of computing in shaping the modern world cannot be overstated. This success is, in large part, due to the development of ever more revolutionary and natural ways of specifying complex computations that machines need to perform to accomplish tasks. Despite these successes, programming remains a complex task – one which requires a long time to master. Computer Scientists have long worked on the problem of program synthesis *ie* the task of automatically discovering a program that is consistent with a given user intent (specification) [38, 6, 22]. The impact of Program Synthesis is not just limited to democratizing programming, but as will see below, it allows us to program computers to accomplish tasks that were not possible earlier.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 16; pp. 16:1–16:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Considering the field of machine learning from the perspective of Program Synthesis, the specification takes the form of input-output examples (training data), and programs are restricted to certain restricted languages. For instance, linear regression involves synthesizing programs that involve a single linear expression, neural networks involve synthesizing programs that are composed of sequence of tensor operations, and decision trees involve synthesizing programs composed of nested if-then conditions. The availability of large scale training data and compute, along with the development of approaches to search-over the afore-mentioned restricted families of programs have allowed machine learning based systems to be extremely successful in many domains. This is particularly true in the case of perceptual tasks such as image [23] and speech understanding [18], where machine learning has led to dramatic recent breakthroughs in the form of development of systems whose abilities go beyond humans themselves [16, 40]. While the programs learnt using machine learning approaches have been very effective, they are restricted by limited domain specific languages they employ. Moreover, the programs learnt by such techniques are hard to interpret, verify, and correct.

Program Synthesis has also seen a substantial amount of research in the Programming Languages community. While earlier approaches to tackle this problem were mostly based on deductive reasoning [25, 26, 27], several approaches based on inductive reasoning have been recently proposed. These new approaches have exploited advances in computational power, algorithmic advances in constraint-solving, and application-specific domain insights [1]. These approaches can be broadly divided into four categories based on the search strategy they employ: (i) enumerative [39], (ii) stochastic [35], (iii) constraint-based [36, 37], and (iv) version-space algebra based [14, 30]. The enumerative approaches enumerate programs in a structured hypothesis space and employ smart pruning techniques to avoid searching a large space. The stochastic techniques use a cost function to induce a probability distribution over the space of programs conditioned on the specification, which is used to sample the desired program. The constraint-based techniques encode the search problem in low-level SAT/SMT constraints and solve them using off-the-shelf constraint solvers. Finally, the Version-space algebra based techniques learn programs in specialized DSLs to perform an efficient divide-and-conquer based search.

While there has been a significant progress in synthesizing richer and larger programs, these synthesis approaches suffer from a number of challenges. The first and most daunting challenge is the search problem for searching over the large space of programs. Modern approaches try to overcome this problem by carefully designing the DSL [14]. Not only this approach restricts the expressivity of the language but it is also extremely time consuming as the DSL designer needs to encode several domain-specific heuristics in terms of DSLs, pruning strategy, cost function etc. Second, these synthesis algorithms do not learn from previously solved tasks, i.e. they do not evolve and get better over time. Finally, these algorithms are typically designed to handle one form of specification (such as input-output examples or partial programs). It is difficult to handle multi-modal specifications such as combination of input-output examples, natural language description, partial programs, and program invariants all together as one combined specification.

In this paper, we argue that the above-mentioned challenges can be tackled by learning synthesizers automatically from a large amount of training data. We believe this new approach will dramatically expand the applicability and effectiveness of Program Synthesis and has the potential to have a massive impact on the development of complex intelligent software systems of the future. We present a first step in this direction by describing a novel synthesis approach based on two neural architectures for tackling the two key challenges of *Learning to understand specifications* and *Learning to search programs.* The first neural architecture called the *Spec Encoder* computes a continuous representation of the specification, whereas the second neural architecture called the *Program Generator* incrementally constructs programs in a hypothesis space that is conditioned by the specification encoding vector. The key idea of the approach is to train these architectures using a large set of (ϕ, P) pairs, where P denotes a program sampled from the DSL L and ϕ denotes the corresponding specification satisfied by P.

We present two preliminary instantiations of this approach. The first instantiation, Neural FlashFill [29], corresponds to the domain of string manipulation programs similar to that of FlashFill [13, 14]. The second domain considers string transformation programs consisting of composition of API functions [4]. The specification mechanisms in both of these domains is input-output string examples and we develop an R3NN (Reverse-Recursive-Reverse Neural Network) to incrementally generate program trees in the DSL that is conditioned on the input-output examples. We show that even with preliminary encodings, the neural system is able to perform quite well in learning a large majority of programs from few input-output examples. We finally conclude with some challenges and exciting future directions.

Related work on Learning to Program. There have some recent proposals to use neural network based encodings for synthesizing programs. Neural RAM [24] constructs an end-to-end differentiable model representing the sequential compositions of a given set of modules (gates), and learns a controller defining the compositions of modules to obtain a program (circuit) that is consistent with a given set of examples. DeepCoder [2] embeds input-output examples of integers to learn a distribution over likely functions that would be useful for the task, and uses off-the-shelf synthesis techniques such as enumerative and SKETCH [36] to learn the program. Terpret [9] and Forth [34] are probabilistic programming languages that allow programmers to write high-level programs with holes, which are then completed using a gradient-descent based search. While these systems have shown a lot of promise, they typically learn simple short programs, need a lot of compute resources per synthesis task, and do not learn how to perform efficient search.

There has been a number of recent proposals of neural architectures to perform *program induction*, where the goal is to learn a network that can learn the functional behavior of a program. These architectures are inspired from computation modules (Turing Machines, RAM, GPU) [12, 24, 21, 33, 28] or common data structures such as stacks [20]. The key idea in these approaches is to represent the operations in a differentiable form, which allows for efficient end-to-end training of a neural controller. However, unlike our approach that learns interpretable programs, these approaches learn only the program behavior. There is also an exciting line of work on learning probabilistic models of code from *big code* [32, 5, 17], which are used for different applications such as variable and method name inference, code-completion, generating method summaries etc.

2 Overview of Neural Program Synthesis

The overview of our approach is shown in Figure 1. We assume that the hypothesis space of programs is given in the form of grammar similar to that of SyGuS [1] and the goal is to find a derivation (program) from the grammar that is consistent with the specification provided by a user. There are two key neural modules: (i) Specification Encoder (Learning to understand specification) and (ii) Program Generator (Learning to search over programs). The specification encoder generates a continuous vector representation of specification (potentially in multiple formats). The program generator takes the specification vector and the DSL L

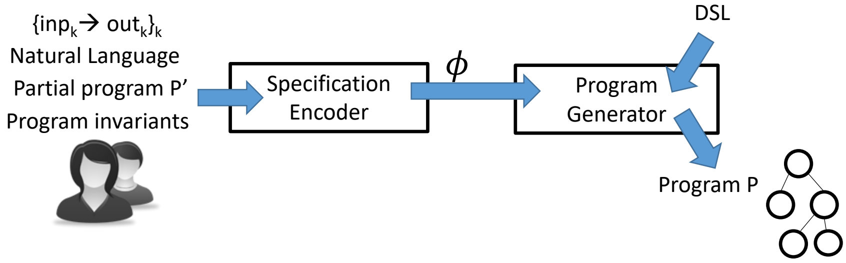


Figure 1 An overview of the neural program synthesis approach consisting of two components. The specification encoder generates a distributed representation of the specification, which is then used to condition the program generator module that incrementally generates programs in a DSL.

as inputs, and generates a program $P \in L$ that conforms to the specification. We briefly describe the two modules and different challenges associated with them.

2.1 Neural Architectures

Specification Encoder: The challenge of designing a specification encoder is that it needs to handle multiple forms of specifications, but at the same time it also needs to maintain a differentiable representation that can be efficiently learnt. Some possible specification modalities we aim to support in our architecture include input-output examples, natural language descriptions, partial programs, and program invariants. The challenge in encoding input-output examples is that the set of examples can be of variable size and each example can be of different length. The encoding should also be invariant to the order of examples. Moreover, the inputs and outputs can be of different types such as strings, integers, arrays, etc., which adds another challenge to the encoder. For encoding partial programs, the encoder needs to take into account the tree structure of parse tree of the program compared to the simpler sequence structure of base types.

Program Generator: The program generator component needs to generate a program from the DSL that is consistent with the specification vector obtained from the specification encoder. One approach is to model program semantics in the continuous domain and have the neural network perform optimization over that space. Since programs are typically discrete in nature, i.e. a small change in inputs (or in programs) can lead to big changes in output, embedding continuous representation of program semantics is challenging. Another approach can be to instead learn a controller that selects different choices from the hypothesis space (DSL) to construct a program. This controller needs to encode partial program trees and learn how to incrementally expand the trees. Depending on the complexity of the DSL, it may additionally need to encode program states, memory, and stack information.

2.2 Training the Neural Architectures

One of the reasons for success of neural networks recently has been the availability of large amounts of training data. While for domains such as natural language processing and computer vision, acquiring good labeled data is a challenging task, it is relatively easier for our domain of program synthesis. Given a DSL, we use a program sampler to sample million of programs from the language and order them using different metrics such as program size, operator usage etc. Given a sampled program, we can then design a rule-based approach to generate corresponding specifications that are consistent with the sampled program.

However, there are also several challenges in generating the training data. First, sampling programs from a context-free grammar is relatively straightforward, but sampling programs from a more complex stateful grammar is more complex as the sampler needs to ensure that the program is well-formed, e.g. there are no variable usage before define or array out-of-bound errors. Second, the generation of consistent specification for a program can also be challenging as the sampler needs to ensure that the inputs satisfy the pre-conditions of the sampled programs. For generating natural language descriptions, one challenge is that there might be multiple different ways to specify the functionality.

Given the training data of input-output examples together with their corresponding programs, the specification encoder and the program generator can be trained in an end-to-end supervised manner using the backpropagation techniques. The goal of training is to learn best parameters for the two neural architectures that result in learning consistent programs for as many training points as possible. There are also several choices for cost function for optimizing the training loss. One choice can be to ensure that the learnt program is syntactically similar to that of training program. This cost function is easy to optimize since we can use the complete supervision for each individual component of the program. However, this cost function also penalizes many good programs that are syntactically different but semantically equivalent with respect to the training program. This is especially true in case of inductive specifications such as input-output examples, where there might be multiple consistent programs. The cost function can be enhanced to optimize over the outputs of the learnt programs instead of their syntactic structure, but this makes the optimization algorithm harder since we no more have intermediate supervisory signal, and we can only check for program correctness after constructing the complete program.

2.3 Synthesizing programs from the learnt architectures

After learning the neural architectures, we can use them to learn programs given some specification. The specification encoder encodes the specification using the learnt parameters and the specification vector is fed to the program generator to construct a consistent program. Another interesting property of the program synthesis domain unlike other domains such as NLP and vision is that we can execute the learnt programs and check if they are correct with respect to the specification at test time, which allows us multiple choices during program generation. We can either generate the 1-best program using the program generator, or we can use the learnt distributions over the grammar expansions to instead sample and generate multiple programs until finding one that is consistent with the given specification. However, checking for correctness might not be feasible for all types of specifications such as Natural Language specifications.

3 Neural FlashFill and NACIO

We present a preliminary instantiation of our framework on two domains – Neural FlashFill and Nacio. The Neural FlashFill system learns regular expression based string transformations in a DSL similar to that of FlashFill given a set of examples. Nacio learns string transformation programs that involves composition of API functions. We use the same neural architectures for Specification Encoder and the Program Generator for both the systems.

$$\begin{array}{rcl}
\text{String } e & := & \texttt{Concat}(f_1, \cdots, f_n) \\
\text{Substring } f & := & \texttt{ConstStr}(s) \\
 & | & \texttt{SubStr}(v, p_l, p_r) \\
\text{Position } p & := & (r, k, \texttt{Dir}) \mid \texttt{ConstPos}(k) \\
\text{Direction } \texttt{Dir} & := & \texttt{Start} \mid \texttt{End} \\
\text{Regex } r & := & s \mid T_1 \cdots \mid T_n
\end{array}$$

(a)

	Input v	Output
1	William Henry Charles	Charles, W.
2	Michael Johnson	Johnson, M.
3	Barack Rogers	Rogers, B.
4	Martha D. Saunders	Saunders, M.
5	Peter T Gates	Gates, P.

(b)

$\texttt{Concat}(f_1, \texttt{ConstStr(", ")}, f_2, \texttt{ConstStr(".")})$,

where $f_1 \equiv \texttt{SubStr}(v, (\text{“ ”}, -1, \texttt{End}), \texttt{ConstPos}(-1))$ and $f_2 \equiv \texttt{SubStr}(v, \texttt{ConstPos}(0), \texttt{ConstPos}(1))$

(c)

Figure 2 (a) The regular expression based string transformation DSL, (b) an example task in the DSL for transforming names to last names followed by first name initial, and (c) an example program in the DSL for this task.

3.1 Domain-specific Language

The domain-specific language for Neural FlashFill is shown in Figure 2(a). The top-level expression is a concatenation of a sequence of substring expressions, where each substring expression is either a constant string s or a substring of an input string between two positions p_l and p_r (denoting the left and right indicies in the input string). A position expression can either be a constant index or a token match expression $(r, k, \texttt{Dir})$, which denotes the `Start` or `End` of the k^{th} match of token r in input string v. A regex token r can either be a constant string s or one of 8 predefined regular expressions such as alphabets, alphanumeric, capital etc. An example benchmark is shown in Figure 2(b) and a possible DSL program for the transformation is: $\texttt{Concat}(f_1, \texttt{ConstStr(", ")}, f_2, \texttt{ConstStr(".")})$, where $f_1 \equiv \texttt{SubStr}(v, (\text{“ ”}, -1, \texttt{End}), \texttt{ConstPos}(-1))$ and $f_2 \equiv \texttt{SubStr}(v, \texttt{ConstPos}(0), \texttt{ConstPos}(1))$. The program concatenates: (i) substring between the end of last whitespace and end of string, (ii) constant string “, ”, (iii) first character of input string, and (iv) constant string “.”.

The top-level expression in the Nacio DSL (Figure 3(a)) is similar to that of the FlashFill DSL consisting of concatenation of a sequence of string expressions. The main difference is that a substring expression can now also be an API expression belonging to one of the three classes of APIs: lookup APIs, regex APIs, and transform APIs, and the DSL allows for both composition and nesting of APIs. The lookup APIs such as `GetCity`, `GetState` etc. comprise a dictionary of a list of strings and perform a lookup on an input string. The regex APIs such as `GetFirstNum`, `GetLastWord`, etc. search for certain regular expression patterns in the input string and return the matched string. Finally, the transform APIs such as `GetStateFromCity` transform strings from one dictionary to another dictionary. In total, the DSL consists of 107 APIs (84 regex, 14 lookup, 9 transform). An example Nacio task shown in Figure 3(b) can be performed by the DSL program $\texttt{GetAirportCode}(\texttt{GetCity}(v))$.

3.2 Specification Encoder

The specification encoder for the Neural FlashFill and Nacio encodes a set of input-output strings to a continuous vector representation. The main idea of the encoder is to first run two separate bidirectional LSTMs [19] over the input and output strings respectively, and then perform compute a cross-correlation vector between the two representations. It then

$$\begin{array}{rcl} \text{String } e & := & \texttt{Concat}(f_1, \cdots, f_n) \\ \text{Substring } f & := & \texttt{CStr}(s) \mid \mathcal{R}(f) \\ & \mid & \mathcal{T}(f) \mid \mathcal{L}(v) \ \mid \ v \\ \text{Lookup API } \mathcal{L} & := & \mathcal{L}_1 \ \mid \ \cdots \ \mid \ \mathcal{L}_m \\ \text{Regex API } \mathcal{R} & := & \mathcal{R}_1 \ \mid \ \cdots \ \mid \ \mathcal{R}_l \\ \text{Transform API } \mathcal{T} & := & \mathcal{T}_1 \ \mid \ \cdots \ \mid \ \mathcal{T}_k \end{array}$$

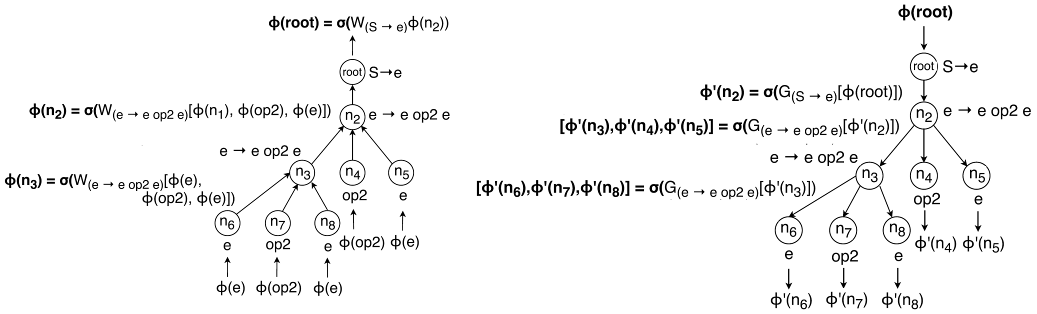
(a)

	Input v	Output
1	Los Angeles, CA	LAX
2	Boston, MA	**BOS**
3	San Francisco, CA	**SFO**
4	Chicago, IL	**ORD**
5	Detroit, MI	**DTW**

(b)

(c) `GetAirportCode(GetCity(`v`))`

Figure 3 (a) The Nacio DSL for API composition, (b) a Nacio task to obtain airport code and (c) a program in the DSL for the task.


ϕ(root) = σ(W(S → e)ϕ(n2))
root S→e
ϕ(n2) = σ(W(e → e op2 e)[ϕ(n1), ϕ(op2), ϕ(e)])
n2 e → e op2 e
e → e op2 e
ϕ(n3) = σ(W(e → e op2 e)[ϕ(e), ϕ(op2), ϕ(e)])
n3 n4 n5
op2 e
n6 n7 n8
e op2 e ϕ(op2) ϕ(e)
ϕ(e) ϕ(op2) ϕ(e)
ϕ(root)
root S→e
ϕ'(n2) = σ(G(S → e)[ϕ(root)])
n2 e → e op2 e
[ϕ'(n3),ϕ'(n4),ϕ'(n5)] = σ(G(e → e op2 e)[ϕ'(n2)])
e → e op2 e
[ϕ'(n6),ϕ'(n7),ϕ'(n8)] = σ(G(e → e op2 e)[ϕ'(n3)])
n3 n4 n5
op2 e
ϕ'(n4) ϕ'(n5)
n6 n7 n8
e op2 e
ϕ'(n6) ϕ'(n7) ϕ'(n8)

(a) Recursive pass (b) Reverse-Recursive pass

Figure 4 (a) The initial recursive pass of the R3NN. (b) The reverse-recursive pass of the R3NN where the input is the output of the previous recursive pass.

concatenates the representations of all input-output examples to get a representation for the set of examples.

3.3 Program Generator

We develop a new R3NN (Recursive-Reverse-Recursive Neural Network) [29] to define a generation model over trees in a DSL (grammar). The R3NN model takes a partial program tree (derivation in the grammar), and decides which non-terminal node in the tree to expand and with which expansion rule, given the I/O encoding vector. The model first starts with the start symbol of the grammar and incrementally incrementally constructs derivations in the grammar until generating a tree with all terminal leaf nodes.

The R3NN model has the following 4 parameters: (i) a vector representation for each symbol in the grammar, (ii) a vector representation for each rule in the grammar, (iii) a deep neural network that takes as input the set of Right-hand side (RHS) symbols of a rule and generates a representation of the corresponding Left-hand side (LHS) symbol, and (iv) a deep neural network that as input the representation of an LHS symbol of a rule and generates a representation for each of the corresponding RHS symbols. The R3NN first assigns a vector representation to each leaf node of a partial tree, and then performs a recursive pass going up in the tree to assign a global representation to the root. It then performs a reverse-recursive pass from the root to assign a global representation to each node in the tree. Intuitively, the idea is to assign a representation to each node in the tree such that the node knows

Sample	Train	Test
1-best	60%	63%
1-sam	56%	57%
10-sam	81%	79%
50-sam	91%	89%
100-sam	94%	94%

(a)

Size	Train	Test
7	45%	37%
8	67%	53%
9	36%	28%
10	41%	33%

(b)

Sample	NeuralFF	Nacio
1-best	8%	15%
1-sam	5%	12%
10-sam	13%	24%
50-sam	21%	34%
100-sam	23%	37%

(c)

Figure 5 (a) Neural FlashFill performance on synthetic data of programs upto size 13, (b) Nacio performance on synthetic data of programs upto 3 concats, and (c) Perfomance of Neural FlashFill and Nacio on 238 real-world FlashFIll Benchmarks.

about every other node in the tree. The R3NN encoding for an example partial tree in the grammar with the recursive and reverse-recursive passes is shown in Figure 4(a) and Figure 4(b) respectively.

More concretely, we first retrieve the distribution representation $\phi(S(l))$ for every leaf node $l \in L$ in the tree and then perform a standard recursive bottom-to-top, RHS→LHS pass by going up the tree and applying $f_{R(n)}$ for every non-leaf node $n \in N$ on its RHS node representations. We continue this pass until we reach the root node, where $\phi(root)$ denotes the global tree representation. This global representation has lost any notion of tree position and we perform a reverse-recursive pass to pass this information to all the leaf nodes of the tree. We start this pass by providing the root node representation $\phi(root)$ as an input to the second set of deep networks $g_{R(root)}$ where $R(root)$ denotes the production rule for expanding the start symbol. This results in a representation $\phi'(c)$ for each RHS node c of $R(root)$. We iteratively apply this procedure to all non-leaf nodes c, i.e., process $\phi'(c)$ using $g_{R(c)}$ to get representations $\phi'(cc)$ for every RHS node cc of $R(c)$. At the end of this pass, we obtain a leaf representation $\phi'(l)$ for each leaf node l, which has an information path to every other node in the tree. Using the global leaf representations $\phi'(l)$, we can generate the scores for each tree expansion e as: $z_e = \phi'(e.l) \cdot \omega(e.r)$, where $e.l$ denotes the leaf node l associated with the expansion e and $e.r$ denotes the expansion rule. The expansion scores can then be used to obtain the expansion probabilities as: $\pi(e) = e^{z_e} \Big/ \sum_{e' \in E} e^{z_{e'}}$.

3.4 Training and Evaluation

The I/O encoder and R3NN models are trained end-to-end over the training set of 2 million programs sampled from the DSL. Because of training complexity of the models, the size of the programs is currently limited to 13 (number of AST nodes) for Neural FlashFill and limited to 3 concats for Nacio. We perform tests on two types of generalization: (i) Input-output generalization: we test the performance of the model on 1000 randomly sample programs that the model has seen during training but with different input-output examples (`Train`) and (ii) Program generalization: performance on 1000 randomly sampled programs that the system has not seen during training (`Test`). We also evaluate the performance of both the systems on real-world FlashFill benchmarks as well. The results are shown in Figure 5.

The Neural FlashFill system after being trained on synthetic programs of size upto 13 is able to successfully synthesize both programs (upto size 13) that it has seen during the training but with different input-output examples, and programs that it has not seen during the training. The 1-best strategy yields an accuracy of about 60% whereas it increases to 94% with 100-samples. Moreover, it is also able to learn desired programs for 55 (23%) of 238 real-world FlashFill benchmarks. The Nacio system is able to get an accuracy of about

41% on Train set and 33% on Test set. Note that the Nacio DSL allows for a much richer class of transformations using complex API functions. However, the Nacio system perform much better on the FlashFill benchmark set with the success rate of 37% with 100-samples. A majority of the unsolved FlashFill benchmarks belong to the category of programs larger than the ones the two systems are trained on.

4 Future Challenges and Other Applications

There are several exciting research challenges in scaling the neural program synthesis framework for larger programs and for synthesizing programs in richer and more sophisticated DSLs. We briefly discuss a few of these directions and also discuss other program analysis applications that can be enabled by such a framework.

Scaling to Larger programs: The current complexity of both the I/O encoder and the R3NN model limits the training capacity to only programs upto a fixed size. We would like to explore new encoders and tree generation architectures for more efficient training.

Modeling program states: The DSLs we have considered till now are functional and do not model stateful assignments. One interesting challenge in the R3NN network is to encode variable-dimensional program states and the imperative state update semantics.

Reinforcement Learning for R3NN: We currently use the supervised training signal to teach the network to generate syntactically similar programs. A key extension to this would be to allow the model to learn semantically equivalent programs (resulting in infrequent reward signal). We believe reinforcement learning techniques can be useful in this setting.

More sophisticated specification encoders: We have only developed a few simple specification encoders for one kind of specification mechanism, i.e. input-output examples over strings. One extension of this would be to consider other data types such as integers, arrays, dictionaries, and trees. Another important extension would be to handle multi-modal specification such as natural language descriptions, partial programs, assertions etc.

Combining Neural approaches with symbolic approaches: The neural architectures are good at recognizing patters in the specifications, but are not good at modeling complex program transformations. A combination of neural architectures with logical reasoning techniques might alleviate some of the function modeling issues.

Learning to Superoptimize: A recent approach based on reinforcement learning was proposed to guide the superoptimization of assembly code [8, 7]. We can use program synthesis for super-optimization with reference implementation as the specification.

Neural Program Repair: The neural representation of programs can also aid in program repair. Several recent approaches learn a language model over programs to perform syntax correction over programs [3, 15, 31]. Neural program synthesis techniques can be extended with distance metrics to correct semantic errors in the programs.

Neural Fuzzing: Fuzzing has proven to be an effective technique for finding security vulnerabilities in software [10]. These techniques have shown impressive results for binary-format file parsers but not for more complex input formats such as PDF, XML etc. parsers, where a grammar needs to be written to define the input formats. Neural architectures can be developed to automatically learn these grammar representations from a set of input examples [11], and the learning can further be guided using metrics such as code coverage.

5 Conclusion

The problem of program synthesis is considered to be the holy grail of Computer Science. Although there has been tremendous progress made recently, the current approaches have either limited scalability or are domain-specific. In this paper, we argued that some of these limitations can be tackled using a learning-based approach that learns to encode specifications and to generate programs from a DSL using a large amount of training data. We presented two preliminary instantiations of the neural program synthesis approach, but we believe this approach can dramatically expand the applicability and effectiveness of Program Synthesis.

References

1 Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015.

2 Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

3 Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016.

4 Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Deep API Programmer: Learning to Program with APIs. *CoRR*, 2017.

5 Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In *ICML*, pages 2933–2942, 2016.

6 Alan W. Biermann. The inference of regular lisp programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, 8(8):585–600, 1978.

7 Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. Learning to superoptimize programs. *CoRR*, abs/1611.01787, 2016.

8 Rudy R. Bunel, Alban Desmaison, Pawan Kumar Mudigonda, Pushmeet Kohli, and Philip H. S. Torr. Adaptive neural compilation. In *NIPS*, pages 1444–1452, 2016.

9 Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016.

10 Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

11 Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. *CoRR*, abs/1701.07232, 2017.

12 Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL: `http://arxiv.org/abs/1410.5401`.

13 Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

14 Sumit Gulwani, William Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, Aug 2012.

15 Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing common C language errors by deep learning. In *AAAI*, 2017.

16 Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *ECCV*, pages 630–645, 2016.

17 Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, 2016.

18 Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

19 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

20 Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 190–198, 2015.

21 Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

22 John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

23 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

24 Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2015. URL: `http://arxiv.org/abs/1511.06392`.

25 Zohar Manna and Richard J. Waldinger. Synthesis: Dreams – programs. *IEEE Trans. Software Eng.*, 5(4):294–328, 1979.

26 Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

27 Zohar Manna and Richard J. Waldinger. Fundamentals of deductive program synthesis. *IEEE Trans. Software Eng.*, 18(8):674–704, 1992.

28 Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

29 Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016.

30 Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

31 Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. *CoRR*, abs/1607.02902, 2016.

32 Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *POPL*, pages 111–124, 2015.

33 Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2016. URL: `https://arxiv.org/abs/1511.06279`.

34 Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *CoRR*, abs/1605.06640, 2016. URL: `http://arxiv.org/abs/1605.06640`.

35 Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic program optimization. *Commun. ACM*, 59(2):114–122, 2016.

36 Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.

37 Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

38 Phillip D. Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

39 Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *PLDI*, pages 287–296, 2013.

40 W. Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. The microsoft 2016 conversational speech recognition system. *CoRR*, abs/1609.03528, 2016.

Migratory Typing: Ten Years Later[*]

Sam Tobin-Hochstadt[1], Matthias Felleisen[2], Robert Bruce Findler[3], Matthew Flatt[4], Ben Greenman[5], Andrew M. Kent[6], Vincent St-Amour[7], T. Stephen Strickland[8], and Asumu Takikawa[9]

1 **PLT & Indiana University, Bloomington, IN, USA**
samth@racket-lang.org
2 **PLT & Northeastern University, Boston, MA, USA**
matthias@racket-lang.org
3 **PLT & Northwestern University, Evanston, IL, USA**
robby@racket-lang.org
4 **PLT & University of Utah, Salt Lake City, UT, USA**
mflatt@racket-lang.org
5 **PLT & Northeastern University, Boston, MA, USA**
ben@racket-lang.org
6 **PLT & Indiana University, Bloomington, IN, USA**
andmkent@racket-lang.org
7 **PLT & Northwestern University, Evanston, IL, USA**
stamourv@racket-lang.org
8 **PLT & Google New York, New York City, NY, USA**
sstrickl@racket-lang.org
9 **PLT & Northeastern University, Boston, MA, USA**
asumu@racket-lang.org

Abstract

In this day and age, many developers work on large, untyped code repositories. Even if they are the creators of the code, they notice that they have to figure out the equivalent of method signatures every time they work on old code. This step is time consuming and error prone.

Ten years ago, the two lead authors outlined a linguistic solution to this problem. Specifically they proposed the creation of typed twins for untyped programming languages so that developers could migrate scripts from the untyped world to a typed one in an incremental manner. Their programmatic paper also spelled out three guiding design principles concerning the acceptance of grown idioms, the soundness of mixed-typed programs, and the units of migration.

This paper revisits this idea of a migratory type system as implemented for Racket. It explains how the design principles have been used to produce the Typed Racket twin and presents an assessment of the project's status, highlighting successes and failures.

1998 ACM Subject Classification D.3 Language Design, F.3.3 Type Structure

Keywords and phrases design principles, type systems, gradual typing

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.17

1 Migratory Typing

In the 1970s and 80s, developers chose Lisp for its flexibility, its libraries, and occasionally Lisp-specific hardware. They argued that higher-order functions, class systems, mixin classes,

[*] Over 10 years, this work was partially supported by our host institutions (Chicago, Northeastern, Northwestern, Indiana, and Utah) as well as AFOSR, DARPA, Mozilla, NSA, and NSF.

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 17; pp. 17:1–17:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

automatic memory management and the absence of *static* types enhanced their productivity. To their credit, statically typed languages with similar features did not exist, and existing compilers used types mostly as hints for choosing data representations and optimizations.

In the 1990s and 00s, developers continued to opt into a wide array of scripting languages that, like Lisp, lacked static types and simultaneously supported a rich collection of powerful features. System administrators picked Perl as their scripting language. Python and Ruby (on Rails) emerged as the primary vehicles for web-server extensions. PHP and JavaScript became the assembly languages of the web (browser). Java, the first soundly typed production language, tried and failed to take the place of these languages.

When the senior authors launched their pedagogical outreach project [15, 16], they settled on an un(i)typed[1] implementation language, Racket, for a mix of the same reasons that the Lispers and their successors had offered before them. They knew that this choice would enable them to rapidly build and deploy teaching languages [17, 18]. Within a few short years, they then faced the task of maintaining a code repository of 500,000 lines of untyped Racket and an exponentially growing web repository of Racket libraries.

Regardless of why developers – consciously or not – choose to develop code in an untyped language, they sooner or later end up in a similar situation. While they design code with type-like ideas in mind, they cannot write down these insights within their programming language and get them cross-checked against code. Every time they have to modify or extend a component, their first chore is to reconstruct the type-like design ideas of the original creator. As any experienced developer knows, this task is time consuming and error prone.

This scenario clearly explains why the authors claim that

even in an untyped world, types are critical for the cost-effective maintenance of software.

Once reliable type information becomes available, developers can comprehend a code base more easily than without types. Similarly, IDEs can exploit types to assist developers with various tasks, because types also help IDEs to reason about code. When code is changed, type checking can catch basic errors even before the unit tests uncover those. Finally, compilers rely on type information to produce performant target code, and software developers may wish to squeeze extra performance percentages out of their untyped code base.

And now the question arises how to equip a large, untyped code base with useful types.

The lead authors' vision paper [48] proposes a linguistic solution to this research problem:

> *the creation of an explicitly-statically typed twin of the given programming language*
> *to which developers can migrate pieces of untyped code in an incremental manner*
> *never losing the ability to test and deploy the software system.*

Their paper also spells out the three basic principles of what such a typed twin language should look like, with direct implications for its development:

1. The type system must cause as few perturbations to the code as possible so as not to "enbug" the code during migration (see Section 3).
2. The mix of untyped and typed code must remain executable, and it must guarantee the soundness of type information, which calls for run-time enforcement [30] (see Section 4).
3. The "unit of migration" must satisfy two opposing desires: (a) It must be small enough to encourage the incremental migration of untyped code into the typed twin language.

[1] With this strange spelling, we emphasize that we are thinking of "safe" untyped languages, and we acknowledge that Dana Scott suggested "unityped" as an alternative characterization.

(b) It must also be large enough to keep values from crossing the language boundary too often because every crossing may trigger a run-time check (see Section 5).

Section 7 provides an assessment of the state of the research program and lessons for language designers. The next section re-tells the pre-history of types for untyped languages from the perspective of the second author, motivating the idea of a typed twin language.

2 Some Pre-history

Most people reply with "type inference" when confronted with the problem of figuring out the types of programs in untyped languages, a solution that comes with an early historical justification [44]. They know about ML, Haskell, and Hindley-Milner type inference, but they rarely appreciate how subtle type inference is. Roughly speaking, the type-inference problem asks for the restoration of missing type declarations for variables and functions – and it thus seems a perfect match for the stated problem. Mathematically speaking, the problem asks for the solution of a system of equations over an uninterpreted algebra of types whose variables stand in for the omitted type declarations:

$$\overrightarrow{\text{(variable expression over uninterpreted type algebra)} = \text{(type)}}$$

Depending on the precise formulation, however, the problem is easily reducible to the Halting Problem or similarly unsolvable problems; for an early example, see Boehm's 1985 paper [6] on the undecidability of a form of polymorphic type inference.

In the context of untyped programming languages, the problem takes on a different form. By definition, a program in an untyped language contains neither any type definitions (say, for recursive data types) nor any declarations (say, for functions). Instead, values come with tags that specify which operations are applicable. Hence the most straightforward way to reason about untyped code is to think in terms of sets and subsets of values, which means the natural mathematical problem statement involves systems of *in*equations, not equations:

$$\overrightarrow{\text{(variable expression over uninterpreted type record algebra)} \subseteq \text{(type)}}$$

The challenge is to develop a method for solving such systems of inequations.

Over the past 30 years, researchers have essentially developed two solution methods:[2]

1. Inspired by operations research, Mike Fagan's "soft typing" approach [13, 11] uses so-called *slack variables* to turn a system of inequations into a system of equations:

   $$\overrightarrow{\text{(variable expression over uninterpreted type record algebra)} \oplus \text{slack-variable} = \text{(type)}}$$

   At this point it is possible to use a relatively standard[3] version of the Hindley-Milner algorithm. If a solution assigns a non-empty (record) type to a slack variable, the program is not typable in the given algebra. The essence of soft typing is to turn a non-empty slack variable into a cast that camouflages the associated type error. Wright [55, 57]'s dissertation shows how to scale the idea to R4RS [12].
2. Inspired by static program analysis, Flanagan's "static debugging" approach [20, 22, 23] uses a modified transitive closure algorithm to solve the system of inequalities *directly*.

[2] Henglein [26] embeds the untyped language into a statically typed language using injections to, and projections from, a universal data type. The solution of IBM's FL group [1, 2] is like the second bullet.

[3] Fagan's approach uses Remy's type algebra of extensible records [34] and is thus not completely standard.

In the experience of the second author[4] of this paper, who used all of the above systems, plain static type inference does not work well for a code base in an untyped language – without modifying the code pervasively, which would explicitly contradict the idiomacy constraint. In a nutshell, any form of inference suffers from some of the following problems:[5]

Syntactic brittleness. Both approaches infer types from the syntactic shape of the program. A simple, semantics-preserving modification may turn a two-line type description into one of ten lines – to the great consternation of a practicing programmer.

Non-actionable type errors. Going back to 1987 [54], researchers have studied the problem of deciphering error messages in the Hindley-Milner setting and making them actionable. Not surprisingly, Wright's Soft Scheme exhibits the same problem as ordinary Hindley-Milner languages. Although people have made some progress on this problem in the past ten years, all of this work applies to the standard type algebra, and it remains an open question how (well) their solutions apply to inference in a completely untyped setting.

Non-modularity. Flanagan's approach greatly simplifies the search for type errors in untyped programs. Unlike unification, the central piece of the Hindley-Milner algorithm, his algorithm is directional and thus connects the source of a type error with its actual manifestation. It comes with the separate disadvantage of requiring a whole-program analysis. Meunier's dissertation [32, 33] overcomes this new problem, but it demands that programmers specify contracts, which is essentially a step toward adding explicit types.

It is this last insight that caused the second author to look for a different solution, the construction of an explicitly and statically typed twin of the given untyped language.

3 Grown Idioms Require Accommodating Type Systems

Ideally the migration of untyped code to a typed twin language adds type definitions and declarations but leaves the code itself alone. While this constraint may not be obvious at the level of a single function, it is indisputable when it comes to modules of hundreds of lines of code or systems consisting of tens of thousands of lines. If developers are forced to change the code while migrating, they may end up "enbugging" their programs. Conversely, the goal of preserving existing code has serious implications for the design of the type system.

In this section, we explore these implications, starting at the function level. Figure 1 displays an idiomatic Racket function. As the comment in line 1 says, the code deals with trees of integers, where a leaf is represented with `#false` and interior nodes are three-element `lists`. The `snap` function traverses the tree, adding `1` to even integers, pruning odd ones, and growing the tree at its leafs. As in any statically typed language, a developer organizes functions on such trees according to the data type definition. Thus, the body of `snap` consists of a two-pronged conditional, one per branch in the informal data type specification. The function recurs on the two branches of a node, which are extracted with `first` and `third`.

Let us compare the code in Figure 1 with the OCaml version in Figure 2. Instead of a comment, the OCaml snippet defines the `tree` type explicitly. The OCaml version of `snap` is nearly identical to the Racket version, except for the use of a pattern matcher[6] and numerous injections into the algebraic datatype. The former hides the projections out of the algebraic datatype and, via the introduction of pattern variables, greatly facilitates Hindley-Milner

[4] He supervised Flanagan's dissertation, jointly supervised Wright's, and was deeply involved with Fagan's.

[5] Cardelli [10] argues that similar problems show up in a statically typed language with Hindley-Milner inference. He therefore suggests to think of type inference as a mere IDE tool.

[6] Racket also comes with `match`. For the purpose of this illustration, we present the function in the style inherited from Scheme, especially because few scripting languages have pattern matching constructs.

Listing 1 A Racket function.

```
1 ;; Tree is either #false or [list Tree Integer Tree]
2
3 (define (snap t)
4   (cond
5     [(false? t) (list #false 1 #false)]
6     [else (define v (second t))
7           (if (odd? v)
8               #false
9               (list (snap (first t)) (+ v 1) (snap (third t))))]))
```

style type inference. The latter is something that “untyped developers” cite as an obstacle to using typed languages and that a migratory type system may not assume.

Listing 2 The OCaml equivalent of `snap`.

```
type tree = False | Triple of tree * int * tree

let rec snap t =
  match t with
  | False -> Triple(False,1,False)
  | Triple(left,v,right) ->
    if (v mod 2 <> 0)
    then False
    else Triple(snap(left),v+1,snap(right))
```

When Racket developers design functions such as `snap`, they reason about its possible inputs as sets. In this specific case, a developer knows that the first line in the conditional subtracts `#false` from the possible inputs. Hence `snap` has to deal with nothing but three-element lists in the second clause, meaning it is safe to extract the `second` item from `t`.

From the perspective of the type system, the developer asserts that the underlined occurrence of `t` in line 6 does not just belong to `Tree` but to `[list Tree Integer Tree]`. More generally, if Racket developers are willing to turn type-oriented comments into type declarations but want to leave function definitions alone during migration, the type checker must assign different sub-types to different occurrences of the same variable, depending on which type predicates govern the occurrence in the flow graph.

We re-use the term *occurrence typing* [29, 49] for this idea. While both Wright’s Soft Scheme [55] and Flanagan’s Spidey Scheme [20] incorporate simple, syntax-oriented variations on this idea, Typed Racket systematically incorporates it into the type judgment. Every function type comes with propositions that hold when the function returns `#false` or a non-false value, respectively. The type checker exploits these propositions and also combines them with propositions in the Racket code, mimicking the kind of propositional reasoning “untyped developers” perform on a daily basis [50].

Now consider the function `false?`, which `snap` uses to discriminate between different trees:

```
false? : [Any -> Boolean : #:+ False #:- (! False)]
```

It uses two optional annotations to articulate two propositions. Specifically, `#:+` says that if `(false? x)` returns `#true`, then `x` has singleton type `False`, i.e., `x` is `#false`; similarly, `#:-` tells the reader and the type checker that if the result is `#false`, then `x` cannot be `#false`.

Listing 3 The Typed Racket version of `snap`.

```
(define-type Tree (U False [List Tree Integer Tree]))

(: snap (Tree -> Tree))
(define (snap t)
  (cond
    [(false? t) (list #false 1 #false)]
    [else (define v (second t))
          (if (odd? v)
              #false
              (list (snap (first t)) (+ v 1) (snap (third t))))]))
```

With this in mind, it is easy to see how a developer can migrate the code from Figure 1 to the one in Figure 3, *without modifying the function at all*. It suffices to define the `Tree` type and to declare the function's type. The occurrence type checker can reconstruct the types of the expressions in `snap` from just these two pieces of type information.

Typed Racket generalizes occurrence typing to higher-order functions, e.g.

```
filter : (All (a b) [ [a -> Any : #:+ b] (Listof a) -> (Listof b) ])
```

Naturally, `filter` is polymorphic. Furthermore, if its predicate argument specifies a positive proposition `b`, its result is a list of elements that satisfy `b`.

Occurrence typing also comes in handy for dealing with the numeric tower [41] that Typed Racket inherits from Racket and Scheme. This numeric tower allows developers to use numbers and operations on them the way mathematicians present them, with computer-based numbers mixed in for performance. Consequently, Racket code comes with numerous idioms that rely on mathematical sets instead of the disjoint numeric types based on machine arithmetic, commonly found in conventional languages.

Typed Racket's corresponding type hierarchy starts with `Complex` numbers, which contain the `Reals`. The latter subdivides into exact `Rationals`, including `Integers`, plus inexact, IEEE `Floats`. Within the `Integers`, Racket is aware of `Fixnum`, `Index`, and `Byte`. Finally, Typed Racket needs the type `Zero` because the value zero shows up in several different, disjoint sets and yet plays a special role in comparisons.

Listing 4 Typed Racket and the numeric tower.

```
(: sum-vector [(Vectorof Integer) -> Integer])
(define (sum-vector v)
  (define n (vector-length v))
  (let loop ([i 0] [sum 0])
    (if (< i n) (loop (+ i 1) (+ sum (vector-ref v i))) sum)))
```

The types for numeric comparisons exploit occurrence typing to reify the numeric subsets:

```
> : [Real Zero -> Boolean : #:+ Positive-Real]
```

This type says that if `(> x 0)` produces `#true`, `x` has type `Positive-Real`. Additionally, reasoning about numeric idioms benefits from a lightweight form of intersection types:

```
+ : (case-> (Integer Integer -> Integer) (Float Float -> Float) ...)
```

This `case->` type lists function types and picks the first one that matches the use. Using such function types plus occurrence typing, Typed Racket can, for example, prove the type correctness of the `sum-vector` function in Listing 4, including the fact that `i` is always a proper vector index. The language's optimizer may then safely exploit this fact [51].

Listing 5 Typed Racket and the numeric tower.

```
(: transpose (All (a ...) [(Listof a) ... -> (Listof (List a ...))]))
(define (transpose . l)
  (if (andmap empty? l)
      '()
      (cons (map first l) (apply transpose (map rest l)))))
```

Like its popular cousins, Racket comes with a large variety of useful accommodations. For example, many Racket primitives take a variable number of arguments, e.g., `map`:

```
(map (lambda (s n) (- (string-length s) n)) '("hello" "world" "bye") '(3 4 1))
```

Of course, developers may also define such functions; see Listing 5 for a simple illustration.

To support multi-variable functions in a sound manner, Typed Racket comes with variable-arity at the type level [43]. Here is the type signature for `map`:

```
map : (All (c a b ...)
         [ [a b ... b -> c] (Listof a) (Listof b) ... b -> (Listof c) ])
```

Like `filter`, `map` is polymorphic, but its type introduces an unbounded number of type variables to describe the number of lists that `map` traverses simultaneously. As Listing 5 shows, Typed Racket is sufficiently powerful to type check uses of `map` via local type inference.

Finally, a lot of Racket code uses class-based, object-oriented programming, which comes with its own idioms [24]. Again like in other untyped languages, classes are first-class run-time values. Using those, Racket developers abstract over classes with functions and methods to construct just the right kind of class hierarchy at run-time. While this well-known "mixin" pattern has a long history, conventional type systems all too frequently identify classes with types and thus cannot deal with either mixins or other idioms using classes as values.

The current version of Typed Racket accommodates both function and method-based mixins, though it took several years to design a sound extension of the type system [47] and two more to implement and evaluate this design [46]. The extension rests on two theoretical novelties: (1) types for classes, for functions on classes, and so on; and (2) a contract system that ensures that class operations in untyped code respect the integrity of typed classes, mixins, etc. Practically the implementation must facilitate the addition of types to classes used in ordinary circumstances and make the protection of classes that flow across type boundaries reasonably efficient. For the former, see Listing 6; for the latter, we point the reader to the original implementation paper [46, section 4] and our recent work on performance evaluation [45], both of which are discussed in the last section.

4 Type Soundness for Mixed-typed Programs

While industrial researchers may trade ideals for practical concerns, especially performance, academic researchers have the moral obligation to strive for them – because nobody else will and society affords them exactly this luxury with generous support. In the context of migratory typing, soundness is the critical ideal.

On one hand, we clearly want the usual soundness of fully typed programs; on the other hand, we also want a generalized notion for programs that link typed and untyped pieces. Formally, the usual soundness theorem states [56] that if a program type checks, running it can have exactly one of three possible outcomes (**MT1**):

1. the program execution terminates, returning and printing values of the predicted types;
2. the execution diverges;
3. the execution ends in one of a number of well-specified exception states. These exceptions are due to partial computational primitives such as division and indexing.

For un(i)typed[1] languages such as plain Racket, (MT1) holds if memory access is safe, but more computational primitives are partial than in a typed setting [14, part I(ch. 5)].

To generalize (MT1) to mixed-type programs, we must make an assumption about the linking of typed and untyped code and hence execution. Given our desire to run mixed-typed programs easily, we clearly do not wish to deal with the twin language as a truly foreign language. Instead we assume that all cross-boundary traffic uses the bit-level representation of values from the original, untyped language and specifies the type of the crossing value in the typed module. Now a value flowing across such a boundary may not meet the expectations expressed as its type, which requires adding one clause to the above three (**MT2**):

4. the execution ends in an exceptional state and points to one of the fixed number of boundaries between a typed piece of code and an untyped one. After all, each boundary represents a distinct, programmer-defined and partial computational primitive.

Let us inspect this generalized type soundness theorem from an operational perspective. As mentioned, a “migrating developer” who links a newly typed module to an untyped code base must add type specifications to all import statements so that the type checker can check their uses statically. Say an untyped function `f` is imported with `(D -> R)` imposed as its type. This type may not match the untyped reality, which is why the typed twin language must insert run-time checks to prevent certain problems. Here are some sample scenarios of how things can go wrong and how run-time checking works:

- `f` cannot cope with elements of `D`. In this case, the run-time checks of the underlying untyped language eventually catch the error and issue an appropriate message. Here it is critical that we assume the same bit-level representation for boundary-crossing values so that the primitives of the untyped function may use the tag bits for run-time checking.
- `f` does not produce elements of `R`. Since the soundness of the type checker depends on the `R`ness of the result value, (1) every `R` must come with a run-time check that can enforce `R`ness and (2) the type checker must insert this check at all call sites of `f`.
- `f`’s domain `D` is a function type, too, say `(D1 -> R1)`. When `f` is called, the typed code sends a typed function `g` into untyped code. In this case, applications of `g` must be protected so that all arguments are values in `D1`; type checking guarantees `R1`ness.

In all cases, the failure of a run-time check might be due to a bad type specification – that is, the untyped code cannot live up to the type imposed by the developer – or an error in the untyped code with respect to an expressed or implied type. And if things do go wrong, Typed Racket’s exceptions come with a highly informative error message that points developers to the specific problem boundary and presents a witness value that explains the mismatch between the value and its type.

4.1 “The Dangers of Moral Turpitude” [35]

A type system that fails to satisfy (MT2) can resurrect all the problems of unsafe languages such as C++, which fails to live up to plain type soundness (MT1). These failures have serious implications for both programmers and ordinary users. Recall that C++ checks types but executes programs without enforcing the interaction between the untyped run-time system and the type-checked code. Hence, C++ programs interpret operations on bit patterns regardless of whether it is appropriate to apply the operation to the data that these bits represent. As long as the hardware does not object, the program execution continues. The program may seemingly terminate “normally,” printing all kinds of output on the way. Alternatively, the misapplication of the operation may eventually trigger an interpretation of bits to which the hardware objects, resulting in a `segfault` long after the original misinterpretation.

Listing 6 A Voting Machine and the Lack of Sound Linking.

```
#lang typed/racket

(provide voting-machine%)

(define-type Count {List String Natural})
(define-type Tally {Listof Count})

(define voting-machine%
  (class object%
    (super-new)

    (init [candidates : [Listof String] '()])

    (field [votes : Natural 0]
           [tally : Tally
            (map (lambda ({s : String}) (list s 0)) candidates)])

    (define/public (show)
      (sort tally second-of-pair->))

    ;; names missing from tally did not get any votes,
    ;; names not on tally are "write ins"
    (define/public (add-votes-from-district {delta : Tally})
      (for ((district-count : Count delta))
        (define-values (name delta) (apply values district-count))
        (define old-count (assoc name tally))
        (set! tally
              (match old-count
                [`(,_name ,old)
                 (define new (+ delta old))
                 (cons (list name new) (remove old-count tally))]
                [#false (cons district-count tally)]))))))
```

To understand how a failure of (MT2) can trigger the first kind of problem, consider the module in Listing 6. It exports a class that represents a simplistic voting machine. An importing module instantiates the class with a list of candidates:

```
(define my-voting-machine
  (new voting-machine% [candidates '("DonaldDuck" "HolyCow")]))
```

Once the votes in a district are tallied, the user can call the `add-votes-from-district` method to consolidate the tally with the running total:

```
(send my-voting-machine add-votes-from-district
  '(("DonaldDuck" 2) ("HolyCow" 4)))
```

This call adds two votes to the total of `"DonaldDuck"` and four for `"HolyCow"`.

Now consider the following method call:

```
(send my-voting-machine add-votes-from-district
  '(("DonaldDuck" 2) ("RonnyM" 3) ("HolyCow" -1)))
```

Given that the type signature of the method demands a *natural* number for vote counts, this call cannot type check in a typed module. Imagine what would happen, however, if an *untyped* module initiated this call. If (MT2) holds, the boundary between this client module and the module from Listing 6 checks that all vote counts are exact, non-negative integers and therefore signals an error when it encounters `-1`. If (MT2) does *not hold*, no such check is run and `"HolyCow"` loses votes and ends up trailing `"DonaldDuck"`. The program execution goes on, and nobody may ever know about the invalid vote subtraction.

A failure to implement (MT2) can also trigger C++-style segfaults if the compiler for typed modules exploits type information for code generation. For example, it may insert an integer multiplication yet the lack of run-time checks may allow floats to flow into this operation. In short, unsound linking really introduces the whole range of problems that the creation of high-level languages – typed and untyped – aims to eliminate for good.

5 Keeping the Cost of Run-time Checks Low

So every single time a (semantic) value flows from an untyped piece of a program to a typed one, a run-time check ensures that the value lives up to the specified type. A flat value, say a number, requires the same kind of checking that any sound, untyped language performs. When a higher-order value such as a function or an object crosses a boundary, checking a type-like property is impossible because, semantically speaking, these values are infinite. Hence the run-time system delays the relevant checks until the function is applied, a message is sent to an object, etc. [19]. Finally, for a compound value, e.g., an array, the run-time check either inspects every element, even if none are actually accessed, or delays the checks until an access is executed. For mutable values, this latter strategy is imperative.

None of these checks come for free. While the checks for flat values are relatively inexpensive for a single crossing, the cost for other checks is non-trivial for single crossings and especially for high-frequency crossings. In Typed Racket, these checks impose two kinds of costs: the allocation of wrappers for delaying the checks and the time for the delayed run-time checks. To make this latter cost concrete, imagine a typed function that flows into an untyped part of the program. If the untyped part applies the function a million times, the argument checks kick in that often. If the already-wrapped function flows back out of typed territory into different untyped code, the boundary check wraps it again, and every application must penetrate two layers of wrapping and execute two delayed checks.

Due to this anticipated cost of boundary crossings, the Typed Racket vision paper [48] argues for reasonably large units of migration. Specifically, it argues that Racket modules hit the sweet spot between the desire to migrate code easily and to keep the run-time cost low.

On one hand, the point of modules is to bundle many functions into one unit, making some visible to client modules, hiding others. An exported function performs a decent amount of work before it hands control back to the client. It may call several hidden functions, usually in a hierarchical manner. By contrast, an individual function often connects to the surrounding context via numerous free identifiers, and the flow of control may cross this boundary much more often than a module boundary. In short, the fewer boundaries the fewer crossings are to be expected, which translates into a lower cost of run-time checking.

On the other hand, most Racket modules are reasonably small. With few exceptions, the modules in known applications consist of several hundred to a couple of thousand lines of code. Although adding types to such modules is clearly much more work than adding types to a single function, the paper assumes that developers work at the level of complete modules and that adding types to a module is a reasonable amount of work. Furthermore, if the IDE comes with approximate type inference tools, developers can rely on those to reduce their work to checking, and correcting, inferred types.

6 Related Work

Migratory typing is one particular instance of the 35-year old idea of *optional typing* for untyped languages, first found in Common Lisp [42]. The basic idea of optional typing is to allow the explicit specification of type information but not necessarily enforce it. Compilers and other development tools may then exploit this information in an appropriate manner. Both StrongTalk [9] and pluggable type systems [8] fall into the same category. Modern incarnations include industrial and quasi-industrial systems such as Hack,[7] Flow,[8] TypeScript,[9] StrongScript[36], and Typed Clojure [7].

Optional type systems usually, but not always, satisfy (MT1) and most fail (MT2). As a result, many suffer from some of the flaws spelled out in Section 4.1. Even though we understand the constraints of their creators as mentioned above and re-iterated below, the situation nevertheless suggests to us that Bertrand Russell's motto of "the advantages of theft over honest toil" has been applied to soundness.

Contemporaries of migratory typing, *hybrid typing* [21] and *gradual typing* [37, 38, 39] are theoretical designs that satisfy both (MT1) and (MT2). The teleology and the technical details differ from design to design. Hybrid typing aims to increase the power of static type checking. It allows programmers to add arbitrary predicates to type specifications, which are checked statically as much as possible and dynamically otherwise. In comparison, gradual typing aims to put static and dynamic typing on equal footing without violating soundness. It thus allows programmers to add type information on a purely optional basis *anywhere* in a program and inserts casts automatically as needed. Finally, the purpose of migratory typing is to support the migration of code from an untyped setting to a typed one, while preserving the ability to run any mixed-typed software system with the same guarantees as the fully untyped or fully typed ones. Clearly gradual typing can be used for migrating code in a sound manner, but it is equally well suited for annotating extremely small fragments with types for documentation of logical invariants or for exploratory coding in the context of a fully-typed system. By contrast, industrial optional type systems also aim to assist with the migration of code but accept temporary or even permanent unsoundness in the process.

Besides Typed Racket, which has been in development for the past ten years, two other academic groups have started efforts to implement optionally typed systems that satisfy (MT1) and (MT2): Reticulated Python [52] and a gradually typed Smalltalk [3]. Both systems are used to experiment with casting strategies [40], including strategies that give programmers some control over the interchange of values [4]. It would be interesting to assess the performance of these systems with the same metrics as Typed Racket; see below.

[7] See `http://hacklang.org`, last visited 18 Mar 2017.
[8] See `https://flowtype.org`, last visited 18 Mar 2017.
[9] See `https://typescriptlang.org`, last visited 18 Mar 2017.

7 Assessing Typed Racket, Lessons for Language Designers

Numerous Racket developers have embraced Typed Racket over the last ten years, including people in the Racket development team. We, the Typed Racket developers, have used Typed Racket extensively, in many different scenarios and situations. While some of the uses correspond to those imagined ten years ago, developers equally often just *add* typed modules to existing applications or create entire new libraries in Typed Racket, e.g., `math`, `pict3d`, `whalesong`. When developers use Typed Racket in this manner, they may use Racket idioms but they may also employ idioms they know from statically typed languages, in which case our type system is typically not what they expect.

We base the following assessment of Typed Racket on this usage history. The assessment accounts for all three principles from Section 1 plus the overall goal of assisting developers.

Programming Idioms. Typed Racket's type system mostly succeeds in accommodating the idioms of untyped Racket. Migrating *mostly-functional* Racket modules requires a relatively small amount of work. For most such programs, it usually suffices to add type definitions that name (recursive) union types and type declarations for functions, variables, and structure fields. Local type inference reduces the burden of adding type declarations in some cases.

Recurring complaints in this setting concern uses of (first-class) polymorphic functions and a lack of (some) refinement typing. When it comes to polymorphic functions, the existing local type inference algorithm fails too often, forcing developers to insert explicit type applications into existing code. Not surprisingly, Racket developers also have a certain amount of "refinement typing" in mind. This observation is the motivation of our recent investigation of refinement typing for Typed Racket [27], an extension that we will soon merge into the production branch.

By contrast, the migration of *object-oriented* Racket modules demands significantly more effort than the migration of functional code, requiring both significantly larger type annotations and many more touch-ups of existing code. For example, developers must write down the types of classes separately from the code for classes – except for those used in a strictly first-order fashion – causing a high degree of redundancy. Similarly, the type system is still missing occurrence typing for fields, which may trigger rewrites of method code.

In general, code migration demands interventions for about 5% to 20% of the existing lines of code. For functional code the number ranges in the lower part of the interval (5% to 10%), down from about 10% to 12% for the first implementation of Typed Racket. Object-oriented code needs interventions for 15% to 20% of the original lines. The difference is partly due to the programming idioms and partly due to maturity of the functional type system, which has been in development since 2007, while the object-oriented one has a mere five-year history.

This experience suggests an important, often overlooked lesson for language designers. *Syntactic engineering* must be taken into account from the very beginning to make the new language constructs as convenient to use as possible. Once developers experience a lot of friction, they might be reluctant to stay the course or take a second look.

The Benefits of Soundness. The system lives up to all expectations that developers have of sound language implementations. The static type checking eliminates many dynamic errors that Racket code suffers from. The newly added dynamic errors resemble the usual run-time exceptions of statically typed programs; they also always point to the boundary where the expectations of a typed module clash with the realities of its untyped surroundings.

Although the implementation comes with the usual errors, the semantic model tends to clarify how to fix these problems quickly. One exception concerns exported parametrically polymorphic functions. To enforce parametricity, the contract system inserts wrappers that subtly change the behavior of the functions and thus the entire program. We do not know how to solve this problem efficiently.

The Cost of Soundness. Performance has emerged as a major problem area over the past couple of years, justifying our original concern about the run-time cost of the inserted dynamic checks. Early indications included posts to our mailing list describing scenarios where mixed-typed programs exhibited abysmal performance. In some cases, developers found satisfactory work-arounds; in others, they solved the problem by abandoning Typed Racket or adding types to all of the code. After analyzing these problems on an *ad hoc* basis for quite some time, we decided to develop an evaluation method.

Our evaluation method [25, 45] calls for measuring *all possible combinations of typed and untyped modules* for every benchmark program. If a program consists of n modules, there are 2^n such configurations, implying an exponential effort. After extensive evaluations, we can now confirm that sampling a *linear* number of configurations suffices [25]. Hopefully this confirmation will encourage others in the field to use the evaluation method.

Applying the method to Typed Racket shows that mixed-type configurations suffer from a huge overhead. Some types compile to expensive run-time checks, and others allocate a lot of memory. Worst, some configurations call functions many times across boundaries.

Applying the method to Reticulated Python[10] suggests that it may suffer from similarly disabling performance problems. Due to Reticulated's transpiler, the performance lattice consists of $1 + 2^n$ programs: the original Python program and 2^n Reticulated configurations. Our measurements suggest a 2x average slow-down from plain Python to unannotated Reticulated program alone and they show that, as types are added, Reticulated inserts additional casts and the programs experience additional slow-down. Nevertheless, Vitousek et. al [53] conjecture that the overhead of Reticulated is an order of magnitude smaller than Typed Racket's – though they do so without having applied the evaluation method.

These performance evaluations suggest three lessons. First, soundness is an ideal that does not come cheap. If academic researchers wish to convince their industrial colleagues that soundness of optional type systems is a feasible idea, they must develop (1) suitable evaluation methods from the get-go and (2) pay attention to performance at every stage. Second, retrofitting implementations is hard. The Typed Racket team is now exploring alternative implementation strategies, especially a just-in-time compiler that can take advantage of the run-time checks [5]. How much this alternative implementation can reduce the overhead of migratory or gradual typing remains an open question. Third, some readers may jump to the conclusion that our industrial colleagues who trade soundness for performance are correct after all. We consider this conclusion premature. As mentioned, we accept the moral obligation of pursuing an ideal until there is conclusive proof of failure but an evaluation of two implementations does *not* disprove any hypothesis. We will continue to investigate sound migratory typing and its implementation until we know for sure that the failure is total.

Migrating Code. As for the original goal – assisting software developers with maintenance via the migration from untyped to typed code – we also have to report mixed insights, all based on anecdotal evidence. On one hand, our experience tells us that migrating entire modules is rarely a problem because the size of Racket modules is (now) reasonably small. On the other hand, outside developers report that modules are too large for a migration

[10] Ben Greenman and Zeina Migeed are currently conducting this investigation.

effort. People would much prefer to add the type invariants of just a key algorithm to a module when they have to revisit the code. This desire suggests a preference for an approach based on gradual typing rather than migratory typing.

The contradictory evidence presents a research challenge. While we are aware of the literature on measuring the benefit of adding types to programs [28, 31], we have not yet figured out how to construct a similar, repeatable experiment on migratory typing from these results. Our ideal scenario would involve a comparative study of a reasonably large code base and several well-qualified developers; moreover, the development task should come with a proper incentive. In other words, we simply do not understand how student experiments on small programs are predictive of real-world behavior, which is what we ultimately aim for.

Until we have answers to both the performance challenge and the benefits question, the entire Typed Racket project remains speculative. While our own experience and anecdotal evidence seem to tell us that Typed Racket adds value to the eco-system of Racket, scientific evidence for these points remains elusive and calls for additional research. We conjecture that gradual typing and other attempts at sound optional typing face similar challenges, and we therefore consider the proposed research the highest priority for this entire area.

Acknowledgements. We thank Eli Barzilay, Ambrose Bonnaire-Sergeant, Robert “Corky” Cartwright, Ryan Culpepper, Earl Dean, Christos Dimoulas, Eric Dobson, Mike Fagan, Cormac Flanagan, Daniel Feltey, Philippe Meunier, Max New, Neil Toronto, Jan Vitek, and Andrew Wright for contributing at various stages to the ideas of typing untyped languages.

References

1 Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *Principles of Programming Languages*, pages 279–290, 1991.

2 Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pages 163–173, 1994.

3 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 96(P1):52–69, December 2014.

4 Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Object-Oriented Programming Systems, Languages & Applications*, pages 251–270, 2014.

5 Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: a tracing JIT for a functional language. In *International Conference on Functional Programming*, pages 22–34, 2015.

6 Hans J. Boehm. Partial polymorphic type inference is undecidable. In *Foundations of Computer Science*, pages 339–345, 1985.

7 Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In *European Symposium on Programming*, pages 68–94, 2016.

8 Gilad Bracha. Pluggable type systems. In *Object-Oriented Programming Systems, Languages & Applications*. Companion (Workshop on Revival of Dynamic Languages), 2004.

9 Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *Object-Oriented Programming Systems, Languages & Applications*, pages 215–230, 1993.

10 L. Cardelli. Typeful programming. Technical Report 45, Digital Systems Research Center, May 1989.

11 Robert Cartwright and Mike Fagan. Soft typing. In *Programming Language Design and Implementation*, pages 278–292, 1991.

12 William Clinger and Jonathan Rees. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

13 Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University, 1991.

14 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

15 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The DrScheme project: An overview. *ACM SIGPLAN Notices*, June 1998. Invited paper.

16 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14:55–77, 2004.

17 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *First Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

18 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.

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

20 Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

21 Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages*, pages 245–256, January 2006.

22 Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Programming Language Design and Implementation*, pages 235–248, 1997.

23 Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Programming Language Design and Implementation*, pages 23–32, May 1996.

24 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, 2006.

25 Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. Performance evaluation for gradual typing. *J. Functional Programming*, 2016. Submitted for publication.

26 Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

27 Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *Programming Language Design and Implementation*, pages 296–309, 2016.

28 Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems? An empirical study. In *International Conference on Program Comprehension*, pages 153–162, 2012.

29 Raghavan Komondoor, G. Ramalingam, Satish Chandra, and John Field. Dependent types for program understanding. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *lncs*, pages 157–173, 2005.

30 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31(3):1–44, 2009.

31 Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented

software. In *Object-Oriented Programming Systems, Languages & Applications*, pages 683–702, 2012.

32 Philippe Meunier. *Modular Set-Based Analysis from Contracts.* PhD thesis, Northeastern University, 2006.

33 Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *Principles of Programming Languages*, pages 218–231, 2006.

34 D. Rémy. Typechecking records and variants in a natural extension of ML. In *Principles of Programming Languages*, 1989.

35 John C. Reynolds. Types, abstraction, and parametric polymorphism. In *IFIP Congress on Information Processing*, pages 513–523, 1983.

36 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for typescript. In *European Conference on Object-Oriented Programming*, pages 76–100, July 2015.

37 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop, University of Chicago, Technical Report TR-2006-06*, pages 81–92, September 2006.

38 Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27, 2007.

39 Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Dynamic Languages Symposium*, pages 1–12, 2008.

40 Jeremy G. Siek, Michael M. Vitousek, and Shashank Bharadwaj. Gradual typing for mutable objects. Unpublished manuscript, available at `http://ecee.colorado.edu/~siek/gtmo.pdf`.

41 Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *Practical Aspects of Declarative Languages*, pages 289–303, 2012.

42 Guy Lewis Steele Jr. *Common Lisp – The Language.* Digital Press, 1984.

43 T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *European Symposium on Programming*, pages 32–46, March 2009.

44 Norihisa Suzuki. Inferring types in Smalltalk. In *Principles of Programming Languages*, pages 187–199, 1981.

45 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Principles of Programming Languages*, pages 456–468, 2016.

46 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*, pages 4–27, 2015.

47 Asumu Takikawa, T. Stephen 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.

48 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Dynamic Language Symposium*, pages 964–974, 2006.

49 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Principles of Programming Languages*, pages 395–406, 2008.

50 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *International Conference on Functional Programming*, pages 117–128, 2010.

51 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.

52 Michael M. Vitousek, Jeremy G. Siek, Andrew Kent, and Jim Baker. Design and evaluation of gradual typing for Python. In *Dynamic Language Symposium*, pages 45–56, 2014.

53 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime. In *Principles of Programming Languages*, pages 762–774, 2017.

54 Mitch Wand. Finding the source of type errors. In *Principles of Programming Languages*, pages 38–43, 1986.

55 Andrew Wright. *Practical Soft Typing*. PhD thesis, Rice University, 1994.

56 Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994.

57 Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *Lisp and Functional Programming*, pages 250–262, June 1994.

Taming the Static Analysis Beast

John Toman[1] and Dan Grossman[2]

1 Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA
jtoman@cs.washington.edu

2 Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA
djg@cs.washington.edu

Abstract

While industrial-strength static analysis over large, real-world codebases has become commonplace, so too have difficult-to-analyze language constructs, large libraries, and popular frameworks. These features make constructing and evaluating a novel, sound analysis painful, error-prone, and tedious. We motivate the need for research to address these issues by highlighting some of the many challenges faced by static analysis developers in today's software ecosystem. We then propose our short- and long-term research agenda to make static analysis over modern software less burdensome.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases static analysis, frameworks, api knowledge, library specifications

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.18

1 Introduction

The ubiquitous use of static analysis to ensure the absence of software defects has been a long-held goal of the static analysis research community. As such, we should marvel at and celebrate the mainstream success of scalable code-analysis tools that are now routine for many projects, including at large software companies (such as Microsoft [29, 43], Google [59], and Facebook [17, 16]). Although we can continue to study why static analyses are not more widely deployed [35, 8], industrial-strength static analyses are finally becoming a reality.

Static analysis researchers also now enjoy excellent tool support. Analysis frameworks exist for several popular languages and platforms. These frameworks handle tedious tasks shared across almost all static analyses, such as translation from bytecode or source-code to an intermediate representation, call-graph construction, type information, string analyses, and points-to information [37, 71, 72, 52, 15]. The developers of these frameworks deserve substantial credit: thanks to these platforms, researchers have been able to ignore complex implementation details and focus solely on implementing their analyses.

Unfortunately, writing a sound static analysis that produces useful results for real programs is now harder than ever. Analysis implementations can easily exceed tens of thousands of lines of code [48, 7]. To understand the sources of complexity, one need look no further than today's software environment. Industrial-strength analyses must handle industrial-strength applications in industrial-strength languages. Analyses must handle objects, the pervasive use of callbacks, threads, exceptions, frameworks, reflection, native code, several layers of indirection, metaprogramming, enormous library dependency graphs, etc. In our experience (and those shared by other static analysis authors), getting a realistic static analysis to

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 18; pp. 18:1–18:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

run on “real” applications requires a combination of luck,[1] multiple heuristics (which may never see the light of day in published papers), engineering effort, manual annotation, and unsatisfying engineering tradeoffs.

Our community has recognized these difficulties and work continues to be published to tackle these challenges. However, developing a novel, sound static analysis *and* testing it accurately on modern applications often remains excruciatingly painful for fundamental reasons. We begin by describing some of these reasons, using examples drawn from our experience building a static analysis for Java.[2] The difficulties we describe are shared by many other researchers. In particular, we focus on the challenges posed by enormous external libraries, pervasive use of frameworks, and the need for high-level, domain knowledge about API behavior (Section 2). We then describe our research vision for addressing each of these three pain points (Section 3), and our vision for the future of static analysis research (Section 4).

2 Static Analysis Challenges

This section describes the challenges today’s static analysis writer faces. Although our descriptions are given in the context of writing an analysis for Java, the challenges we identify are not Java-specific in any fundamental way.

2.1 Libraries

No application is completely self-contained: even a simple “Hello World” application transitively depends on 3,000 classes [41]. The size of an application’s transitive dependencies can dwarf the original application code, sometimes by several orders of magnitude, posing significant scaling challenges for static analysis writers [6]. For example, a highly precise, scalable field-sensitive analysis by Lerch et al. [44] exhausted 25 GB of memory when analyzing the Java Class Library (which is comprised of over 18,000 classes). In the same work, an even less precise analysis exhausted the 25 GB memory limit on 6 of 7 non-trivial applications when including external dependencies. Our own experience broadly mirror this trend: when including all external dependencies an analysis that took under a minute exhausts all available memory after running for over 20 minutes.

Some analyses consider all library code along with application code (e.g., [47, 25]). This often limits the sophistication of an analysis: in general the more expressive or complex the analysis, the less scalable it becomes. We do not suggest that useful static analyses that consider library code cannot or do not exist: as mentioned in Section 1, large companies run static analyses regularly on their codebases. Nevertheless, considering an application and all dependent libraries requires tradeoffs in analysis sophistication and enormous engineering effort.

In practice, the challenges of including all library dependencies means many static analysis writers accept incomplete portions of an application’s class hierarchy and/or call-graph. However, ignoring these missing pieces is clearly unsound. Analysis writers therefore resort to one of several unappealing options. The analysis writer may provide hand-written summaries for all missing methods. This approach is precise but infeasible for even moderately

[1] We found we needed answers for a type of aliasing query unsupported by all existing off-the-shelf pointer analyses. A few weeks later, an analysis designed to answer these queries was published at a top conference.

[2] Currently under anonymous submission.

sized applications. Another option is to apply a notionally conservative summary of missing library behavior; e.g., “all data flowing into a function are propagated to the return value.” This technique is still unsound as it fails to consider “out parameters” and other side effects, and is unacceptably imprecise for pure methods.

In response to this difficulty, several authors have explored how to make analysis tractable in the presence of large libraries. A widely explored technique is caching results across runs of an analysis. Caching forms the core of incremental analyses [55, 65, 5, 16, 51, 50]. However, these approaches can reuse results only from previous executions from the analysis on the same program. If an analysis fails to terminate due to large libraries there is no opportunity for caching. Kulkarni et al. have recently proposed a technique to reuse analysis results on common (i.e., library) code shared between two or more target applications [41]. However, this technique can reuse results only of the same analysis and requires programmer provided predicates describing when cached results may be soundly applied. Even the optimal approach for analyzing libraries in isolation remains an area of active research [57].

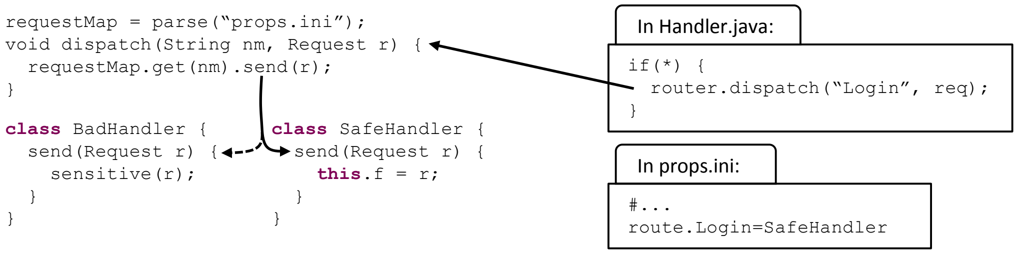
An alternative option is to write modular (or bottom-up) analyses [20, 32]. Instead of generating summaries for multiple (or infinite) calling contexts in a top-down setting, bottom-up analyses may generate summaries for methods (including library code) valid over *all* calling contexts. However, as noted by Zhang et al. [76], bottom-up approaches may ultimately need to analyze exponentially many input states limiting their scalability in practice. Thus, although theoretically appealing, “designing and implementing [modular analyses] for realistic languages is challenging” [41].

Finally, instead of relying on the hand-written or unsound rule-of-thumb summaries described above, many authors have explored automatically inferring specifications for missing library methods [12, 21, 45, 53, 56]. For example, in the context of a taint analysis, Bastani et al. [10] infer the specifications for missing methods needed to complete flows from sources to sinks. These specifications are presented to the user as candidate method specifications. Albarghouthi et al. and Zhu et al. [1, 78] have both explored using abduction to infer the minimal method specifications to verify the absence of errors. These techniques are promising, but they are currently limited to relatively simple specifications, require a human oracle, or focus on inferring preconditions for methods. These limitations mean that these techniques are unlikely to infer, e.g., the behavior of Java’s thread pool or executor APIs.

The decision to exclude library implementations is motivated by scalability concerns but also affects soundness. What impact do these decisions have on the analysis results reported in the literature? It is hard to say: the answer is certainly “a non-zero number” but to our knowledge there is no empirical study on false negatives due to excluded library code nor is this commonly reported in existing analysis results. It is up to analysis evaluators (who are usually also the analysis designers and implementers) to decide if this unsoundness arises in practice for the applications being analyzed. Unless the community can devise convincing experiments that the effects of excluding library code are negligible, the current approaches used may undermine the credibility of static analysis results.

2.2 Frameworks

Applications in complex domains (e.g., web applications, GUI programs) require a common set of functionality that does not vary significantly from application to application. For example, most web applications must parse incoming HTTP requests and dispatch them to the appropriate handler code. Rather than reimplement this functionality, applications use



```
requestMap = parse("props.ini");
void dispatch(String nm, Request r) {
  requestMap.get(nm).send(r);
}

class BadHandler {          class SafeHandler {
  send(Request r) {           send(Request r) {
    sensitive(r);               this.f = r;
  }                           }
}                           }
```

```
if(*) {
  router.dispatch("Login", req);
}
```

```
#...
route.Login=SafeHandler
```

Figure 1 An invented program fragment that demonstrates string indirection commonly found in frameworks. Without the routing information in `props.ini`, the analysis must conservatively assume the program dispatches to `BadHandler` (dashed line). Many framework models incorporate this type of information.

frameworks.[3] Frameworks are skeleton applications with holes for application specific code. Frameworks generally handle "boring" tasks (e.g., parsing HTTP requests or dispatching incoming UI events) and allow the programmer to focus on application specific tasks, e.g., responding to an HTTP request or UI event.

Frameworks are notoriously hard to analyze. In the interest of reusability, framework implementations rely heavily on language features that are difficult or impossible to analyze in general, such as reflection [11, 46]. This design makes basic call-graph construction (a basic requirement of any whole program static analysis) incredibly difficult. In addition to reflection, frameworks often use multiple layers of abstraction that confound most static analyses. For example, in Figure 1, finding the exact callee of `send()` in the `dispatch()` method requires reasoning about the precise key/value pairs present in the `requestMap` variable. Without this information, the static analysis must conservatively assume any handler is invoked, leading to a false report in `BadHandler`.

However, making matters even worse, frameworks are often configured using annotations [27], XML files [66], or other static sources. For example, the mapping information necessary to precisely resolve the `send()` call in Figure 1 is found only in the configuration file `props.ini`! Another example of configurations, simplified from an application we encountered while evaluating our static analysis, is shown in Figure 2. A static analysis must either consider these external artifacts (which requires deep domain knowledge) or make conservative assumptions about the behavior of the framework (leading to a precision loss and corresponding performance hit). Ignoring a framework's code entirely is not a realistic option: applications written using frameworks often lack a distinguished "main" function making even basic call-graph construction impossible.

In practice, static analysis writers either laboriously hand write models[4] of frameworks [70, 7] or avoid evaluating their analysis on framework applications. The latter option is unrealistic considering trends in application engineering but is understandable given the former option: constructing framework models by hand is a time-intensive and frustrating process. Our own experience analyzing Java web applications that use the Servlet framework is representative of this difficulty. The Servlet framework is relatively simple but building a sound model of the framework required reading parts of three specification documents: the Servlet, JSP

[3] The line between a library and framework is fuzzy. In this context, we use framework to refer to code that provides scaffolding upon which an application is built.

[4] A model is a compact, potentially non-executable, description of framework behavior.

```
1 <bean id="filterChain" class="FilterChain">
2   <property name="chain">
3       PATTERN_TYPE_APACHE_ANT
4       /logout=logoutFilter,anonymousProcessingFilter
5       /login=basicProcessingFilter,rememberMeProcessingFilter
6   </property>
7 </bean>
```

Figure 2 A simplified framework configuration fragment. The `filterChain` "bean" is bound to an instance of `FilterChain`. The values of fields of the `chain` bean are configured with property elements (line 2). Lines 3–5 define a tiny url-mapping DSL stored in the `chain` field. Building a complete model of this configuration requires not only a model of bean definitions, but an interpreter for this DSL. In our analysis we opted for a one-off, hand-coded interpretation of the DSL.

(JavaServer Pages) and EL (Expression Language) specifications, which together total 557 pages of prose. The Servlet framework is not an outlier: the reference document for Spring [66], a framework that builds on the Servlet framework, totals *910* pages.

Building a good model requires more than just understanding the framework. In addition to being sound, a model must be precise enough that client analyses can complete in reasonable amounts of time. For example, the largest performance gains in our analysis did not come from optimizations in the core analysis, but from aggressively including more domain specific knowledge into our Servlet model to improve call-graph precision.

Our community recognizes the difficulty of building these models: as recently as 2015 [11], a complete model of the Android framework was a significant research contribution. In addition, there has been work to simplify writing these models using a DSL [67]. However, expecting static analysis writers to build sound and efficient models for every framework is unrealistic. Other techniques [7, 48, 28, 70, 31, 75, 77] also require some form of programmer annotation or development which limits their adoption to new frameworks. However, evaluating new analyses on applications that use older, simple-to-model frameworks is equally undesirable as it ignores trends in modern software development.

2.3 High-Level API Knowledge

Analysis writers often require domain knowledge about the behavior of an API. For example, to soundly construct call graphs, analyses must handle the concurrency and reflection APIs of the Java Class Library. The reflection and concurrency APIs are just one example: many different analyses need high-level knowledge about an API. For example:

- What methods read or write from the database? [68]
- What methods return personal or sensitive information? [7]
- What methods may block execution of the current thread? [40]
- What methods and classes are part of a container abstraction? [23]

The answers to these questions are difficult to extract automatically and require reading the relevant documentation. The unfortunate state-of-the-art is that a static analysis developer interested in the answers to these questions must therefore manually audit an API to find the methods of interest. This is no trivial task: the reflection API alone contains over one hundred methods spread across 17 different interfaces and classes. The methods found during the audit are then usually added to a list of "special" methods; the analysis developer must then incorporate *ad hoc* handling for these methods to the analysis. For example, the call-graph construction facility of Soot [71], a popular analysis framework for Java, contains a hard-coded list of reflection and thread methods. WALA [72], another framework for Java, maintains its own list in an external XML file.

For many combinations of analysis domains and APIs, it is likely another analysis author has already performed a similar audit. However, no shared infrastructure exists to reuse and share the results of these audits, condemning analysis writers to re-audit APIs. In addition to wasting time, this process is error prone: failure to properly account for high-level API knowledge may make an analysis unsound. We encountered an otherwise sound and precise alias analysis that failed to consider `Class.newInstance()` an allocation site and therefore could not find aliases of reflectively instantiated objects.

3 Future Directions

The problems described in the previous section are not insurmountable. This section sketches future research directions, community initiatives to overcome some of these challenges, and our research agenda.

3.1 Toward Sound Library Handling

As noted in Section 2.1, the size of application code is often dwarfed by library code, leading static analysis authors to exclude the library code out of scalability concerns. We sketch future research directions to address these concerns.

Exhaustive Top-down Summaries. Top-down function summaries are difficult to reuse across analysis runs, as they are highly context-dependent and the probability of reaching a calling context identical to one in cache decreases as the complexity of the domain increases. Exhaustively enumerating input calling contexts as proposed in some work [58] becomes infeasible as the complexity and size of possible input contexts grows.[5] Recent work on StubDroid [6] by Artz and Bodden addresses some of these issues by soundly handling holes in the library call-graph and automatically computing the input contexts of interest. However, their approach assumes a specific representation of dataflow facts within a particular analysis domain. Nevertheless, this technique represents a promising step forward toward library summary precomputation. Our community should explore how to generalize these techniques to work on any combination of dataflow facts and analysis.

Analyzing Analyses. We plan to explore developing automated techniques to compare the power of two or more analyses. In particular, we plan to develop an automated semi-decision procedure that can determine if one analysis always over-approximates another analysis on all code fragments. In other words, the procedure will decide if the results of one analysis imply the results of another analysis on all programs. Recent work on comparing the behaviors allowed by memory models [74] has shown that it is possible to answer these types of queries using automated theorem provers such as Z3 [22].

This technique will have several important applications. This research may enable sound reuse of cached analysis results from different analyses. If the procedure determines analysis A over-approximates analysis B, then cached results from B may be safely reused within A (with some loss of precision). The developed procedure will also allow our community to

[5] Even for seemingly simple domains (e.g., access-paths [69] limited to length 1), this approach is unlikely to scale.

compare the precision of two or more analyses. Finally, this procedure could find soundness bugs in analyses. A developer may choose a concrete interpreter as one "analysis", and query the semi-decision procedure to verify her analysis over-approximates the concrete interpreter.

Analysis Semi-Refinement. Recent work on caching and incremental analysis provide a promising approach to solving scalability concerns on large codebases. However, in the current state-of-the-art, cached results cannot be used across analyses, so every new analysis effectively begins with a blank slate of results to draw upon. No amount of caching helps if the initial run of an analysis never terminates! The research sketched above potentially alleviates this issue, but only if cached results *always* soundly over-approximate the analysis using these results. However, we expect that only highly related analyses in the same problem domain will exhibit this property, which in turn limits opportunity for reuse.

We hypothesize that there are analyses that may not always produce over-approximate results but may sometimes agree under certain conditions. We hope to explore automatically determining when one analysis *conditionally* over-approximates another. For example, two analyses may model the heap incompatibly, but otherwise produce the same results on code with no heap accesses. In this case, cached results for a code fragment may be shared between analyses if the fragment does not access the heap. Given two analyses A and B, we aim to synthesize a predicate such that analysis B over-approximates A on fragments of code for which the predicate is true. If the predicate reduces to a simple syntactic check, results or summaries from unrelated analyses can be easily reused by another analysis to improve efficiency.

Automatic Synthesis of Weakened Analyses. A common technique for static analysis is to add precision "knobs" to an analysis [34, 37, 38]. These knobs allow the analysis user to trade performance for precision. However, constructing these knobs requires careful engineering on the part of the analysis designer and implementer. Similarly, staged analyses (e.g., [33, 26, 36]) exploit a precision/performance tradeoff by iteratively applying more and more precise analyses to suppress false positives or discharge verification of conditions not provable by less precise analyses. Unfortunately, the staged analysis designer must either "luck" into two or more analyses that yield compatible results with different levels of precision, or (more likely) design and implement multiple (related) versions of an analysis.

We plan to research techniques to *synthesize* less precise (but more scalable) versions of existing analyses. These weakened analyses may be used as fast preanalyses, or to handle large library codebases. One possible direction for this work is to develop a technique to take flow-sensitive analysis and create a flow-*insensitive* version (in the style of Andersen points-to analysis [4]). In addition to this flow-sensitive/-insensitive tradeoff (which is well-known within our community), we plan to explore other axes along which analyses may be transformed for performance gains.

3.2 Sound, Automated Handling of Frameworks

The techniques discussed in Section 2.2 for handling frameworks all rely on some manual effort by analysis writers. At times, new frameworks become popular and old ones make changes so fast that keeping up disincentivizes work in the space.[6] We therefore propose two possible research directions that handle frameworks without programmer intervention.

[6] Krishnamurthi reports this experience in work on semantics for JavaScript [39].

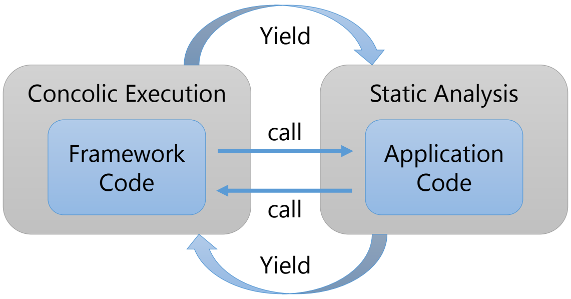


Figure 3 High-level architecture of concolic analysis.

Concolic Analysis. Framework implementations are difficult to analyze statically, but when executed often follow only a handful of paths determined by static values that can be easily accessed at analysis time (e.g., a configuration file or annotations). These characteristics suggest that concolic execution [62, 63, 30] can be an effective approach to analyzing framework implementations. Concolic execution extends traditional symbolic execution by falling back on concrete execution for code that cannot be modeled symbolically (e.g., due to expressions unsupported in the underlying automated theorem prover). For example, a concolic executor can precisely handle framework code that reflectively instantiates objects based on static configuration values by simply executing the relevant code.

However, concolic execution by itself cannot analyze entire framework-based applications. Although the scalability of both concolic and symbolic executors has improved, and there have been amazing advances on semi-decision procedures like CEGAR [19], it remains a challenge to verify programs with many paths of execution and complicated data dependencies. In particular, on programs with infinite paths of execution, these techniques either fail to terminate, artificially finitize the program, or limit tool execution with a fixed time budget. Thus, concolic execution will struggle to verify, e.g., Android applications that process unbounded streams of input events, or web applications that accept infinite sequences of requests.

Given the scalability concerns of concolic execution (and other formal methods techniques) and the difficulty of precisely analyzing extremely flexible framework implementations, we believe that a single, unified analysis approach is insufficient to verify or analyze framework-based applications. We instead plan to explore a *hybrid* analysis technique that combines concolic execution and traditional static analysis. We have termed this technique concolic analysis. Under concolic analysis, framework code is executed concolically, whereas application code is over-approximated with a meet-over-all-paths static analysis. A visualization of this technique is shown in Figure 3. When control passes from the framework to the application, the concolic executor yields to a static analysis. Similarly, calls from the application back into the framework cause the static analysis itself to yield to the concolic executor. By using the best approach on each part of a program, concolic analyses combine the efficiency of static analysis and the completeness and precision of concolic execution. For example, reflective operations in framework code can be concretely executed, while unbounded loops in application code can be efficiently over-approximated using fixpoint iteration.

Although other authors have examined combining concrete execution and static analyses, these approaches have either used information recorded during executions in a static analysis [13, 24, 73], or used dynamic analysis as a post-processing step to prune false positives or discharge verification conditions [9, 42, 18]. In contrast, concolic analysis tightly couples different analysis techniques to cooperate concurrently to analyze different parts of a monolithic application.

Automatic Model Synthesis. Recent advances in synthesis technology and techniques have enabled automatic synthesis of complex hardware memory models [14] that were previously hand-axiomatized through multiple iterations of publications [61, 64, 54, 2, 3, 60, 49]. This pattern echoes current work on building models for the Android framework: several papers have been published over the years, each claiming more precise (and sound) models of a single framework. Our community should also focus on automatic synthesis of framework models.

Full specifications of complex frameworks are likely more complicated than those for hardware memory models, so complete specification synthesis is likely intractable. Thus, we envision focusing on synthesizing specifications describing framework behavior for a single, specific application. For example, the input/output behavior of framework methods can be recorded during either directed randomized testing (e.g., [30]) or execution of a program's functional test suite. These traces can be used as inputs to a synthesis procedure that generates specifications (expressed in a DSL) for framework methods. The quality of these generated specifications necessarily depends on the completeness of the observed traces. However, as noted above, frameworks are often driven by static, deterministic configurations and annotations, so we expect only a handful of executions will provide relatively complete set of input/output examples for the framework methods executed by an application.

3.3 Infrastructure for Sharing API Knowledge

Given the overlap in knowledge needs of static analysis writers, the static analysis community would benefit from an open platform to share API knowledge. The high-level API knowledge described in Section 2.3 can often be expressed in a few short English words. Concise tags therefore are a good format to express this API knowledge. We propose the community create and maintain a shared, open database that associates API elements (i.e., classes, methods, etc.) with tags that express the high-level knowledge needed by analysis developers.

Each tag would express a property that is common to multiple methods in different APIs. For example, the `TelephonyManager.getDeviceId()` method of the Android framework returns a unique identifier and is treated as a source of sensitive data for information integrity analyses. This method, and the analogous `UIDevice.uniqueIdentifier()` of the iOS framework could be tagged with the tag `"sensitive-source"`. The collection of methods associated with this tag in the proposed database would replace the hand-curated list of source methods used by many security analyses. Similarly, methods from the reflection API of the Java Class Library (e.g., `Class.newInstance` or `Method.invoke`) would be associated with the tag `"indirect-flow"` indicating that these methods indirectly invoke another method by name. As with the sensitive source example, the methods associated with this tag would replace the hard-coded lists found in many program analysis frameworks' call graph construction facilities.

The implementation and deployment of the tag database poses no major technical hurdles: similar web applications are widely deployed in industry and enjoy extensive library and framework support. We foresee there will be two major challenges. First, as tags are expressed using natural language, different users of the database may interpret the same tag differently. However, we are confident that the community can standardize around a set of tags with widely accepted and understood definitions. Second, although some tags may be assigned automatically (e.g., tags identifying setter and getter methods) other tags require human knowledge. For this type of information, we envision that after analysis developers manually collect domain knowledge for an API, they then tag the API elements in the shared database.

Although the development and deployment of the shared database does not present core PL research opportunities, this initiative will have an immediate impact on the community. For one, it will free the analysis developers from the tedious, error-prone task of auditing APIs, and improve the soundness of analyses by ensuring no important methods are missed (as in the alias analysis described in Section 2.3). Further, analyses in related domains could be fairly compared as all analyses would consider the same methods of interest (e.g., sources and sinks).

4 Conclusion

Despite advances in tooling and mainstream success, static analysis development is still a painful process. We have outlined our research vision for tackling some of these pain points. Our proposals do not represent the full space of solutions, and there are other difficult aspects of analysis development we have not addressed. Mitigating or eliminating the challenges faced by static analysis writers is a ripe area for research. We believe using static analysis and formal methods techniques to tackle these difficulties (i.e., static analyses for static analyses) is a particularly exciting research direction. In addition, we hope the community will invest in sharing knowledge and results across research projects. Our proposed tag database initiative is a potential first step; there are even more opportunities for community-wide collaboration to ease the burden of constructing static analyses.

Acknowledgments. The authors would like to thank Doug Woos, James Bornholt, Jared Roesch, Zachary Tatlock, and Chandrakana Nandi for feedback on early versions of this work. We would also like to thank our shepherd, Ruzica Piskac, and the anonymous reviewers for their insightful comments.

References

1 Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.

2 Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP*, 2009.

3 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *CAV*, 2010.

4 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

5 Steven Arzt and Eric Bodden. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *ICSE*, 2014.

6 Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *ICSE*, 2016.

7 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

8 Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

9 Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Symposium on Security and Privacy*, 2008.

10 Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, 2015.

11 Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2015.

12 Sam Blackshear and Shuvendu K. Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *PLDI*, 2013.

13 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

14 James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *PLDI*, 2017.

15 David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *CAV*, 2011.

16 Cristiano Calcagno and Dino Distefano. Infer: an automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, 2011.

17 Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. Open-sourcing facebook infer: Identify bugs before you ship. `https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/`, 2015.

18 Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for Javascript. In *PLDI*, 2009.

19 Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

20 Patrick Cousot and Radhia Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, 2002.

21 Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *CAV*, 2015.

22 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

23 Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

24 Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA*, 2007.

25 William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2), 2014.

26 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2), 2008.

27 Apache Foundation. Apache struts 2. `https://struts.apache.org/`.

28 Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android. Technical Report CS-TR-4991, University of Maryland, November 2009.

29 Patrice Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE software*, 25(5), 2008.

30 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

31 Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.

32 Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, 2007.

33 Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, 2011.

34 Ben Hardekopf, Ben Wiedermann, Berkeley Churchill, and Vineeth Kashyap. Widening for control-flow. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2014.

35 Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE*, 2013.

36 Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, 2007.

37 Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: a static analysis platform for javascript. In *FSE*, 2014.

38 Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale javascript applications (t). In *ASE*, 2015.

39 Shriram Krishnamurhti. Personal Communication, 2016.

40 Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for c/pthreads. In *ASE*, 2016.

41 Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. Accelerating program analyses by cross-program training. In *OOPSLA*, 2016.

42 Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *Partial Evaluation and Semantics-based Program Manipulation*, 2008.

43 James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE software*, 21(3), 2004.

44 Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *ASE*, 2015.

45 Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, 2009.

46 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, JoséNelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2), 2015.

47 Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014.

48 Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

49 Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWERmultiprocessors. In *CAV*, 2012.

50 Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *FSE*, 2013.

51 Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient incremental static analysis using path abstraction. In *FASE*, 2014.

52 George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, 2002.

53 Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, 2002.

54 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

55 Lori L. Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12), 1989.

56 Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *PLDI*, 2007.

57 Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *FSE*, 2016.

58 Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, 2008.

59 Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, 2015.

60 Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

61 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 *POPL*, 2009.

62 Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.

63 Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

64 Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7), 2010.

65 Amie L. Souter and Lori L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *ICSM*, 2001.

66 Spring framework. `http://spring.io/`.

67 Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA*, 2011.

68 Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep typechecking and refactoring. In *OOPSLA*, 2008.

69 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, 2013.

70 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI*, 2009.

71 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, 1999.

72 WALA – T. J. Watson Libraries for Analysis. `http://wala.sf.net/`.

73 Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *ISSTA*, 2013.

74 John Wickerson, Mark Batty, Tyler Sorensen, and George A Constantinides. Automatically comparing memory consistency models. In *POPL*, 2017.

75 Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *ICSE*, 2015.

76 Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. In *PLDI*, 2014.

77 Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

78 Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *ASPLAS*, 2013.

Programming Language Abstractions for Modularly Verified Distributed Systems

James R. Wilcox[1], **Ilya Sergey**[2], **and Zachary Tatlock**[3]

1 **University of Washington, Seattle, WA, USA**
`jrw12@cs.washington.edu`
2 **University College London, London, UK**
`i.sergey@ucl.ac.uk`
3 **University of Washington, Seattle, WA, USA**
`ztatlock@cs.washington.edu`

Abstract

Distributed systems are rarely developed as monolithic programs. Instead, like any software, these systems may consist of multiple program components, which are then compiled separately and linked together. Modern systems also incorporate various services interacting with each other and with client applications. However, state-of-the-art verification tools focus predominantly on verifying standalone, *closed-world* protocols or systems, thus failing to account for the compositional nature of distributed systems. For example, standalone verification has the drawback that when protocols and their optimized implementations evolve, one must re-verify the entire system from scratch, instead of leveraging compositionality to contain the reverification effort.

In this paper, we focus on the challenge of modular verification of distributed systems with respect to *high-level protocol invariants* as well as for *low-level implementation safety properties*. We argue that the missing link between the two is a programming paradigm that would allow one to reason about both high-level distributed protocols and low-level implementation primitives in a single verification-friendly framework. Such a link would make it possible to reap the benefits from both the vast body of research in distributed computing, focused on modular protocol decomposition and consistency properties, as well as from the recent advances in program verification, enabling construction of provably correct systems implementations. To showcase the modular verification challenges, we present some typical scenarios of decomposition between a distributed protocol and its implementations. We then describe our ongoing research agenda, in which we are attempting to address the outlined problems by providing a typing discipline and a set of domain-specific primitives for specifying, implementing and verifying distributed systems. Our approach, mechanized within a proof assistant, provides the means of decomposition necessary for modular proofs about distributed protocols and systems.

1998 ACM Subject Classification D.3.3 [Programming Languages] Language Constructs and Features, F.3.1 [Logics and Meanings of Programs] Specifying and Verifying and Reasoning about Programs

Keywords and phrases Distributed systems, program verification, distributed protocols, domain-specific languages, type systems, dependent types, program logics

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.19

1 Introduction

As with any software, distributed systems are not built as standalone pieces of code: rather they are assembled from multiple independently developed components. For instance, different nodes may communicate using message passing, components of a particular implementation

2nd Summit on Advances in Programming Languages (SNAPL 2017).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 19; pp. 19:1–19:12
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

may be compiled separately, and different systems may interact with each other and with the client applications via regular program flow and by imposing implicit invariants on each other's behavior.

There is a vast amount of work dedicated to establishing and verifying invariants of standalone *distributed protocols*, such as Paxos [23, 24, 61], Raft [46], *etc*, formulated as abstract high-level state-transition systems (see, *e.g.*, [62, 61, 20, 44] for references). Furthermore, several impressive advances have been recently made in verifying specific realistic *systems implementations* with respect to fixed properties [64, 16, 27, 11, 48, 65, 22, 32]. However, the modular nature of these systems is not fully matched by state-of-the art verification techniques, which still follow a "whole-program" approach. Specifically, most of the verification methodologies to date require a complete revision of the proofs (or are not applicable at all) in the following scenarios, which occur regularly in the life cycle of distributed software:

1. A high-level protocol $\mathcal{P}$ (*e.g.*, Paxos) remains the same, but its implementation run by a particular node is updated (*e.g.*, replaced by an optimized one [4]). Naturally, one should now establish that the new implementation *refines* (*i.e.*, exhibits the same externally observable behavior as) the same abstract protocol [1], while all proofs concerning the protocol itself should not change.
2. As a variant of the previous scenario, an optimization in $\mathcal{P}$'s implementation might *delegate* some of the computation to another node, possibly following another protocol $\mathcal{P}'$ [61]. In this case, one should establish that, under certain assumptions about $\mathcal{P}'$, the resulting implementation of $\mathcal{P}$ still refines its specification.
3. An implementation, interacting with other nodes under a protocol $\mathcal{P}$, may make specific assumptions about the initial state of the system, thus restricting the set of reachable states. This is captured by strengthening the protocol's state-space invariant, thus permitting the implementation to leverage additional facts about its state. Such strengthening should *not* cause the proofs of implementations run by other involved nodes to be revised.

The first scenario is fairly standard: one should always be able to make low-level optimizations in an actual implementation, as long as these changes *are not observable* on the abstract level, with the high-level protocol serving as a system specification. The existing solutions [16, 64] for this modularity challenge rely on the classical technique of establishing a *refinement* [1, 26] between an actual implementation (the code) and a specification (a protocol) via forward-backwards simulation [37]. That said, in the presence of program-level composition (*e.g.*, third-party libraries), recursion, and higher-order programming primitives, proving refinement in a modular way becomes a notoriously difficult problem, requiring a non-trivial relational semantics and dedicated program logics. While such logics exist for shared-memory concurrency [59, 30, 56], none exist for distributed systems. The situation is even more complicated in the presence of *fine-grained* communication primitives, such as send and receive (as opposed to synchronous models [13]), that are used for implementing non-blocking message-passing. To the best of our knowledge there is no program logic that supports reasoning about fine-grained message-passing distributed systems in a modular way, and the state-of-the-art approaches either avoid fine-grained operations all together [10, 11], thus sacrificing potential performance gains, or employ first-order reduction techniques [16, 31, 12].

The second scenario demonstrates an interplay between properties of a protocol and proofs of an implementation that relies on them: indeed, the correctness of a refinement by the latter depends on the invariants of the former. Yet, from a programmer's perspective this is just another program optimization, so the proofs should not be that different from those in the Scenario **1**. However, we are not aware of any verification frameworks allowing one to modularly prove refinement between an implementation and its protocol in this case.

The third scenario demonstrates a common pattern where a protocol implementor assumes the system is initialized to a certain "good" state. This implies any subsequent state of the system is *reachable* from the good state, which can be used to establish additional safety properties. This scenario allows different client implementations using a protocol to rely on different assumptions about its initial conditions and different system invariants. Combined with the second scenario, this means that one should be able to impose custom (but valid) invariants when proving an implementation-specific refinement!

To make things more concrete, let us imagine implementing an optimization of a straightforward distributed computation (*e.g.*, MapReduce), run by a node, that memoizes its past results using some third-party distributed storage. Then, an important invariant of a storage protocol, required for justifying such an optimization, should state that the stored values are never dropped or replaced. However, another client application, which only *queries* the storage but does not *write* into it might be verified under a weaker invariant. From this observation we conclude that *one and the same* distributed protocol might be a subject of different application-specific invariants (since the strongest possible invariant is not alway possible to foresee in advance) and initial state assumptions, but imposing a different inductive invariant should not affect already verified protocol implementations and their proofs.

From the discussion above, it seems that the proofs of refinement, *i.e.*, that an implementation "does not go wrong", are unavoidable for formally establishing the correspondence between the code of an implementation and its abstract protocol specification. In this line of research, in an attempt to overcome the complexity of the refinement proofs, which become especially acute in the presence of horizontal composition of interacting distributed services (*i.e.*, Scenario **2**) and client-specific invariants (*i.e.*, Scenario **3**), we have decided to adopt a different approach for proving programs well-behaved: by means of type theory.

2 A Type-Based Approach to Distributed System Verification

We have drawn inspiration from results on Hoare Type Theory (HTT) [43, 42, 41] and specifically its recent variants, which support specifying and verifying fine-grained shared-memory concurrent algorithms [40, 50, 51, 52]. In HTT, an effectful, imperative, potentially higher-order program e is given a Hoare type $\mathsf{HT}\ \{\lambda s.\ A\}\{\lambda r\ s'.\ B\}$, where A is a predicate constraining the pre-state s (*e.g.*, a heap), and B constrains the result r and the post-state s'. That is, the pre-/postconditions A and B declaratively specify the effect of e with respect to the state it might affect. Furthermore, the original HTT incorporated Separation Logic-style specifications [41] and adopted *fault-avoiding* semantics [49], thus ensuring that well-typed programs are *memory-safe*. The concurrent extensions of HTT extended the notion of type safety to account for *data race freedom* [28] and *coherence* of a concurrently used resource [40].

Distributed Hoare Types. In this work, we extend the notion of Hoare types to distributed system implementations, whose "state" captures both local components (*e.g.*, a heap) and a global component, namely the (multi-)set of messages exchanged by the nodes involved in the system. In this way, "effects" correspond to interactions in a distributed environment between nodes via message passing. Each such interaction (*i.e.*, sending or receiving a message) is synchornized with a change in a node's local state (*e.g.*, updating a set of local permissions). These changes follow one of several available "atomic" transitions, which are provided by user-defined high-level protocols $\mathcal{P}_1, \mathcal{P}_2$, *etc*, which are encoded as state transition systems. All together, they form a part of the type environment when assigning a type to such a program. Thus, the Hoare type judgements assigning types to distributed

$$\frac{\mathcal{P}_1 \vdash e : \mathsf{DHT}\{\lambda s.\ A\}\big\{\lambda r\ s'.\ B\big\} \qquad A, B, R \text{ are stable} \qquad R \text{ constrains state related to } \mathcal{P}_2}{\mathcal{P}_1, \mathcal{P}_2 \vdash e : \mathsf{DHT}\{\lambda s.\ A \wedge R(s)\}\big\{\lambda r\ s'.\ B \wedge R(s')\big\}}\ \textsc{Inject}$$

$$\frac{\mathcal{P} \vdash e : \mathsf{DHT}\{\lambda s.\ A\}\big\{\lambda r\ s'.\ B\big\} \qquad I \text{ is inductive } \textit{wrt.}\ \mathcal{P}}{\mathsf{WithInv}(\mathcal{P}, I) \vdash e : \mathsf{DHT}\{\lambda s.\ A \wedge I(s)\}\big\{\lambda r\ s'.\ B \wedge I(s')\big\}}\ \textsc{WithInv}$$

Figure 1 Selected type inference rules of Distributed Hoare Types.

implementations are of the shape $\mathcal{P}_1, \ldots, \mathcal{P}_n \vdash e : \mathsf{DHT}\{\lambda s.\ A\}\{\lambda r\ s'.\ B\}$, where the typing context $\mathcal{P}_1, \ldots, \mathcal{P}_n$ lists all of the abstract protocols that the program e can exercise, and the pre/postcondition constrain the state of the protocol-related part of the network. Each protocol defines the per-node local state, which is governed by the protocol's transitions. One node can possibly host disjoint pieces of local state that "belong" to different protocols, which is crucial to allow composing multiple protocols together to form useful systems. In addition to the send/receive primitives, all the standard programming constructs, such as conditionals, recursion, and higher-order functions can be used, and the typing rules for them are straightforward.

In any interesting distributed protocol, there are dependencies between messages about to be sent and the protocol-specific local state of a node that can send them. These dependencies are what our rich type system is designed to *enforce*. For instance, in any Paxos implementation, a replica can only send a response to a client when it is certain that agreement has been reached [61]. A protocol for Paxos would enforce this by constraining the *precondition* of sending a response to require that agreement had been reached. These constraints are manifested in the Hoare types, which are derived for the basic send/receive commands from the definitions of the transitions they follow. Since there is *no other way* to interact but by relying on the protocol-supplied transitions, this provides a powerful mechanism for enforcing system-specific constraints. For instance, in a well-typed program e, following a protocol $\mathcal{P}$, it will be only possible to send a certain message if the precondition in the corresponding transition τ_s, is satisfied by the node's local state.[1]

The notion of well-typedness for Distributed Hoare Types incorporates program well-formedness with respect to the protocols in its typing context: no matter how complex the program is, if it is well-typed, then each of its externally observable transitions "faithfully" follows a transition of some of the protocols from its typing context, *i.e.*, it *does not go wrong* [38]. Summarizing the high-level overview of our approach, to enable language-based verification [53] of distributed systems, we have introduced the two following program- and type-level mechanisms to the otherwise well-studied model of higher-order effectful programs [50]:

(a) Instrumented message-passing primitives (send/receive), derived from protocol transitions, serving as basic building blocks for distributed programs;

(b) Distributed Hoare Types (DHT), an extension of Hoare Types [41], as a compositional approach to verify well-behaved programs in a context of arbitrary user-provided protocols.

Addressing the Modularity Challenges. Let us now see how the type-based approach helps alleviate the main difficulties of modular refinement proofs, outlined in Section 1.

[1] In fact, our type system allows for more general assertions, constraining the *global* state of the system.

1. Since any well-typed implementation must follow the protocol, type safety immediately implies refinement Moreover, Distributed Hoare Type Theory enjoys the standard substitution principle, which allows one to replace any program of a type $\mathsf{DHT}\{\lambda s.\ A\}\{\lambda r\ s'.\ B\}$ by any other program with the same type without compromising type safety.
2. From the perspective of a type system, there is no difference between a value obtained as a result of a local computation or the one received from a remote service, as long as it allows the desired Hoare type to be derived. Furthermore, Distributed Hoare Types allow for a form of *context weakening*, making it possible include more protocols (and account for interactions involving these protocols) into the typing context by adapting the pre/postconditions appropriately via the rule Inject from Figure 1.[2] The stability requirement on R is standard for concurrency program logics and means that the assertions should be invariant with respect to possible concurrent changes in the network state [60].
3. The proof of an invariant I being inductive with respect to a protocol $\mathcal{P}$ is not tied to a specific implementation e, and, therefore, can be discharged via an external verification tool (*e.g.*, Ivy [47]). That said, the invariant itself, once proven, can be used for strengthening the type of e, possibly enabling one to prove some properties of e's clients. The interaction between protocol-level proofs and program-level verification is enabled by the typing rule WithInv from Figure 1. The *protocol combinator* WithInv enhances the state-space invariants of $\mathcal{P}$ conjoining them with the invariant I.

Relation to Refinement Proofs. Our careful choice of basic programming primitives, namely, *protocol transitions*, is the trick that allowed us to replace expensive proofs of program refinement with a far less complicated (although still non-decidable) and uniform type derivation mechanism. While this model might seem to be too "coarse-grained" in the sense that it forces changes in the protocol-relevant local state to be atomically synchronized with sending/receiving messages, the model nevertheless leaves a lot of room for possible program-level optimizations. Specifically, it allows one to combine the transitions in any well-typed way, as well as allowing one to make use of any internal state and higher-order programming primitives. What is more important is that our model explicitly identifies valid *linearization points* [17] in the implementations (they correspond precisely to the taken transitions), thus adopting a well-established proof method for observational refinement [14].

3 Language-Based Verification with Distributed Hoare Types

Distributed Hoare Types can be effectively represented as dependent types, parametrized by the protocol contexts and pre/postconditions [42]. This allowed us to implement the type-based verification approach, sketched in Section 2, in a verification tool Disel, by embedding our type system, its semantic foundations, and inference rules into the Coq proof assistant [6]. In this section, we outline the layout of specifications and proofs using a characteristic example of a widely-used distributed system: Two-Phase Commit (2PC) [63, §19].

The goal of the 2PC protocol is to achieve agreement among several nodes about whether a transaction should be committed or aborted (*e.g.*, as part of a distributed database). Since the system may execute in an asynchronous environment where message delivery is unreliable and machines may experience transient crashes, achieving agreement requires care. The protocol designates a single node as the *coordinator*, which is in charge of managing the

[2] These rules allow us to consider Distributed Hoare Types as a program logic-based verification framework.

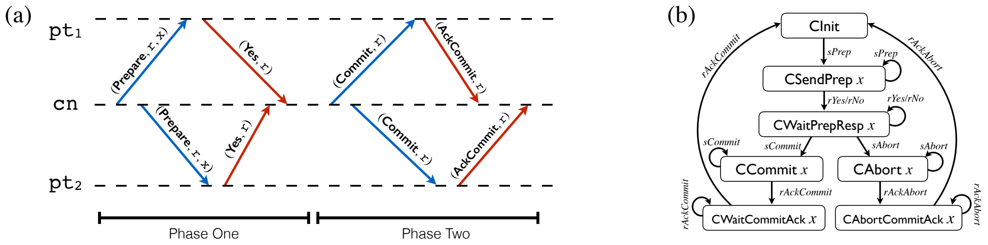


Figure 2 One round of the 2PC algorithm (a) and state-space of the coordinator (b).

```
Definition c_recv_step (r : round) (cs : CState)
            (log : Log) (tag : nat) (mbody : seq nat) :=
  | CWaitPrepResp x ⇒ if (* received all votes *)
     then (r, if (* all votes yes *) then CCommit x else CAbort x, log)
     else (r, CWaitPrepResp, log)
  (* ... more cases depending on cs, tag, mbody ... *)
  end.
```

Figure 3 Example receive transitions of the coordinator.

commit process; other nodes participating in the protocol are *participants*. The protocol proceeds in a series of rounds, each of which makes a single decision. Each round consists of two phases; an example round execution is shown in Figure 2(a). In phase one, the coordinator notifies the participants of the transaction being committed by sending *prepare* messages and receives *votes* from the participants about whether the transaction should proceed. In the figure, both participants vote Yes, so the coordinator enters phase two, during which it notifies all participants of its decision to commit or abort the transaction.

Formalizing this description into a protocol consists in describing the local state of each node as well as the valid transitions. Figure 2(b) shows the relevant portions of the local state of the coordinator and its transitions. Between rounds, the coordinator waits in the CInit state. Then, the coordinator makes transitions following the informal description above; these are formalized by the step-function, one case of which is shown in Figure 3. The additional state components keep track of the round number and a log of all processed transactions.

With the protocol instance in hand, we can now proceed to build programs that implement the participant and coordinator and assign them Hoare-style specifications. A possible implementation of a single round of the coordinator and its Hoare type are shown in Figure 4. The function `coordinator_round` takes as an argument the transaction data to be processed in this round. The type `DHT [cn, TPC]` is parametrized by the coordinator node id `cn` and a 2PC protocol instance `TPC`. The precondition requires that the coordinator is in the CInit state, with an arbitrary round number and log. The postcondition ensures that the local state has returned to CInit, the round number has been incremented, and the return value accurately reflects the decision made on the data, which is also reflected in the updated log. The code proceeds along the lines required by the protocol, but nothing prevents us from writing an optimized implementation, *adhering to the very same type*, which could, for instance, send abort-request upon receiving the first Phase One Abort response.

The type ascribed to `coordinator_round` above only constrains the local state of the coordinator, but in fact the protocol maintains stronger global invariants. For example, imagine using the Two-Phase Commit protocol as part of a larger distributed database system. Database nodes participate in several copies of the Two-Phase Commit protocol, one per node, so that each node is the coordinator of one copy of the protocol. Nodes

```
Program Definition coordinator_round (d : data) :
  {r log}, DHT [cn, TPC] (fun s ⇒ loc cn s = (r, CInit, log),
                          fun res s' ⇒ loc cn s' = (r+1, CInit, log ++ [(res, d)])) :=
  Do (r ← read_round;
      send_prep_loop r d;;
      res ← receive_prep_loop r;
      b ← read_resp_result;
      (if b then send_commits r d;;
                 receive_commit_loop r
       else send_aborts r d;;
            receive_abort_loop r);;
       return b).
```

Figure 4 Distributed Hoare type and code of a single coordinator round.

can then commit transactions by initiating Two-Phase Commit in the copy of the protocol they coordinate. The database might like to conclude that between rounds, all logs are in agreement. This strong global agreement property is not directly implied by the protocol as it stands, so we must prove an inductive invariant that implies it. Finding such invariants typically requires several iterations before converging on a property that is inductive and implies the desired spec. In this case, a state invariant Inv that closely follows the intuitive execution of the protocol suffices to prove the global log agreement property. For example, when the coordinator is in the CSendCommit state, the invariant ensures that all participants are either waiting to hear about the decision, have received the decision but not acknowledged it, or have acknowledged the decision and returned to the initial state. The invariant also implies a simple statement of global log agreement, shown below.

```
Lemma cn_log_agreement (s : state) (r : round) (log : Log) :
  Inv s → loc cn s = (r, CInit, log) → ∀ pt, pt ∈ pts → loc pt s = (r, PInit, log).
```

In other words, when the coordinator cn is in the CInit state, all participants pt ∈ pts must be in the PInit state with the same round number and log.

We can freely use the strengthened invariant in proofs of programs. For example, in the hypothetical database example, the programs implementing the database can now conclude global log agreement from the fact that the local state is CInit.

4 Related Work

Type-based reasoning about concurrent and distributed systems

Session Types (ST) [18] are one of the most established approaches for lightweight verification of message-passing programs. ST were originally designed to constrain two-party channel communications, enforcing a particular interaction protocol; they were later extended to specify interactions between several parties [19, 8] and quantify over values of messages [55]. This has culminated in research on *choreographies* [3], which identify allowable orderings of message exchanges in a distributed system. Even though (Multiparty) Session Types (MST) [19] and Distributed Hoare Types pursue the same goal, namely, enforce the protocol discipline in an distributed setting with asynchronous message-passing, they seem to achieve this by different means. The underlying semantic formalism of MST is π-calculus [39], in which computations communicate via dedicated *session channels* that are a central notion for enforcing the well-formedness of executions via a tailored type system. In contrast, DHT adopts a model similar to those from modern program logics for fine-grained shared-memory concurrency [9, 40, 57, 54], in which messages of a specific protocol are treated as a *shared* *state*, related to local state of specific nodes via the protocol invariants and a subject to change as defined by the transitions.

While the precise relation between MST and DHT is still to be determined, we believe that our representation of distributed protocols via transition systems governing local/shared state is much closer in spirit to the models employed by the distributed systems community to describe the high-level logic of state-of-the art consensus and replication algorithms and their properties [25, 34]. It is not immediately clear to us how to encode Paxos, Raft or Two-Phase Commit using MST. Furthermore, the only language-level extension required to support a DHT programming model was the introduction of protocol-aware send/receive primitives and typing rules for them; the remaining language fragment is entirely standard. For instance, in our implementation the host language is Coq's Gallina [6] extended, via monadic embedding, with general recursion and message passing. This has the benefit that one can use the full power of Gallina to implement distributed programs. Finally, MST provide little support for reasoning about protocols themselves, separately from the programs they implement. This is something that is afforded for DHT using the WithInv rule.

A very close type-based formalism to DHT are RGRefs [15], allowing one to enforce a Rely/Guarantee-discipline [21] for mutable references in a shared-memory concurrency setting. That said, while RGRefs are suitable for showing that a program follows specific Rely/Guarantee-protocol, they are too weak to prove its invariants or functional correctness.

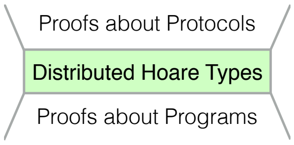
Modular verification of distributed protocols

Compositional verification of invariants of distributed protocols is an area of active research in the Distributed Computing community (*cf.* [2, 33, 62]). There, it is common to reduce the reasoning about message-passing concurrency to reasoning about shared-memory mechanisms. For example, Boichat *et al.* [2] suggest a series of abstractions, such as *round-based consensus* and *round-based register*, that make it possible to deconstruct a family of Paxos algorithms into a set of reusable primitives. Input/Output automata [36] are another high-level formalism allowing for a form of protocol composition by coupling the automatas' actions [35]. At the moment, all these constructions are only studied at the level of reasoning about protocols, without any relation to implementations. We believe that these abstractions can be incorporated into the framework of DHT by generalizing the notion of the shared state to incorporate both message-passing (which is currently the case) and shared memory. Such a unification would make it possible to immediately reuse many of the existing specification and proof techniques from the logics for shared-memory concurrency, for instance, when defining custom correctness conditions [52].

Datta *et al.* [7] propose Protocol Composition Logic (PCL) as a way to combine security properties of multiple distributed protocols governing processes, communicating with each other. The programming component of PCL is a conventional process calculus. At the moment, it is not clear to us the extent to which PCL can be employed to verify, e.g., consensus protocols such as 2PC, or to be employed for reasoning about higher-order code.

5 Concluding Remarks

We have outlined the main ideas behind Distributed Hoare Types – a typing discipline that allows one to enforce high-level protocol logic in a low-level implementation via dependent types.



We believe that DHT serves as a link, connecting proofs of protocol properties and program properties in the same logical framework while providing modular reasoning. This modularization hints for a number of follow-up extensions, moving both up and down the abstraction stack.

Moving up: Reasoning about protocols. Thanks to the rule WithInv, reasoning about inductive protocol invariants can be conducted independently of the program-level verification. At the moment such proof obligations are discharged via Coq's native machinery for interactive proofs, and we are planning to investigate the possibility to delegate these proofs to third-party tools, such as Ivy [47], which is designed for this specific purpose. Furthermore, there is currently only one linguistic way to formulate protocols in the framework of DHT: by synchronizing the state changes with sending/receiving. This model is sufficiently fine-grained to be able to encode more transitional I/O Automata [35] or the round-based model [13] by establishing simulation *on the level of protocols* and generalizing the DHT semantics. Such a generalization is of practical interest, as it will allow us to port existing invariant proofs in other frameworks (*e.g.*, Verdi [64, 65]) that follow the I/O Automata model.

Moving down: Reasoning about programs. The immediate advantage of employing protocol-aware primitives for implementing provably correct distributed systems is the ability to use them in combination with higher-order functions and other programming mechanism. For instance, we were able to define loops and blocking receive just as syntactic sugar, relying on primitive commands and higher-order combinators. Even further, the shallow encoding of DHT into the Calculus of Constructions made it possible for use to take advantage of Coq's powerful abstraction mechanisms, providing reusable specifications for programs in terms of *abstract predicates* [9] rather then referring to concrete protocols. Finally, realistic distributed applications, such as multi-Paxos [61, 4] are far from being simple first-order code with message sending and receiving: they employ advanced features, such as per-node fork/join concurrency, higher-order iteration and client-side libraries. In order to establish the correctness of such implementations, one would have to relate the protocol-specific logic to those programming mechanisms – precisely what DHT enables.

That said, in the current formulation, the programming component of DHT is a pure functional language with general recursion and message passing. Imperative state and a form of exceptions can be encoded by means of "effect-passing" style, thus allowing some optimizations. For more low-level reasoning about highly optimized implementations in the presence of *native* mutable state, local faults, and per-node concurrency, we are planning to extend the reasoning with *low-level* versions of separation logic, adopting the ideas from the corresponding recent verification efforts [45, 5], as well as the idea of *transitions-as-resources* [29, 58] as a way to account for local concurrency, allowing several protocol branches to be exercised by a node in parallel [61].

References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *LICS*, pages 165–175. IEEE Computer Society, 1988.

2 Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.

3 Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. In *CONCUR*, pages 47–62. Springer, 2014.

4 Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC*, pages 398–407. ACM, 2007.

5 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *SOSP*, pages 18–37. ACM, 2015.

6 Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.5pl3*, 2016. `http://coq.inria.fr`.

7 Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol composition logic (PCL). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

8 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446. ACM, 2011.

9 Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

10 Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. The Need for Language Support for Fault-Tolerant Distributed Systems. In *SNAPL*, volume 32 of *LIPIcs*, pages 90–102. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015.

11 Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415. ACM, 2016.

12 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15. ACM, 2009.

13 Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.

14 Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

15 Colin S. Gordon. *Verifying Concurrent Programs by Controlling Alias Interference.* PhD thesis, University of Washington, 2014.

16 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *SOSP*, pages 1–17. ACM, 2015.

17 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

18 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

20 Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, 2005. URL: `https://www.isa-afp.org/entries/DiskPaxos.shtml`.

21 Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

22 Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *PLDI*, pages 179–188. ACM, 2007.

23 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

24 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

25 Leslie Lamport and Fred B. Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, volume 190 of *LNCS*, pages 203–285. Springer, 1985.

26 Butler W. Lampson. How to build a highly available system using consensus. In *WDAG*, volume 1151 of *LNCS*, pages 1–17. Springer, 1996.

27 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370. ACM, 2016.

28 Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, pages 561–574. ACM, 2013.

29 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.

30 Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, pages 455–468, 2012.

31 Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

32 Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *OOPSLA*, pages 395–410, New York, NY, USA, 2012. ACM.

33 Giuliano Losa, Sebastiano Peluso, and Binoy Ravindran. Brief announcement: A family of leaderless generalized-consensus algorithms. In *PODC*, pages 345–347. ACM, 2016.

34 Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., 1996.

35 Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151. ACM, 1987.

36 Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

37 Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

38 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

39 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I, II. *Inf. Comput.*, 100(1):1–40, 1992.

40 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014.

41 Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare Type Theory. In *ICFP*, pages 62–73. ACM, 2006.

42 Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240. ACM Press, 2008.

43 Aleksandar Nanevski, Viktor Vafeiadis, and Josh Berdine. Structuring the verification of heap-manipulating programs. In *POPL*, pages 261–274. ACM, 2010.

44 Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.

45 Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. Fault-tolerant resource reasoning. In *APLAS*, volume 9458 of *LNCS*, pages 169–188. Springer, 2015.

46 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319. USENIX Association, 2014.

47 Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630. ACM, 2016.

48 Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. In *AVOCS*. EASST, 2015.

49 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

50 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.

51 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, volume 9032, pages 333–358. Springer, 2015.

52 Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *OOPSLA*, pages 92–110. ACM, 2016.

53 Tim Sheard, Aaron Stump, and Stephanie Weirich. Language-based verification will change the world. In *FoSER*, pages 343–348. ACM, 2010.

54 Kasper Svendsen and Lars Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.

55 Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *PPDP*, pages 161–172. ACM, 2011.

56 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390. ACM, 2013.

57 Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.

58 Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *POPL*, pages 343–356. ACM, 2013.

59 Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In *POPL*, pages 247–258. ACM, 2011.

60 Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*, volume 4703, pages 256–271. Springer, 2007.

61 Robbert van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, 2015.

62 Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Trans. Dependable Sec. Comput.*, 12(4):472–484, 2015.

63 Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

64 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368. ACM, 2015.

65 Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for change in a formal verification of the Raft Consensus Protocol. In *CPP*, pages 154–165. ACM, 2016.