3rd Summit on Advances in Programming Languages

SNAPL 2019, May 16–17, 2019, Providence, RI, USA

Edited by

Benjamin S. Lerner
Rastislav Bodík
Shriram Krishnamurthi

Editors

Benjamin S. Lerner
Northeastern University, USA
benjamin.lerner@gmail.com

Rastislav Bodík
University of California Berkeley, USA
bodik@cs.washington.edu

Shriram Krishnamurthi
Brown University, USA
sk@cs.brown.edu

ACM Classification 2012
Software and its engineering → General programming languages; Software and its engineering → Semantics

ISBN 978-3-95977-113-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-113-9.

Publication date
July, 2019

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 https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): https://creativecommons.org/licenses/by/3.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.SNAPL.2019.0

ISBN 978-3-95977-113-9 **ISSN 1868-8969** **https://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.

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Christel Baier (TU Dresden)
- Mikolaj Bojanczyk (University of Warsaw)
- Roberto Di Cosmo (INRIA and University Paris Diderot)
- Javier Esparza (TU München)
- Meena Mahajan (Institute of Mathematical Sciences)
- Dieter van Melkebeek (University of Wisconsin-Madison)
- Anca Muscholl (University Bordeaux)
- Luke Ong (University of Oxford)
- Catuscia Palamidessi (INRIA)
- Thomas Schwentick (TU Dortmund)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi 0:vii

Regular Papers

Overparameterization: A Connection Between Software 1.0 and Software 2.0
Michael Carbin 1:1–1:13

Blame Tracking and Type Error Debugging
Sheng Chen and John Peter Campora III 2:1–2:14

What is a Secure Programming Language?
Cristina Cifuentes and Gavin Bierman 3:1–3:15

From Theory to Systems: A Grounded Approach to Programming Language Education
Will Crichton 4:1–4:9

From Macros to DSLs: The Evolution of Racket
Ryan Culpepper, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi . 5:1–5:19

The Dynamic Practice and Static Theory of Gradual Typing
Michael Greenberg 6:1–6:20

A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity
Lenny Truong and Pat Hanrahan 7:1–7:21

Version Control Is for Your Data Too
Gowtham Kaki, KC Sivaramakrishnan, and Suresh Jagannathan 8:1–8:18

The Next 700 Semantics: A Research Challenge
Shriram Krishnamurthi, Benjamin S. Lerner, and Liam Elberty 9:1–9:14

Toward Domain-Specific Solvers
for Distributed Consistency
Lindsey Kuper and Peter Alvaro 10:1–10:14

A Tour of Gallifrey, a Language for Geodistributed Programming
Mae Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers 11:1–11:19

Formal Verification vs. Quantum Uncertainty
Robert Rand, Kesha Hietala, and Michael Hicks 12:1–12:11

3rd Summit on Advances in Programming Languages (SNAPL 2019).
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 third running of the **S**ummit o**N** **A**dvances in **P**rogramming **L**anguages (SNAPL), a relatively new venue for the programming languages community. The goal of SNAPL is to complement existing conferences by discussing big-picture questions. After the first two events, held in 2015 and 2017, we hope to continue evolving 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. After two runnings in Asilomar, California, we decided to hold the third SNAPL in Providence, RI, USA, hoping that the East Coast location will make the conference more accessible to attendees from Europe.

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 at 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 ideas, progress reports, retrospectives, analyses of mistakes, calls to action, and more. Thus, a SNAPL submission should be viewed more as a "request to give an invited talk."

Overall, the submissions suggest SNAPL remains an interesting and valuable venue. We have received 21 submissions and decided to accept twelve. The program will also include talks by members of the program committee. We look forward to discussions that will hopefully be as lively as in the previous instances of SNAPL.

Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi
May, 2019

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Overparameterization: A Connection Between Software 1.0 and Software 2.0

Michael Carbin
MIT CSAIL, Cambridge, MA, USA
mcarbin@csail.mit.edu

Abstract

A new ecosystem of machine-learning driven applications, titled *Software 2.0*, has arisen that integrates neural networks into a variety of computational tasks. Such applications include image recognition, natural language processing, and other traditional machine learning tasks. However, these techniques have also grown to include other structured domains, such as program analysis and program optimization for which novel, domain-specific insights mate with model design. In this paper, we connect the world of Software 2.0 with that of traditional software – *Software 1.0* – through *overparameterization*: a program may provide more computational capacity and precision than is necessary for the task at hand.

In Software 2.0, overparamterization – when a machine learning model has more parameters than datapoints in the dataset – arises as a contemporary understanding of the ability for modern, gradient-based learning methods to learn models over complex datasets with high-accuracy. Specifically, the more parameters a model has, the better it learns.

In Software 1.0, the results of the approximate computing community show that traditional software is also overparameterized in that software often simply computes results that are more precise than is required by the user. Approximate computing exploits this overparameterization to improve performance by eliminating unnecessary, excess computation. For example, one – of many techniques – is to reduce the precision of arithmetic in the application.

In this paper, we argue that the gap between available precision and that that is required for either Software 1.0 or Software 2.0 is a fundamental aspect of software design that illustrates the balance between software designed for general-purposes and domain-adapted solutions. A general-purpose solution is easier to develop and maintain versus a domain-adapted solution. However, that ease comes at the expense of performance.

We show that the approximate computing community and the machine learning community have developed overlapping techniques to improve performance by reducing overparameterization. We also show that because of these shared techniques, questions, concerns, and answers on how to construct software can translate from one software variant to the other.

2012 ACM Subject Classification Software and its engineering → General programming languages; Computing methodologies → Machine learning

Keywords and phrases Approximate Computing, Machine Learning, Software 2.0

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.1

Funding This work was supported in part by the Office of Naval Research (ONR N00014-17-1-2699) and the National Science Foundation (NSF CCF-1751011).

Acknowledgements We would like to thank Jonathan Frankle, Benjamin Sherman, Jesse Michel, and Sahil Verma for the research contributions summarized in this work.

© Michael Carbin;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 1; pp. 1:1–1:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Software 2.0

Software 2.0 is the vision that future software development will consist largely of developing a data processing pipeline and then streaming the pipeline's output into a neural network to perform a given task [25]. The implication is then that *Software 1.0* – the development of software using traditional data structures, algorithms, and systems – will move lower in the software stack, with less of the overall software engineering effort dedicated to building software in this traditional way.

In domains such as image recognition and natural language processing, this Software 2.0 vision has been largely realized: neural networks have replaced hand-engineered models for vision, natural language processing, and other traditional, machine learning tasks. The key observation in these domains is that building models by hand is labor-intensive, requires significant expertise in domain-specific modeling, and is difficult to adapt across similar tasks. Such engineering can include both hand-developing analytical models as well as engineering features to use as inputs to machine learning models, such as Support Vector Machines.

The promise of Software 2.0 is that neural networks have the capability to automatically learn high-dimensional representations of the system's inputs from raw data alone. For example, the word2vec [33] model automatically learns high-dimensional vector-valued representations of natural language words that capture semantic meaning and can automatically be used in a variety of natural language tasks. This stands in contrast to traditional approaches that require experts to develop algorithms that manually identify important words or subwords within a piece of text. With this opportunity, Software 2.0 – in its most extreme form – holds out the promise of replacing large, hand-developed components of traditional software systems with neural networks.

1.1 Overparameterization

A key component of the success of Software 2.0 is overparameterization. Contemporary explanations for the relative ease of training neural networks on large, complicated datasets with relatively simple optimization methods such as gradient descent posit that overparameterization is a key ingredient. Specifically, overparameterization results in improved learning both in total accuracy and accuracy as a function of data points of training [29, 1, 15, 2].

Overparameterization is a condition in which a machine learning model – such as a neural network – has more parameters than datapoints. In such a regime, the contemporary understanding is that a learning algorithm can identify parameters for the model that can perfectly memorize the data. The connotation of the term *over*parameterization is therefore a note that – in principle, for well-posed datasets – it is possible to design a model that has fewer parameters than the number of datapoints for which a learning algorithm can identify an effective setting for those parameters for the task at hand.

Although overparameterization seems uniquely restricted to the domain of machine learning models and Software 2.0, in this paper we argue that overparameterization is a key ingredient in the development of Software 1.0 and that therefore overparameterization is binding force to relate Software 1.0 and Software 2.0.

1.2 Overparmeterization in Software 1.0

We define overparameterization in Software 1.0 as a condition in which the system performs more computation than is necessary for the task at hand. As a simple example, the traditional software development practice of uniformly choosing single- or double-precision for all real-like

numbers in a system is a simplifying assumption that ignores the fact that different quantities in the program may require different precisions. Choosing precision in this regime requires the precision of all operations to be that of the operation that the program needs to be the most precise. However, for some applications, data and operations may require only limited precision, such as half-precision or even less, while still enabling the application to produce an acceptable result.

The difference between selecting uniform precision and selecting an appropriate precision per operation is an inherent trade-off: uniformly selecting a high-precision is easier than the detailed numerical analysis required to soundly select a precision per operation [12]. However, uniformly selecting precision may not yield optimal performance: more precise floating-point operations are more computationally and memory intensive than less precise operations.

1.3 Reducing Overparameterization (Approximate Computing)

In both Software 1.0 and Software 2.0, researchers have sought techniques to address the fact that overparameterization results in reduced performance, increased energy consumption, and decreased ubiquity (requiring significant resources from the user to execute the software).

For Software 1.0, the approximate computing community has shown that it is possible to eliminate unnecessary computation and, correspondingly, improve performance. Specific techniques have included floating-point precision tuning (i.e., choosing less precise arithmetic) [12, 11, 44, 10, 34], loop perforation (eliding computations entirely) [36, 35, 49], and function substitution (replacing entire sub-computations whole cloth with less expensive, approximate implementations) [23], and more [13, 16, 17, 27, 38, 39, 45, 9].

For Software 2.0, the machine learning community has developed a variety of new techniques to reduce overparameterization in neural networks. *Quantization* chooses new, low-precision representations of the parameters and arithmetic in neural networks [40]. *Pruning* ignores subsets of a neural network's parameters [26, 21, 41]. *Distillation* trains a new, smaller network to mimic the behavior of a large, well-trained network [5, 22]. Each of these techniques have direct analogues to techniques in approximate computing: precision-tuning, loop perforation, and function substitution, respectively.

1.4 Shared Questions

The core premise of this paper is that overparameterization in both Software 1.0 and Software 2.0 and the overlap in techniques for reducing overparameterization in both, enable us to interpolate between Software 1.0 and Software 2.0, mapping observations and questions from one software construction methodology to the other and vice-versa.

For example, the first question we ask is, is overparameterization is necessary? For Software 1.0, it is assumed that developers – perhaps with significant effort – can develop optimized implementations from the outset given a specification of the system's requirements. The analogous question for Software 2.0 is if it is possible to train – from scratch – a neural network with significantly fewer parameters to model a given problem. We recount our results on the Lottery Ticket Hypothesis [18] that demonstrate that from scratch training for standard problems is possible.

Second, a claim for Software 2.0 is that the parameters of its design – such as the specific neural network architecture – can be optimized alongside the system's objective, provided that these parameters are differentiable. The second question we therefore ask is, can we integrate approximation transformations from the start – and throughout the lifetime – of a Software 1.0 system? We survey our recent results on noise-based sensitivity analysis for

programs by showing that if we embrace Software 2.0's aim to minimize its expected behavior – versus its worst-case behavior – then it is possible to integrate approximations into the optimization of the system's overall objective using gradient descent.

Finally, we discuss several questions on the compositionality and correctness of future of Software 1.0 and Software 2.0 systems and propose directions forward.

1.5 Directions

Software 1.0 and Software 2.0 share the phenomenon of overparameterization, share the same approaches for reducing overparameterization, and – once connected – share overlapping challenges to their construction. By drawing these connections, we hope to identify principled techniques to approach software design, correctness, and performance jointly for both Software 1.0 and Software 2.0.

2 Reducing Overparameterization

Researchers in both approximate computing and machine learning have sought techniques to automatically eliminate overparameterization. In the machine learning setting, there are a variety of techniques that reduce parameter counts by more than 90% while still maintaining the accuracy of the neural network on the end task. The goal of reducing parameter counts is multi-fold: 1) reducing the representation size of the network reduces storage and communication costs [21, 22], 2) reducing parameters eliminates computation, and 3) the combination of effects overall improves performance and energy consumption [50, 37, 31].

For example, the following code implements a neural network layer that computes a matrix-vector product of its internal weights (`weights`) with its `m`-dimensional input (`x`). The result is an `n`-dimensional vector passed through a Rectified Linear Unit activation function (`max(0, output)`). The `n`-dimensional result denotes the output of `n` neurons.

```
1  float x[] = { ... };
2  float weights[][] = { ... };
3  float output[] = { ... };
4  for (int i = 0; i < n; ++i)
5  {
6      for (int j = 0; j < m; ++j)
7      {
8          output[i] += weights[i][j] * x[j];
9      }
10 }
11 return max(0, output);
```

Quantization reduces the number of bits used to represent each weight and compute each operation within the neural network's computation. We can capture this optimization as classic precision selection where, for example, this program could be written to use 16-bit precision floating-point instead of 32-bit `float` types.

Pruning takes a trained large model and eliminates weights by, for example, removing the weights of the smallest magnitude. For this example program, this is equivalent to eliding a subset of the loop iterations – i.e., loop perforation – based upon a pre-determined mask. Eliding iterations of the loop over `j`, elides individual weights in each of the `n` neurons while eliding iterations of the loop over `i` elides neurons in their entirety. Both options have been explored in the literature [21].

Distillation takes a large, trained model and trains a smaller network to mimic the outputs of this model. In this example, this entire layer could be substituted with an alternative implementation. For example, this computation is no more than a matrix-vector product and therefore it is to possible accelerate this computation by instead learning a fast, low-rank approximation of `weights` and computing with that instead [14].

3 Fewer Parameters from the Start

The standard conceptualization of the approximate computing workflow applies approximation transformations after an initial, end-to-end development of a system. Specifically, the standard workflow requires that a developer write an additional *quality-of-service* specification [36] after developing their program. This specification states how much error – measured with respect to the ground truth – that the user can tolerate in their program's output. Given this specification, an approximate computing system then searches the space of approximation transformations to produce a program that meets the quality-of-service specification.

Reducing overparameterization in Software 2.0 follows a similar methodology. Standard methodologies apply pruning, quantization, and distillation to a fully trained neural network. However, the reality that these techniques can be applied lends itself to the question: why not simply use a smaller, more efficient neural network from the start?

Until our recent results [18], contemporary understanding in the machine learning community was that small neural networks are harder to train, reaching lower accuracy than the original networks when trained from the start. Alternatively, overparameterization is required for effective learning.[1] Figure 1 illustrates this phenomenon. In this experiment, we randomly sample and train small networks. These networks are sampled from the set of subnetworks of several different neural network architectures for the MNIST digit recognition benchmark and the CIFAR10 image recognition benchmark. Across various sizes relative to the original reference networks, the sparser the network (fewer parameters), the slower it learned and the lower its eventual test accuracy.

However, this figure also shows the results of our techniques for identifying *winning tickets* [18]: small networks that do in fact train as well (or better) than the original network. The solid lines in the graph are winning tickets, for which these graphs show that even down to a small size, these networks achieve high-accuracy as well as train quickly.

We identify winning tickets by pruning: randomly initialize a neural network $f(x; W_0)$, train it to completion, and prune the weights with the lowest magnitudes (and repeat over multiple iterations). To initialize the winning ticket, we reset each weight that survives the pruning process back to its value in W_0.

The success of this technique has lead us to pose and test the *lottery ticket hypothesis*: any randomly-initialized, neural network contains a subnetwork that is initialized such that – when trained in isolation – it can learn to match the accuracy of the original network after at most the same number of training iterations. In our work, we have supported this hypothesis through experimental evidence on a variety of different neural network architectures. However, our approach to identify winning ticket still requires training the full network: the remaining science is to determine a technique to identify winning tickets earlier in the training process.

[1] "Training a pruned model from scratch performs worse than retraining a pruned model, which may indicate the difficulty of training a network with a small capacity." [28] "During retraining, it is better to retain the weights from the initial training phase for the connections that survived pruning than it is to re-initialize the pruned layers...gradient descent is able to find a good solution when the network is initially trained, but not after re-initializing some layers and retraining them." [21]

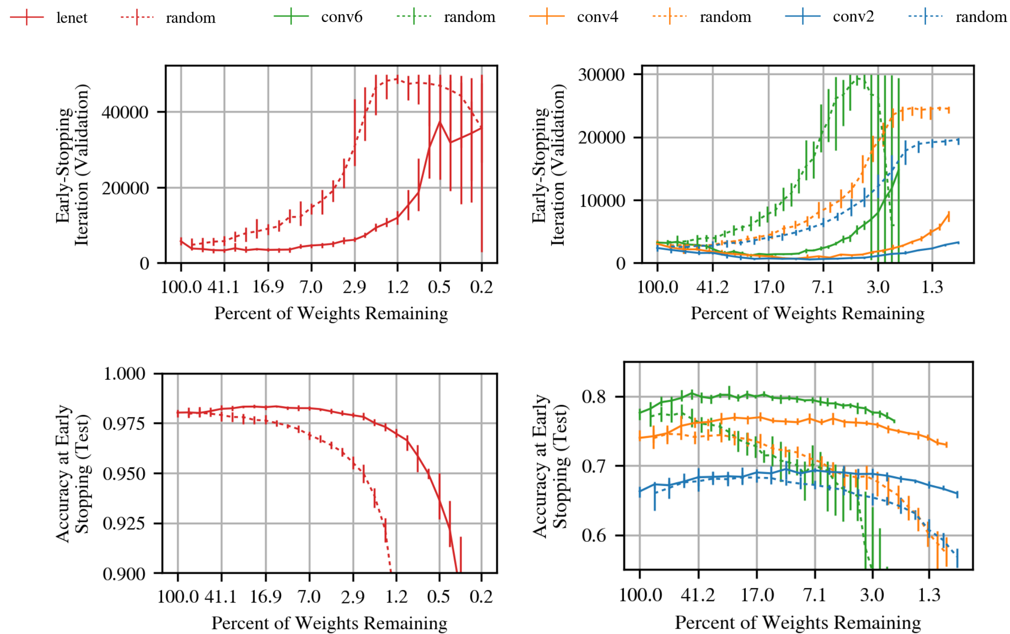


Figure 1 The iteration at which early-stopping would occur (left) and the test accuracy at that iteration (right) of the lenet architecture for MNIST and the conv2, conv4, and conv6 architectures for CIFAR10 ([18] Figure 2) when trained starting at various sizes. Dashed lines are randomly sampled sparse networks (average of ten trials). Solid lines are winning tickets (average of five trials).

4 Approximation from the Start

Recent work has investigated techniques to introduce pruning, quantization, early in the training process with limited success [40, 30, 19]. The key idea behind these approaches is that the parameters of these techniques can be designed to be differentiable. Notably, in the standard Software 2.0 methodology, a learning algorithm identifies settings of the system's parameters that minimize the system's expected error with respect to a ground truth given by a training dataset. Within this umbrella, an error specification is therefore builtin to this methodology and – typically – the learning algorithm selects the system's parameters through gradient descent. Therefore, if the parameters of these reduction techniques are differentiable, then they can be learned alongside the system's standard parameters.

Approach

Inspired by Software 2.0, in recent results we have developed a sensitivity and precision selection technique for traditional numerical programs that is differentiable. Our approach models sensitivity as random noise – e.g., sampled from a gaussian distribution – added to each operation in the program such that the standard deviation of the noise's distribution indicates the sensitivity of the program's expected error to changes in the operation's output.

If an operation's noise distribution can have large standard deviation without perturbing the expected error of the program, then the program is relatively less sensitive to perturbations in that operation's results. On the other hand, if an operation's noise distribution must have small standard deviation to avoid perturbing the expected error of the program, then the program is relatively more sensitive to changes in that operation's result.

Table 1 We compare the absolute error bound from FPTuner against the empirically determined root mean squared error from our approach. Mean bits is the average number of bits in the mantissa of the approximate program in contrast to FPTuner's 52-bit mantissa (from doubles).

Benchmarks	FPTuner	RMSE	Mean Bits
verlhulst	3.79e-16	1.13e-16	51
sineOrder3	1.17e-15	7.64e-16	50
predPrey	1.99e-16	1.90e-16	50
sine	8.73e-16	8.34e-17	51
doppler1	1.82e-13	3.30e-14	53
doppler2	3.20e-13	3.30e-14	53
doppler3	1.02e-13	8.22e-14	53
rigidbody1	3.86e-13	1.37e-13	51
sqroot	7.45e-16	4.00e-16	50
rigidbody2	5.23e-11	6.08e-12	51
turbine2	4.13e-14	2.57e-14	50
carbon gas	1.51e-08	3.01e-09	49
turbine1	3.16e-14	8.69e-15	51
turbine3	1.73e-14	5.09e-15	52
jet	2.68e-11	2.45e-11	54

The goal of our approach is to identify the maximum standard deviations for the distribution of each operator such that the resulting program still delivers acceptable expected error. Our technique poses this goal as an optimization problem and solves the problem through stochastic gradient descent. We have shown that these sensitivities are informative by developing a precision selection approach that takes as input the set of sensitivities for the operations in the program and produces an assignment of precision to each operator.

Case Study

We have applied our approach to a set of scientific computing benchmarks used by FPTuner's developers to develop and evaluate FPTuner [10]. FPTuner is a tool that can identify an assignment of precisions to operators such that the resulting program satisfies a user-provided worst-case error. FPTuner uses a combination of static error analysis and quadratic programming to automatically identify an assignment that satisfies the provided bound. To evaluate our sensitivity analysis, we have devised a technique to map the sensitivity of each operator to the number of bits to use in an arbitrary precision library (i.e., MPFR).

Table 1 presents our preliminary results of our approach. We configured our approach to produce sensitivities such that expected error of the program is less than FPTuner's absolute error. The results are that our approach generates tighter expected error bounds using fewer bits than FPTuner for 10 out of 15 benchmarks. For the remaining 5 benchmarks, our approach requires at most 2 extra bits on average than in FPTuner.

Sensitivity analysis provides critical information to an approximate computing system – the sensitivity of the program's output to changes in semantics of an operation. By embracing expected error and working within a differentiable setting, we have arrived at an approach that mates Sofware 1.0 (approximate computing) with Software 2.0.

5 Discussion

Software 2.0 – replacing core components of traditional-developed software systems with learned components – is a lofty goal. A particular challenge to this goal is that an integral component of Software 2.0 – neural networks – have so far proved to be difficult to interpret and reason about for the purposes of validating the resulting system's behavior. However, the approximate computing community has faced similar difficulties with 1) understanding the composition of approximation with the original system, 2) giving the resulting approximate system a useful behavioral specification, and 3) developing analysis frameworks for reasoning about that behavioral specification. Through the shared tie of overparameterization, Software 2.0 and approximate computing can share techniques to solve their common challenges.

Compositionality

In a Software 2.0 system that composes both neural networks and traditional computation, reasoning about the behavior of the resulting system is challenging. Specifically, the high-dimensional representations that neural networks learn are not necessarily directly interpretable by humans. For example, word2vec represents words as n-dimensional vectors, where n is often large (i.e., greater than 256), with limited semantic meaning assigned to each dimension. Moreover, interpreting the composed behavior of the neural network with the larger system is challenging because the network's task may not easily permit a compositional specification of its behavior for which global faults can be reduced to local reasoning.

The approximate computing community has faced similar challenges. In its most idealized form, the community's agenda has advocated for a variety of techniques that are together composed with the program in a blackbox manner with limited interaction with the developer or user. Techniques such as loop perforation and function substitution may remove or replace large fractions of a system's computation with the result being limited understanding of the system's semantics. As a consequence, a developer may receive an approximate system for which failures are hard to address because it is not clear if they are resident in the original, non-approximate program program or a created anew through approximation [6].

One avenue for Software 2.0 to follow is the direction of our work in approximate computing to apply the concept of *non-interference* to support compositional reasoning [6]. Our proposed programming methodology argues that a developer should develop a program and establish its *acceptability properties* – the basic invariants that must be true of the program to ensure that its execution and ultimate results are acceptable for the task at hand [42]. Example invariants include standard safety properties – such as memory safety – but also include application-specific *integrity properties* [6, 7]. For example, a computation that computes a distance metric between two values should return a nonnegative result, regardless of the extent of its approximation.

Given a program and its acceptability properties, the developer communicates to the approximate computing system points in the application at which approximation opportunities are available and do not interfere with the properties established for the original program. Therefore by non-interference, we mean that if the original program satisfies these properties, then the approximate program satisfies these properties. Reasoning about non-interference can include reasoning about information flow to ensure that approximations do not change the values of data and computations that are involved in the invariants. For Software 2.0, this methodology could enable existing software components to be replaced with learned variants that do not interfere with the program's acceptability properties. This framework can, for example, underpin recent work on adding assertions to machine learning models [24].

Correctness

The act of developing a specification of the full functional correctness of a software system as can be done for traditional software does not directly translate to either Software 2.0 or approximate systems. For many Software 2.0 systems – such as those in computer vision and natural language processing – the correctness specifications for these problems is either not known or not well posed. For example, specifying that the system correctly classifies an image of a cat as a cat does not have a declarative, logical specification. Therefore developers typically evaluate a Software 2.0 system based on its expected error over a sample of data from its input distribution. The correctness specification for approximate systems is challenging in that – by definition – an approximate system returns different results from its original implementation. By this nature, an approximate system's natural measure of its behavior is its error with respect to the original program.

In both domains, there has developed a shared understanding that statistically bounding the error of the system is a potential direction for improving the confidence in a system. The core conceptual challenge is that while the error of either type of system can be measured on a test set drawn as a sample of the system's input distribution to give confidence, the key property to bound is the system's *generalization*: its error on unseen data.

In the statistics and machine learning community, such bounds are known as *generalization bounds*. A simplified structure for these bounds is that for $\delta \in (0, 1)$, with probability $1 - \delta$ over a random sample s from an input distribution $\mathcal{D}$, $\mathcal{L} \leq B(\hat{\mathcal{L}}(s), \delta)$. Here $\mathcal{L}$ is the expected error (or loss) of the system on unseen data, $\hat{\mathcal{L}}(s)$ is the observed error of the system on s, and B denotes a function that computes the bound. The computed bound is at least $\hat{\mathcal{L}}(s)$ – i.e., the error on unseen data is no better the error on observed data – and decreasing in δ – the less confidence one requires of the bound, the closer it is to the observed error.

The approximate computing community has also developed statistical bounds on the behavior of approximate systems. These analyses include bounding the probability the system produces the correct result [8, 34], bounding the probability it produces a result exceeding a specified distance from that of the original program [43, 35], and bounding its expected error [51]. The shared focus and results on statistical bounds between Software 2.0 and approximate computing suggests that these bounds may be integral specifications for ensuring the behavior of future systems, including both Software 2.0 and Software 1.0.

Analysis

If Software 2.0 takes hold, then the reasoning methods we have used to build software will need to change. Traditional software construction methodologies have been designed around the classic building blocks of Computer Science: discrete and deterministic math, algorithms, and systems. However, the basic analysis that underpins reasoning about Software 2.0 is based on continuous math and statistics: neural networks are formalized as functions on real numbers and formalizing generalization relies on statistical analysis. The required analysis for approximate systems also relies on continuous math and statistics: these systems compute on reals in their idealized mathematical specification and statistical bounds are the preferred framework for reasoning about their correctness.

A resulting challenge is that the formal methods, programming languages, and systems communities – communities that are major contributors to the mission to formalize and deliver automated reasoning systems – has invested less in understanding real-valued and/or probabilistic computations than for discrete computations. The result is that there is a significant gap between the needs of future systems and the capabilities of existing analysis. Therefore, the next generation of systems will need new computational building blocks.

For example, our work on programming with continuous values makes it possible to soundly compute on real numbers to arbitrary precision as well as soundly combine real-valued computation with discrete computation [47, 46]. Specifically, discrete computations on the reals – e.g., testing if two real numbers are equal – is undecidable in general. This fact stands as a contradiction to modern programming languages that expose floating-point as an approximation of the reals and permit developers to test them for equality.

As another example, many of the reasoning tasks for future software – such as profiling or computing bounds on their behavior – will be probabilistic computations. One can pose these analyses as queries on the behavior of probabilistic models, as currently captured by the community around *probabilistic programming*: representing probabilistic models as programs with stochastic choices [20]. In these systems, the supporting programming system can perform *inference* to compute answers to questions such as, what's the probability that the program produces a value that exceeds a given bound?

However, beneath the covers of these programming systems lies a space of inference algorithms, which are algorithms that manipulate probability distributions. If probabilistic computations are to be integral to future software, then future and developers and systems will need to understand and manipulate inference algorithms and therefore need to understand and manipulate probability distributions as first-class values. In this space, there are open questions about the architecture of programming systems for implementing inference algorithms that we – along with others – are exploring [4, 3, 32, 48].

In sum, the future of Software 2.0 – and the extent to which we can reason about its behavior – critically depends on the development of new programming models and abstractions for continuous math and statistics.

6 Conclusion

Software 1.0 and Software 2.0 appear radically different. The development methodology for Software 1.0 revolves around developers manually architecting the overall structure and constituent algorithms of a system. In contrast, the mantra of Software 2.0 is to delegate much of the system's algorithms and – perhaps – even its structure to neural networks or other machine learning methods. However, overparameterization is a shared connection between both methodologies.

In the case of Software 1.0, developers rely on coarse, general-purpose abstractions that are easy to program with but that perform more computation than is necessary for the task at hand. In the case of Software 2.0, results have shown that larger neural networks learn more easily than their smaller counterparts, but – in principle – smaller networks are capable of representing the task. The trade-off for both of these methodologies is that the increased ease in development comes at the expense of performance.

To address this problem, both the approximate computing community and the machine learning community have coalesced on techniques to reduce overparameterization in Software 1.0 and Software 2.0, respectively, while still preserving ease of development. Based on this shared goal, this paper offers the viewpoint that questions, challenges, and techniques from both communities can translate from one to the other. As the Software 2.0 future unfolds, new questions about the composition and correctness of these systems will arise. However, these questions can be addressed jointly within both Software 1.0 and Software 2.0.

References

1. Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A Convergence Theory for Deep Learning via Over-Parameterization. In *International Conference on Machine Learning*, pages 242–252. PMLR, 2019.
2. Sanjeev Arora, Simon S. Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-Grained Analysis of Optimization and Generalization for Overparameterized Two-Layer Neural Networks. In *International Conference on Machine Learning*, pages 322–332. PMLR, 2019.
3. E. Atkinson, C. Yang, and M. Carbin. Verifying Handcoded Probabilistic Inference Procedures. *CoRR*, 2018. `arXiv:1805.01863`.
4. Eric Atkinson and Michael Carbin. Towards Correct-by-Construction Probabilistic Inference. In *NIPS Workshop on Machine Learning Systems*, 2016.
5. Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, 2014.
6. M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *Conference on Programming Language Design and Implementation*, pages 169–180. ACM, 2012. `doi:10.1145/2254064.2254086`.
7. M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Verified integrity properties for safe approximate program transformations. In *Workshop on Partial Evaluation and Program Manipulation*, pages 63–66. ACM, 2013. `doi:10.1145/2426890.2426901`.
8. M. Carbin, S. Misailovic, and M. Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *International Conference on Object Oriented Programming Systems Languages & Applications*, pages 33–52, 2013. `doi:10.1145/2509136.2509546`.
9. S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. In *Symposium on the Foundations of Software Engineering and European Software Engineering Conference*, pages 102–112, 2011. `doi:10.1145/2025113.2025131`.
10. Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. Rigorous Floating-point Mixed-precision Tuning. In *Symposium on Principles of Programming Languages*, pages 300–315. ACM, 2017.
11. Eva Darulova, Einar Horn, and Saksham Sharma. Sound Mixed-precision Optimization with Rewriting. In *International Conference on Cyber-Physical Systems*, pages 208–219. IEEE Press, 2018. `doi:10.1109/ICCPS.2018.00028`.
12. Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Symposium on Principles of Programming Languages*. ACM, 2014. `doi:10.1145/2535838.2535874`.
13. M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *International Symposium on Computer Architecture*, pages 497–508. ACM, 2010. `doi:10.1145/1815961.1816026`.
14. Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation. In *Advances Neural Information Processing Systems*, 2014.
15. Simon S. Du, Jason D. Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient Descent Finds Global Minima of Deep Neural Networks. In *International Conference on Machine Learning*, pages 1675–1685. PMLR, 2019.
16. D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *International Symposium on Microarchitecture*, pages 7–18, 2003. `doi:10.1109/MICRO.2003.1253179`.
17. H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312. ACM, 2012. `doi:10.1145/2150976.2151008`.
18. Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *International Conference on Learning Representations*, 2019.
19. Trevor Gale, Erich Elsen, and Sara Hooker. The State of Sparsity in Deep Neural Networks. *CoRR*, 2019. `arXiv:1902.09574`.

20 Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Conference on Uncertainty in Artificial Intelligence*, pages 220–229. AUAI Press, 2008.

21 Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, 2015.

22 Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. In *NIPS Workshop on Deep Learning*, 2014. arXiv:1503.02531.

23 H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic Knobs for Responsive Power-Aware Computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–212. ACM, 2011. doi:10.1145/1950365.1950390.

24 Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. Model Assertions for Debugging Machine Learning. In *NeurIPS MLSys Workshop*, 2018.

25 Andrej Karpathy. Software 2.0, November 2017.

26 Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, 1990.

27 L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. ERSA: error resilient system architecture for probabilistic applications. In *Design, Automation and Test in Europe*, pages 1560–1565, 2010. doi:10.1109/DATE.2010.5457059.

28 Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations*, 2017.

29 Yuanzhi Li and Yingyu Liang. Learning Overparameterized Neural Networks via Stochastic Gradient Descent on Structured Data. In *International Conference on Neural Information Processing Systems*, pages 8168–8177. Curran Associates Inc., 2018.

30 Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through L_0 regularization. In *International Conference on Learning Representations*, 2018.

31 Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *International Conference on Computer Vision*, pages 5068–5076, 2017. doi:10.1109/ICCV.2017.541.

32 Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic Programming with Programmable Inference. In *Conference on Programming Language Design and Implementation*, pages 603–616, 2018. doi:10.1145/3192366.3192409.

33 Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, 2013.

34 S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *International Conference on Object Oriented Programming Systems Languages & Applications*, pages 309–328. ACM, 2014. doi:10.1145/2660193.2660231.

35 S. Misailovic, D. Roy, and M. Rinard. Probabilistically Accurate Program Transformations. In *International Symposium on Static Analysis*, pages 316–333. Springer, 2011. doi:10.1007/978-3-642-23702-7_24.

36 S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *International Conference on Software Engineering*, pages 25–34, 2010. doi:10.1145/1806799.1806808.

37 Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *International Conference on Learning Representations*, 2017.

38 S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. In *Design, Automation and Test in Europe*, pages 335–338. IEEE Computer Society, 2010. doi:10.1109/DATE.2010.5457181.

39 K. Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers*, 2005.

40 Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. In *International Conference on Learning Representations*, 2018.

41 Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. Optimizing CNNs on Multicores for Scalability, Performance and Goodput. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 267–280. ACM, 2017. `doi:10.1145/3037697.3037745`.

42 M. Rinard. Acceptability-oriented computing. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 221–239. ACM, 2003. `doi:10.1145/949344.949402`.

43 M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *International Conference on Supercomputing*, pages 324–334. ACM, 2006. `doi:10.1145/1183401.1183447`.

44 C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 27:1–27:12. ACM, 2013. `doi:10.1145/2503210.2503296`.

45 A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Conference on Programming Language Design and Implementation*, pages 164–174. ACM, 2011. `doi:10.1145/1993498.1993518`.

46 Benjamin Sherman, Jesse Michel, and Michael Carbin. Sound and robust solid modeling via exact real arithmetic and continuity. In *International Conference on Functional Programming*. ACM, 2019.

47 Benjamin Sherman, Luke Sciarappa, Adam Chlipala, and Michael Carbin. Computable decision making on the reals and other spaces: via partiality and nondeterminism. In *Symposium on Logic in Computer Science*, pages 859–868. ACM, 2018. `doi:10.1145/3209108.3209193`.

48 Benjamin Sherman, Jared Tramontano, and Michael Carbin. Constructive probabilistic semantics with non-spatial locales. In *Workshop on Probabilistic Programming Languages, Semantics, and Systems*, 2018.

49 S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs With Loop Perforation. In *Symposium on the Foundations of Software Engineering*, pages 124–134. ACM, 2011. `doi:10.1145/2025113.2025133`.

50 Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Conference on Computer Vision and Pattern Recognition*, pages 6071–6079, 2017. `doi:10.1109/CVPR.2017.643`.

51 Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized Accuracy-Aware Program Transformations for Efficient Approximate Computations. In *Symposium on Principles of Programming Languages*, pages 441–454. ACM, 2012. `doi:10.1145/2103656.2103710`.

Blame Tracking and Type Error Debugging

Sheng Chen
University of Louisiana at Lafayette, USA
http://www.ucs.louisiana.edu/~sxc2311
chen@louisiana.edu

John Peter Campora III
University of Louisiana at Lafayette, USA
campora@louisiana.edu

Abstract

In this work, we present an unexpected connection between gradual typing and type error debugging. Namely, we illustrate that gradual typing provides a natural way to defer type errors in statically ill-typed programs, providing more feedback than traditional approaches to deferring type errors. When evaluating expressions that lead to runtime type errors, the usefulness of the feedback depends on *blame tracking*, the defacto approach to locating the cause of such runtime type errors. Unfortunately, blame tracking suffers from the *bias* problem for type error localization in languages with type inference. We illustrate and formalize the bias problem for blame tracking, present ideas for adapting existing type error debugging techniques to combat this bias, and outline further challenges.

2012 ACM Subject Classification Theory of computation → Type structures

Keywords and phrases Blame tracking, type error debugging, gradual typing, type inference

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.2

Funding This work is partially supported by the NSF grant CCF-1750886.

© Shen Chen and John P. Campora III;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 2; pp. 2:1–2:14

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Static and dynamic typing have different strengths [38, 53, 54]. For example, static typing can detect more errors at compile time but delivers no runtime feedback for programs that have static errors, while dynamic typing can run any program and provide feedback but defers error detection to runtime. Gradual typing [45, 52, 47] is a new language design approach that can integrate both typing disciplines in a single language, allowing different interacting program regions to use static or dynamic typing as needed. Adopting gradual typing has been popular with both statically typed languages (for example C#[3]) and dynamically typed languages (for example JavaScript and Flow [8]). Additionally, gradual typing has been adapted to work with many advanced language features [44, 31, 46, 37, 16, 30].

In particular, much research has explored the interaction of gradual typing with type inference [46, 37, 16, 4, 5]. There are several reasons that this interaction has been explored. First, inference can benefit the usability of gradual typing without programmers having to manually modify numerous type annotations, as argued by [36]. Second, inferred types can be synchronized to help improve gradual typing performance [36, 37, 5, 57]. Third, inference can aid in the detection of inconsistencies in programs [16, 4, 30].

This paper explores an intriguing connection between gradual typing and type inference, the *blame tracking* in gradual typing and *type error debugging* in type inference. In gradual typing, well-typed programs may still encounter runtime type errors since type-checkers allow statically typed contexts to accept values produced by dynamically typed expressions. At runtime these dynamic values may not have the desired runtime types that the context expected, and consequently a runtime type error may be triggered. *Blame safety*, adapted from [15] by Wadler and Findler [61] and further developed in [1, 60, 43, 2], is a well accepted approach in the gradual typing community for indicating which code region is responsible for a runtime type error. Blame safety states that when a cast fails the blame is always assigned to the more dynamic part, under the slogan that "well-typed programs can't be blamed".

Type error localization provides a similar purpose in type error debugging. It specifies which code region is responsible for a static type error. Inaccurate type error localization produces poor error messages [62, 24], and type inference often leads to poor localization. Unlike in gradual typing where blame tracking is the de facto mechanism for enforcing blame safety, numerous approaches have been developed to improving error localization and error reporting in the presence of type inference errors in the last three decades [7, 21, 19, 51, 24, 28, 17, 14, 13, 10, 68, 33, 27, 9, 12, 32, 26, 55, 64, 65, 42, 11].

In this paper, we explore the relations between gradual typing and type error debugging. Our exploration includes the following directions. First, we investigate whether gradual typing and blame tracking improve type error debugging for type inference. Intuitively, we can turn a program with type errors into a well-typed gradual program by annotating certain expressions with dynamic types. We can then obtain and observe runtime behaviors of the program using gradual typing, which may give the users a better understanding of the type error. Through a few examples, we conjecture that gradual typing improves type error debugging, but blame tracking does not. We present the details in Section 2.

Second, by drawing an analogy from type error localization, we reflect the usefulness of blame tracking for providing debugging information when gradual programs encounter dynamic type errors. A fundamental difficulty in type error localization is that type inference is biased. Specifically, it tends to attribute the type error to a later part of the program's syntax tree while in fact the error may have been caused by an earlier part. Unfortunately, we observe that blame tracking is also biased. It tends to blame a later untyped region along

the execution path leading to a dynamic type error, while the error may have been caused by an earlier region. We prove this problem in Section 3. Thus, we suggest that while blame tracking is well-accepted in the gradual typing community, its helpfulness in debugging is questionable, particularly in languages supporting type inference.

Third, as blame tracking suffers from the same problem as type inference does, we are interested in knowing the potential of adapting existing type error localization and debugging approaches to improve blame tracking. We discuss the potential of several approaches and highlight the challenges in adapting them in Section 4. We conclude in Section 5.

2 Gradual Typing and Blame Tracking for Type Error Debugging

This section investigates how gradual typing and blame tracking can help with type error debugging in Sections 2.1 and 2.2, respectively.

2.1 Gradual Typing for Type Error Debugging

We use the example factorial program from Figure 1 to illustrate how gradual typing provides new insights for type error debugging. The type error in Figure 1a is caused by the value `true`, which should instead be `1`. Types for the program are inferred by solving constraints that are generated by the structure of the program. For example, in each function call a constraint is generated ensuring that the argument type of the function must match the type of the argument, and similarly different branches in a single function are constrained into returning the same type. These constraints are collected and solved while traversing program structure. If the constraints can't be solved, then a type error is raised.

To make type inference feasible, constraint solving must be *most general* [59]. Most-general constraint solving pushes back failure detection as late as possible, and thus the reported location is likely not the real error cause. For example, Helium [20], a Haskell type error debugger, attributes the type error to `*`, later than the real error cause `true`, as can be seen from Figure 1b. The standard Haskell compiler GHC also suffers from the right-biased problem, although it reports the whole `else` branch as the error cause.

Most type error debugging approaches prevent the running of ill-typed programs. However, the ability to run ill-typed programs is believed to help program understanding [41, 58], which may in turn help programmers fix their type errors. A main approach that enables the execution of ill-typed programs is deferring type errors to runtime [58]. In a program that contains both well-typed and ill-typed functions, deferring type errors allows programmers to run functions that are not involved in the type error. It, however, provides little help to fix the type error itself. To illustrate, consider running the program in Figure 1a with GHC and deferred type errors enabled. After loading the function, we can invoke `fac` in GHCi. While it looks reasonable to expect the result `true` when we run `fac 0`, since the branch that will be executed does not contain a type error, GHCi actually dumps its deferred compile-time type error message, as shown in Figure 1d. This message is biased since it does not mention the real error cause `true`. The work by Seidel et al. [41] can also run ill-typed programs, but it supports much fewer language features.

An alternative to the previously mentioned approaches is to use gradual typing, facilitated by ascribing certain subexpressions with the dynamic type (denoted by ?). For `fac` in Figure 1a, we make it well-typed by ascribing ? to `*`, as shown in Figure 1c. This new expression, which we name `facG`, is well-typed because the dynamically typed operator (`*`:?) can accept values of any type as arguments. We run `facG` with the gradual evaluator developed by Miyazaki et al. [30], and the result is shown in Figure 1e. The output first shows

```
fac n = if n == 0 then true
                  else n * fac (n-1)
```

(a) An ill-typed function, adopted from [41].

```
(3,43): Type error in infix application
 expression        : n * fac (n - 1)
 operator          : *
   type            : a    -> a    -> a
   does not match  : Int -> Bool -> Bool
```

(b) Output from Helium [20] for `fac`.

```
facG n = if n == 0 then true
               else n (*:?) facG (n-1)
```

(c) A well-typed gradual program by ascribing `*` in Figure 1a to have a dynamic type ?.

```
*Main> fac 0
*** Exception: Fac.hs:3:41: error:
• Couldn't match expected type 'Bool'
                   with actual type 'Int'
• In the expression: n * fac (n - 1)
  In the expression:
    if n == 0 then True else n * fac (n - 1)
    ...
(deferred type error)
```

(d) Output from running `fac` with GHC 8.0.2 and deferring type errors enabled.

```
# facG 0;;
- : bool = true
```

(e) Result of running `facG` with the idea from [30].

Figure 1 An ill-typed function `fac` and outputs from various tools and approaches. To reconcile language differences, we use `true` for `True` in Haskell.

the type and then the result after the equals sign. Interestingly, gradual typing produces `true` for the expression `facG 0`. With this output, a keen programmer should already be able to fix the type error in `fac` because the factorial function should never return a boolean value. She may thus change `true` to `1` to remove the type error.

Because gradual typing generally provides more feedback when running "statically" ill-typed (but gradually well-typed) programs than deferred type errors, we believe that gradual typing does offer additional insights beyond existing type error debugging approaches. However, a main hindrance of adopting gradual typing for debugging type errors is that current implementations supporting gradual typing with type inference and parametric polymorphism are limited. The work in [30] (building on much previous work on some mix of polymorphism, inference, and blame [16, 1, 2, 22, 23]) provides an implementation, but it is restricted to a small set of language features.

A particular twist in using gradual typing to improve type error debugging is that the user needs to provide appropriate arguments to ill-typed functions. Specifically, gradual typing will show execution results only if the execution does not encounter a dynamic type error. For example, only when the argument to `facG` is `0` will the resulting `true` be shown. Otherwise, blaming information will be shown. In such cases, the usefulness of gradual typing for type error debugging depending on that of blame tracking, which we investigate in Section 2.2.

2.2 Blame Tracking for Type Error Debugging

Blame tracking specifies which subexpression should be blamed if the execution of an expression encounters a runtime error. The standard goal of blame tracking in gradual typing is to preserve blame safety [61, 60], which attributes the blame for runtime type errors to subexpressions with more dynamic types. We use the following table to illustrate the behaviors of blame safety. In the "Expressions" column, we use a `;` to denote a sequential expression. The "Blames" column are the expressions blamed after running the corresponding expression with the evaluator from [30]. Our goal in the expressions (1) through (4) is to use gradual typing to debug the type error in `fac` (Figure 1a). The goal of expressions (5) through (11) is to debug the type error in `(\x y -> if true then x else y) 2 false`.

Ids	Expressions	Blames
(1)	`fac1 n = if n == 0 then true else n (*:?) fac1 (n-1); fac1 1`	`*`
(2)	`fac2 n = if n == 0 then (true:?) else n * fac2 (n-1); fac2 1`	`true`
(3)	`fac3 n = if n == 0 then (true:?) else n (*:?) fac3 (n-1); fac3 1`	`*`
(4)	`fac4 n = if n > 0 then n (*:?) fac4 (n-1) else (true:?); fac4 1`	`*`
(5)	`(\x y -> if true then x else y) 2 (false:?)`	`false`
(6)	`(\x y -> if true then x else y) (2:?) false`	`2`
(7)	`(\x y -> if true then x else y) (2:?) (false:?)`	`false`
(8)	`(\x y -> if true then x else y) (false:?) (2:?)`	`2`
(9)	`(\x y -> if true then x else y) ((succ:?) 2) (false:?)`	`false`
(10)	`(\x y -> if true then x else y) ((succ:?) (2:?)) (false:?)`	`false`
(11)	`(\x y -> if true then x else y) ((\x -> x) (succ:?) 2) (false:?)`	`false`

From the table, we make two observations. First, when an expression contains only one subexpression with a dynamic type ?, then that subexpression will be blamed for causing the runtime type error. This is intuitive, because that subexpression is more dynamic than all other subexpressions and will alone be responsible for type mismatches at runtime. The expressions (1), (2), (5), and (6) in the table all belong to this case. Note that the anonymous function is statically-typed (since the parameters do not have the type ?), and the conditional branches are required to have the same type. As a result, the expressions (5) through (11) will raise blame, rather than returning the first argument. We observe that blame tracking does blame the subexpression that caused the type error if that subexpression happens to have the dynamic type, as in the expression (2) above. However, if the user knows where to add ? she probably already knows how to fix the type error.

Second, when an expression contains multiple subexpressions that have the dynamic type, then blame safety is *biased* in attributing the dynamic type error. Specifically, because blame safety is connected to the expression that triggers the runtime type error, it always blames the most recently encountered dynamic context in program execution, even if the true cause of the error was due to an expression evaluated much earlier. For example, in both expressions (3) and (4), the subexpression `true`:? is returned earlier than it is being used as a multiplicand to `*`:?. In other words, `true` is being executed before `*`:?. As a result, `*`:? is blamed in both subexpressions, regardless of their ordering in the conditional branches. Alternatively, we can view `true`:? as injecting `true` to a dynamic value, and when it is used in `*`:? a projection happens. This fits in our description of blame, since blame tracking always blames projections that follow injections.

In expressions (7) through (11), the anonymous function is first applied to the first argument, substituting the argument into `x` in the body and also instantiating the type of both parameters `x` and `y`. When it is applied to the second argument, its type is ensured to be the same as the instantiation. Therefore, we observe that the first argument is executed first and the second argument later. Unsurprisingly, blame safety always assigns the blame to the second argument. The expression (8) in the table demonstrates that that the ordering of types is responsible for determining the blamed expression. The expressions (9) through (11) confirm this observation, even as expressions become more complicated.

We observe that when an expression has multiple subexpressions with ?, blame tracking may provide little help to type error debugging. There is little context in the program that indicates that the blamed subexpression is the true cause of the type error. This phenomenon is well-understood in the type error debugging research community. For example, while `*` is blamed in the expressions (3) and (4), we already know that that does not cause the

Term variables	x, y	Type variables	X, Y
Type constants	ι	Blame labels	ℓ
Static types	$T \ ::= \ \iota \mid X \mid T \to T$	Gradual types	$U \ ::= \ ? \mid \iota \mid X \mid U \to U$
Ground types	$G \ ::= \ \iota \mid ? \to ?$		

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & x \mid c \mid op(e,e) \mid \lambda x : U.e \mid e \ e \mid e : U \Rightarrow^{\ell} U \\ \text{Values} & w & ::= & c \mid op(w,w) \mid \lambda x : U.e \mid w : U \to U \Rightarrow^{\ell} U \to U \mid w : G \Rightarrow^{\ell} ? \end{array}$$

Figure 2 Syntax of types, expressions, and values.

type error. In subexpressions (7) and (9) through (11), while `false` may be the error cause (because changing it to some integer value will in fact fix the type error), `2` or/and `succ` are equally likely to have caused the type error. For example, the user may have intended to use a boolean value where `2` is, or they may have intended to use a boolean valued function such as `even` or `odd` instead of `succ`.

Based on these two observations, we conclude that blame tracking offers very little additional help to type error debugging. Moreover, our second observation implies that blame safety may not be an ideal way to report causes for dynamic type errors, since it is biased. In Section 3 we prove that this bias is indeed a general problem.

3 Blame Tracking is Biased

To prove that blame tracking is biased, our high-level idea is to show that gradual program execution embodies type constraint generation and solving, which are well-known to be biased in the type inference research community. Our idea is inspired by the work of dynamic type inference (DTI) for gradual typing [30]. In combining gradual typing and type inference, some type variables are left undecided at compile time due to the interaction of dynamic types. Consider, for example, the expression $(\lambda x : ?.x \ 2) \ (\lambda y.y)$. During type inference, the type, say Y, for the parameter y cannot be decided because it is solely required to be consistent with ? (which every type is consistent with). However, the choice of Y has a significant impact on the execution result. If Y is chosen to be `Int`, then the expression runs correctly. Otherwise, the expression leads to dynamic type errors. The challenge here is that it is statically difficult to decide that Y should be `Int`.

DTI addressed this issue by keeping Y as y's static type and deferring the instantiation of it to runtime. Eventually, $\lambda y.y$ will be applied to 2, making it clear that Y needs to be instantiated with `Int` to make this application succeed. Miyazaki et al. [30] proved that DTI is both sound and complete in the sense that if a term is evaluated successfully then some correct instantiation (such as instantiating Y with `Int` at compile time) of the term will execute successfully in the blame calculus [61, 1, 2].

This example illustrates that DTI mixes reductions and type instantiations without generating type constraints explicitly. To investigate whether existing type error localization and debugging approaches can help address the bias in blame tracking, we instead separate constraint generation (during program execution) and constraint solving (after the execution is finished). If constraint solving succeeds, then the program executes without raising blame in DTI. Otherwise, blame will be raised. The advantage of this separation is that it allows us to make a clear connection between constraint solving and blame tracking.

$$
\begin{array}{rcll}
op(w_1, w_2) & \longrightarrow_G & ([\![op]\!](w_1, w_2),\ \{\}) & \\
(\lambda x : U.e)\ w & \longrightarrow_G & (e[w/x],\ \{\}) & \\
w : \iota \Rightarrow^{\ell} \iota & \longrightarrow_G & (w,\ \{\}) & \\
w : ? \Rightarrow^{\ell} ? & \longrightarrow_G & (w,\ \{\}) & \\
(w_1 : U_1 \to U_2 \Rightarrow^{\ell} U_3 \to U_4)\ w_2 & \longrightarrow_G & ((w_1\ (w_2 : U_3 \Rightarrow^{\overline{\ell}} U_1)) : U_2 \Rightarrow^{\ell} U_4,\ \{\}) & \\
w : U \Rightarrow^{\ell} ? & \longrightarrow_G & (w : U \Rightarrow^{\ell} G \Rightarrow^{\ell} ?, \{\}) \quad (\text{if } U \neq ?, U \neq G, U \sim G) & \\
w : ? \Rightarrow^{\ell} U & \longrightarrow_G & (w : ? \Rightarrow^{\ell} G \Rightarrow^{\ell} U, \{\}) \quad (\text{if } U \neq ?, U \neq G, U \sim G) & \\
w : G_1 \Rightarrow^{\ell_1} ? \Rightarrow^{\ell_2} G_2 & \longrightarrow_G & (w, \{G_1 \cdot \overset{\ell_2}{=} \cdot G_2\}) & \textsc{Meet} \\
w : \iota \Rightarrow^{\ell_1} ? \Rightarrow^{\ell_2} X & \longrightarrow_G & (w, \{\iota \cdot \overset{\ell_2}{=} \cdot X\}) & \textsc{Base} \\
w : ? \to ? \Rightarrow^{\ell_1} ? \Rightarrow^{\ell_2} X & \longrightarrow_G & (w : ? \to ? \Rightarrow^{\ell_2} X_1 \to X_2, \{X \cdot \overset{\ell_2}{=} \cdot X_1 \to X_2\}) & \textsc{Arrow}
\end{array}
$$

Figure 3 Reduction rules. X_1 and X_2 in the ARROW rule are fresh.

3.1 Syntax

We consider a cast calculus with the type and expression syntax given in Figure 2. The definition is standard compared to other cast calculi [30, 47]. In the figure, $e : U_1 \Rightarrow^{\ell} U_2$ denotes a cast that ensures e has the type U_2 at runtime and raises blame at location ℓ otherwise. Both the cast and the label ℓ are inserted while translating gradual programs into programs in a cast calculus [47, 30].

The main difference between our calculus and others is that we do not have an explicit `blame` construct and do not terminate programs early, while others do [30, 47]. To avoid early termination, we add a value form $op(w_1, w_2)$ that interprets possible computations like `3 + true` as the value `+(3,true)`. The main reason for this addition is that we do not want to terminate program execution once a cast error would be encountered but rather collect all type constraints until program reduction finishes. We will show that our calculus and corresponding reduction rules yield the same result as others–that is they succeed with the the same value or fail with the same blame label.

3.2 Dynamic Constraint Generation and Solving

The reduction and constraint generation rules are presented in Figure 3. Our reductions have the form $e_1 \longrightarrow_G (e_2, C)$, where C is a set of constraints. A constraint has the form $U_1 \cdot \overset{\ell}{=} \cdot U_2$, denoting that U_1 and U_2 are required to be the same type, and ℓ will be blamed if they can not be made the same. We write $e \longrightarrow_G^* (e_n, C)$ if $e \longrightarrow_G (e_1, C_1)$, $e_1 \longrightarrow_G (e_2, C_2)$, $\ldots$, and $e_{n-1} \longrightarrow_G (e_n, C_n)$, and $C = C_1 \cup C_2 \cup \cdots \cup C_n$.

Since our semantics is designed to collect all type constraints, our reduction rules differ from those in [30] only when typing constraints are generated, or when a primitive operation would produce an error. The rules are the same in all other cases. Specifically, the first seven rules in Figure 3 are mostly standard among other cast dynamic semantics [30, 47]. Since no typing constraints are generated, the constraint set is empty in these rules.

The MEET rule handles two cases that are dealt separately in other cast semantics. In the first case, G_1 and G_2 are the same, and the cast will be successful and will thus be dropped. In the second case, G_1 and G_2 are different, and the cast will fail and blame ℓ_2. When our approach attempts to solve the constraint $G_1 \cdot \overset{\ell_2}{=} \cdot G_2$, it will behave as one of these cases. Specifically, if G_1 and G_2 are the same, then the constraint solving succeeds. Otherwise, constraint solving fails and blames ℓ_2.

In BASE, a dynamically typed value with primitive type ι can be projected into having type X if and only if X of is the same as ι. As a result, a constraint $\iota \cdot \stackrel{\ell_2}{=} \cdot X$ is generated. Similarly, in ARROW, when we project a value that has a function type $(? \rightarrow ?)$, our minimal expectation of the value is that it is a function $(U_1 \rightarrow U_2)$. Both of these rules are similar to those in [30], but they solve the constraint immediately while we collect and solve them later.

For a generated constraint set, the Robinson's unification algorithm [39] will suffice to solve it. For simplicity, we use $\mathcal{U}$ to denote that algorithm with following simple extensions. For any constraint set C, $\mathcal{U}(C)$ returns a substitution θ if constraint solving succeeds. When constraint solving fails, it returns $(\ell,\ \theta)$, where ℓ is the label of the constraint that fails to solve and must be blamed, and θ is the substitution accumulated so far. Note that $\mathcal{U}$ solves the constraints in the ordering they are added to the constraint set, as what classic type inference does [35].

In general, our approach of separating reduction and constraint solving is correct, as expressed in the following theorem, where $\longrightarrow_D^*$ denotes the reduction relation that mixes constraint solving and reduction, as defined in [30].

▶ **Theorem 1** (Correctness of $\longrightarrow_G$ and $\mathcal{U}$). *Given any expression e, let $e \longrightarrow_G^* (w_1, C)$.*

- *$\mathcal{U}(C) = \theta$ if and only if $e \longrightarrow_D^* w_2$ and $\theta(w_1) = w_2$.*
- *$\mathcal{U}(C) = (\ell,\theta)$ if and only if $e \longrightarrow_D^*$ `blame` ℓ.*

Intuitively, the theorem states that both approaches compute the same correct result or blame the same location. The theorem can be proved by induction over the $\longrightarrow_G$ relation defined in Figure 3 and the relation $\longrightarrow_D$ in [30].

The following example shows the reduction sequences of $\longrightarrow_G^*$ (between the two rules) and $\longrightarrow_D^*$ (after the second rule) for the expression $(\lambda x\ y.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ x\ \mathbf{else}\ y)\ (2{:}?)\ (5{:}?)$. Note, $\rightsquigarrow$ denotes the step of type inference and cast insertion. The two parameters have the same type variable X after type inference because (1) they have no ?s and normal type inference applies to them and the subexpressions using them and (2) they are the two branches of the same conditional, which are required to have the same type [16, 30].

$$(\lambda x\ y.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ x\ \mathbf{else}\ y)\ (2{:}?)\ (5{:}?)$$
$$\rightsquigarrow (\lambda x : X\ y : X.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ x\ \mathbf{else}\ y)\ (2 : \mathtt{Int} \Rightarrow^{\ell_2} ? \Rightarrow^{\ell_2} X)\ (5 : \mathtt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} X)$$

Reduction	Constraints
$\longrightarrow_G (\lambda x : X\ y : X.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ x\ \mathbf{else}\ y)\ 2\ (5 : \mathtt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} X)$	$\{\mathtt{Int} \cdot \stackrel{\ell_2}{=} \cdot X\}$
$\longrightarrow_G (\lambda y : X.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ 2\ \mathbf{else}\ y)\ (5 : \mathtt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} X)$	$\{\mathtt{Int} \cdot \stackrel{\ell_2}{=} \cdot X\}$
$\longrightarrow_G (\lambda y : X.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ 2\ \mathbf{else}\ y)\ 5$	$\{\mathtt{Int} \cdot \stackrel{\ell_2}{=} \cdot X, \mathtt{Int} \cdot \stackrel{\ell_5}{=} \cdot X\}$
$\longrightarrow_G^* 2$	$\{\mathtt{Int} \cdot \stackrel{\ell_2}{=} \cdot X, \mathtt{Int} \cdot \stackrel{\ell_5}{=} \cdot X\}$
$\longrightarrow_D (\lambda x : \mathtt{Int}\ y : \mathtt{Int}.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ x\ \mathbf{else}\ y)\ 2\ (5 : \mathtt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} \mathtt{Int})$	$\{X \mapsto \mathtt{Int}\}$
$\longrightarrow_D (\lambda y : \mathtt{Int}.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ 2\ \mathbf{else}\ y)\ (5 : \mathtt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} \mathtt{Int})$	
$\longrightarrow_D (\lambda y : \mathtt{Int}.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ 2\ \mathbf{else}\ y)\ 5$	
$\longrightarrow_D 2$	

For this example, the reduction $\longrightarrow_D^*$ produces the result 2. The relation $\longrightarrow_G^*$ also produces that result, but generates an additional constraint set $\{\mathtt{Int} \cdot \stackrel{\ell_2}{=} \cdot X, \mathtt{Int} \cdot \stackrel{\ell_5}{=} \cdot X\}$, which has the solution $\{X \mapsto \mathtt{Int}\}$ after being solved by the solver $\mathcal{U}$. Therefore, both reductions succeed and produce 2.

The following example shows the reduction sequences of $\longrightarrow_G^*$ (between the two rules) and $\longrightarrow_D^*$ (after the second rule) for the expression $(\lambda x\ y.\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ x\ \mathbf{else}\ y)\ (2{:}?)\ (\mathtt{false}{:}?)$.

$$
\begin{array}{ll}
& (\lambda x\ y.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ (2{:}?)\ (\texttt{false}{:}?) \\
\rightsquigarrow & (\lambda x : X\ y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ (2 : \texttt{Int} \Rightarrow^{\ell_2} ? \Rightarrow^{\ell_2} X)\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_f} ? \Rightarrow^{\ell_f} X)
\end{array}
$$

$$
\begin{array}{lll}
\longrightarrow_G & (\lambda x : X\ y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ 2\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_f} ? \Rightarrow^{\ell_f} X) & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X\} \\
\longrightarrow_G & (\lambda y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_f} ? \Rightarrow^{\ell_f} X) & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X\} \\
\longrightarrow_G & (\lambda y : X.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ \texttt{false} & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Bool} \cdot \overset{\ell_f}{=} \cdot X\} \\
\longrightarrow_G^* & 2 & \{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Bool} \cdot \overset{\ell_f}{=} \cdot X\}
\end{array}
$$

$$
\begin{array}{lll}
\longrightarrow_D & (\lambda x : \texttt{Int}\ y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ x\ \textbf{else}\ y)\ 2\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_f} ? \Rightarrow^{\ell_f} \texttt{Int}) & \{X \mapsto \texttt{Int}\} \\
\longrightarrow_D & (\lambda y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_f} ? \Rightarrow^{\ell_f} \texttt{Int}) & \\
\longrightarrow_D & (\lambda y : \texttt{Int}.\textbf{if}\ \texttt{true}\ \textbf{then}\ 2\ \textbf{else}\ y)\ (\texttt{blame}\ \ell_f) & \\
\longrightarrow_D & \texttt{blame}\ \ell_f &
\end{array}
$$

For this example, the reduction $\longrightarrow_D^*$ blames the location $\ell_{\texttt{f}}$ because `false` does not have the expected type `Int` at runtime. The relation $\longrightarrow_G^*$ produces the result 2 with the constraint set $\{\texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Bool} \cdot \overset{\ell_f}{=} \cdot X\}$. When solving this set in the ordering that constraints were added to this set, $\mathcal{U}$ fails to solve the second constraint because X will be updated to `Int` after the first constraint is solved. As a result, solving the second constraint leads the program label $\ell_{\texttt{f}}$ being blamed. Overall, both reductions have the same behavior of blaming $\ell_{\texttt{f}}$ for this example.

Although the two reductions blame same locations for expressions that have runtime type errors, the reduction $\longrightarrow_G^*$ can extract more useful information that can be exploited by existing type error debugging approaches to provide better blaming information (We briefly explore this idea in Section 4). The following example `cond3` illustrates this aspect. For the cast inserted program, the relation $\longrightarrow_D^*$ blames ℓ_2 and so does $\longrightarrow_G^*$. However, our reduction rules also collect the constraint set $C_{\texttt{cond}} = \{\texttt{Bool} \cdot \overset{\ell_f}{=} \cdot X, \texttt{Int} \cdot \overset{\ell_2}{=} \cdot X, \texttt{Int} \cdot \overset{\ell_5}{=} \cdot X\}$.

$$
\begin{array}{ll}
& \texttt{cond3} = (\lambda x\ y\ z.\textbf{if}\ \texttt{false}\ \textbf{then}\ x\ \textbf{else}\ (\textbf{if}\ \texttt{true}\ \textbf{then}\ y\ \textbf{else}\ z))\ (\texttt{false}{:}?)\ (2{:}?)\ (5{:}?) \\
\rightsquigarrow & (\lambda x : X\ y : X\ z : X.\textbf{if}\ \texttt{false}\ \textbf{then}\ x\ \textbf{else}\ (\textbf{if}\ \texttt{true}\ \textbf{then}\ y\ \textbf{else}\ z)) \\
& \qquad (\texttt{false} : \texttt{Bool} \Rightarrow^{\ell_f} ? \Rightarrow^{\ell_f} X)\ (2 : \texttt{Int} \Rightarrow^{\ell_2} ? \Rightarrow^{\ell_2} X)\ (5 : \texttt{Int} \Rightarrow^{\ell_5} ? \Rightarrow^{\ell_5} X)
\end{array}
$$

Based on Theorem 1, we can reduce blame tracking to constraint generation and solving in our approach. Since constraint solving is biased [14, 25, 29, 66, 10, 17], blame tracking is also biased. The difference between type inference and gradual typing is that constraints are collected at compile time in the former while at runtime in the latter. This means that in type inference the bias happens along the abstract syntax traversal ordering while in gradual typing the bias happens along the execution ordering.

4 Type Error Debugging for Blame Tracking

The previous section shows that blame tracking is biased similar to type inference, albeit with constraint collection happening at different times. This inspires that existing work in type error debugging may be adapted to alleviate the bias problem in blame tracking.

Reordering constraint solving. A common idea to combat the bias in the standard unification algorithm is to reorder the unification problems being solved [14, 25, 29, 66]. For example, if we solve the constraints in $C_{\texttt{cond}}$ from the last to the first, then the location ℓ_f, corresponding to `false`, will be blamed. While these approaches can improve blame tracking in some cases, they are in general still biased in the orderings they solve constraint problems.

Error slicing. Instead of just blaming one location, type error slicing approaches [50, 17, 40, 48, 49, 63] highlight all of the program locations relevant to a type error. For example, for the expression `cond3` in Section 3, although only ℓ_2 looks to cause the dynamic type error, all three locations ℓ_f, ℓ_2, and ℓ_5 will be identified by slicing approaches due to the connection of the common type variable X in all constraints. The downside of slicing approaches is that the user still must determine the real error cause among all those identified.

Error localization. Many approaches [24, 21, 18, 20, 34, 33, 27, 67, 68, 69] have been developed to exploit context information to locate the most likely error location among a set of locations. From the constraint set $C_{\texttt{cond}}$, all these approaches will blame ℓ_f as the error cause because the type variable X is unified with `Bool` once but `Int` twice. This result makes sense because changing `false` modifies only one subexpression, whereas the other fix requires changing 2 and 5. Most of these approaches, however, lack a concrete message about how to fix the type error.

Discussion. Some challenges exist in adopting existing type error debugging for improving blame tracking, including: (1) current gradual typing implementations do not facilitate constraint collection at runtime, (2) the assumptions that hold for type error debugging may not hold for blame tracking, and (3) shortcomings with error debugging approaches will also be transferred to those for blame tracking.

The real challenge is that, in some situations, neither blame tracking nor type error debugging help to debug the error, illustrated by the following expression.

```
(\x y -> if true then x else y) ((\x -> (x:?)) succ (2:?)) (false:?)
```

Assume the user intended to use `even` instead of `succ` in the expression, meaning that `succ` is the error source. Similar to the expression (11) (Section 2.2), this expression will cause a dynamic type error. However, neither blame tracking nor a potentially adopted type error debugging approach will be able to locate `succ` as the error source since it is not annotated with a ?. Worse, since this expression is well-typed, no existing type error debugging approach can help, leaving it to the user to determine the real problem.

One may suggest to remove all ?s in gradual programs and employ existing type error debugging approaches to assign blame for such programs. This idea is particularly intriguing as recent work on a user study of gradual typing behaviors [56] suggests that both experienced and novice programmers value static feedback for programs that have runtime type errors. However, the suggested idea fails because it may report errors in programs that do not fail. To illustrate, consider the following expression `condxy`, adopted from [6].

$$\texttt{condxy} = (\lambda x\ (y :?).\textbf{if}\ x\ \textbf{then}\ \texttt{even}\ y\ \textbf{else}\ \texttt{not}\ y)\texttt{true}\ 2$$

Assume `even` has the type `Int` → `Bool` and `not` has the type `Bool` → `Bool`. This gradual expression runs without blaming any subexpression. However, if we remove the ? for the parameter y, we receive an error that blames some subexpression in `condxy`. Consequently, this idea may yield too many false positive error reports.

5 Conclusion

Type error debugging for fixing and understanding type errors when type inference fails is a well-studied subject. Blame tracking, though relatively new, is a well-accepted mechanism for assigning blames to program locations when gradually-typed programs encounter runtime

type errors. This paper explores connections between these two error debugging mechanisms, focusing on how one can help the other. A fundamental observation in our exploration is that blame tracking can be reduced to constraint collection and solving, the two main components of type inference, indicating that the well-accepted gradual typing error debugging mechanism suffers from the bias problem in type inference. This illustrates two problems. First, it limits the ability to use gradual typing as a type-error debugging approach similar to deferred type errors. Second, it means that blame tracking in general may not help programmers find the cause of their runtime type errors. This calls for more research into understanding and improving of blame tracking, particularly in gradual languages that employ type inference to recover type information.

References

1. Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for All. *SIGPLAN Not.*, 46(1):201–214, January 2011. doi:10.1145/1925844.1926409.
2. Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for Free for Free: Parametricity, with and Without Types. *Proc. ACM Program. Lang.*, 1(ICFP):39:1–39:28, August 2017. doi:10.1145/3110283.
3. Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding Dynamic Types to C♯. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming*, pages 76–100, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
4. John Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating Gradual Types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '18, New York, NY, USA, 2018. ACM.
5. John Peter Campora, Sheng Chen, and Eric Walkingshaw. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proc. ACM Program. Lang.*, 2(ICFP):98:1–98:30, July 2018. doi:10.1145/3236793.
6. Giuseppe Castagna and Victor Lanvin. Gradual Typing with Union and Intersection Types. In *ACM SIGPLAN International Conference on Functional Programming*, ICFP 2017, 2017. To appear.
7. Christopher Chambers, Sheng Chen, Duc Le, and Christopher Scaffidi. The function, and dysfunction, of information sources in learning functional programming. *Journal of Computing Sciences in Colleges*, 28(1):220–226, 2012.
8. Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.*, 1(OOPSLA):48:1–48:30, October 2017. doi:10.1145/3133872.
9. S. Chen and M. Erwig. Guided Type Debugging. In *Int. Symp. on Functional and Logic Programming*, LNCS 8475, pages 35–51, 2014.
10. Sheng Chen and Martin Erwig. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 583–594, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535863.
11. Sheng Chen and Martin Erwig. Systematic identification and communication of type errors. *Journal of Functional Programming*, 28:e2, 2018. doi:10.1017/S095679681700020X.
12. Sheng Chen, Martin Erwig, and Karl Smeltzer. Exploiting diversity in type checkers for better error messages. *Journal of Visual Languages & Computing*, 39:10–21, 2017. Special Issue on Programming and Modelling Tools. doi:10.1016/j.jvlc.2016.07.001.
13. Olaf Chitil. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *ACM Int. Conf. on Functional Programming*, pages 193–204, September 2001.
14. Hyunjun Eo, Oukseh Lee, and Kwangkeun Yi. Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Generation Computing*, 22(1):1–36, 2004. doi:10.1007/BF03037279.

15 Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-order Functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM. doi:10.1145/581478.581484.

16 Ronald Garcia and Matteo Cimini. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 303–315, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676992.

17 Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *European Symposium on Programming*, pages 284–301, 2003.

18 Jurriaan Hage and Bastiaan Heeren. Heuristics for Type Error Discovery and Recovery. In *Implementation and Application of Functional Languages*, pages 199–216. Springer, 2007.

19 Jurriaan Hage and Peter Van Keeken. Mining for Helium. *Technical report UU-CS*, 2006.

20 Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 62–71, New York, NY, USA, 2003. ACM. doi:10.1145/871895.871902.

21 Bastiaan J. Heeren. *Top Quality Type Error Messages.* PhD thesis, Universiteit Utrecht, The Netherlands, September 2005. URL: http://www.cs.uu.nl/people/bastiaan/phdthesis.

22 Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. On Polymorphic Gradual Typing. *Proc. ACM Program. Lang.*, 1(ICFP):40:1–40:29, August 2017. doi:10.1145/3110284.

23 Lintaro Ina and Atsushi Igarashi. Gradual Typing for Generics. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 609–624, New York, NY, USA, 2011. ACM. doi:10.1145/2048066.2048114.

24 Gregory F. Johnson and Janet A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *ACM Symp. on Principles of Programming Languages*, pages 44–57, 1986. doi:10.1145/512644.512649.

25 Oukseh Lee and Kwangkeun Yi. A Generalized Let-Polymorphic Type Inference Algorithm. Technical report, Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 2000.

26 B. Lerner, M. Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *ACM Int. Conf. on Programming Language Design and Implementation*, pages 425–434, 2007. doi:10.1145/1250734.1250783.

27 Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. A Practical Framework for Type Inference Error Explanation. In *OOPSLA*, pages 781–799, 2016.

28 Bruce J. McAdam. How to repair type errors automatically. In Kevin Hammond and Sharon Curtis, editors, *Selected papers from the 3rd Scottish Functional Programming Workshop (SFP01), University of Stirling, Bridge of Allan, Scotland, August 22nd to 24th, 2001*, volume 3 of *Trends in Functional Programming*, pages 87–98. Intellect, 2001.

29 Bruce J McAdam. *Reporting Type Errors in Functional Programs.* PhD thesis, Larboratory for Foundations of Computer Science, The University of Edinburgh, 2002.

30 Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. Dynamic Type Inference for Gradual Hindley–Milner Typing. *Proc. ACM Program. Lang.*, 3(POPL):18:1–18:29, January 2019. doi:10.1145/3290331.

31 Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. In *OOPSLA*, New York, NY, USA, 2017. ACM. doi:10.1145/3133880.

32 Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In *ACM Int. Conf. on Functional Programming*, pages 15–26, 2003. doi:10.1145/944705.944708.

33 Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding Minimum Type Error Sources. In *OOPSLA*, pages 525–542, 2014.

34 Zvonimir Pavlinovic, Tim King, and Thomas Wies. Practical SMT-based Type Error Localization. In *ICFP*, pages 412–423, 2015.

35 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

36 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The Ins and Outs of Gradual Type Inference. *SIGPLAN Not.*, 47(1):481–494, January 2012. doi:10.1145/2103621.2103714.

37 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *POPL*, 2015.

38 Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM. doi:10.1145/2635868.2635922.

39 J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965. doi:10.1145/321250.321253.

40 Thomas Schilling. Constraint-Free type error slicing. In *Trends in Functional Programming*, pages 1–16. Springer, 2012.

41 Eric L. Seidel, Ranjit Jhala, and Westley Weimer. Dynamic Witnesses for Static Type Errors. In *ACM SIGPLAN International Conference on Functional Programming*, 2016. to appear.

42 Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. Learning to Blame: Localizing Novice Type Errors with Data-driven Diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA):60:1–60:27, October 2017. doi:10.1145/3138818.

43 Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and Coercion: Together Again for the First Time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 425–435, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737968.

44 Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In *In Proceedings of the 24th European Symposium on Programming (ESOP'15)*, volume 9032 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-662-46669-8_18.

45 Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

46 Jeremy G. Siek and Manish Vachharajani. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 7:1–7:12, New York, NY, USA, 2008. ACM. doi:10.1145/1408681.1408688.

47 Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

48 Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in Haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 72–83, 2003. doi:10.1145/871895.871903.

49 Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *ACM SIGPLAN Workshop on Haskell*, pages 80–91, 2004. doi:10.1145/1017472.1017486.

50 F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. on Software Engineering and Methodology*, 10(1):5–55, January 2001. doi:10.1145/366378.366379.

51 Ville Tirronen, SAMUEL UUSI-MÄKELÄ, and VILLE ISOMÖTTÖNEN. Understanding beginners' mistakes with Haskell. *Journal of Functional Programming*, 25:e11, 2015.

52 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 964–974, New York, NY, USA, 2006. ACM. doi:10.1145/1176617.1176755.

53 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328486.

54 Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.SNAPL.2017.17`.

55 Kanae Tsushima and Olaf Chitil. A Common Framework Using Expected Types for Several Type Debugging Approaches. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming*, pages 230–246, Cham, 2018. Springer International Publishing.

56 Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The Behavior of Gradual Types: A User Study. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pages 1–12, New York, NY, USA, 2018. ACM. `doi:10.1145/3276945.3276947`.

57 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 762–774, New York, NY, USA, 2017. ACM. `doi:10.1145/3009837.3009849`.

58 Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: a compiler pearl. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 341–352, 2012. `doi:10.1145/2364527.2364554`.

59 Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, September 2011.

60 Philip Wadler. A Complement to Blame. In *SNAPL*, 2015.

61 Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag. `doi:10.1007/978-3-642-00590-9_1`.

62 Mitchell Wand. Finding the source of type errors. In *ACM Symp. on Principles of Programming Languages*, pages 38–43, 1986. `doi:10.1145/512644.512648`.

63 Jeremy Richard Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions*. PhD thesis, The University of Melbourne, January 2006.

64 Baijun Wu, John Peter Campora III, and Sheng Chen. Learning User Friendly Type-error Messages. *Proc. ACM Program. Lang.*, 1(OOPSLA):106:1–106:29, October 2017. `doi:10.1145/3133930`.

65 Baijun Wu and Sheng Chen. How Type Errors Were Fixed and What Students Did? *Proc. ACM Program. Lang.*, 1(OOPSLA):105:1–105:27, October 2017. `doi:10.1145/3133929`.

66 Jun Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Heriot-Watt University, May 2001.

67 Danfeng Zhang and Andrew C. Myers. Toward General Diagnosis of Static Errors. In *ACM Symp. on Principles of Programming Languages*, pages 569–581, 2014.

68 Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. Diagnosing Type Errors with Class. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–21, 2015.

69 Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. SHErrLoc: A Static Holistic Error Locator. *ACM Trans. Program. Lang. Syst.*, 39(4):18:1–18:47, August 2017. `doi:10.1145/3121137`.

What is a Secure Programming Language?

Cristina Cifuentes
Oracle Labs, Australia
cristina.cifuentes@oracle.com

Gavin Bierman
Oracle Labs, UK
gavin.bierman@oracle.com

Abstract

Our most sensitive and important software systems are written in programming languages that are inherently insecure, making the security of the systems themselves extremely challenging. It is often said that these systems were written with the best tools available at the time, so over time with newer languages will come more security. But we contend that all of today's mainstream programming languages are insecure, including even the most recent ones that come with claims that they are designed to be "secure". Our real criticism is the lack of a common understanding of what "secure" might mean in the context of programming language design. We propose a simple data-driven definition for a secure programming language: that it provides first-class language support to address the causes for the most common, significant vulnerabilities found in real-world software. To discover what these vulnerabilities actually are, we have analysed the National Vulnerability Database and devised a novel categorisation of the software defects reported in the database. This leads us to propose three broad categories, which account for over 50% of all reported software vulnerabilities, that *as a minimum* any secure language should address. While most mainstream languages address at least one of these categories, interestingly, we find that none address all three.

Looking at today's real-world software systems, we observe a paradigm shift in design and implementation towards service-oriented architectures, such as microservices. Such systems consist of many fine-grained processes, typically implemented in multiple languages, that communicate over the network using simple web-based protocols, often relying on multiple software environments such as databases. In traditional software systems, these features are the most common locations for security vulnerabilities, and so are often kept internal to the system. In microservice systems, these features are no longer internal but external, and now represent the attack surface of the software system as a whole. The need for secure programming languages is probably greater now than it has ever been.

2012 ACM Subject Classification Software and its engineering → Language features; Security and privacy → Software security engineering

Keywords and phrases memory safety, confidentiality, integrity

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.3

© Cristina Cifuentes and Gavin Bierman;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 3; pp. 3:1–3:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Analysing software vulnerabilities

The National Vulnerability Database (NVD) is the US government repository of standards-based vulnerability management data. It takes as its data feed the vulnerabilities given in the Common Vulnerabilities and Exposures (CVE) directory, and performs various analyses to determine impact metrics (CVSS), vulnerability types (CWE), and applicability statements (CPE), and other useful metadata.

For the past few decades, this data has been extensively analysed, aggregated and categorised, and gives probably the best insight into the most common security issues facing the software industry. In 1995, the NVD contained only 25 entries; in 2017, this had ballooned to more than 10,000 entries for that year alone. The software industry clearly faces a serious problem. Moreover, there is strong evidence that this is an *underestimation* of the actual problem. For example, many cloud software defects are not even assigned a CVE, because cloud software is typically architected to be continuously updated, making it difficult to be tracked by services such as the CVE list.

The main technical contributions of this paper are an analysis of the labelled data in the NVD for the five years from 2013 to 2017, where we have normalised and aggregated the data related to source code vulnerabilities into broad categories, and a more precise and meaningful definition of a *secure* language.

In our analysis of the NVD data, interestingly, we discovered that three of the top four most common vulnerabilities are actually issues that can be considered to be in the realm of programming language design. Moreover, when combined, these three categories of vulnerabilities represent 53% of all labelled exploited vulnerabilities listed in the NVD for that period. The complete analysis is presented in Appendix A. The three categories and the number of reported vulnerabilities they represent are as follows:

- 5,899 buffer errors
- 5,851 injection errors[1]
- 3,106 information leak errors.

It is a little depressing that two of these vulnerability categories are very old. The Morris worm [16] exploited a buffer error in the Unix finger server in 1988 – over 30 years ago. SQL injections [14] and XSS exploits [2] have been documented in the literature since 1998 and 2000, respectively.

A 2018 study [9] revealed that the average total cost of a data breach is US$3.86 million, with an average cost per record of US$157 due to hacker or criminal attacks. With all the additional costs of software defects, including reputational loss, productivity loss, among many others, there is an obvious incentive to try to address more than 50% of vulnerabilities by better programming language design.

This paper is organised as follows. In §2 we give the definition and an example of each of the three types of vulnerabilities discussed in this paper. In §3 we take a look at mainstream languages, and consider their support for the three categories of software vulnerabilities. We consider this problem abstractly, examining the trade-offs when using new abstractions provided by a programming language. In §4 we look at the common programming language abstractions to address buffer overflows. In §5 we look at the common programming language abstractions to address the most frequently occurring forms of injection error. Finally, in §6 we consider the issue of information leak errors, and point to a promising technique from the research community. We conclude in §7, and give details of our categorisation of the CWE enumerations used in the NVD, and analysis of recent five years of NVD data in Appendix A.

[1] Injection errors include Cross-Site Scripting (XSS), SQL injection, Code injection, and OS command injection.

2 The Three Categories of Vulnerabilities

In this paper we focus on three vulnerabilities that are widely exploited year over year, and yet, those vulnerabilities could be prevented through programming language design. Throughout this section we make use of NIST's Computer Security Resource Center definitions [4] and CWE examples from Mitre [5].

2.1 Buffer Errors

A buffer overflow attack is a method of overloading a predefined amount of memory storage in a buffer, which can potentially overwrite and corrupt memory beyond the buffer's boundaries. Buffer overflow errors can be stack-based or heap-based, depending on the location of the memory that the buffer uses.

An example of a stack-based buffer overflow, CWE-121, follows. In this C code, the function `host_lookup` allocates a 64-byte buffer, `hostname`, to store a hostname. However, there is no guarantee in the code that the hostname will not be larger than 64 bytes, and hence, this is a vulnerability. An attacker can attack this vulnerability by supplying an address that resolves to a large hostname, overwriting sensitive data or even relinquishing control to the attacker:

```
void host_lookup(char *user_supplied_addr){
struct hostent *hp;
in_addr_t *addr;
char hostname[64];
in_addr_t inet_addr(const char *cp);

/*routine that ensures user_supplied_addr is in the right format
 for conversion */

validate_addr_form(user_supplied_addr);
addr = inet_addr(user_supplied_addr);
hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
strcpy(hostname, hp->h_name);
}
```

A canonical example of a heap-based buffer overflow, CWE-122, follows. In this C code, a heap-based buffer `buf` is allocated of a given size. When a copy of the string `argv[1]` into that buffer is made, there is no check on the size of the buffer, leading to an overflow for strings larger than the allocated size. This is a vulnerability because the user/attacker has control over the string `argv[1]`.

```
#define BUFSIZE 256
int main(int argc, char **argv) {
char *buf;
buf = (char *)malloc(sizeof(char)*BUFSIZE);
strcpy(buf, argv[1]);
}
```

2.2 Injection Errors

Injection errors include several types of vulnerabilities, the most common ones being: cross-site scripting (XSS), SQL injection, code injection, and OS command injection. We provide examples for the first two types of vulnerabilities.

Cross-site scripting is a vulnerability that allows attackers to inject malicious code into an otherwise benign website. These scripts acquire the permissions of scripts generated by the target website and can therefore potentially compromise the confidentiality and integrity of data transfers between the website and client. Websites are vulnerable if they display user-supplied data from requests or forms without sanitising the data so that it is not executable. There are three main kinds of XSS, all part of CWE-79: reflected XSS (or non-persistent); stored XSS (persistent); and DOM-based XSS.

An example of a reflected XSS follows. The JSP code fragment reads an employee ID, `eid`, from an HTTP request and displays it to the user without sanitising the employee ID. This vulnerability can be exploited by an attacker by including meta-characters or source code in the input, then that code will be executed by the web browser as it displays the HTTP response.

```
<% String eid = request.getParameter("eid"); %>
...
Employee ID: <%= eid %>
```

An example of a stored XSS follows. The JSP code fragment queries a database for a particular employee ID and prints the corresponding employee's name. This is a vulnerability if the value of the name originates from user-supplied data and has not been validated. As such, without proper input validation on all data stored in the database, this vulnerability can be exploited by an attacker by executing malicious commands on the user's web browser.

```
<%Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select␣*␣from␣emp␣where␣id="+eid);
if (rs != null) {
 rs.next();
 String name = rs.getString("name");
}%>

Employee Name: <%= name %>
```

An SQL injection vulnerability is one that allows attackers to execute arbitrary SQL code on a database back-end. A canonical example of an SQL injection vulnerability, CWE-89, follows. In this C$^\sharp$ code fragment, an SQL query is dynamically constructed that searches for items matching a currently authenticated user name. Because the query concatenates a user-defined string, `ItemName.Text`, the query will behave correctly only if the item name does not contain a single-quote character. This vulnerability is exploited by an attacker with the user name wiley by entering an item name such as `name’ OR ‘a’=‘a`, leading to the SQL query `SELECT * FROM items WHERE owner = ‘wiley’ AND itemname = ‘name’ OR ‘a’=‘a’;`, which effectively becomes `SELECT * FROM items;` as the `WHERE` clause evaluates to true. Even worse, if the SQL API supports multiple statements, then a malicious user would enter an item name such as `‘ OR 1=1; DROP TABLE users; SELECT * FROM secretTable WHERE ‘t’ = ‘t` that would delete the table of users and extract all the details from a secret table.

```
...
string userName = ctx.getAuthenticatedUserName();
string query = "SELECT␣*␣FROM␣items␣WHERE␣owner␣=␣'" + userName +
 "'␣AND␣itemname␣=␣'" + ItemName.Text + "'";
sda = new SqlDataAdapter(query, conn);
DataTable dt = new DataTable();
sda.Fill(dt);
...
```

2.3 Information Leak Errors

Information leakage is the intentional or unintentional release of information to an untrusted environment. There are many forms of information leakage in today's software systems. Information can be leaked through log files, caching, environment variables, test code, shell error messages, servlet runtime error messages, Java runtime error messages, and more. The most common type of information leak is through log files (whether debug logs, server logs, or other).

An example of an information leak through log files, CWE-532, follows. In this Java code fragment, the `this` object contains the location of the user. When the application encounters an exception, it will write the user object to the log, including the location information.

```
locationClient = new LocationClient(this, this, this);
locationClient.connect();
currentUser.setLocation(locationClient.getLastLocation());
...

catch (Exception e) {
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Sorry,␣this␣app␣has␣experienced␣an␣error.");
AlertDialog alert = builder.create();
alert.show();
Log.e("ExampleActivity", "Caught␣exception:␣" + e + "␣While␣on␣User:"
 + User.toString());
}
```

3 Mainstream languages and vulnerabilities

For the purposes of this paper, we restrict our attention only to what we term mainstream languages. We use this term a little loosely, and we are certainly not arguing the relative merits of these languages over each other, or any other language for that matter. To be concrete, we used the data from the respected TIOBE Index [17], though interestingly, other similar indexes yield very similar, if not identical, results. According to the TIOBE index of January 2019, the current top 10 mainstream programming languages are: Java, C, Python, C++, Visual Basic .NET, JavaScript, C$^\sharp$, PHP, SQL and Objective-C. However, for this paper, we take the cumulative data for the past 10 years, and therefore we consider a language to be mainstream if it has been in the top 10 for every year of the past 10 years. Accordingly, we take Java, C, C++, Python, C$^\sharp$, PHP, JavaScript and Ruby as the mainstream programming languages to be considered in this paper.

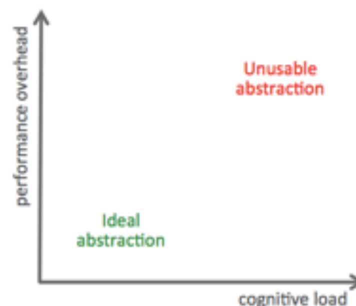
So, we have our mainstream languages, and from our analysis of the NVD we have our three security vulnerability categories. Our thesis is that any secure programming language worthy of its name should be one that has first-class support for all three categories. Put diagrammatically, a secure language is one that lives in the intersection of the following Venn diagram:

Unfortunately, given this definition, it is the case that *none of our mainstream languages can be considered secure.*

It is worth taking a step back, and considering the causes of vulnerabilities in software. Developers do not write incorrect code because they want to. Vulnerable code is written inadvertently because our mainstream programming languages do not provide the right abstractions to support developers in writing vulnerable-prone code. Buffer errors are introduced because of manual management (allocation, reallocation, and deallocation) of pointers. Injection errors are introduced because of the representation of code as strings, use of manual string concatenation, and sanitisation of strings that reach a sensitive location (e.g., an SQL execute statement). And information leak errors are introduced because of manual tracking of sensitive data, trying to ensure it does not leak to less sensitive objects.

Abstractions in programming languages introduce different levels of cognitive load. The easier it is for an abstraction to be understood, the more accepted that abstraction becomes. For example, managed memory is an abstraction that frees the developer from having to manually keep track of memory (both allocation and deallocation). As such, this abstraction is widely used in a variety of mainstream and non-mainstream languages. At the same time, performance of the developed code is also relevant to developers. If an abstraction introduces a high performance overhead, it makes it hard for that abstraction to be used in practice in some domains. For example, managed memory is not often used in embedded systems or systems programming due to the difficulty of predicting its performance overhead.

The space of these two criteria that needs to be considered when developing safe abstractions can be visualised as follows:



The ideal abstraction is clearly one that has low cognitive load and low performance overhead. By contrast, abstractions that provide a high cognitive load and high performance overhead will never be used in a mainstream programming language. There are abstractions in between that provide a low cognitive load yet a high performance overhead. Those abstractions may be usable in certain domains, as are abstractions that may have a high cognitive load and low performance overhead.

For the issues addressed in this paper, we are interested in abstractions that address the vulnerabilities from our analysis of the NVD. We refer to these abstractions as *safe abstractions*. Managed memory is thus an example of a safe abstraction with respect to buffer errors. In §§4–6 we provide examples of other safe abstractions.

4 Language support addressing buffer overflows

Providing language support that addresses buffer overflows is almost as old as programming language design itself. The traditional technique is to use managed memory, discussed in §4.1. However, this technique has not had great acceptance in the systems community, where concerns about performance overheads, as per the discussion in the previous section, have dominated. In §4.2 we mention an interesting new development addressing buffer overflows using an ownership-inspired type system.

4.1 Managed memory

In 1958, the LISP language introduced the concept of managed memory by adding garbage collection in the runtime of the language. Garbage collection provides a solution that frees up developers from the cognitive load of having to allocate, reallocate and deallocate memory objects correctly. Developers do not need to specify any allocation and deallocation instructions because it is done under the hood by the runtime of the language.

Managed memory took some time to become widely used due to its performance overhead. With the advent of faster computers, it is now widely used by object-oriented languages such as Smalltalk, Java, C$^\sharp$, JavaScript and Go; functional languages such as ML, Haskell and APL; and dynamic languages such as Ruby, Perl and PHP. Note that many of today's mainstream languages use managed memory, however, in some domains, such as embedded systems and operating systems, the performance overhead of a managed memory runtime is still not feasible.

LISP's managed memory abstraction was introduced to free up developers from the cognitive load of allocating and deallocating memory objects manually. At the time, exploitation of buffer errors was not even known, c.f., the Morris worm didn't appear until 1988. As such, this abstraction was not introduced for security, however, it is a great example of a safe abstraction that can be used to provide security against buffer errors and other types of memory-related errors.

4.2 Ownership and borrowing

Rust is a systems programming language introduced in 2009 that runs fast, prevents memory corruption, and is designed to guarantee memory and thread safety, though has not been formally verified yet. Not only does it prevent buffer errors, it prevents various other types of memory corruptions, such as null pointers and use after free. This feature is provided by the introduction of ownership and borrowing into the type system of the language:

- Ownership is a programming pattern employed in C++ and other languages whereby a resource can have only one owner. Ownership with "resource acquisition is initialisation" (RAII) ensures that whenever an object goes out of scope, its destructor is called and its owned resource is freed. Ownership of a resource is transferred (i.e., moved) through assignments or passing arguments by value. When a resource is moved, the previous owner can no longer access it, therefore preventing dangling pointers.

- Borrowing is an abstraction that allows a reference to a resource to be made available in a secure way – either through a shared borrow (`&T`), where the shared reference cannot be mutated, or a mutable borrow (`&mut T`), where the shared reference cannot be aliased – but not both at the same time. Borrowing allows for data to be used elsewhere in the program without giving up ownership. It prevents use after free and data races.

Ownership and borrowing provide abstractions suitable for memory safety, and prevent buffer errors from happening in the code. Anecdotal evidence seems to suggest that the learning curve to become fluent in the use of these abstractions can be long, pointing to a high cognitive load. Nevertheless, given its low performance overhead, this safe abstraction with respect to memory-related errors is an exciting development in the systems programming domain.

5 Language support addressing injection errors

No mainstream language offers a general safe abstraction for injection errors. .NET offers support for language-level query abstractions, including queries over a relational database. This removes the possibility of SQL injection errors, which we consider in more detail in §5.1. Although Perl is not strictly a mainstream language by our criteria, in §5.2 we consider its taint tracking supported, that was later extended in Ruby (which is a mainstream language by our criteria). Taint tracking could form the basis of a more general abstraction to address injection errors.

5.1 LINQ to SQL

Language INtegrated Query (LINQ) is a framework introduced in 2007 by Microsoft for .NET 3.5 that adds language-level query facilities to the .NET languages. It extends the languages by adding first-class query expressions (along with some supporting extensions to the type system and other language extensions) that can be used to extract and process data from any source that supports a predefined set of methods, known as the standard query operator API. A data type that supports this API is known as a LINQ provider. By default, all arrays and enumerable classes are LINQ providers. This allows developers to write high-level declarative queries over their in-memory collections.

Moreover, .NET also provides other implementations of LINQ providers, in particular, LINQ-to-SQL. This allows queries to be written in a .NET language and be translated into a semantically equivalent query that executes on an SQL engine. In addition, it comes with an object-relational mapping framework that automatically generates strongly typed .NET class declarations that correspond to tables in the database. LINQ-to-SQL also solves the SQL injection problem because the programmer no longer constructs SQL code using strings and string concatenation (the source of SQL injection errors) but uses the language-level queries, which themselves pass all data to the database using injection-safe SQL parameters.

LINQ-to-SQL allows for a large percentage of SQL queries to be written in a simpler language, but not all SQL queries can be represented. A number of more advanced SQL queries, for example, that involve selecting into temporary tables and then querying those tables, predicated updates and bulk inserts, and triggers, are not supported. Vendor-specific extensions to SQL are also not supported. These features have to be accessed through stored procedures that have to be defined on the database side.

5.2 Taint tracking

Perl is a rapid prototyping language introduced in 1987. In 1989, Perl 3 introduced the concept of taint mode to track external input values (which are considered tainted by default), and to perform runtime taint checks to prevent direct or indirect use of a tainted value in any command that invokes a sub-shell, or any command that modifies files/directories/processes, except for arguments to `print` and `syswrite`, symbolic methods and symbolic subreferences, or hash keys. Default tainted values include all command-line arguments, environment variables, locale information, results of some system calls (`readdir()`, `readlink()`), etc. Importantly, taint mode is supported in the runtime of the language. Perl 5 supports taint mode, but Perl 6 does not. Note that because there is no tracking of taint to `print` statements, cross-site scripting attacks on web applications are still possible.

Ruby is a dynamic programming language developed in 1993 with a focus on simplicity and productivity. It supports multiple programming paradigms, including functional, object-oriented, imperative, and reflective. Ruby extends Perl's taint mode to provide more flexibility. There are four safe levels available, of which the first two are as per Perl, as follows:

0. No safety.
1. Disallows use of tainted data by potentially dangerous operations. This level is the default on Unix systems when running Ruby scripts as `setuid`.
2. Prohibits loading of program files from globally writable locations.
3. Considers all newly created objects to be tainted.

In Ruby, every object has a `Trusted` flag. There are methods to make an object tainted, check whether the object is tainted, or untaint the object (only for levels 0–2). At runtime, Ruby tracks direct data flows through levels 1–3; it does not track indirect/implicit data flows.

The taint tracking abstraction provides a way to prevent some types of injection errors with low cognitive load on developers. Apart from Perl and Ruby, PHP [18] and version 1.1. of JavaScript implemented taint tracking in the runtime. Trade-offs in performance overhead need to be made to determine how much data can be tracked, what target locations should be tracked, and whether direct and indirect uses can be tracked. Livshits [10] suggests several reasons why taint tracking is still an unsolved problem including the following: predictable performance needs; whether control flow tracking is needed; value tracking of data that may be safe though deemed to be potentially tainted at runtime; determining declassifiers; specification inference at runtime; and configurable runtime support for taint. Proposals for reducing performance overhead include combining the approach with static analysis. Further, for some applications, values need to be tracked not only for strings, but also for primitive values, collections, or other objects, resulting in performance overheads.

6 Language support addressing information leak errors

As far as we are aware, there is no language-level feature in any mainstream language to address vulnerabilities resulting from information leak errors. This is clearly an area requiring further research. We briefly review one interesting approach from the research community, although other approaches such as tracking sensitive data [6], data shadowing of sensitive data [8], and a DSL for data-centric applications [13] are being pursued.

6.1 Faceted values

Jeeves is an experimental academic language for automatically enforcing information flow policies, first released in 2014 [21]. It is implemented as an embedded DSL in Python. Jeeves makes use of the faceted values abstraction, which is a data type used for sensitive values

that stores within it the secret `s` (high-confidentiality) and non-secret `ns` (low-confidentiality) values, guarded by a policy `p`, e.g., `<s | ns> (p)`. A developer specifies policies outside the code, making the code policy-agnostic, and the language runtime enforces the policy by guaranteeing that a secret value may flow to a viewer only if the policies allow the viewer to view secret data.

Many applications today make use of a database. To make the language practical, faceted values need to be introduced into the database when dealing with database-backed applications. A faceted record is one that guards a secret and non-secret pair of values. Jacqueline, a web framework developed to support faceted values in databases, automatically reads and writes metadata in the database to manage relevant faceted records. The developer can use standard SQL databases through the Jacqueline object relational mapping.

The faceted values abstraction provides a way to prevent information leaks with low cognitive load on developers, at the expense of performance overhead. This work is yet to determine the lower bound on performance overhead, in order to provide direct and indirect tracking of the data flows for leaks of sensitive data purposes.

We note that faceted values could also be used to prevent injection errors, as shown by the DroidFace implementation for an intermediate Java-like language [15]. Conversely, taint tracking can also be used to track sensitive user data leaving an application through a network, file system, or similar [6].

7 Concluding Remarks and Further Work

The mainstream languages over the past 10 years are Java, C, C++, Python, C♯, PHP, JavaScript and Ruby. In this paper we have considered these languages with respect to their support for features that directly address the sorts of vulnerabilities in real-world software systems.

To identify these vulnerabilities, we have analysed the labelled data in the NVD for the five-year period of 2013–2017. Employing a novel categorisation, we have identified that three of the top four categories of vulnerabilities for that period actually represent issues that can be considered to be in the realm of programming language design; namely, prevention of buffer errors, injection errors, and information leak errors.

We observe that none of today's mainstream languages provide safe abstractions that address all three of these prominent types of exploited vulnerabilities. By our criteria, we claim that none of our mainstream languages can be considered secure.

Buffer errors are addressed by all mainstream languages, apart from C, C++, and PHP, through use of the managed memory. We note that taint tracking is in place in only the Ruby and PHP (through an extension) mainstream languages to avoid injection errors, and that some classes of injection errors have been addressed by LINQ for the .NET languages. Taint tracking solutions are not yet mature enough to capture all aspects of what needs to be tracked with a good performance trade-off. As far as we are aware, no mainstream language supports a safe abstraction to deal with information leak errors.

We note that a couple of upcoming languages, Rust and Pony [3], have designed their languages to not only avoid buffer errors, but also concurrency (data race) errors. They use different safe abstractions for data race prevention, and neither provides abstractions to prevent injection or information leak errors.

Many languages today provide a foreign function interface (FFI) to "escape" into another language that normally provides other properties, such as performance or low-level system access. Such an FFI escape hatch breaks any concept of a safe abstraction in the language because the compiler and/or runtime does not track information across the language boundaries.

Ideally, in order to help developers in writing vulnerability-prone code, we need to provide safe abstractions that complement existing mainstream languages, because it is impractical to migrate the millions of lines of code already written and relied upon by millions of developers. Indeed, much of the existing software is developed in languages that are distinctly *in*secure! Two interesting and somewhat orthogonal approaches to this problem are being explored in the research community:

1. One approach is to explore multilingual compilers and runtimes that have been architected to simultaneously support multiple languages, with differing security abstractions (e.g. [20]). More foundational work in this area is also encouraging [12].
2. Another approach is to maintain a monolingual runtime but employ possibly many stages of secure compilation; that is, compilation that preserves security properties via translations that are fully abstract (e.g. [1, 7]).

Further work also involves continued analysis of the vulnerability data. While we have paid attention to the largest categories, and hope that they will be addressed, the remaining categories will become increasingly more relevant. Put another way, these other vulnerability categories should be drivers for programming language design of the future.

Finally, we conclude by drawing attention to the significant change in the design of real-world systems from large, standalone software systems to software-as-a-service. This reflects both the change in underlying infrastructure from in-house systems to rented cloud platforms, and also the requirements of systems to seamlessly scale up/down. Many existing applications, as well as new applications, are being re-architected using quite radically different design patterns, such as microservices (functions-as-a-service). While it should be expected that familiar vulnerabilities will be reported from these applications, it is entirely reasonable to expect that either new vulnerabilities will emerge, or existing ones will become much more common. There is some work on security aspects of microservice-style programming – for example, Whip [19] proposes a formal contracts system for microservices code that prevents software errors at the edge – but it is clear that further research is needed, for example, how to prevent second-order SQL injections in a microservice.

References

1 Martín Abadi. Protection in Programming-Language Translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, pages 19–34, 1999. doi:10.1007/3-540-48749-2_2.

2 CERT. Malicious HTML Tags, 2000.

3 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA, 2015. ACM. doi:10.1145/2824815.2824816.

4 NIST Computer Security Resource Center – Glossary, Last accessed April 8, 2019. URL: https://csrc.nist.gov/Glossary.

5 Common Weakness Enumeration – CWE list version 3.2, Last accessed April 8, 2019. URL: https://cwe.mitre.org/data/index.html.

6 William Enck, Peter Gilbert, 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. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1924943.1924971.

7 Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 371–384, 2013. doi:10.1145/2429069.2429114.

8 Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren't the Droids You'Re Looking for: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM. doi:10.1145/2046707.2046780.

9 Ponemon Institute. 2018 Cost of Data Breach Study: Global Overview, July 2018. URL: https://databreachcalculator.mybluemix.net/assets/2018_Global_Cost_of_a_Data_Breach_Report.pdf.

10 Benjamin Livshits. Dynamic Taint Tracking in Managed Runtimes. Technical Report MSR-TR-2012-114, Microsoft Research, November 2012.

11 National Vulnerability Database – NVD CWE Slice, Last accessed February 7, 2019. URL: https://nvd.nist.gov/vuln/categories.

12 Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. *CoRR*, abs/1711.04559, 2017. arXiv:1711.04559.

13 Nadia Polikarpova, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Enforcing Information Flow Policies with Type-Targeted Program Synthesis, 2018. arXiv:1607.03445v2.

14 rain.forest.puppy. NT web technology vulnerabilities. *Phrack Magazine*, 8(54), 1998. URL: http://www.phrack.org/archives/issues/54/8.txt.

15 Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. Let's Face It: Faceted Values for Taint Tracking. In *ESORICS (1)*, volume 9878 of *Lecture Notes in Computer Science*, pages 561–580. Springer, 2016.

16 E. H. Spafford. Crisis and Aftermath. *Commun. ACM*, 32(6):678–687, June 1989. doi:10.1145/63526.63527.

17 TIOBE index, February 2019. URL: http://www.tiobe.com/tiobe-index.

18 Wietse Venema. Taint Support for PHP, Last modified: 2017/09/22. URL: https://wiki.php.net/rfc/taint.

19 Lucas Waye, Stephen Chong, and Christos Dimoulas. Whip: higher-order contracts for modern services. *PACMPL*, 1(ICFP):36:1–36:28, 2017. doi:10.1145/3110280.

20 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062381.

21 Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, Dynamic Information Flow for Database-backed Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 631–647, New York, NY, USA, 2016. ACM. doi:10.1145/2908080.2908098.

22 Yves Younan. 25 Years of Vulnerabilites: 1988–2012. Research report, Sourcefire, 2013.

A Appendix – NVD data analysis

CWE Classification

The Common Weakness Enumeration (CWE) [11] specification provides a classification of software security vulnerabilities as they are found in source code, design, or system architecture. An individual CWE represents a single vulnerability type. CWEs are organised in a hierarchical structure – CWEs at the highest levels of the structure provide an overview of a vulnerability category (e.g., "data handling"), and the leaves of the tree represent actual vulnerability types (e.g., "out-of-bounds read", "SQL injection", "cross-site scripting"). The National Vulnerability Database makes use of 34 CWEs for its classification of the various reported CVEs. We focus on the ones related to source code errors.

The first 25 years of vulnerability data, from 1988 to 2012, were collated and analysed by Sourcefire [22]. From September 2007, the NVD changed its methodology and adopted 19 CWE categories to classify vulnerabilities, but chose not to reclassify the earlier ones. Sourcefire normalised the data by manually mapping from the old categories to the 19 new ones. NVD also makes use of two CWEs that do not provide information for classification purposes and are not counted towards the 19. They are "Insufficient Information" and "Other". The 19 CWE categories are:

1. Buffer Errors
2. SQL Injection
3. Information Leak
4. Configuration
5. Credentials
6. Link Following
7. Race Conditions
8. Cross-Site Scripting
9. Input Validation
10. Resource Management
11. Numeric Errors
12. Crypto
13. OS Command Injection
14. Access Control
15. Code Injection
16. Path Traversal
17. Authentication
18. CSRF
19. Format String

Figure 1 shows the subtree of the CWE tree hierarchy that represents vulnerabilities in the source code. As can be seen, there are four main CWE categories: (1) indicator of poor code quality, (2) data handling, (3) security features, and (4) time and state.

Data for 2013–2017

We took the data from the NVD for the years 2013 to 2017 and recategorised the 32 CWEs that have been used since July 2016 into the 19 CWE NVD categories of 2007 because the 12 new CWEs are all inner nodes of the CWE tree hierarchy. As such, they represent a subcategory of vulnerabilities rather than a vulnerability type *per se*. There are 16 leaves that are part of the 2007 vulnerability types, with two others ("input validation", and "path traversal") that are inner nodes of the subtree, and one other ("configuration") that is not part of the source code CWE subtree hierarchy.

The summary data from NVD for the years 2013–2017, categorised into 19 categories after removing the entries "Not Enough Information" and "Other", is as follows:

```
1- CWE-398: Indicator of poor code quality
 CWE-399: Resource management errors
2- CWE-19: Data handling
 CWE-20: Input validation
 CWE-119: Buffer errors
 CWE-134: Format string vulnerability
 CWE-74: Injection
 CWE-77: Command injection
 CWE-78: OS command injection
 CWE-89: SQL injection
 CWE-79: XSS
 CWE-94: Code injection
 CWE-21: Path equivalence
 CWE-22: Path traversal
 CWE-59: Link following
 CWE-189: Numeric errors
 CWE-129: Improper validation of array index
 CWE-190: Integer overflow or wraparound
 CWE-682: Incorrect calculation
 CWE-190: Integer overflow or wraparound
 CWE-191: Integer underflow
 CWE-369: Divide by zero
 CWE-199: Information management errors
 CWE-200: Information Leak
3- CWE-254: Security features
 CWE-287: Authentication issues
 CWE-345: Insufficient verification of data authenticity
 CWE-352: CSRF
 CWE-255: Credentials management
 CWE-264: Permissions, privileges and access control
 CWE-284: Improper access control
 CWE-287: Authentication issues
 CWE-310: Cryptographic issues
4- CWE-361: Time and state
 CWE-362: Race conditions
```

Figure 1 Subtree of the CWE tree hierarchy that focuses on source code.

Vulnerability Type	Number of Unique Exploits
Buffer errors	5899
Permissions, privileges and access control	3951
Cross-site scripting	3914
Information leak	3106
Input validation	2618
Cryptographic issues	1873
Resource management	1422
SQL injection	1299
Cross-site request forgery	1062
Path traversal	690
Authentication	485
Code injection	450
Numeric errors	420
Credentials management	370
Race conditions	256
OS command injection	188
Link following	112
Configuration	38
Format string	29

We aggregated the data from all forms of injection attacks (namely, “cross-site scripting”, “SQL injection”, “code injection”, and “OS command injection”) into a single category, and then considered the resulting top four vulnerability categories.

Vulnerability Category	Count
Buffer errors	5899
Injections	5851
Permissions, privileges and access control	3951
Information leak	3106

It is important to note that these top four categories represent **64%** of all labelled data for the five years.

The categories “buffer errors”, “injections”, and “information leak” represent categories of vulnerabilities that could be prevented through first-class language support. However, the category “Permissions, privileges and access control” represents vulnerabilities in the implementation of a security solution (whether in the implementation of the language solution or through own implementation of a solution in the source code). It remains future work to establish to what degree language-level support could alleviate vulnerabilities in this category. Accordingly, we have focused our attention in this paper on the other three categories, which still represent **53%** of all labelled data in the NVD for the recent five years of 2013–2017.

From Theory to Systems: A Grounded Approach to Programming Language Education

Will Crichton
Stanford University, CA, USA
wcrichto@cs.stanford.edu

Abstract

I present a new approach to teaching a graduate-level programming languages course focused on using systems programming ideas and languages like WebAssembly and Rust to motivate PL theory. Drawing on students' prior experience with low-level languages, the course shows how type systems and PL theory are used to avoid tricky real-world errors that students encounter in practice. I reflect on the curricular design and lessons learned from two years of teaching at Stanford, showing that integrating systems ideas can provide students a more grounded and enjoyable education in programming languages. The curriculum, course notes, and assignments are freely available: `http://cs242.stanford.edu/f18/`

2012 ACM Subject Classification Social and professional topics → Computing education

Keywords and phrases programming languages, programming language education

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.4

Acknowledgements I am eternally grateful to my advisor, Pat Hanrahan, for his support and feedback through the years. His positivity, encouragement, and wisdom continue to push me and this course to ever greater heights.

© Will Crichton;
licensed under Creative Commons License CC-BY

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 4; pp. 4:1–4:9

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A deep divide exists today between courses on programming languages across academia. Some PL courses focus on theory and formal methods, such as CMU [1], UPenn [2], and Princeton [3]. Some combine PL and compilers to cover the implementation of grammars, typecheckers, and interpreters like Northeastern [9], UC Berkeley [4], and UIUC [5]. Others yet focus on teaching existing paradigms of programming languages like functional and logic programming at places like UW [7] and (formerly) Stanford [10]. (And others still blaze new trails, like Brown [17].)

While I strongly believe in the educational value of programming language theory, I worry that the theory-oriented PL courses (e.g. courses using [16, 14]) end up only interesting the math-oriented programmers that appreciate the intrinsic beauty of concise programming models and proofs. Anecdotally, my peers and I experienced this during undergrad at CMU. I heard sentiments about PL theory like, "I get it, it's cool, but I'm not designing functional programming languages or doing PL research, so it's not for me."

By contrast, while implementation-oriented or "programming language zoo" courses are looked down upon by theorists, these courses are undeniably *fun* in ways theory courses often are not. The journey of discovering new languages, the excitement of exploring unfamiliar paradigms – for me, these moments, arcs, and emotions carried my passion for programming languages through the years moreso than abstract theory. Motivating students to continue learning and using what we teach is just as important as the content itself.

Over the last two years, I redesigned Stanford's graduate PL course, CS 242, in an attempt to bridge this divide. I developed a novel curriculum that integrates modern systems programming languages like WebAssembly and Rust to contextualize and motivate discussions of functional programming, type theory, and operational semantics. In this paper, I will discuss the key design decisions of this curriculum and my experience teaching it to 70 Stanford CS students.

2 Pedagogy

The target audience for this course is the average Stanford senior undergraduate or master's student. This student has completed the computer science core (data structures, computer systems, algorithms, discrete mathematics, etc.) and has no formal exposure to functional programming (CS 242 is the only Stanford course that teaches FP). They probably intend to become a software engineer at a large tech company after graduation, or perhaps already are – Google and Facebook engineers have taken CS 242 through job training programs. This student does not necessarily come predisposed with a love of mathematical theory, of functional programming, or of programming languages at all.

For such a student, I believe the primary benefit of learning PL theory is to provide a simple model of fundamental concepts in computation. I call this "computational literacy," i.e. not just knowing how to program, but understanding the essence of ideas like variables, functions, and types. From simplicity comes two key benefits:

1. Improving mental models of computation: most students learn programming ad hoc through courses, extracurriculars, and internships. These experiences provide a working knowledge of computational concepts through the lens of existing programming tools, but such concepts often carry significant baggage due to reasons of poor design, legacy code, or limitations of the underlying hardware. For example, a student who learns C assumes a

mental model of functions that involves multiple/variadic arguments, void returns, top-level vs. nested functions, recursion, control flow through early returns, the call stack, and so on. By contrast, a student who learns the lambda calculus can understand functions solely as abstraction of code over a single variable (along with variable scope). I strongly believe that providing students a clear mental model of computational concepts should be a core mission of not just a PL course, but of a broader CS curriculum. I suspect that these concepts are essential in acquisition and transfer of computational skills like building new algorithms or learning new programming languages, although our current understanding of the programmer's psyche is perilously insufficient to fully justify this claim.

2. Enabling formal reasoning: when a computational model is simple (has few rules, syntactic constructs, edge cases, etc.), then the model becomes significantly easier to analyze for correctness, computational complexity, and many other such properties. For students both using and potentially designing computational models, simplicity becomes a criterion for understanding and evaluating the amenability of a model to analysis. Moreover, students can gain exposure to a wider variety of program analysis techniques than just those traditionally applied to programming languages. Existential types, ownership types, session types, refinement types, etc. all serve to expand a student's idea of the realm of possibilities in language design.

My goal for CS 242 is to demonstrate how programming language theory provides a simpler view on computational concepts, and to contextualize the benefits of simplicity against problems familiar to students. All CS students at Stanford are required to take an introduction to computer systems course, i.e. assembly, C, pointers and memory, bit manipulation, and so on. This provides an opportunity to ground the theoretical framework of PL in their concrete experiences writing low-level programs, showing that PL theory could solve problems they empathize with. What if a static type system could prevent you from ever getting a segfault? What if assembly actually had a type system? Answering these questions both provides concrete applications of PL theory in practice, but also motivates students by addressing problems they understand viscerally: just ask any student how long they've spent staring at x86 or debugging a segfault.

3 Curriculum

CS 242 has seven arcs across eight weeks of core content, roughly 50/50 between theory and systems. The theory arcs are largely derived from a subset of *Types and Programming Languages* [16], while the systems arcs are entirely my own invention. Students are expected to do 12-15 hours of work per week including lectures and weekly assignments.

3.1 Logic and semantics

We first establish the logical system of judgments and inference rules, followed by a presentation of the untyped lambda calculus and its operational semantics. The assignment, like many in this course, is structured around introducing new concepts and getting students to transfer knowledge from lecture to new contexts. This is as opposed to, say, just implementing an interpreter for the abstractions defined in class. Rather than teach students the Church encoding in class, we provide an untyped lambda calculus interpreter with OCaml-ish syntax and have students build the encoding themselves. This requires students to transfer an understanding of the lambda calculus execution model from the simple examples shown in class to the more abstract ones in the Church encoding.

Similarly, to test the student's ability to interpret semantic rules, we introduce the semantics for a dynamically-scoped lambda calculus and ask students to complete a proof of an execution trace and provide a semantic rule for a new syntactic construct. This both prepares students to understand proof contexts used later for typing judgments, and it also demonstrates how to reformulate language semantics under a different design decision, i.e. dynamic vs. lexical scoping.

3.2 Type theory

We next move to the basics of static semantics and the typed lambda calculus. We discuss structural induction along with progress and preservation, both to highlight a topic of intrinsic interest (demonstrating the safety of a language), but also to show a first example of proving a complex property about a language. Given the limited course time, we do not discuss proofs of totality or other such properties. Because functional programming is not a course prerequisite, we simultaneously introduce OCaml, framing the language as a featureful typed lambda calculus interpreter.

For the assignment, students implement an interpreter for the typed lambda calculus with algebraic data types in OCaml, learning both languages at the same time. This more standard exercise is supplemented with a language design challenge: the student is given the static and dynamic semantics for two proposed language extensions (let bindings and induction over naturals). One violates progress or preservation and the other does not, and the student must figure out which is which and provide the corresponding proof/counterexample. From my interactions with students, this kind of written exercise is a necessary complement to the interpreter, since it's very possible for students to successfully write the OCaml program without fully internalizing how the lambda calculus actually works.

3.3 Functional programming

We then discuss three important areas in type systems: algebraic data types, recursion, and parametric polymorphism. On each topic, I give a practical introduction as one might find in an introduction to functional programming course, followed by a discussion of formal semantics in the typed lambda calculus. For example, when teaching ADTs, I will give examples of how sum types in OCaml better represent error conditions than error codes and null pointers, followed by a presentation of their static and dynamic semantics in the typed lambda calculus along with a progress/preservation proof.

For this assignment, students implement a polymorphic collections library in OCaml and translate their implementation by hand into typed lambda calculus. The coding portion is of the kind one would find in an introduction to functional programming class, the goal being to have students write functional programs of moderate complexity that aren't interpreters. The written portion is part of a broader strategy of learning by analogy, i.e. students are more likely to recognize the similarities of the OCaml and lambda calculus representations if asked to explicitly translate between the two. Similarly in lecture, if I show a new functional programming construct, I will consistently translate back to a familiar language like C, Java, or Python to help students relate to languages and problems they know.

3.4 WebAssembly

This is the major divergence point with existing curricula. We start our shift into systems by asking: now that we understand the basics of a mathematical framework for programming languages, how can we use this knowledge to improve the design of real-world languages?

WebAssembly is a perfect example, as it is a popular, cutting-edge language for high performance web development, and its authors have presented a formal semantics [13] and a proof of progress and preservation [19] for the language. WebAssembly introduces the notion that even assembly languages don't have to be unsafe, and could support a type system with careful language design (e.g. no jumps). I use its semantics as a motivating example to introduce formal representations of mutability and non-local control flow. The actual semantics are too complicated to be feasibly taught in a single lecture, so I instead designed a distilled version of the language, e.g. separating the instruction sequence from the value stack and eliminating block contexts for branch semantics [11].

The assignment has students implement a memory allocator using plain WebAssembly and formulate semantics for adding exceptions into the language. The allocator uses a simple implicit free list that all students are familiar with from their introduction to systems course, but this time recast into WebAssembly instead of C. This helps students internalize WebAssembly semantics and identify the key differences with x86 and C like structured branching. The written portion asks students to provide operational semantics for a basic exception mechanism given an English specification. This task both provides students an opportunity to design semantics for a nontrivial language feature, and also highlights the difference between static/dynamic control flow with WebAssembly's branches versus the proposed exceptions.

3.5 Rust

While we use WebAssembly to demonstrate a direct application of PL theory, we introduce Rust to demonstrate more broadly the utility of type systems in low-level programming. We focus primarily on using the borrow checker (i.e. affine and ownership types) for resource management, especially with respect to memory safety and concurrency. I also emphasize the trait system for understanding a more compositional approach to modular programming than the traditional inheritance-based object-oriented style. I do not use formal semantics to teach either topic, as that would take up too much time. My goal is just to convince students that tricky systems problems like avoiding segfaults and data races can be categorically eliminated through static analysis combined with careful language design.

This arc covers two assignments: first, students implement a WebAssembly interpreter in Rust to learn Rust basics and reinforce knowledge of WebAssembly semantics. The WebAssembly semantics are significantly more complex than the typed lambda calculus, having local mutable variables, a function call stack, and an addressable memory. Students experience the contrast between implementing a simple interpreter in OCaml versus a complex interpreter in Rust.

Second, students implement a combinator-based futures library [8] to explore the interaction of memory management, concurrency, and traits in a DSL. Most systems concepts like concurrency and memory management are taught in languages with impoverished type systems like C, which shapes the student's mental model around ugly interfaces like pthreads and epoll. This assignment shows how a proper type system provides the foundation for expressing domain concepts with better abstractions, e.g. using parametric polymorphism instead of `void*` for generic thread callbacks, or using sum types and pattern matching to manage communication between threads instead of tagged unions or `#define`.

3.6 Session types

In the same way UW's "Hack Your Language" course [6] promotes an ethos where anyone can implement an embedded DSL, I want to convey the same lesson but for domain-specific type theories. To do this by example, we introduce two-party session types [18] along with their standard theory (e.g. computing duals), and then walk through an implementation of session types in Rust [15]. Session types pull together many of the course's lessons: articulating a type theory using mathematical notation, translating inference rules into code with traits, avoiding dangerous aliasing with affine types, implementing lexically scoped variables with de Bruijn indices, and defining recursion through fix points. We also discuss more general notions of typestate, such as statically verified finite state machines.

Students then design and build a session-typed (simplified) TCP implementation in Rust. We provide a high-level English specification and an example communication trace with sufficient detail to ensure that, although the students design the session type themselves, there is only one possible type they can correctly derive. The student's implementation contains the session type and an implementation of the server and client, which we test by compiling an adversarial server/client against their code, checking that the session types match, and then verifying the correctness properties not captured in the type.

3.7 Dynamic typing

In the course's final arc, I conclude by asking: static types are great, but what do we lose by having them in our language? We review many of the language constructs in the course, e.g. polymorphic types, exceptions, etc., and consider how dynamically-typed languages allow programmers to get these features "for free" while statically-typed languages require significant machinery in the compiler. I show how dynamically-typed languages make it easy to implement new language features. We specifically look at building a multiple-inheritance object-oriented programming system from scratch within Lua using prototypes.

For their final two-week-long assignment, students implement a variant of this OOP system, build a small ASCII-art adventure game on top of it, then rewrite core components in Rust using Lua's C API. In the first section, students implement a class-based (as opposed to prototype-based) OOP system so students can draw parallels to the litany of class-based OOP languages they know. In the second section, we provide students most of the infrastructure for running the game, except that it relies on their class library (thus providing a substantial integration test). We have students implement a simple AI by introducing coroutines to represent state machines. Finally, the game's vector library is rewritten in Rust to show an example of how Lua's API enables the integration of statically typed, GC-free code with dynamically typed, garbage-collected code.

3.8 Final project

CS 242 does not have a midterm – the majority of its grade comes from the eight assignments described above. I am still in the process of determining the the best way to do a final. In the first iteration, students did final projects for three weeks. Students enjoyed the opportunity to do a deep dive on a PL-related subject of personal interest, and two-thirds said they preferred doing a project over an exam. However, the prompt was too open-ended and the three weeks required cut too much into class time that would otherwise be used for more directed assignments.

On the latest iteration, I had students learn to use the Lean theorem prover [12] on their own and prove a number of simple theorems (i.e. a take-home final, still not an exam). The goal of the final was to show that learning about PL theory and functional programming

would make students more productive in settings of self-directed learning that involved foreign PL concepts, such as grappling with dependent types and the Curry-Howard correspondence. While students were able to effectively pick up the core concepts just by reading the Lean documentation (a reflection perhaps on the high quality of the Lean documentation!), several students provided feedback that the final felt disjointed from the course. In future iterations, I plan to make the final more systems-related while still preserving the same themes, perhaps by learning and using TLA+ to verify an actual distributed system instead of arbitrary logic theorems.

4 Outcomes

I taught this version of the curriculum in Autumn 2018 to a class of 77 Stanford students, mostly computer science majors, and mostly masters and upper-level undergraduate students. 85% of the students provided anonymized course feedback, mostly numeric with a subset of students leaving comments.

Students enjoyed the course and felt they learned a good amount of material. Students were asked "How much did you learn from this course?" with responses of "A great deal" (5), "A lot" (4), "A moderate amount" (3), "A little" (2), "Nothing" (1). The response average was 4.3 with 87% of students answering 4 or above. When prompted to articulate specific skills learned, most respondents wrote about: 1) PL theory and proofs about programs, 2) functional programming and new models of computation, or 3) learning to quickly acquire new languages. I believe this variety of responses reflects positively on the curriculum – different students enter with different personal learning goals, and the course's breadth allows students to find the aspect of programming languages they enjoy most.

The survey also prompted students with "What would you like to say about this course to a student who is considering taking it in the future?" I reproduce a few responses below.

> [CS 242] reflects on the translation that occurs between the model of computation you have in your head and how you express that model of computation using the tools a programming language gives you.
>
> It's still not really clear to me how the Lambda Calculus practically relates to the rest of the course, so the first few weeks were a bit frustrating, but I found it more enjoyable after that.
>
> It was not what I expected and was a lot of work (office hours were a blessing) but it introduced me (thoroughly!) to functional programming concepts that I am now looking into further on my own because I am interested in different ways of thinking about computation and how that will affect the way I approach programming problems in the future.
>
> One of my favorite classes at Stanford (my 4th year for context) and has really opened my eyes to the world of PL and how I actually really enjoy it and might be looking into pursuing research or further education in PL related work.
>
> This is one of the best CS classes I've taken at Stanford–if not hands down the best. Lectures are always so exciting, and the homework is a ton of fun. I really like the mix of theory and systems-level stuff, as well as the mix of programming and written assignments, where both feel like they are valuable and adding to the other. I also feel like I'm being challenged to think in different ways than I do in most CS classes (mostly because of the functional paradigm).

I firmly believe that students learn most from assignments. Many students enjoyed the course likely because I put significant effort into developing assignments that didn't just reinforce the course content, but did so in an exciting way. This is my biggest critique of the definitional interpreters approach to PL education: while it showcases the content, it gets boring quickly. Implementing TCP, an adventure game, a memory allocator – this variety keeps students engaged while they pick up new PL concepts. Moreover, the focus on real-world systems promotes creativity in assignments, as you naturally get access to a wider range of libraries and applications than when using niche teaching languages.

5 Discussion

Overall, I consider the focus on bridging theory and systems to be a major success. Students leave the course with different takeaways on the most important material, but almost everyone leaves understanding something about the role and importance of programming language theory in practice. Going forward, I will continue working to draw explicit connections between the different parts of the course – as one comment mentioned, some students are still struggling to see the relationships between the abstract ideas of the lambda calculus and the everyday concerns of systems programming.

In particular, I would like to refine the lecture and assignments to reflect the role of PL theory in the process of designing computational systems. Operational semantics and the like shouldn't just be seen as verification tools for functional-esque programming languages, but instead as a design tool for any system with a compositional model. Proofs like progress and preservation aren't check boxes on the way to language safety, but part of reasoning about unforeseen edge cases. Put another way: I believe we should seek to integrate lessons from the theorem-prover-oriented PL courses (e.g. Princeton [3], UPenn [2]) where possible.

Lastly, if any instructors would like to use part or all of this curriculum in their own courses, I have made all course materials are freely available. The course website (`http://cs242.stanford.edu/f18/`) contains a mini-textbook of typeset notes for each lecture. Starter code for all assignments is open-source on GitHub (`https://github.com/stanford-cs242/f18-assignments`).

References

1 15-312 Principles of Programming Languages. URL: `https://www.cs.cmu.edu/~rwh/courses/ppl/`.

2 CIS 500: Software Foundations. URL: `https://www.seas.upenn.edu/~cis500/current/index.html`.

3 COS 510: Programming Languages. URL: `https://www.cs.princeton.edu/courses/archive/spring18/cos510/`.

4 CS 164: Programming Languages and Compilers. URL: `http://www-inst.eecs.berkeley.edu/~cs164/sp18/`.

5 CS 421: Programming Languages and Compilers. URL: `https://courses.engr.illinois.edu/cs421/fa2018/CS421D/lectures/index.html`.

6 CSE 401: Hack Your Language! URL: `https://courses.cs.washington.edu/courses/cse401/16wi/index.html`.

7 CSE341: Programming Languages, Autumn 2018. URL: `https://courses.cs.washington.edu/courses/cse341/18au/`.

8 futures-rs. URL: `https://github.com/rust-lang-nursery/futures-rs`.

9 Programming Languages. URL: `https://pl.barzilay.org/syllabus.html`.

10 Programming Languages. URL: `https://suclass.stanford.edu/courses/course-v1:Engineering+CS242+Fall2016/about`.

11 WebAssembly Semantics. URL: `http://cs242.stanford.edu/f18/lectures/04-2-webassembly-theory.html`.

12 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

13 Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Notices*, volume 52 (6), pages 185–200. ACM, 2017.

14 Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

15 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, pages 13–22. ACM, 2015.

16 Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002.

17 Justin Pombrio, Shriram Krishnamurthi, and Kathi Fisler. Teaching Programming Languages by Experimental and Adversarial Thinking. *2nd Summit on Advances in Programming Languages*, page 15, 2017.

18 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *International Conference on Parallel Architectures and Languages Europe*, pages 398–413. Springer, 1994.

19 Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 53–65. ACM, 2018.

From Macros to DSLs: The Evolution of Racket

Ryan Culpepper
PLT
ryanc@racket-lang.org

Matthias Felleisen
PLT
matthias@racket-lang.org

Matthew Flatt
PLT
mflatt@racket-lang.org

Shriram Krishnamurthi
PLT
sk@racket-lang.org

Abstract

The Racket language promotes a language-oriented style of programming. Developers create many domain-specific languages, write programs in them, and compose these programs via Racket code. This style of programming can work only if creating and composing little languages is simple and effective. While Racket's Lisp heritage might suggest that macros suffice, its design team discovered significant shortcomings and had to improve them in many ways. This paper presents the evolution of Racket's macro system, including a false start, and assesses its current state.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases design principles, macros systems, domain-specific languages

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.5

Funding Over 20 years, this work was partially supported by our host institutions (Brown University, Northeastern University, Prague Technical University, and University of Utah) as well as several funding organizations (AFOSR, Cisco, DARPA, Microsoft, Mozilla, NSA, and NSF).

Acknowledgements The authors thank Michael Ballantyne, Eli Barzilay, Stephen Chang, Robby Findler, Alex Knauth, Alexis King, and Sam Tobin-Hochstadt for contributing at various stages to the evolution of Racket's macro system and how it supports language-oriented programming. They also gratefully acknowledge the suggestions of the anonymous SNAPL '19 reviewers, Sam Caldwell, Will Clinger, Ben Greenman and Mitch Wand for improving the presentation.

© Ryan Culpepper, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 5; pp. 5:1–5:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Macros and Domain-Specific Languages

The Racket manifesto [20, 21] argues for a *language-oriented programming* (LOP) [13, 45] approach to software development. The idea is to take Hudak's slogan of "languages [as] the ultimate abstractions" [33] seriously and to program with domain-specific languages (DSLs) as if they were proper abstractions within the chosen language. As with all kinds of abstractions, programmers wish to create DSLs, write programs in them, embed these programs in code of the underlying language, and have them communicate with each other.

According to the Lisp worldview, a language with macros supports this vision particularly well. Using macros, programmers can tailor the language to a domain. Because programs in such tailored languages sit within host programs, they can easily communicate with the host and each other. In short, creating, using, and composing DSLs looks easy.

Macros alone do not make DSLs, however, a lesson that the Racket team has learned over 20 years of working on a realization of language-oriented programming. This paper recounts Racket's history of linguistic mechanisms designed to support language-oriented programming; it formulates desiderata for DSL support based on, and refined by, the Racket team's experiences; it also assesses how well the desiderata are met by Racket's current capabilities. The history begins with Scheme's hygienic macros, which, in turn, derive from Lisp (see sec. 2). After a false start (see sec. 4), the Racket designers switched to procedural, hygienic macros and made them work across modules; they also strictly separated expansion time from run time (see sec. 5). Eventually, they created a meta-DSL for writing macros that could properly express the grammatical constraints of an extension, check them, and synthesize proper error messages (see sec. 6). A comparison between general DSL implementation desiderata (see sec. 3) and Racket's capabilities shows that the language's support for a certain class of DSLs still falls short in several ways (see sec. 7).

▶ Note. This paper does *not* address the safety issues of language-oriented programming. As a similar set of authors explained in the Racket Manifesto [20], language-oriented programming means programmers use a host language to safely compose code from many small pieces written in many different DSLs and use the very same host language to implement the DSLs themselves. Hence language-oriented programming clearly needs the tools to link DSLs safely (e.g., via contracts [22] or types [41]) and to incorporate systems-level protection features (e.g., sandboxes and resource custodians [30]).

Some Hints For Beginners on Reading Code

We use Racket-y constructs (e.g., `define-syntax-rule`) to illustrate Lisp and Scheme macros. Readers familiar with the original languages should be able to reconstruct the original ideas; beginners can experiment with the examples in Racket.

Lisp's S-expression construction		Racket's code construction	
`'`	S-expression quote	`#'`	code quote
`` ` ``	Quine quasiquote	``#` ``	code quasiquote
`,`	Quine unquote	`#,`	code unquote
`@,`	list splicing	`#@,`	code splicing

Figure 1 Hints on strange symbols.

For operations on S-expressions, i.e., the nested and heterogeneous lists that represent syntax trees, Lisp uses `car` for `first`, `cdr` for `rest`, and `cadr` for `second`. For the convenient construction of S-expressions, Lisp comes with an implementation of Quine's quasiquotation

idea that uses the symbols shown on the left of fig. 1 as short-hands: quote, quasiquote, unquote, and unquote-splicing. By contrast, Racket introduces a parallel data structure (syntax objects). To distinguish between the two constructions, the short-hand names are prefixed with #.

2 The Lisp and Scheme Pre-history

Lisp has supported macros since 1963 [32], and Scheme inherited them from Lisp in 1975 [43]. Roughly speaking, a Lisp or Scheme implementation uses a *reader* to turn the sequence of characters into a concrete tree representation: *S-expressions*. It then applies an *expander* to obtain the abstract syntax tree(s) (AST). The expander traverses the S-expression to find and eliminate uses of macros. A macro rewrites one S-expression into a different one, which the expander traverses afresh. Once the expander discovers a node with a syntax constructor from the core language, say `lambda`, it descends into its branches and recursively expands those. The process ends when the entire S-expression is made of core constructors.

A bit more technically, macros are functions of type

```
S-expression ⟶ S-expression
```

The `define-macro` form defines macros, which are written using operators on S-expressions. Each such definition adds a macro function to a table of macros. The expander maps an S-expression together with this table to an intermediate abstract syntax representation:

```
S-expression × TableOf[MacroId, (S-expression ⟶ S-expression)] ⟶ AST
```

The `AST` is an internal representation of an S-expression of only core constructors.

See the left-hand side of fig. 2 for the simple example of a `let` macro. As the comments above the code say, the `let` macro extends Racket with a block-structure construct for local definitions. The macro implementation assumes that it is given an S-expression of a certain shape. Once the definition of the `let` macro is recognized, the expander adds the symbol `'let` together with the specified transformer function to the macro table. Every time the macro expander encounters an S-expression whose head symbol is `'let`, it retrieves the macro transformer and calls it on the S-expression. The function deconstructs the given S-expression into four pieces: `decl`, `lhs`, `rhs`, `body`. From these, it constructs an S-expression that represents the immediate application of a `lambda` function.

```
;; PURPOSE extend Racket with block-oriented, local bindings
;;
;; ASSUME the given S-expression has the shape
;;  (let ((lhs rhs) ...) body ...)
;; FURTHERMORE ASSUME:
;; (1) lhs ... is a sequence of distinct identifiers
;; (2) rhs ..., body ... are expressions
;; PRODUCE
;;  ((lambda (lhs ...) body ...) rhs ...)
```

```
(define-macro (let e)
  (define decl (cadr e))
  (define lhs  (map car decl))
  (define rhs  (map cadr decl))
  (define body (cddr e))
  ;; return
  `((lambda ,lhs ,@body) ,@rhs))
```

```
(define-syntax-rule
  (let ((lhs rhs) ...) body ...)
  ;; rewrites above pattern to template below
  ((lambda (lhs ...) body ...) rhs ...))
```

Figure 2 Macros articulated in plain Lisp vs Kohlbecker's macro DSL.

Macros greatly enhance the power of Lisp, but their formulation as functions on S-expressions is both error-prone and inconvenient. As fig. 2 shows, the creator of the function makes certain assumption about the shape of the given S-expression, which are not guaranteed by the macro expansion process. Yet even writing just a transformation from the assumed shape of the S-expression to the properly shaped output requires bureaucratic programming patterns, something a macro author must manage and easily causes omissions and oversights.

For concreteness, consider the following problems in the context of `let` macro:

1. The S-expression could be an improper list. The transformer, as written, does not notice such a problem, meaning the compilation process ignores this violation of the implied grammar of the language extension.
2. The S-expression could be too short. Its second part might not be a list. If it is a list, it may contain an S-expression without a `cadr` field. In these cases, the macro transformer raises an exception and the compilation process is aborted.
3. The S-expression has the correct length but its second part may contain lists that contain too many S-expressions. Once again, the macro transformer ignores this problem.
4. The S-expression may come with something other than an identifier as the `lhs` part of a local declaration. Or, it may repeat the same identifier as an `lhs` part of the second clause. In this case, the macro generates code anyways, relying on the rest of the compilation process to discover the problems. When these problems are discovered,
   a. it may have become impossible to report the error in terms of source code, meaning a programmer might not understand where to look for the syntax error.
   b. it has definitely become impossible to report errors in terms of the language extension, meaning a programmer might not comprehend the error message.
5. The author of the macro might forget the unquote `,` to the left of `lhs`. In many members of the Lisp family, the resulting code would be syntactically well formed but semantically rather different from the intended one. In particular, conventional Lisp would generate a function that binds all occurrences of `lhs` in `body` via this newly created `lambda` – a clear violation of the intended scoping arrangements expressed in the comments.

In short, if the S-expression fails to live up to the stated assumptions, the macro transformation may break, ignore code, or generate code that some later step in the compilation process recognizes as an error but describes in inappropriate terms. If the programmer makes even a small mistake, strange code may run and is likely to cause inexplicable run-time errors.

Kohlbecker's dissertation research on macros greatly improves this situation [35, 36, 37]. His macro system for Scheme 84 adds two elements to the macro writer's toolbox. The first is a DSL for articulating macro transformations as rewriting rules consisting of a pattern and a template. The revised macro expander matches S-expressions against the specified patterns; if there is a match, the template is instantiated with the resulting substitution. This DSL removes programming patterns from macro definitions and, to some extent, eliminates problems 1 through 3 from above. For an example, see the right-hand side of fig. 2.

▶ Note. We call Lisp-style macros *procedural* and Kohlbecker's approach *declarative*.

The second novel element is hygienic expansion. By default, Kohlbecker's macro expander assumes that identifiers contained in the source must be distinct from macro-generated identifiers in binding positions. As such, it eliminates the need to explicitly protect against accidental interference between the macro's lexical scopes and those of its use contexts – that is, yet another programming pattern from macro code. At a minimum, this hygienic expander would not bind `lhs` in `body` as indicated in problem 5 above.

Further work [2, 8, 15] refined and improved the pattern-oriented approach to specifying macros as well as hygienic macro expansion. The `define-syntax-rule` construct and hygienic expansion became part of the Scheme standard by the late 1990s [1]. Starting in 1988 and in parallel to the Scheme standardization process, Dybvig et al. [15] designed and implemented a macro definition construct, `define-syntax-cases` (in actual code it requires a combination of `define-syntax` and `syntax-case`) that merged the procedural and declarative elements of the Lisp world. Dybvig et al. also switched from S-expressions to trees of syntax objects. These trees included source locations so that the error handling code could try to point back to the surface code (problem 4a above).

Starting in the late 80s, researchers explored other ways to facilitate the work of macro authors, including two relevant to creating DSLs from macros. Dybvig et al. [14] invented expander-passing macros. Macro authors would write their own expanders and use different ones in different macros. At an abstract level, expansion-passing style anticipates the need for checking static attributes. Blume [3] as well as Kelsey and Reese [34] added modules that could export and import macros. Such modules allow macro programmers to encapsulate bundles of macros, a first step towards encapsulating a DSL's design and implementation.

3 DSLs Require More than Bunches of Macros

Scheme-style macros greatly improve on Lisp's as far as the *extension* of an existing language is concerned. A developer can add concise and lexically correct macros to a program and may immediately use them, for writing either ordinary run-time code or additional macros. This immediacy is powerful and enticing because a programmer never has to leave the familiar programming environment, use external tools, or mess with scaffolding setups.

The idea of macros is also easy to comprehend at the abstract level. Conceptually, a macro definition adds a new alternative to one of Racket's grammatical productions: definitions or expressions. The declarative approach makes it easy to specify simple S-expression transformers in a straightforward manner; hygienic macro expansion guarantees the integrity of the program's lexical scope.

The problem is that a language extension provides only a false sense of a purpose-tailored language. On one hand, a programmer who uses a bunch of macro-based language extensions as if it were a self-contained DSL must code with an extreme degree of self-discipline. On the other hand, the macro system fails to support some of the traditional advantages of using DSLs: catching mistakes in the parsing or type-checking stage; exploiting constraints to generate optimized code; or link with/target tailor-made run-time functions.

Conventionally, the creation of DSLs demands a pipeline of compiler passes:

1. a *parser*, based on explicit specification of a domain-specific *vocabulary* and a *grammar*, that reports errors at the DSL's source level;
2. a *static semantics*, because one goal of migrating from an application interface to a DSL is to enforce certain constraints statically;
3. a *code generation* and *optimization* pass, because another goal of introducing DSLs is to exploit the static or linguistic constraints for improved performance; and,
4. a *run-time system*, because (1) the host language may lack pieces of functionality or (2) the target language might be distinct from the host language.

Scheme macros *per se* do not support the creation of such pipelines or its proper encapsulation.

The Racket designers noticed some of these problems when they created their first teaching languages [18, 19]. In response, they launched two orthogonal efforts to support the development of DSLs via language-extension mechanisms with the explicit goal of retaining the ease of use of the latter:

- One concerned the encapsulation of DSLs and support for some traditional passes. This idea was to develop a module system that allows the export and import of macros and functions while also retaining a notion of separate compilation for modules.
- The other aimed at a mechanism for easily expressing a macro's assumptions about its input and synthesizing error messages at the appropriate level, i.e., the problems from sec. 2. The results would also help with implementing DSLs via modules.

While sec. 4 reports on an ambitious, and abandoned, attempt to address these problems all at once, secs. 5 and 6 describe the tools that Racket eventually provided to DSL designers and implementors.

4 Ambitious Beginnings

When the Racket designers discovered the shortcomings of a traditional Scheme macro system, they decided to address them with three innovations. First, they decided to move beyond the traditional S-expression representation of syntax and instead use a richly structured one (see sec. 4.1). Second, they realized that macros needed to work together to implement context-sensitive checks. To this end, they supplemented declarative macros with procedural *micros* that could deal with attributes of the expansion context (see sec. 4.2). Finally they decided to use modules as the containers of macro-based DSL implementations as well as the units of DSL use (see sec. 4.3).

4.1 From S-expressions to Syntax Objects

To track source locations across macro expansion, Racket – like Dybvig's Chez Scheme – introduced a *syntax object* representation of the surface code, abandoning the conventional S-expression representation. Roughly speaking, a syntax object resembles an S-expression with a structure wrapped around every node. At a minimum, this structure contains source locations of the various tokens in the syntax. Using this information, a macro expander can often pinpoint the source location of a syntax error, partially solving problem 4a from sec. 3.

4.2 The Vocabularies of Micros

Recall that a macro is a function on the syntax representation. Once this representation uses structures instead of just S-expressions, the signature of a macro has to be adapted:

```
Syntax-Object ⟶ Syntax-Object
```

Of course, this very signature says that macros cannot naturally express[1] communication channels concerning attributes of the expansion context.

Krishnamurthi et al.'s work [39] supplements macros with *micros* to solve this issue. Like `define-macro`, `define-micro` specifies a function that consumes the representation of a syntax. Additionally, it may absorb any number of `Attribute` values so that collections of micros can communicate contextual information to each other explicitly:

```
Syntax-Object ⟶ (Attribute ... ⟶ Output)
```

[1] A macro author could implement this form of communication via a protocol that encodes attributes as syntax objects. We consider an encoding unnatural and therefore use the phrase "naturally express."

As this signature shows, a micro also differs from a macro in that the result is some arbitrary type called `Output`. This type must be the same for all micros that collaborate but may differ from one collection of micros to another. For macro-like micros, `Output` would equal `Syntax-Object`. By contrast, for an embedded compiler `Output` would be `AST`, meaning the type of abstract syntax trees for the target language. This target language might be Racket, but it could also be something completely different, such as GPU assembly code. The Racket team did not explore this direction at the time.

As this explanation points out, micros for DSLs must be thought of as members of a collection. To make this notion concrete, Krishnamurthi et al. also introduce the notion of a *vocabulary*. Since collections of macros and micros determine the “words” and “sentence structure” of a DSL, a vocabulary represents the formal equivalent of a dictionary and grammar rules. The micros themselves transform “sentences” in an embedded language into meaningful – that is, executable – programs.

In Krishnamurthi et al.'s setting, a vocabulary is created with `(make-vocabulary)` and comes with two operations: `define-micro`, which adds a micro function to a given vocabulary, and `dispatch`, which applies a micro to an expression in the context of a specific vocabulary.

```
;; type Output = RacketAST

(define compiler (make-vocabulary))

_ _ _ elided _ _ _

(define-micro compiler
  (if cond then else)
  ==>
  (lambda ()
    (define (expd t)
      ((dispatch t compiler)))
    (define cond-ir (expd cond))
    (define then-ir (expd then))
    (define else-ir (expd else))
    (make-AST-if
      cond-ir then-ir else-ir)))

_ _ _ elided _ _ _

(define compiler-language
  (extend-vocabulary
    base-language
    compiler))
```

```
;; type Output = RacketType
(define type-check (make-vocabulary))

_ _ _ elided _ _ _

(define-micro type-check
  (if cond then else)
  ==>
  (lambda (Γ)
    ;; first block
    (define (tc t)
      ((dispatch t type-check) Γ))
    (define cond-type (tc cond))
    (unless (type-== cond-type Boolean)
      (error _ _ _ elided _ _ _))
    (define then-type (tc then))
    (define else-type (tc else))
    (unless (type-== then-type else-type)
      (error _ _ _ elided _ _ _))
    then-type))

_ _ _ elided _ _ _
```

Figure 3 Micros and vocabularies.

Fig. 3 collects illustrative excerpts from a vocabulary-micro code base. The left-hand column sets up a `compiler` vocabulary, which expresses transformations from the surface syntax into Racket's core language. Among other micros, the `if` micro is added to `compiler` because it is a core construct. The final definition shows how to construct a complete language implementation by mixing in vocabularies into the common base language.

Like Scheme's macro definitions, micro definitions use a pattern DSL for specifying inputs. As for the `Attribute ...` sequence, micros consume those via an explicit `lambda`. To create its output, the `if` micro allocates an AST node via `make-AST-if`. The pieces of this node are the results of expanding the three pieces that make up the original `if` expression. The expansions of these sub-expressions employ `dispatch`, a function that consumes the expression to be expanded together with the contextual vocabulary and the attributes (none here) in a staged fashion.

The right-hand side of fig. 3 shows how to add an `if` micro for a type-checking variant of the DSL. The code introduces a second vocabulary for the type checker. The `if` micro for this additional vocabulary implements the type checking rule for `if` in a standard manner, reporting an error as soon it is discovered.

Once the `type-check` vocabulary is in place, a developer can use it independently or in combination with the `compiler` vocabulary. For example, Racket's soft typing system [24] needed a distinct interpretation for the language's `letrec` construct, i.e., a distinct `type-check` vocabulary unrelated to the actual compiler. A variant of Typed Racket [44] could be implemented via the composition of these two vocabularies; in this context, the composition would discard the result of the pass based on the `type-check` vocabulary.

In general, DSL creators get two advantages from vocabularies and micros. First, they can now specify the syntax of their languages via explicit collections of micros. Each micro denotes a new production in the language's expression language, and the input patterns describe its shape. Second, they can naturally express and implement static checking. The micro's secondary arguments represent "inherited" attribute, and the flexible `Output` type allows the propagation of "synthesized" ones.

Implementing complete DSLs from vocabularies becomes similar to playing with Legos: (1) vocabularies are like mixins [31], (2) languages resemble classes, and (3) `dispatch` is basically a method invocation. Hence creating a variety of similar DSLs is often a game of linking a number of pieces from a box of language-building blocks. For the team's rapid production and modification of teaching languages in the mid 1990s, vocabularies were a critical first step.

4.3 Languages for Semantic Modules

According to sec. 3 the implementation of any language combines a compiler with a run-time system. This dictum also applies to DSLs, whether realized with macros or micros. Both translate source code to target code, which refers to run-time values (functions, objects, constants, and so on). Such run-time values often collaborate "via conspiracy," meaning their uses in target code satisfies logical statements – invariants that would not hold if all code had free access to these values. That is, the implementor of a DSL will almost certainly wish to hide these run-time values and even some of the auxiliary compile-time transformations. All of this suggests that macros, micros and vocabularies should go into a module, and such modules should make up a DSL implementation.

Conversely, the implementors of DSLs do not think of deploying individual constructs but complete languages. Indeed, conventional language implementors imagine that DSL programmers create self-contained programs. By contrast, Lispers think of their language extensions and imagine that DSL programmers may wish to escape into the underlying host language or even integrate constructs from different DSL-like extensions at the expression level. The question is whether a macro-micro based approach can move away from the "whole program" thinking of ordinary DSLs and realize a Lisp-ish approach of deploying languages for small units of code.

Krishnamurthi's dissertation [38] presents answers to these two questions and thus introduces the first full framework for a macro-oriented approach to language-oriented programming. It combines macros with the first-class modules of the 1990s Racket, dubbed units [29], where the latter becomes both the container for DSL implementations as well as the one for DSL deployment. Technically, these units have the shape of fig. 4. They are parameterized over a `Language` and link-time imports, and they export values in response.

```
(unit/lang Language
  (ImportIdentifier ...)
  (ExportIdentfier ...)
  Definitions-and-Expressions ...)
```

Figure 4 Language-parameterized, first-class units.

A DSL implementation is also just a `unit/lang` that combines macros, micros, and run-time values. It is not recognized as a valid `Language` until it is registered with a *language administrator*. The latter compiles `unit/lang` expressions separately to plain `units`. For this compilation, the language administrator expands all uses of macros and micros and then resolves all the names in the generated code – without exposing any of them to other code. In particular, the developer does not need to take any action, such as adding the names of run-time values to export specifications of `Language`s or to `unit/lang`s that use a `Language`. The result of a compilation is a collection of plain Racket `units`, and the Racket compiler turns this collection into a running program.

In principle, Krishnamurthi's `unit/lang` system addresses all four DSL criteria listed in sec. 3. The micro-vocabulary combination can enforce syntax constraints beyond what macros can do. They are designed to express static processing in several passes and explicitly accommodate target languages distinct from Racket. And, the implementations of DSLs as `unit/lang`s encapsulates the compiler component with a run-time component.

What this system fails to satisfy is the desire to synthesize DSL implementation techniques with Lisp's incremental language-extension approach. The main problem is that a programmer has to parameterize an entire unit over a complete language. It is impossible to selectively import individual macros and micros from a `unit/lang`, which is what Racket developers truly want from a modular macro system. After a few years of using plain `units`, the Racket team also realized that first-class units provided more expressive power than they usually needed, meaning the extra complexity of programming the linking process rarely ever paid off in the code base.

Additionally, the `unit/lang` system was a step too far on the social side. Racket – then called PLT Scheme – was firmly in the Scheme camp and, at the time, the Scheme community had developed and open-sourced a new syntax system [15] that quickly gained in popularity. This macro system combined the declarative form of Krishnamurthi's macros with the procedural form of his micros into a single `define-syntax-cases` form. Furthermore, this new macro system came with the same kind of syntax-object representation as Krishnamurthi's, allowing source tracking, hygienic expansion, and other cross-expansion communication. In other words, the new system seemed to come with all the positive aspects of Krishnamurthi's without its downsides. Hence, the Racket team decided to adapt this macro system and create a module system around it.

5 Languages From Syntactic Modules

The Racket designers started this rebuilding effort in 2000. The goal was to create a module system where a developer could write down each module in a DSL that fit the problem domain and where a module could export/import individual macros to/from other modules – and this second point forced them to reconsider the first-class nature of modules.

Flatt's "you want it when" module-macro system [25] realizes this goal. It introduces a module form, which at first glance looks like `unit/lang`. Like the latter, `module` explicitly specifies the language of a module body, as the grammar in fig. 5 shows. Otherwise the

```
(module Name Language

  {    ProvideSpecification
     | RequireSpecification
     | Definition
     | Expression }*
  )
```

```
#lang Language                                   Name.rkt

{    ProvideSpecification
   | RequireSpecification
   | Definition
   | Expression }*
```

Figure 5 Language-parameterized, first-order modules and their modern abbreviation.

grammar appears to introduce a new expression form whose internals consist of a sequence of exports, imports, definitions and expressions. A small difference concerns the organization of the module body. The import and export specifications no longer need to show up as the first element of the module; they can appear anywhere in the module. Appearances are deceiving, however, and the `Name` part suggests the key difference. A module is *not* an expression but a first-order form, known to the expander.

When the expander encounters `module`, it imports the `Language`'s provided identifiers. This step establishes the base syntax and semantics of the module's expressions, definitions, imports, and exports. Next the expander finds the imported and locally-defined macros in the module body. The search for imported macros calls for the expansion and compilation of the referenced modules. It is this step that requires the restriction to first-order modules, because the expander must be able to identify the sources of imported macros and retrieve their full meaning. Finally, the expander adds those imported and local macros to the language syntax and then expands the module body properly, delivering an abstract-syntax representation in the Racket core language.

One consequence of this arrangment is that the expansion of one module may demand the evaluation of an entire tower of modules. The first module may import and use a macro from a second module, whose definition relies on code that also uses language extensions. Hence, this second module must be compiled after expanding and compiling the module that supplies these auxiliary macros.

```
#lang racket                         loop.rkt

(provide inf-loop)

(define-syntax-cases
  [(inf-loop e)
   (begin
     (displayln "generating inf-loop")
     #'(do-it (lambda () e)))])

(define (do-it th)
  (th)
  (do-it th))
```

```
#lang racket                         use-loop.rkt

(provide display-infinitely-often)

(require "loop.rkt")

(define (display-infinitely-often x)
  (inf-loop (do-it x)))

(define (do-it x)
  (displayln x))
```

Figure 6 Exporting macros from, and importing them into, modules.

The right-hand side of fig. 5 also shows the modern, alternative syntax for modules. The first line of code specifies only the language of the module via a `#lang` specification; the name of the file (boxed) determines the name of the module. Fig. 6 illustrates how two modules interact at the syntax and run-time level. The module on the left defines the language extension `inf-loop`, whose implementation generates code with a reference to the function `do-it`. The module on the right imports this language extension via the `require` specification. The Racket compiler retrieves the macro during compile time and uses it to

expand the body of the `display-infinitely-often` function – including a reference to the `do-it` function in module `loop.rkt`. Cross-module hygienic expansion [25, 27] ensures that this macro-introduced name does not conflict in any way with the `do-it` function definition of the `use-loop.rkt` module. Conceptually, the expansion of `display-infinitely-often` looks like the following definition:

```
(define (display-infinitely-often x)
  (loop.rkt-do-it (lambda () (use-loop.rkt-do-it x))))
```

with the two distinct, fully-resolved names guaranteeing the proper functioning of the code according to the intuitive surface meaning.

Flatt's module-macro system allows the use of both declarative and procedural language extensions. To illustrate the latter kind, the `inf-loop` macro uses `define-syntax-cases`. If the expander can match a piece of syntax against one of the left-hand-side patterns of `define-syntax-cases`, it evaluates the expression on the right. This evaluation must produce code, which is often accomplished via the use of templates (introduced by `#'`) whose pattern variables are automatically replaced by matching pieces of syntax. But, as the definition of `inf-loop` suggests, the right-hand side may contain side-effecting expressions such as `displayln`. While these expressions do not become a part of the generated code as the above snippet shows, their side effects are observable during compile time.

To enable separate compilation, Racket discards the effects of the expansion phase before it moves on to running a module. Discarding such effects reflects the Racket designers' understanding that language-extensions are like compilers, which do not have to be implemented in the same language as the one that they compile and which are not run in the same phase as the program that they translate. Phase separation greatly facilitates reasoning about compilation, avoiding a lot of the pitfalls of Lisp's and Chez Scheme's explicit `eval-when` and `compile-when` instructions [5, 25, 42].

```
#lang racket                    math.rkt

(provide Ack)

;; Number Number -> Number
(define (Ack x y)
  (cond
    [(zero? x) (+ y 1)]
    [(and (> x 0) (zero? y))
     (Ack (- x 1) 1)]
    [else
     (Ack (- x 1) (Ack x (- y 1)))]))
```

```
#lang racket                    use-acker.rkt

(require (for-syntax "math.rkt"))

(define-syntax-cases ()
  [(static-Ack x y)
   ;; rewrites the pattern to a template
   ;; via some procedural processing
   (let* ((x-e (syntax-e #'x))
          (y-e (syntax-e #'y)))
     (unless (and (number? x-e) (number? y-e))
       (raise-syntax-error #f "not numbers"))
     (define ack (Ack x-e y-e))
     #`(printf "the Ack # is ~a" #,ack))])

(static-Ack 1 2)
```

Figure 7 Importing at a different phase.

Phase separation imposes some cost on developers, however. If a module needs run-time functions for the definition of a language construct, the import specification must explicitly request a phase shift. For an example, see fig. 7. The module on the right defines `static-Ack`, which computes the Ackermann function of two numbers at compile time. Since the Ackermann function belongs into a different library module, say `math`, the `use-ack` module most import it from there. But, because this function must be used at compile time, the `require` specification uses the (underlined) `for-syntax` annotation to shift the import to this early phase. The Racket designers' experience shows that phase-shifting annotations are still significantly easier to work with than Lisp's and Scheme's `expand-when` and `eval-when` annotations.

Like Krishnamurthi's `unit/langs`, Flatt's `modules` allow developers to write different components in different languages. In the case of `modules`, the `Language` position points to a module itself. The exports of this `Language` module determine the initial syntax and semantics of a client module.

In contrast to an ordinary `module`, a `Language module` must export certain macros, called interposition points; it may export others. An interposition point is a keyword that the macro expander adds to some forms during its traversal of the source tree. Here are the two most important ones for `Language` modules:

- `#%module-begin` is the (invisible) keyword that introduces the sequence of definitions and expressions in a module body. A `Language` module must export this form.
- `#%top-interaction` enables the read-eval-print loop for a `Language`, i.e., dynamic loading of files and interactive evaluation of expressions.

Other interposition points control different aspects of a `Language`'s meaning:

- `#%app` is inserted into function applications. In source code, an application has the shape `(fun arg ...)`, which expands to the intermediate form `(#%app fun arg ...)`.
- `#%datum` is wrapped around every literal constant.
- `#%top` is used to annotate module-level variable occurrences.

In practice, a developer creates a `Language` by adding features to a base language, subtracting others (by not exporting them), and re-interpreting some. Here "features" covers both macros and run-time values. The `#%module-begin` macro is commonly re-interpreted for a couple of reasons. Its re-definition often helps with the elimination of boilerplate code but also the communication of context-sensitive information from one source-level S-expression (including `modules`) to another during expansion.

```
#lang racket                                  lazy.rkt

(provide
 (except-out (all-from-out racket) #%app)
 (rename-out [lazy-app #%app]))

(define-syntax-rule
  (lazy-app f a ...)
  (#%app f (lambda () a) ...))
```

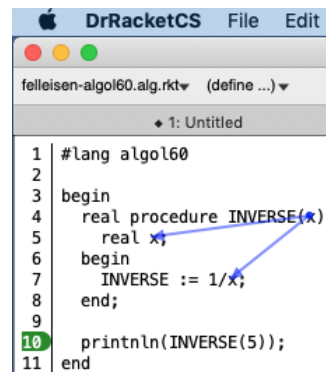
```
#lang "lazy.rkt"                     no-error.rkt

; a constant function
(define (f x y)
  10)

; called on two erroneous terms
(f (/ 1 0) (first '()))
; evaluates to 10
```

Figure 8 Building an embedded DSL from modules and macros.

Fig. 8 indicates how a developer could quickly build a language that looks like Racket but uses call-by-name instead of call-by-value. The module on the left is the language implementation. It starts from Racket and re-exports all of its features, including `#%module-begin`, except for function application. The module re-interprets function application via the second part of `provide`. Technically, a re-interpretation consists of a macro definition that is re-named in a `provide`. The `lazy` module comes with a `lazy-app` macro, which rewrites `(lazy-app fun arg ...)` to `(#%app fun (lambda () arg) ...)`. By static scope, the `#%app` in the expansion refers to the function application form of Racket. Since this macro is provided under the name `#%app`, a client module's function applications – into which the expander inserts `#%app` – eventually expand according to `lazy-app`. In particular, the two exception-raising expressions in the `no-error` module are wrapped in `lambda`; because `f` is a constant function that does not evaluate its arguments, these errors are never reported. (For additional details on `lazy`, see the last chapter of *Realm of Racket* [16].)

```
#lang racket                    all-in-one.rkt

(module lazy-impl racket

  (provide
   (except-out (all-from-out racket) #%app)
   (rename-out [lazy-app #%app]))

  (define-syntax-rule
    (lazy-app f a ...)
    (#%app f (lambda () a) ...)))

(module lazy-client (submod ".." lazy-impl)

  (define (f x y)
    10)

  (f (/ 1 0) (first '())))

(require (submod "." lazy-client))
```

(a) DSL development in one module.



(b) Algol 60 as a Racket DSL.

Figure 9 Developing and deploying DSLs in Racket.

Modules and macros jointly make DSL development an interactive activity in the Racket ecosystem. A programmer can open two tabs or windows in an IDE to use one for the DSL implementation and another for a DSL program. Or, a programmer can place a DSL-implementing submodule [26] and a DSL-using submodule into a single file, which can then be edited and executed within a single editor window of the preferred IDE. Fig. 9a shows how to combine the modules of fig. 8 into a single file. This program consists of three pieces. The first one is a submodule that implements the lazy language, while the second uses the first one in the `Language` position. Hence the first submodule is the programming language of the second. The last piece of the program requires and thus evaluates the client module. Any change to the first submodule is immediately visible in the second.

A developer may also equip a DSL with any desired syntax, not just build on top of Racket's beautiful parentheses. To support this kind of syntax, a `Language` module may export a new reader. Recall from sec. 2 that a Lisp reader turns the stream of characters into a sequence of S-expressions (or `Syntax-Objects`, in the case of Racket). The rest of the implementation can then use the usual mix of macros and functions. Butterick's *Beautiful Racket* [4] is a comprehensive introduction to this strategy and comes with a powerful library package for lexing and parsing.

In the context of modular macros, a developer may also create a conventional compiler with the macro infrastructure. Instead of just expanding to Racket, a DSL implementation may use a combination of macros and compile-time functions to perform conventional type checking or other context-sensitive checks.

Fig. 9b presents a simple example of a Racket DSL program in conventional syntax. Except for the first line, the code represents a standard Algol 60 program. The first line turns this program into a Racket DSL and thus allows Racket to parse, type check, compile, and run this program. Because the DSL implementation turns the Algol 60 program into syntax objects and implements its semantics via macro expansion, DrRacket (the Racket IDE [23]) automatically adapts itself to this new language. For example, fig. 9b illustrates how DrRacket connects the binding occurrence of `INVERSE`'s parameter to its bound ones.

In sum, Racket's modules simultaneously allow the incremental and interactive construction of language extensions as well as the construction of complete DSLs with their own vocabulary. The key design decision is to turn macros into entities that first-order modules can export, import, hide, and re-interpret. It does necessitate the introduction of strict phase separation between the expansion phase and run-time phase to obtain separate compilation.

6 Syntax Done Properly With Parsing Macros

The implementation of a DSL's syntax consists of two essential parts: parsing syntactically legitimate sentences, and reporting violations of the syntactic rules. Both aspects are equally important, but for 40 years, the macro community mostly neglected the second one.

Sec. 2 lists five problems with parsing via Lisp-style macros. Kohlbecker's rewriting DSL – based on patterns and templates – eliminates all of them except for problem 4. To appreciate the complexity of this particular problem, consider the actual grammatical production of `let` expressions in classical BNF notation:

$$\texttt{(let (\{[id expression]\}}^{*}\texttt{) expression}^{+}\texttt{)}$$

Kohlbecker's pattern-based meta-DSL addresses this context-free shape specification with the elegant trick of using ellipses (...) for * and unrolling for $^+$:

```
(let ([id expression] ...) expression expression ...)
```

What Kohlbecker's notation cannot express is the side condition of fig. 2:

```
id ... is a sequence of distinct identifiers
```

Indeed, Kohlbecker's notation cannot even specify that `id` must stand for an identifier.

So now imagine a programmer who writes

```
(let (((+ 1 2) x)) (* x 3))  ;; => ((lambda ((+ 1 2)) (* x 3)) x)
```

or

```
(let ((x 1) (x 2)) (* x 3))  ;; => ((lambda (x x) (* x 3)) 1 2)
```

In either case, a pattern-oriented language generates the `lambda` expression to the right of the `=>` arrow. Hence, the resulting syntax errors speak of `lambda` and parameters, concepts that the grammatical description of `let` never mentions. While a reader might be tempted to dismiss this particular error message as "obvious," it is imperative to keep in mind that this `let` expression might have been generated by the use of some other macro, which in turn might be the result of some macro-defining macro, and so on.

Dybvig's `define-syntax-cases` slightly improves on Kohlbecker's DSL. It allows the attachment of of *fenders* – Boolean expressions – to a macro's input patterns. With such fenders, a macro developer can manually formulate conditions that check such side conditions. Even in such simple cases as `let`, however, the error-checking code is many times the size of the rewriting specification. And this is why most macro authors fail to add this code or, if they do, fail to write comprehensive checks that also generates good error messages.

Culpepper's DSL for defining macros solves this problem with two innovations [9,10,11,12]. The first is an augmentation of the pattern-matching DSL with "words" for articulating classification constraints such as those of the `let` macro. The second is a DSL for specifying new classifications. Together, these innovations allow programmers to easily enforce assumptions about the surface syntax, synthesize error messages in terms of the specification, and deconstruct the inputs of a macro into relevant pieces.

```
(define-syntax-class distinct-bindings
 #:description "sequence of distinct binding pairs"
 (pattern (b:binding ...)
  #:fail-when (check-duplicate-id #'(b.lhs ..))
              "duplicate variable name"
  #:with (lhs* ...) #'(b.lhs ...)
  #:with (rhs* ...) #'(b.rhs ...)))
```

```
(define-syntax-class binding
 #:description "binding pair"
 (pattern (lhs:id rhs:expr)))
```

Figure 10 Syntax classifications.

Following our discussion above, the specification of `let` needs two syntax classifications: one to say that the second part of `let`'s input is a sequence and another one to say that the elements of this sequence are identifier-expression pairs. Fig. 10 shows how a programmer can define these classifications in Culpepper's meta-DSL. A classification must come with at least one `pattern` clause, which spells out the context-free shape of the form and names its pieces. For example, the `binding` class uses the pre-defined classifications `id` (for identifier) and `expr` (for expression) to say that a binding has the shape `(id expr)` and that the name of the `id` is `lhs` and the name of `expr` is `rhs`. Any use of such a syntax class, for example the one in the definition of `distinct-bindings`, may refer to these attributes of the input via a dot notation. Thus, `b.lhs ...` in `distinct-bindings` denotes the sequence of identifiers. As this example also shows, a syntax-class definition may also defer to procedural code, such as `check-duplicate-id` to process the input. A `fail-when` clause allows macro developers to specify a part of the synthesized error message (when the default is not sufficiently clear).

```
(define-syntax-parser let
  [(_ bs:distinct-bindings body:expr ...+)
   ;; rewrites the pattern to a template
   #'((lambda (bs.lhs* ...) body ...) bs.rhs* ...)])
```

Figure 11 Macros via parsing macros.

Using these two syntax classes, specifying the complete shape of `let` is straightforward; see fig. 11. The `:distinct-bindings` classification of `bs` introduces names for two pieces of the input syntax: a sequence of identifiers (`bs.lhs*`) and a sequence of right-hand-side expressions (`bs.rhs*`), one per variable. The syntax template uses these pieces to generate the same target code as the macros in fig. 2.

A comparison of figs. 2 and 11 illustrates the advantages as well as the disadvantages of Culpepper's DSL for writing macros. On the positive side, the size of the Culpepper-style macro definition appears to remain the same as the one for the Kohlbecker-style one. The revised definition merely adds classifications to the macro's pattern and attribute selections to the macro's template. This shallow size comparison camouflages that these small changes cause the macro to check *all* constraints on the shape of `let` and formulate syntax errors in terms of the specified surface syntax. As Culpepper [10, page 469] explains, implementing the same level of assumption checking and error reporting via procedural macros increases the code size by "several factors." Furthermore the "primary benefit [of this meta-DSL] ... is increased clarity" of a macro's input specification and its code template.

On the negative side, macro programmers are now expected to develop syntax classifications such as those in fig. 10 and use them properly in macro definitions, as in fig. 11. While the development of syntax classifications clearly poses a new obstacle, their use comes with a significant payoff *and* most end up as reusable elements in libraries. Hence the cost of developing them is recouped through reuse. As for the use of syntax classifications in macro templates and patterns, experience shows that most macro programmers consider the annotations as type-like notions and the attribute selections as a natural by-product.

In short, Culpepper's meta-DSL completely replaces the `define-syntax-cases` meta-DSL for macro definitions. By now, the large majority of Racket programmers develop macros in Culpepper's DSL and contribute to the ever-expanding collection of syntax classifications.

	lang. extens. (sec. 3)	lexical scope (2)	classify syntax (1)	error messages (1)	separate compil. (4)	run-time encaps. (4)	code gen. opt. (3)
Lisp macros	✓	–	–	–	–	–	–
Scheme							
– syntax-rules	✓	✓	patterns	–	–	–	–
– syntax-case	✓	✓	patterns & fenders	–	–	–	–
Racket – syntax-parse	✓	✓	patterns & syn. classes	✓	✓& phases	✓	`module` only

– means programmers have the tools to design manual solutions

Figure 12 A concise overview of Lisp-family language extension features.

7 DSL Creators Need Still More Than Modular, Parsing Macros

Racket has made great progress in improving the state of the art of macros with an eye toward both language extension and DSL implementation. Fig. 12 surveys the progress in roughly the terms of sec. 3's criteria. The `syntax-parse` DSL for defining macros can express almost every context-free and -sensitive constraint; macro developers get away with a few hints and yet get code that reports syntax errors in terms of the macro-defined variant. The `module` system supports both the fine-grained export/import of macros for language extensions and the whole-cloth implementation of DSLs.

At this point, implementing DSLs is well within reach for Racket beginners [4, 16] and easy for experts. While beginners may focus on `module`-based DSLs, experts use macros to create fluidly embedded DSLs. Examples are the DSL of pattern-matching for run-time values, the `syntax-parse` DSL itself, and Redex [17, 40]. In this domain, however, the macro framework falls short of satisfying the full list of desiderata for from sec. 3.

To explain this gap, let us concisely classify DSLs and characterize Racket's support:

- *stand-alone DSLs*
  These are the most recognized form in the real world. Racket supports those via `module` languages with at least the same conveniences as other DSL construction frameworks.
- *embedded DSLs with a fixed interface*
  All programming languages come with numerous such sub-languages. For example, `printf` interprets the format DSL – usually written as an embedded string – for rendering some number of values for an output device. In Racket, such DSLs instead come as a new set of expression forms with which programmers compose domain-specific programs. Even in Racket, though, such DSLs allow only restricted interactions with the host.
- *embedded and extensible DSLs with an expression-level interface*
  Racket's DSLs for pattern matching and structure declarations are illuminating examples of this kind. The former allows programmers to articulate complex patterns, with embedded Racket expressions. The latter may contain patterns, which contain expressions, etc. The pattern DSL is extensible so that, for example, the DSL of structure definitions can automatically generate patterns for matching structure instances. Naturally, this DSL for structure declarations can also embed Racket expressions at a fine granularity.

With regard to the third kind of DSL, Racket's macro approach suffers from several problems. A comparison with the criteria in sec. 3 suggests three obvious ones.

```
(define-typed-syntax (if cond then else)
  [⊢ cond >> cond-ir ⟹ cond-type]
  [⊢ then >> then-ir ⟹ then-type]
  [⊢ else >> else-ir ⟸ else-type]
   -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -
  [(AST-if cond-ir then-ir else-ir) ⟶ then-type])
```

Figure 13 Type-checking from macros.

The first concerns DSLs that demand new syntactic categories and, in turn, good parsing and error reporting techniques. While syntax classes allow DSL creators to enumerate the elements of a new syntactic category, this enumeration is fixed. Experienced DSL implementors can work around this restriction, just like programmers can create extensible visitor patterns in object-oriented languages to allow the blind-box extension of data types.

The second problem is about context-sensitive language processing. The existing macro framework makes it difficult to implement context-sensitive static checking, translations, and optimizing transformations – even for just Racket's macros, not to mention those that define new syntactic categories. Chang and his students [6, 7] have begun to push the boundaries in the realm of type checking, a particular rich form of context-sensitivity. Specifically, the team has encoded the rich domain of type checking as a meta-DSL. In essence, this meta-DSL enables DSL creators to formulate type checking in the form of type elaboration rules from the literature (see fig. 13), instead of the procedural approach of fig. 3. However, their innovation exploits brittle protocols to make macros work together [28]. As a result, it is difficult to extend their framework or adapt it to other domains without using design patterns for macro programming.

Finally, the DSL framework fails to accommodate languages whose compilation target is not Racket. Consider an embedded DSL for Cuda programming that benefits from a fluid integration with Racket. Such a DSL may need two interpretations: on computers with graphical co-processors it should compile to GPU code, while on a computer without such a processor it may need to denote a plain Racket expression. Implementing a dependent-type system in the spirit of Chang et al.'s work supplies a second concrete example. The language of types does not have the semantics of Racket's expressions and definitions. Although it is possible to expand such DSLs through Racket's core, it forces DSL developers to employ numerous macro-design patterns.

The proposed work-arounds for these three problems reveal why the Racket team does not consider the problem solved. Racket is all about helping programmers avoid syntactic design patterns. Hence, the appearance of design patterns at the macro level is antithetical to the Racket way of doing things, and the Racket team will continue to look for improvements.

References

1 H. Abelson, R.K. Dybvig, C.T. Haynes, G.J. Rozas, N.I. Adams, D.P. Friedman, E. Kohlbecker, G.L. Steele, D.H. Bartley, R. Halstead, D. Oxley, G.J. Sussman, G. Brooks, C. Hanson, K.M. Pitman, and M. Wand. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

2 Alan Bawden and Jonathan Rees. Syntactic Closures. In *Symposium on Lisp and Functional Programming*, pages 86–95, 1988.

3 Matthias Blume. Refining Hygienic Macros for Modules and Separate Compilation. Technical report tr-h-171, ATR Human Information Processing Research Laboratories, Kyoto, Japan, 1995. URL: `people.cs.uchicago.edu/~blume/papers/hygmac.pdf`.

4 Matthew Butterick. Beautiful Racket, 2013. URL: `https://beautifulracket.com/`.

5 Cadence Research Systems. *Chez* Scheme Reference Manual, 1994.

6 Stephen Chang, Alex Knauth, and Ben Greenman. Type Systems as Macros. In *Symposium on Principles of Programming Languages*, pages 694–705, 2017.

7 Stephen Chang, Alex Knauth, and Emina Torlak. Symbolic Types for Lenient Symbolic Execution. In *Symposium on Principles of Programming Languages*, pages 40:1–40:29, 2018.

8 William Clinger and Jonathan Rees. Macros That Work. In *Symposium on Principles of Programming Languages*, pages 155–162, 1991.

9 Ryan Culpepper. *Refining Syntactic Sugar: Tools for Supporting Macro Development.* PhD thesis, Northeastern University, 2010.

10 Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22(4–5):439–476, 2012.

11 Ryan Culpepper and Matthias Felleisen. Taming Macros. In *Generative Programming and Component Engineering*, pages 225–243, 2004.

12 Ryan Culpepper and Matthias Felleisen. Fortifying Macros. In *International Conference on Functional Programming*, pages 235–246, 2010.

13 Sergey Dmitriev. Language-oriented Programming: the Next Programming Paradigm, 2004.

14 R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-Passing Style: A General Macro Mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, January 1988.

15 R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

16 Matthias Felleisen, Forrest Bice, Rose DeMaio, Spencer Florence, Feng-Yun Mimi Lin, Scott Lindeman, Nicole Nussbaum, Eric Peterson, Ryan Plessner, David Van Horn, and Conrad Barski. *Realm of Racket.* No Starch Press, 2013. URL: `http://www.realmofracket.com/`.

17 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009.

18 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs. Second Edition.* MIT Press, 2001–2018. URL: `http://www.htdp.org/`.

19 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The Structure and Interpretation of the Computer Science Curriculum. *Journal of Functional Programming*, 14(4):365–378, 2004.

20 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*, pages 113–128, 2015.

21 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A Programmable Programming Language. *Communications of the ACM*, 61(3):62–71, February 2018.

22 R. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pages 48–59, 2002.

23 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

24 Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Conference on Programming Language Design and Implementation*, pages 23–32, 1996.

25 Matthew Flatt. Composable and Compilable Macros: You Want it *When?* In *International Conference on Functional Programming*, pages 72–83, 2002.

26 Matthew Flatt. Submodules in Racket: you want it when, again? In *Generative Programming and Component Engineering*, pages 13–22, 2013.

27 Matthew Flatt. Binding As Sets of Scopes. In *Symposium on Principles of Programming Languages*, pages 705–717, 2016.

28 Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that Work Together: Compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming*, 22(2):181–216, March 2012.
29 Matthew Flatt and Matthias Felleisen. Cool Modules for HOT Languages. In *Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
30 Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine). In *International Conference on Functional Programming*, pages 138–147, September 1999.
31 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Symposium on Principles of Programming Languages*, pages 171–183, 1998.
32 Timothy P. Hart. MACROS for LISP. Technical Report 57, MIT Artificial Intelligence Laboratory, 1963.
33 Paul Hudak. Modular Domain Specific Languages and Tools. In *Fifth International Conference on Software Reuse*, pages 134–142, 1998.
34 Richard Kelsey and Jonathan Rees. A Tractable Scheme Implementation. *Lisp and Symbolic Computation*, 5(4):315–335, 1994.
35 Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, 1986.
36 Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic Macro Expansion. In *Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
37 Eugene E. Kohlbecker and Mitchell Wand. Macros-by-Example: Deriving Syntactic Transformations from their Specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.
38 Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, 2001.
39 Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From Macros to Reusable Generative Programming. In *International Symposium on Generative and Component-Based Software Engineering*, pages 105–120, 1999.
40 Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems. In *Rewriting Techniques and Applications*, pages 2–16, 2004.
41 Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In *Summit on Advances in Programming Languages*, pages 12:1–12:15, 2017.
42 Guy Lewis Steele Jr. *Common Lisp – The Language*. Digital Press, 1984.
43 Gerald L. Sussman and Guy Lewis Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report 349, MIT Artificial Intelligence Laboratory, 1975.
44 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, pages 395–406, 2008.
45 Martin P. Ward. Language Oriented Programming. *Software Concepts and Tools*, 15:147–161, April 1994.

The Dynamic Practice and Static Theory of Gradual Typing

Michael Greenberg
Pomona College, Claremont, CA, USA
`http://www.cs.pomona.edu/~michael/`
michael@cs.pomona.edu

Abstract

We can tease apart the research on gradual types into two 'lineages': a pragmatic, implementation-oriented dynamic-first lineage and a formal, type-theoretic, static-first lineage. The dynamic-first lineage's focus is on taming particular idioms – 'pre-existing conditions' in untyped programming languages. The static-first lineage's focus is on interoperation and individual type system features, rather than the collection of features found in any particular language. Both appear in programming languages research under the name "gradual typing", and they are in active conversation with each other.

What are these two lineages? What challenges and opportunities await the static-first lineage? What progress has been made so far?

2012 ACM Subject Classification Social and professional topics → History of programming languages; Software and its engineering → Language features

Keywords and phrases dynamic typing, gradual typing, static typing, implementation, theory, challenge problems

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.6

Acknowledgements I thank Sam Tobin-Hochstadt and David Van Horn for their hearty if dubious encouragment. Conversations with Sam, Ron Garcia, Matthias Felleisen, Robby Findler, and Spencer Florence improved the argumentation. Stephanie Weirich helped me with Haskell programming; Richard Eisenberg wrote the program in Figure 3. Ross Tate, Neel Krishnaswami, Ron, and Éric Tanter provided helpful references I had overlooked. Ben Greenman provided helpful feedback. The anonymous SNAPL reviewers provided insightful comments as well as the CDuce code in Section 3.4.3. Finally, I thank Stephen Chong and Harvard Computer Science for hosting me while on sabbatical.

© Michael Greenberg;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 6; pp. 6:1–6:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 A tale of two gradualities

> It was the best of types, it was the worst of types,
> it was the age of static guarantees, it was the age of blame,
> it was the epoch of implementations, it was the epoch of core calculi,
> it was the season of pragmatism, it was the season of principles.
>
> – with apologies to Charles Dickens

In 2006, the idea of gradual typing emerged in two papers. Tobin-Hochstadt and Felleisen introduced the idea of mixing untyped and typed code such that "code in typed modules can't go wrong" using *contracts* [102, 34]; Siek and Taha showed how to relax the simply typed lambda calculus (plus some extensions) to allow for unspecified "dynamic" types to be resolved at runtime via *casts* [87].[1]

In these two papers, two parallel lines of research on gradual typing began with quite different approaches. Sam Tobin-Hochstadt summarized the distinction as 'type systems for existing untyped languages' [Tobin-Hochstadt and Felleisen] and 'sound interop btw typed and untyped code' [Siek and Taha] [101]. I draw slightly different lines, identifying one lineage as being "dynamic-first" and the other as "static-first". That is: one can think about taking a dynamic language and building a type system for it, or one can think about taking a statically typed language and relaxing it to allow for dynamism.

The differences at birth between these two approaches are still evident, and the latter approach has an opportunity for interesting new discoveries from proof-of-concept (and more serious) implementations.[2]

Disclaimer: I have made an effort to be thorough but not comprehensive in my citations. Readers looking for a comprehensive survey will enjoy Sam Tobin-Hochstadt's "Gradual Typing Bibliography" [117]. Even so, I make general claims about trends in gradual types. I try to mention the inevitable exceptions to my generalizations, but I may have missed some.

1.1 The dynamic-first approach

Tobin-Hochstadt and Felleisen use a "macro" approach, where the unit of interoperation is the module. They are directly inspired by Racket's module system. They see the dynamic language as being somehow primary, with a static layer above:

> First, a program is a sequence of modules in a safe, but dynamically typed programming language. The second assumption is that we have an explicitly, statically typed programming language that is [a] variant of the dynamically typed language. Specifically, the two languages share run-time values and differ only in that one has a type system and the other doesn't. [102]

---

[1] Flanagan showed how to use a similar cast framework to relax a fancy subset type system to a series of dynamic checks [36]. Dynamic checking is necessary in Flanagan's *hybrid typing*, because not every refinement is easy to send to an SMT solver. While the approach is different, the spirit is similar: there must have been something in the water.

[2] There are three other distinctions one could make. First, the macro/micro distinction from Takikawa et al. and Greenman et al. [98, 49]; second, the latent/manifest distinction [45, 46]; and third, the distinction between languages with static semantics that influence runtime behavior (e.g., type classes) and those languages where types can be erased. These distinctions are important but less salient for my analysis.

Their paper takes an "expanded core calculus" approach, defining an extension of the lambda calculus with a notion of module (untyped, contracted, or typed).

Dynamic-first gradual typing is about accommodating particular programming idioms in programs that allow legacy untyped code to interoperate with the newly typed fragment. Typed Racket is a canonical example, though TypeScript's various dialects, Dart, DRuby/Rubydust/rtc, Clojure's `specs`, Gradualtalk, and Reticulated Python are all comparable efforts in the research community [103, 39, 8, 81, 80, 5, 112]. These languages all share an approach going back chiefly to the 1990s but also earlier: we have a dynamic language in hand and we'd like the putative benefits of static typing (for, e.g., maintenance [59], documentation [64], performance) [20, 100, 17, 21, 50].

Dynamic idioms vary widely [82, 4], but a common theme is untypability in conventional systems. Accordingly, the type systems used in the dynamic-first approach tend to the unfamiliar, with features designed to capture particular language idioms: occurrence typing [103, 104, 56], "like" types [119], severe relaxations of runtime checking disciplines to avoid disrupting reference equality [112, 113], and ad hoc rules for inferring particular types (e.g., telling the difference between a tuple and an array in TypeScript or Flow).

1.2 The static-first approach

Siek and Taha take a "micro" approach, where the unit of interoperation is the expression [87]. They are inspired by Thatte's quasi-static typing and Oliart's algorithmic treatment thereof [100, 73]. While they imagine migrating programs from dynamic to static – would one ever want to go the other way? – they implicitly see the type system as primary, and gradual types as a relaxation. In their contributions:

> We present a formal type system that supports gradual typing for functional languages, providing the flexibility of dynamically typed languages when type annotations are omitted by the programmer and providing the benefits of static checking when function parameters are annotated.

Siek and Taha's paper does not, however, identify any particular dynamic idioms they want to write but that their static type discipline disallows. Such an example might serve as motivation for wanting to relax the type system, either to accommodate existing dynamic code that uses hard-to-type idioms (e.g., as in Takikawa et al. [99]) or to write new code that goes beyond their system (e.g., as in Tobin-Hochstadt and Findler [106]). To be sure, adding the dynamic type does add a new behavior – nontermination [1]. Siek and Taha don't explicitly observe as much beyond mentioning that the untyped lambda calculus embeds in their system. The code of their two lambda calculus interpreters is identical (their Figure 1; reproduced in our Figure 1); only the type annotations change.

According to Siek et al.'s refined definition [90], gradual typing "provides seamless interoperability, and enables the convenient evolution of code between the two disciplines"; it is critical to their conception of gradual typing that it "relates the behavior of programs that differ only with respect to their type annotations". Lacking particular dynamic idioms to accommodate, the examples in static-first papers tend to be toy snippets mixing static and dynamic code to highlight this interoperation, even when pointing out the oversight (e.g., Section 6 from Garcia and Cimini [41]).

Work in the "static-first" lineage cites interoperation as a motivation, not only in Siek and Taha's seminal paper [87] but especially in Wadler and Findler [114]. Later papers take interesting type feature X and show how to relax the typing rules, resolving static imprecision with dynamic checks. X ranges widely: objects [88], polymorphism [2, 3, 108], typestate [118],

```
(define interp
  (λ (env e)
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")])))
```

```
(type expr (datatype (Var ,symbol)
                     (Int ,int)
                     (App ,expr ,expr)
                     (Lam ,symbol ,expr)
                     (Succ ,expr)))
(type envty (listof (pair symbol ?)))

(define interp
  (λ ((env : envty) (e : expr))
    (case e
      [(Var ,x) (cdr (assq x env))]
      [(Int ,n) n]
      [(App ,f ,arg) (apply (interp env f) (interp env arg))]
      [(Lam ,x ,e) (list x e env)]
      [(Succ ,e) (succ (interp env e))])))

(define apply
  (λ (f arg)
    (case f
      [(,x ,body ,env)
       (interp (cons (cons x arg) env) body)]
      [,other (error "in application, expected a closure")])))
```

Figure 1 The lambda calculus interpreter from Siek and Taha (Figure 1 [87]).

information flow control [29, 33, 107], ownership types [85], effects [9], session types [55], etc. The process of relaxation was characterized and made beautifully concrete in Garcia, Clark, and Tanter's "Abstracting Gradual Typing" (AGT) [42]. In AGT, one "abstracts" a gradual type system starting from a syntax-directed, fully static type system that enjoys a preservation-based proof of type safety. Matteo Cimini and Jeremy Siek built the Gradualizer, a tool for automatically turning a variety of type systems gradual [27]. AGT is a human methodology, but has been shown to apply to a broad swath of systems [9, 61, 107, 108]. The Gradualizer is automatic, but is substantially less general than the principles in AGT.

The type systems in the static-first lineage tend to look much more like those found in the conventional types literature... unsurprising, in light of AGT! The resulting theories are typically conservative extensions of their original system, where statically typed programs remain acceptable – satisfying the static gradual guarantee (removing type information retains typeability) [90]. Many systems also enjoy the dynamic gradual guarantee (removing type information retains successful runs of the program), though notably not for several type systems implementing hyperproperties [107, 108].

2 Dynamic trouble in static paradise

> It is easy to design a type system, and it is reasonably straightforward to validate some theoretical property. However, the true proof of a type system is a pragmatic evaluation. To this end, it is imperative to integrate the novel ideas with an existing programming language. Otherwise it is difficult to demonstrate that the type system accommodates the kind of programming style that people find natural and that it serves its intended purpose.
>
> To evaluate occurrence typing rigorously, we have implemented Typed Scheme.
>
> – Tobin-Hochstadt and Felleisen [103]

2.1 A distinction without a difference?

Does it matter whether one starts from dynamic typing and works up to static typing or starts with static typing and relaxes to allow dynamic typing [53]? Only the dynamic-first lineage addresses particular examples and the particular difficulties they introduce into the resulting systems.

```
1 (define (flatten x)
2   (cond
3     [(null? x) '()]
4     [(cons? x) (append (flatten (car x)) (flatten (cdr x)))]
5     [else      (list x)]))
```

```
> (flatten '(1 (2 3) (((4) (5)) (6 7 8 (9)))))          ; example
'(1 2 3 4 5 6 7 8 9)
```

Figure 2 The `flatten` function in Scheme/Racket.

Dynamic-first gradual typing is motivated by particular, existing legacy code in particular, existing languages. Whatever theory dynamic-first systems come up with must be accommodated to the host language's pre-existing conditions.

> [D]ynamic language programmers often employ programming idioms that impede precise yet sound static analysis. For example, programmers often give variables flow-sensitive types that differ along different paths, or add or remove methods from classes at run-time using dynamic features such as reflection and `eval`. [8]

Static-first gradual typing typically lacks such concrete examples as motivation, studying interoperation more abstractly. Static-first gradual typing often studies type system *features* without any attempt to accommodate the idiosyncrasies of any particular implementation of those features. (There are, of course, laudable exceptions [85, 6].)

The distinction becomes clear when we see what is actually implemented: the overwhelming majority of the existing implementations of gradual typing start with a dynamic language and grow an appropriate type system for it [103, 39, 8, 81, 80, 5, 112]. There are several notable exceptions: Nom and Grift are direct implementations of the static-first theory of gradual typing for new static languages [68, 60]; Thorn invents its own theory of "like" types [13]; C# is a statically typed language which grew a dynamic runtime unrelated to the theory of gradual types [65].

It is surprising that the theory takes a static-first approach, but the practice takes a dynamic-first one. It would seem that nobody has tried to apply the static-first theory to a pre-existing statically typed language. A set of concrete, desirable idioms from dynamic typing would allow the dynamic-first and static-first lineages to address the same challenges and benefit more from each other's insights. I offer one such challenge in detail, followed by some higher level challenges (Section 3).

2.2 A dynamic idiom: flatten

A canonical example of a dynamic programming idiom is the `flatten` function (Figure 2). The `flatten` function takes arbitrarily nested lists (formed by `cons` cells) and produces a single flat list containing all of the elements in a left-to-right traversal. Thinking of such nested lists as trees, `flatten` computes the fringe of the tree. The `flatten` function works because there are predicates `null?` and `cons?` of conceptual type $? \to$ `bool`. While it is a perfectly safe function – nothing in it can go wrong at runtime – it is hard to assign a type to `flatten`, since the type of uniformly constructed, heterogeneous lists cannot be written down in simple type languages. Like in Tobin-Hochstadt and Findler's "gradual typing poem" [106], we assign the dynamic type to patch over a programming idiom that our type system cannot account for (there, cyclic data structures; here, heterogeneity and arbitrary nesting).

2.2.1 flatten in dynamic-first gradual typing

Occurrence typing captures the reasoning in `flatten` perfectly, allowing Typed Racket to infer the type of flatten without any annotations.[3]

Occurrence typing is not a standard type system feature. It is not even a particularly desirable one according to the tastes of the static typing community, as evidenced by its lack of adoption there. Folks who like static types seem to prefer dependent pattern matching for flow-sensitive reasoning. Occurrence typing is used in Typed Racket because it suffices to characterize many of the idioms used: it “accommodates ... modes of reasoning ... programmers use” – Typed Racket was designed “to support Scheme idioms and programming styles” [103]. To put it in terms of Ron Garcia’s 2018 ICFP keynote, Typed Racket is an exercise in “type system anthropology”, finding the folk type system that corresponds to Racket programmers’ mental models [40].

2.2.2 flatten in static-first gradual typing

How might one write `flatten` in the static-first lineage? First, let us be clear that statically typed languages can already more or less accommodate the `flatten` function! Zhe Yang implemented it in SML two ways: once with functors, and once with embeddings to and projections from a universal-datatype [121]; the embedding/projection model is not too hard to use but is not the most efficient [10, 11]. One can implement `flatten` in Haskell using recent reflection support (see Figure 3). ℂDuce can express this function directly (see Figure 4).

Static languages accommodate `flatten` with either significant runtime cost or fancy type systems. Work in the static-first lineage of gradual typing has only recently devised systems that can accommodate this simple function. Most static-first gradual type systems don’t offer type tests, though there are noteworthy exceptions [62, 63, 16]. Siek and Tobin-Hochstadt’s true union types [89] can handle the definition at the same moral type of $? \to$ `list` $?$ (in their notation, $\star \to \mu X.\mathtt{unit} \cup \star \times X$). Recent work by Castagna, Lanvin, and others might be able to accommodate the idiom, as well [24, 25].

3 An opportunity

We ought to (a) identify the particular new programs gradual typing allows us to write or interoperate with and (b) verify that we can implement gradual type systems accommodating these new programs. Enumerating concrete examples and implementing the theory will stress-test our understanding, leading to refinements and improvements in both theory and practice.

Since there are multiple motivations for wanting gradual typing, I’ve broken the challenges up into sections by motivation: added expressiveness (Section 3.1), interoperation (Section 3.2), and types themselves (Section 3.3). I conclude by addressing possible objections (Section 3.4).

[3] Typed Racket assigns the type `(-> Any (Listof Any))`. Unfortunately, Typed Racket cannot express the negation lurking in the codomain under the `Listof`, where one might want to write `(-> Any (Listof (- Any (Listof Any))))`.

3.1 Gradual typing for expressiveness

> For any interesting programming language, there will always be some programs that [the] user must rewrite to accommodate a static type checker.
>
> – Mike Fagan's *Fundamental Theorem of Static Typing* [32]

If one studies gradual typing in order to be able to write new kinds of programs, I offer three examples of dynamic idioms that might serve as motivating examples: heterogeneous structures, semi-structured data, and an object annotation strategy drawn from the "middleware" approach to webservers.

3.1.1 Heterogeneous structures

It is very common to program with uniformly constructed, heterogeneous data structures in dynamic programming languages: the lists in `flatten` nest arbitrarily and hold arbitrary values (heterogeneity) but are constructed using only `'()` and `cons` (uniformity). While `flatten` is a "toy" function, it manipulates the heterogeneous lists with a non-trivial use of type predicates in a way that is simultaneously realistic but also challenging to existing static-first type theories. Fagan's PhD thesis is rich in such examples [32].

Not only do static type systems limit the kinds of values that get put into data structures, they often limit the shapes of those data structures themselves. It is not a trivial exercise to construct a non-statically known, immutable, circular list in OCaml. Programming in a static language, I might want to temporarily "cheat" and view my structured data a little less formally than the type system would ordinarily allow. For example, one might temporarily allow mutation to make a list circular before "freezing" it as an immutable one [106]. How can such shenanigans be safely accommodated in languages that want types to mean things? What does heterogeneity mean for more complicated structures like tree-based sets and maps that, e.g., compare values to maintain invariants?

3.1.2 Semi-structured data, like JSON, YAML, and XML

A great deal of information is stored and exchanged in semi-structured formats like JSON, YAML, and XML. Even when these formats don't take advantage of recursion, they represent heterogeneous data that isn't easily accommodated by type systems. Much of XML can be handled with some moderately fancy type system features – unions and recursive types [12] – but a proper account of names has proven elusive, in part due to the challenging type system features necessary (e.g., row types and first-class names). While Typed Racket is good at working with heterogeneous structures, it accommodates semi-structured data less well. Can gradual typing help us work with these common structures? Might we be able to gradualize recent advances in reasoning about rows [67]? Might we offer a gradual treatment of the row-based metaprogramming of Ur/Web [26]?

3.1.3 Attaching information to HTTP request and response objects

So-called "middleware" in web servers is typically implemented as a quasi-continuation-passing function $\texttt{mw} : \texttt{Req} \times \texttt{Resp} \times (\texttt{unit} \to \alpha) \to \alpha$, where `Req` and `Resp` are (mutable) HTTP request and response objects and the third argument is a (thunked) continuation. Middleware can be used for many tasks: application transformers which, e.g., compress outgoing data with gzip, but also session management and authentication regimes for tracking which user a request belongs to. Such authentication middleware might look up user information and then *attach* that user information to the request object, making it available for other portions of the web application that rely on such user information being present.

Authentication middleware amounts to a form of strong update, where a record – the HTTP request object – gets a new field, or its field changes type. Sound gradual systems can support strong update [91]; can we extend and implement these systems to write *mostly* typed web-servers that can accommodate this "attachment" idiom?

3.2 Gradual typing for interoperation

If one studies gradual typing in order to interoperate programs from different idioms, what better way to show it than by implementing an interoperation library for, say, OCaml and Python or Haskell and Julia or Scala and Clojure?

There are several challenges left unaddressed by theoretical treatments of interoperation. High level concerns include design issues surrounding numerics, annotations, type-driven features, and how data structures (and their invariants) can be ported from one language to another. There are also critical lower level concerns, like garbage collection, linking, and debugging.

3.2.1 Numerics

Dynamic languages typically have a "numeric tower" with rules for when values move from more precise types (e.g., unbounded bignum integers or precise rationals) to less precise ones (e.g., fixed or floating point numbers). Statically typed languages typically require explicit coercions (e.g., `fromIntegral` and other conversions in Haskell's numeric type classes) and sometimes have separate operations for each numeric type (e.g., `+` and `+.` in OCaml).

For static languages to interoperate with dynamic ones, the promotion rules will leak. A statically polymorphic function run in the dynamic side could result in a promotion, which might violate parametricity. These thorny questions have been studied for Racket's complicated numeric tower already [93]; what should happen in other settings?

3.2.2 Data structures, interfaces vs. translations, and guarantees

Tobin-Hochstadt and Felleisen assume that "the two languages share run-time values and differ only in that one has a type system and the other doesn't" [102]. This will not generally hold. The representation of Racket and OCaml strings are different, but so are their interfaces: string constants in Racket are immutable,[4] while OCaml's are mutable.

When we move a value from language A to language B, we may want to send it over as an object with an interface – allowing B to use the object with A's semantics – or to translate it to one of many possible targets in B. Such translations will come with a computational cost – typically linear but sometimes worse! – but allow several benefits: it may be more efficient to avoid the A/B language barrier, B may have more efficient representations in general, and B may provide guarantees that A does not. Can gradual typing theory or implementations help us think about these questions? There has been some related work for contracts and data structures [35]; how might it port over?

3.2.3 Type-driven features

Muehlboeck and Tate have shown that a variety of type-based features in C# lead to violations of the dynamic gradual guarantee [68]. The core issue is that the runtime semantics of some features depend on the decisions made during static typing. Haskell's type classes are

[4] Though Racket documentation indicates that not all strings are immutable [77].

another example of this phenomenon: it is determined statically how to resolve each call of a constrained function, which determines, e.g., which monad to use. How might Haskell mix with dynamic code that performs IO or other effects? How might dynamic values in Haskell enjoy the `Ord` instances necessary to build, e.g., heterogeneous sets?

3.2.4 Minimizing annotation overhead

Static-first gradual typing typically studies elaborated core calculi – many papers do not describe the surface language that generates the runtime checks. How can we minimize the annotation overhead in the source language? What check insertion strategies are appropriate? Swamy et al. give a theoretical starting point for thinking about coercions more generally [96], subsuming Henglein's seminal work [50] but not offering an algorithm.. Allende et al. offer a concrete analysis of the issue and a novel, hybrid cast insertion strategy in an object-oriented setting [7]. Greenman and Felleisen consider "a spectrum of type soundness" for cast insertion but not alternative sound strategies [47]; What tool support do we need – inference [86]? Something more exploratory, along the lines of Campora et al. [18, 19]? More tools for eliminating checks [71]?

The idea of minimizing annotation overhead is implicit in gradual versions of fancy type systems, where the "dynamic" side is a typical static type system and the "static" side is a fancier type system (e.g., information flow [29, 33, 107]). Experiments with an implementation are a natural next step.

3.2.5 Lower-level concerns: garbage collection, linking, and debugging

When two languages interoperate, which is responsible for allocating and deallocating? When does each language's GC run? This question is a serious one: in Ramsey's Lua-ML, the thorny issue of whose garbage collector is in charge led him to reimplement Lua in OCaml [79]! Not only is such a "duplication of effort ... regrettable", it means that Ramsey's Lua may not behave identically to the original Lua and won't necessarily keep up as Lua evolves.

What is the right object/header format? It is a shame that if we were to try to link Rust and Haskell, we would probably have to go through a C API! How does one take the hodgepodge of stack frames, thunks, and continuations from mixing two real languages and produce something intelligible?

3.3 Gradual typing for typing's sake

One could summarize the gradual types approach as finding runtime-enforceable safety properties that simultaneously (a) allow one to relax the strictures of type checking in part of one's program while (b) not compromising the safety guarantees in the checked parts of the program. But types are more than safety guarantees! Folklore and substantial engineer experience assign high value to what I called before the "putative benefits of static typing". Types are executable documentation [64]; they rule out whole classes of errors, assist in maintenance [59], and can lead to more efficient code. So far, the literature on gradual types has focused on the "rule out whole classes of errors" benefit. But there are others! To what extent do the existing, implemented dynamic-first systems buy you the benefits of static typing? To what extent would implementations of theoretical, static-first systems buy you those same benefits?

For efficiency, it's a mixed bag. Typed Racket sometimes generates better code than one would get without the type systems [78]. But Typed Racket has a hard row to hoe: it lives within the strictures of Racket's dynamic-first world, where the primitives by default perform

runtime checks. Typed Racket already adds its own checks at module boundaries; avoiding checks on internal uses of primitives takes both effort and care. Working in ActionScript, which has a less constrained runtime, Rastogi et al. find an average 1.6x improvement when using their type inference algorithm over partially annotated programs. Nom and Grift both show that gradual types can be implemented efficiently [68, 60]; while both can generate good static code, neither recovers global, type-based optimizations.

For maintenance, Typed Racket exhibits some of the desirable behaviors: on changing an algebraic datatype definition to include new possibilities, the type system can find functions that are now missing cases. But having the right features in the small and behaving correctly in the large are two different things. A ten-year retrospective on Typed Racket's "migratory typing" suggests that deeper study is required [105].

3.4 In which I am gravely mistaken

"No, no," you say, "that's not right. We can already do all of this!" There has indeed been good progress towards meeting these challenges! I don't mean to diminish the substantial literature on these topics, but rather to help direct its aims. What do we have so far, and how close are we to meeting the challenges I've laid out?

3.4.1 Static languages can accommodate those idioms

You can just *make* a datatype for JSON; OCaml already has S-expression support instead of the general dynamic type. Polymorphic variants offer some of the flexibility and reuse of dynamic types while also admitting a meaningful typing discipline [43].

Standard examples like heterogeneous lists and maps are typeable using some of the fancier features of Haskell's type system [58, 57, 115]. Haskell has dependency, `Data.Dynamic` and `Type.Reflection`, and the Aeson library.

▶ Response. Maybe the static world never really wanted to interoperate with dynamic types. But there are still challenges.

Type-based programming in Haskell is strong medicine, and every project has a limited complexity budget. Not everyone wants to spend their complexity budget on types, even if some claim (tongue in cheek) that Haskell is already gradually typed [31]. For example, the `flatten` function can be written in Haskell (Figure 3), but it is somewhat less readable than its Racket counterpart. The definition itself is not so much longer than the Racket code, and the supporting functions could live in a library. One could presumably write a similar program in Scala using the `Dynamic` trait.

Or consider Yesod, a mature Haskell web application framework [92]. Yesod uses idioms like routing and middleware for specifying servers, as is common in dynamic web frameworks like ExpressJS [37].

Yesod supports sessions directly, using cookies to allow a notion of continuity in stateless HTTP sessions. Yesod's sessions are implemented as maps from `Text` to `ByteString`. Could one build a version of Yesod that used Haskell's type system to guarantee that user information was available in the session map for stages of processing that needed it? Could we construct a gradual version of that system?

3.4.2 We can use linking types

Patterson and Ahmed's linking types solve this problem [76].

▶ Response. Let's implement it! Linking types have been successful for proving things about translations [15, 14]. Do they have any bearing on implementations? Work by Matthews et al. offers some gestures in this direction [63, 62];

```haskell
{-# LANGUAGE
    GADTs, TypeApplications, ScopedTypeVariables, ViewPatterns,
    PolyKinds, DataKinds
#-}
module Flatten where

import Data.Dynamic
import Type.Reflection

data MaybeMatch (a :: k1) (b :: k2) where
  Match :: MaybeMatch a a
  NoMatch :: MaybeMatch a b

isType :: forall a b. Typeable a => TypeRep b -> MaybeMatch a b
isType (eqTypeRep (typeRep @a) -> Just HRefl) = Match
isType _ = NoMatch

smartToDyn :: TypeRep a -> a -> Dynamic
smartToDyn (isType @Dynamic -> Match) x = x
smartToDyn rep                         x = Dynamic rep x

flatten :: [Dynamic] -> [Dynamic]
flatten [] = []
flatten (dx@(Dynamic rep x):dxs) = x' ++ flatten dxs
  where
    x' | App (isType @[] -> Match) arg <- rep
       = flatten (map (smartToDyn arg) x)
       | otherwise
       = [dx]
```

Figure 3 A version of `flatten` using Haskell's `Dynamic` type.

3.4.3 These ideas are already implemented

Typed Racket, Gradualtalk, DRuby/rtc/Rubydust, Reticulated Python, Thorn, Nom and Grift are implementations [103, 5, 39, 8, 81, 112, 13, 68, 60]; some theoretical work offers web interfaces for experimentation with their type theory [107].

▶ Response. Let's do more! Let's scale them to real, existing languages; let's implement the various challenge problems I've described.

GradualTalk, DRuby/Rubydust/rtc, Reticulated Python, and the various TypeScript dialects are all more or less in the dynamic-first lineage, since they are put on top of existing dynamic languages. While Reticulated Python is inspired by gradual typing, the transient checking strategy they invented for it only loosely corresponds to anything found in the static-first lineage [113] (see Greenman and Felleisen [47] and Greenman and Migeed [48]).

Of course, there are exceptions. C# is an example of a statically typed language that added dynamic features; this effort doesn't seem particularly informed by gradual typing theory, but its dynamic language runtime draws on some of the challenges here as motivation, e.g., working with JSON and XML [65].

Nick Benton showed how to use `call/cc` to get clean errors at the dynamic/static interface when mixing untyped and typed code [11], but never scaled up to a real language. Similar embedding/projection pairs can be found in Yang's work, Benton's earlier work on embedding languages in ML, and Ramsey's Lua-ML [121, 10, 79]. The idea of embedding/projection

```
1  let flatten ( Any -> [ (Any\[Any*])* ] )
2    | []     -> []
3    | (h,t) -> (flatten h)@(flatten t)
4    | x      -> [x]
5
6  type Tree('a) = ('a\[Any*]) | [ (Tree('a))* ]
7
8  let flattenTree ( (Tree('a)) -> ['a*] )
9     | []       -> []
10    | [h ; t] -> (flattenTree h)@(flattenTree t)
11    | x        -> [x]
```

Figure 4 Flatten in ℂDuce, with and without a custom type.

pairs as a core idea in gradual types goes back at least to Henglein [51], but has seen a resurgence in recent work by New et al. [69, 70]. Only Ramsey's work seems to have ever enjoyed a serious implementation; even so, it seems that the up-to-date Lua-OCaml interface is via a more conventional, `ctypes`-based API [30]. Gray et al.'s reflection-based approach offer a substantial interface between two very different languages (Java and Scheme) [44]. Their interface compiles the Java to Scheme, though, sidestepping the "two runtimes" issue in much the same way Ramsey does.

ℂDuce deserves particular attention, as its set-theoretic types allow for concise implementations of functions like `flatten` [12]. We can write two good implementations of `flatten` in ℂDuce (Figure 4): the first version uses ordinary types, while the second uses a custom type definition to better characterize the list structure of the input. ℂDuce doesn't address issues of interoperation at all, but recent work has shown that ℂDuce's set-theoretic types are relevant to gradual typing [25]. How much of the reasoning done in dynamic languages can be done using set-theoretic types? What kinds of reasoning lie outside those types?

Various recent systems have moved beyond core calculi, studying surface syntax directly [120, 66, 24, 25]; why not try practical experiments?

3.4.4 That's not what gradual typing is about

The dynamic and static ends of the gradual typing spectrum are relative. As Ron Garcia shows in his 2018 ICFP keynote, one could consider SML as a dynamic language in light of the static verification of partial pattern matches used in the datasort refinement system of Refined ML [38, 40]. Some of the gradual notions of, e.g., security typing [29, 33, 107], have this flavor.

▶ Response. The particular challenge problems may change, but a focus on implementations will help the community relate efforts in the world of gradual types and efforts outside.

Taking security typing as a concrete instance, there are already researchers working hard on runtime enforcement mechanisms for information flow control [95, 54, 52, 94, 110]. While these systems generally don't support a notion of graduality, they *do* support important security properties overlooked in the gradual types approaches (for example, protection from side channels and termination sensitivity). It would be productive to hold a "Build It, Break It, Fix it" contest between gradual and conventional systems [83]. LIO's implementation has already been used to build systems of moderate size [75]; why not build a comparable system in a gradual security-typed language?

There has been some work on enforcing linearity at runtime, with ALMS being a realistic implementation [109]; Scherer et al.'s theory comes with a toy implementation [84]. To test these ideas, why not implement such a system for Rust, allowing runtime checking for affinity as an alternative to unsafe blocks?

Finally, the first question asked after Ron Garcia's keynote was, "I'd like to use 'dynamic' Haskell programs from Agda. What do I do about nontermination?" A good deal of work studies how to gracefully allow nontermination in languages with consistent logical interpretations of their types [116, 22, 74, 23, 97, 111, 28]. What might a "gradual" system look like here? Does the AGT methodology help?

The gradual types approach is to find safety properties – i.e., runtime enforcement mechanisms – that are sufficient to guarantee the desirable properties of a given type system. Nguyen et al.'s runtime semantics for checking size-change termination is related: their runtime checks can be lifted to static ones via an analysis [72]. Their implementation isn't "gradual", though, as there's no notion of "mixed" dynamic and static checking.

More broadly, is "graduality" even the right fit for thinking about nontermination? There is some recent evidence that the dynamic gradual guarantee – which some see as essential to gradual typing [70] – is incompatible with various hyperproperties, like noninterference [107] and parametricity [108]. Binary formulations of parametricity generalize unary formulations of termination arguments. Can a language satisfy the gradual guarantees but also preserve strong properties like logical consistency in mixed programs?

4 Conclusion

I come to praise gradual types, not to bury them.

– with apologies to William Shakespeare

Gradual types is a thriving research area. Work is plentiful in both the implementation-focused dynamic-first lineage and the theoretically-focused static-first lineage. Our community has made good progress so far, and recent implementation efforts give me hope that the two lineages will enter into a more fruitful dialog. Significant challenges remain for both the theory and practice of gradual types; there are thorny practical questions that deserve immediate attention, as they will determine the direction of our efforts. Which programs do we want to write? How do theoretical models generalize to whole languages? What of the low-level concerns of interoperation, viz. the "two runtimes" problem?

Finally, perhaps I am wrong. Maybe the distinction between these lineages is a trivial one, and the theory is already applicable to practice. The best proof that the distinction I draw is trivial would be an interoperation layer for an existing language that follows existing theory directly, without any need to adapt to pre-existing conditions. I would welcome such proof, and I encourage the gradual types community to take advantage of this opportunity and implement their ideas.

References

1 M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic Typing in a Statically-typed Language. In *Principles of Programming Languages (POPL)*, pages 213–227, New York, NY, USA, 1989. ACM. doi:10.1145/75277.75296.

2 Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Principles of Programming Languages (POPL)*, pages 201–214, 2011. doi:10.1145/1926385.1926409.

3 Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. Theorems for free for free: parametricity, with and without types. *PACMPL*, 1(ICFP):39:1–39:28, 2017. doi:10.1145/3110283.

4 Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. Tracing Dynamic Features in Python Programs. In *Working Conference on Mining Software Repositories (MSR)*, pages 292–295, New York, NY, USA, 2014. ACM. doi:10.1145/2597073.2597103.

5 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual Typing for Smalltalk. *Sci. Comput. Program.*, 96(P1):52–69, December 2014. doi:10.1016/j.scico.2013.06.006.

6 Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 251–270, 2014. doi:10.1145/2660193.2660222.

7 Esteban Allende, Johan Fabry, and Éric Tanter. Cast Insertion Strategies for Gradually-typed Objects. In *Dynamic Languages Symposium (DLS)*, pages 27–36, New York, NY, USA, 2013. ACM. doi:10.1145/2508168.2508171.

8 David An, Avik Chaudhuri, Jeffrey Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *Principles of Programming Languages (POPL)*, pages 459–472. ACM, 2011.

9 Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A Theory of Gradual Effect Systems. In *International Conference on Functional Programming (ICFP)*, pages 283–295, New York, NY, USA, 2014. ACM. doi:10.1145/2628136.2628149.

10 Nick Benton. Embedded Interpreters. *J. Funct. Program.*, 15(4):503–542, July 2005. doi:10.1017/S0956796804005398.

11 Nick Benton. Undoing Dynamic Typing (Declarative Pearl). In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Functional and Logic Programming*, pages 224–238, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

12 Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *International Conference on Functional Programming (ICFP)*, pages 51–63. ACM, 2003. doi:10.1145/944705.944711.

13 Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 117–136, 2009. doi:10.1145/1640089.1640098.

14 William J. Bowman and Amal Ahmed. Typed Closure Conversion for the Calculus of Constructions. In *Programming Language Design and Implementation (PLDI)*, pages 797–811, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192372.

15 William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. Type-preserving CPS Translation of Σ and Π Types is Not Not Possible. *Proc. ACM Program. Lang.*, 2(POPL):22:1–22:33, December 2017. doi:10.1145/3158110.

16 John Boyland. The Problem of Structural Type Tests in a Gradual-Typed Language. In *FOOL*, 2014.

17 Gilad Bracha. Pluggable type systems, October 2004. URL: http://bracha.org/pluggableTypesPosition.pdf.

18 John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating Gradual Types. *Proc. ACM Program. Lang.*, 2(POPL):15:1–15:29, December 2017. doi:10.1145/3158103.

19 John Peter Campora, Sheng Chen, and Eric Walkingshaw. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proc. ACM Program. Lang.*, 2(ICFP):98:1–98:30, July 2018. doi:10.1145/3236793.

20 Robert Cartwright. User-Defined Data Types as an Aid to Verifying LISP Programs. In *ICALP*, 1976.

21 Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI)*, pages 278–292, New York, NY, USA, 1991. ACM. doi:10.1145/113445.113469.

22 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Step-Indexed Normalization for a Language with General Recursion. In *MSFP*, volume 76, pages 25–39, 2012.

23 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *Principles of Programming Languages (POPL)*, pages 33–46. ACM, 2014.

24 Giuseppe Castagna and Victor Lanvin. Gradual Typing with Union and Intersection Types. *Proc. ACM Program. Lang.*, 1(ICFP):41:1–41:28, August 2017. doi:10.1145/3110285.

25 Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.*, 3(POPL):16:1–16:32, January 2019. doi:10.1145/3290329.

26 Adam Chlipala. Ur: Statically-typed Metaprogramming with Type-level Record Computation. In *Programming Language Design and Implementation (PLDI)*, pages 122–133, New York, NY, USA, 2010. ACM. doi:10.1145/1806596.1806612.

27 Matteo Cimini and Jeremy G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *Principles of Programming Languages (POPL)*, pages 443–455, 2016. doi:10.1145/2837614.2837632.

28 Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. Foundations of dependent interoperability. *Journal of Functional Programming*, 28:e9, 2018. doi:10.1017/S0956796818000011.

29 Tim Disney and Cormac Flanagan. Gradual Information Flow Typing. In *Workshop on Script-to-Program Evolution (STOP)*, 2011.

30 Paolo Donadeo. Lua-OCaml. URL: https://pdonadeo.github.io/ocaml-lua/.

31 Richard Eisenberg. Haskell as a gradually typed dynamic language, January 2016. URL: https://typesandkinds.wordpress.com/2016/01/22/haskell-as-a-gradually-typed-dynamic-language/.

32 Mike Fagan. *Soft typing: An approach to type checking for dynamically typed languages*. PhD thesis, Rice University, 1991. URL: https://scholarship.rice.edu/handle/1911/16439.

33 L. Fennell and P. Thiemann. Gradual Security Typing with References. In *Computer Security Foundations Symposium (CSF)*, pages 224–239, June 2013. doi:10.1109/CSF.2013.22.

34 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, pages 48–59, 2002. doi:10.1145/581478.581484.

35 Robert Bruce Findler, Shu-Yu Guo, and Anne Rogers. Lazy Contract Checking for Immutable Data Structures. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages*, pages 111–128. Springer-Verlag, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-85373-2_7.

36 Cormac Flanagan. Hybrid type checking. In *Principles of Programming Languages (POPL)*, pages 245–256, 2006. doi:10.1145/1111037.1111059.

37 The Node Foundation. ExpressJS. URL: https://expressjs.com/.

38 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI)*, June 1991.

39 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. Static type inference for Ruby. In *Symposium on Applied Computing (SAC)*, pages 1859–1866, 2009. doi:10.1145/1529282.1529700.

40 Ronald Garcia. Gradual Typing (keynote), 2018. ICFP. URL: https://www.youtube.com/watch?v=fQRRxaWsuxI.

41 Ronald Garcia and Matteo Cimini. Principal Type Schemes for Gradual Programs. In *Principles of Programming Languages (POPL)*, pages 303–315, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676992.

42 Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Principles of Programming Languages (POPL)*, pages 429–442, 2016. doi:10.1145/2837614.2837670.

43 Jacques Garrigue. Code reuse through polymorphic variants. In *FOSE*, 2000.

44 Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained Interoperability Through Mirrors and Contracts. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 231–245, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094830.

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

46 Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. *J. Funct. Program.*, 22(3):225–274, 2012.

47 Ben Greenman and Matthias Felleisen. A Spectrum of Type Soundness and Performance. *Proc. ACM Program. Lang.*, 2(ICFP):71:1–71:32, July 2018. doi:10.1145/3236766.

48 Ben Greenman and Zeina Migeed. On the Cost of Type-Tag Soundness. In *Partial Evaluation and Program Manipulation PEPM*, pages 30–39, 2018. doi:10.1145/3162066.

49 Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *Journal of Functional Programming*, 29:e4, 2019. doi:10.1017/S0956796818000217.

50 Fritz Henglein. Dynamic Typing: Syntax and Proof Theory. In *European Symposium on Programming (ESOP)*, pages 197–230, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V. URL: http://dl.acm.org/citation.cfm?id=197475.190867.

51 Fritz Henglein. Dynamic Typing: Syntax and Proof Theory. *Sci. Comput. Program.*, 22(3):197–230, June 1994. doi:10.1016/0167-6423(94)00004-2.

52 Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *POST*, volume 9036, pages 11–31. Springer, 2015.

53 David Van Horn, September 2018. URL: https://twitter.com/lambda_calculus/status/1039702266679369730.

54 Cătălin Hriţcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All Your IFCException Are Belong to Us. In *Security and Privacy (SP)*, pages 3–17, 2013. I am deeply embarrassed by the title of this paper. doi:10.1109/SP.2013.10.

55 Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *PACMPL*, 1(ICFP):38:1–38:28, 2017. doi:10.1145/3110282.

56 Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence typing modulo theories. In *Programming Language Design and Implementation (PLDI)*, pages 296–309. ACM, 2016.

57 Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. *CoRR*, abs/cs/0509027, 2005.

58 Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly Typed Heterogeneous Collections. In *ACM SIGPLAN Workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM. doi:10.1145/1017472.1017488.

59 S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, and E. Tanter. Do static type systems improve the maintainability of software systems? An empirical study. In *IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162, June 2012. doi:10.1109/ICPC.2012.6240483.

60 Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. Efficient Gradual Typing. In *Programming Language Design and Implementation (PLDI)*, 2019. To appear. arXiv:1802.06375.

61 Nico Lehmann and Éric Tanter. Gradual Refinement Types. In *Principles of Programming Languages (POPL)*, pages 775–788, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009856.

62 Jacob Matthews and Amal Ahmed. Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices! In *European Symposium on Programming (ESOP)*, pages 16–31, 2008. doi:10.1007/978-3-540-78739-6_2.

63 Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3):12:1–12:44, 2009. doi:10.1145/1498926.1498930.

64 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 and Applications (OOPSLA)*, pages 683–702, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384666.

65 Microsoft. Dynamic Language Runtime overview. URL: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview.

66 Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. Dynamic Type Inference for Gradual Hindley–Milner Typing. *Proc. ACM Program. Lang.*, 3(POPL):18:1–18:29, January 2019. doi:10.1145/3290331.

67 J. Garrett Morris and James McKinna. Abstracting Extensible Data Types: Or, Rows by Any Other Name. *Proc. ACM Program. Lang.*, 3(POPL):12:1–12:28, January 2019. doi:10.1145/3290325.

68 Fabian Muehlboeck and Ross Tate. Sound Gradual Typing is Nominally Alive and Well. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, New York, NY, USA, 2017. ACM. doi:10.1145/3133880.

69 Max S. New and Amal Ahmed. Graduality from embedding-projection pairs. *PACMPL*, 2(ICFP):73:1–73:30, 2018.

70 Max S. New, Daniel R. Licata, and Amal Ahmed. Gradual type theory. *PACMPL*, 3(POPL):15:1–15:31, 2019.

71 Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *PACMPL*, 2(POPL):51:1–51:30, 2018. doi:10.1145/3158139.

72 Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Size-Change Termination as a Contract. In *Programming Language Design and Implementation (PLDI)*, 2019. To appear. arXiv:1808.02101.

73 Alberto Oliart. An algorithm for inferring quasi-static types. Technical Report 1994-013, Boston University, 1994. URL: http://www.cs.bu.edu/techreports/pdf/1994-013-quasi-static-types.pdf.

74 Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. Dependent interoperability. In *PLPV*, pages 3–14. ACM, 2012.

75 James Parker, Niki Vazou, and Michael Hicks. LWeb: Information Flow Security for Multi-tier Web Applications. *Proc. ACM Program. Lang.*, 3(POPL):75:1–75:30, January 2019. doi:10.1145/3290388.

76 Daniel Patterson and Amal Ahmed. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *Summit on Advances in Programming Languages (SNAPL)*, volume 71, pages 12:1–12:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SNAPL.2017.12.

77 PLT. URL: https://docs.racket-lang.org/reference/strings.html.

78 PLT. URL: https://docs.racket-lang.org/ts-guide/optimization.html.

79 Norman Ramsey. Embedding an Interpreted Language Using Higher-order Functions and Types. In *Workshop on Interpreters, Virtual Machines and Emulators (IVME)*, pages 6–14, New York, NY, USA, 2003. ACM. doi:10.1145/858570.858571.

80 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Principles of Programming Languages (POPL)*, pages 167–180, 2015. doi:10.1145/2676726.2676971.

81 Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. The ruby type checker. In *Symposium on Applied Computing (SAC)*, pages 1565–1572, 2013. doi:10.1145/2480362.2480655.

82 Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do. In Mira Mezini, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

83 Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. Build It, Break It, Fix It: Contesting Secure Development. In *Computer and Communications Security (CCS)*, pages 690–703, New York, NY, USA, 2016. ACM. doi:10.1145/2976749.2978382.

84 Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. Fabous Interoperability for ML and a Linear Language. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures (FoSSaCS)*, pages 146–162, Cham, 2018. Springer International Publishing.

85 Ilya Sergey and Dave Clarke. Gradual Ownership Types. In *European Symposium on Programming (ESOP)*, pages 579–599, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-28869-2_29.

86 Uri Shaked. TypeWiz, 2019. URL: https://github.com/urish/typewiz.

87 Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

88 Jeremy G. Siek and Walid Taha. Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27, 2007. doi:10.1007/978-3-540-73589-2_2.

89 Jeremy G. Siek and Sam Tobin-Hochstadt. The Recursive Union of Some Gradual Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 388–410, 2016. doi:10.1007/978-3-319-30936-1_21.

90 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *Summit on Advances in Programming Languages (SNAPL)*, volume 32, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.SNAPL.2015.274.

91 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic References for Efficient Gradual Typing. In Jan Vitek, editor, *European Symposium on Programming (ESOP)*, pages 432–456, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

92 Michael Snoyman et al. Yesod. URL: https://www.yesodweb.com/.

93 Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the Numeric Tower. In *Practical Aspects of Declarative Languages (PADL)*, pages 289–303, 2012. doi:10.1007/978-3-642-27694-1_21.

94 Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. Flexible dynamic information flow control in the presence of exceptions. *J. Funct. Program.*, 27:e5, 2017.

95 Deian Stefan, Alejandro Russo, John Mitchell, and David Maziéres. Flexible Dynamic Information Flow Control in Haskell. In *Haskell Symposium*, 2011.

96 Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A Theory of Typed Coercions and Its Applications. In *International Conference on Functional Programming (ICFP)*, pages 329–340, New York, NY, USA, 2009. ACM. doi:10.1145/1596550.1596598.

97 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-monadic Effects in F*. In *Principles of Programming Languages (POPL)*, pages 256–270, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837655.

98 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Position Paper: Performance Evaluation for Gradual Typing. In *Workshop on Script-to-Program Evolution (STOP)*, New York, NY, USA, 2015. ACM. URL: `http://www.ccs.neu.edu/home/types/publications/pe4gt/pe4gt.pdf`.

99 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 and Applications (OOPSLA)*, pages 793–810, 2012. `doi:10.1145/2384616.2384674`.

100 Satish R. Thatte. Quasi-Static Typing. In *Principles of Programming Languages (POPL)*, pages 367–381, 1990. `doi:10.1145/96709.96747`.

101 Sam Tobin-Hochstadt, September 2018. URL: `https://twitter.com/samth/status/1039707471290478595`.

102 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 964–974, 2006. `doi:10.1145/1176617.1176755`.

103 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Principles of Programming Languages (POPL)*, pages 395–406, New York, NY, USA, 2008. ACM. `doi:10.1145/1328438.1328486`.

104 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *International Conference on Functional Programming (ICFP)*, pages 117–128. ACM, 2010.

105 Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *Summit on Advances in Programming Languages (SNAPL)*, volume 71, pages 17:1–17:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.SNAPL.2017.17`.

106 Sam Tobin-Hochstadt and Robert Bruce Findler. Cycles Without Pollution: A Gradual Typing Poem. In *Workshop on Script to Program Evolution (STOP)*, pages 47–57, New York, NY, USA, 2009. ACM. `doi:10.1145/1570506.1570512`.

107 Matías Toro, Ronald Garcia, and Éric Tanter. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.*, 40(4):16:1–16:55, 2018. URL: `https://dl.acm.org/citation.cfm?id=3229061`.

108 Matías Toro, Elizabeth Labrada, and Éric Tanter. Gradual parametricity, revisited. *PACMPL*, 3(POPL):17:1–17:30, 2019. URL: `https://dl.acm.org/citation.cfm?id=3290330`.

109 Jesse A. Tov and Riccardo Pucella. Practical Affine Types. In *Principles of Programming Languages (POPL)*, 2011. `doi:10.1145/1926385.1926436`.

110 Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. From fine- to coarse-grained dynamic information flow control and back. *PACMPL*, 3(POPL):76:1–76:31, 2019.

111 Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, University of California, San Diego, 2016. URL: `https://escholarship.org/uc/item/8dm057ws`.

112 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Symposium on Dynamic Languages (DLS)*, pages 45–56, New York, NY, USA, 2014. ACM. `doi:10.1145/2661088.2661101`.

113 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big Types in Little Runtime: Open-world Soundness and Collaborative Blame for Gradual Type Systems. In *Principles of Programming Languages (POPL)*, pages 762–774, New York, NY, USA, 2017. ACM. `doi:10.1145/3009837.3009849`.

114 Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In *European Symposium on Programming (ESOP)*, pages 1–16, 2009. `doi:10.1007/978-3-642-00590-9_1`.

115 Stephanie Weirich. The influence of dependent types (keynote). In *Principles of Programming Languages (POPL)*, page 1. ACM, 2017.

116 Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Aaron Stump, Harley Eades, Peng (Frank) Fu, Garrin Kimmell, Tim Sheard, Ki Yung Ahn, and Nathan Collins. The Preliminary Design of the Trellys Core Language, 2011. Discussion session at PLPV.

117 Sam Tobin-Hochstadt with Jeremy Siek, Asumu Takikawa, Ben Greenman, et al. URL: `https://github.com/samth/gradual-typing-bib`.

118 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual Typestate. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 459–483, 2011. `doi:10.1007/978-3-642-22655-7_22`.

119 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Principles of Programming Languages (POPL)*, pages 377–388. ACM, 2010.

120 Ningning Xie, Xuan Bi, and Bruno C. d. S. Oliveira. Consistent Subtyping for All. In Amal Ahmed, editor, *European Symposium on Programming (ESOP)*, pages 3–30, Cham, 2018. Springer International Publishing.

121 Zhe Yang. Encoding Types in ML-like Languages. In *International Conference on Functional Programming (ICFP)*, pages 289–300, New York, NY, USA, 1998. ACM. `doi:10.1145/289423.289458`.

A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity

Lenny Truong
Stanford University, USA
lenny@cs.stanford.edu

Pat Hanrahan
Stanford University, USA
hanrahan@cs.stanford.edu

Abstract

Leading experts have declared that there is an impending golden age of computer architecture. During this age, the rate at which architects will be able to innovate will be directly tied to the design and implementation of the hardware description languages they use. Thus, the programming languages community stands on the critical path to this new golden age. This implies that we are also on the cusp of a golden age of hardware description languages. In this paper, we discuss the intellectual challenges facing researchers interested in hardware description language design, compilers, and formal methods. The major theme will be identifying opportunities to apply programming language techniques to address issues in hardware design productivity. Then, we present a vision for a multi-language system that provides a framework for developing solutions to these intellectual problems. This vision is based on a meta-programmed host language combined with a core embedded hardware description language that is used as the basis for the research and development of a sea of domain-specific languages. Central to the design of this system is the core language which is based on an abstraction that provides a general mechanism for the composition of hardware components described in any language.

2012 ACM Subject Classification Hardware → Hardware description languages and compilation

Keywords and phrases Hardware Description Languages

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.7

Funding The authors would like to thank the DARPA DSSoC program (contract # FA8650-18-2-7861), the AHA and SystemX affiliate programs, and Intel's Agile ISTC for supporting this work.

Acknowledgements The authors would like to thank the anonymous reviewers and Will Crichton for providing feedback on paper drafts. They would also like to thank their colleagues at Stanford, particularly members of the HardwarePL Reading Group and the AHA Agile Hardware Center, as many of the ideas in this paper are the direct or indirect result of the many intellectual discussions that they have collectively participated in. Finally, the authors would like to thank the organizers of SNAPL for creating a venue for visionary papers on programming languages.

© Lenny Truong and Pat Hanrahan;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 7; pp. 7:1–7:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Turing award winners John Hennessy and David Patterson recently declared that we are on the cusp of a new golden age of computer architecture [29, 30]. Current trends in silicon manufacturing are signaling the end of Moore's law and Dennard scaling. This, combined with the inherent inefficiencies in general-purpose processor design, indicates that new innovations in computer architecture will come from the design of domain-specific architectures. The recent proliferation of application accelerators, such as Apple's neural engine for the A12 and Google's Tensor Processing Unit, support the idea that the hardware community is transitioning to specialized chip design. This shift signals a new golden age because researchers have an opportunity to develop radically different architectures rather than incremental improvements to existing processor designs.

Domain-specific chips necessarily target smaller markets which implies that design teams will become smaller and demand more of their tools in order to be productive. This indicates that we are also on the cusp of a new golden age of hardware description languages (HDLs) because hardware designers are actively seeking radically new technologies that will dramatically reduce design time and cost. A major impediment to design productivity is the fact that the hardware ecosystem is a monoculture comprised of a few chip designs from a small number of manufacturers. It is essential that the hardware community shifts towards a diverse ecosystem of easily accessible intellectual property blocks that can be composed to construct new chip designs. Underlying this shift will be advances in HDLs that promote the proliferation of hardware libraries.

The development of HDL abstractions that promote reuse, correctness, and performance represents the main challenge for this new golden age of HDLs. Fortunately, the programming languages (PL) community has enjoyed a rich history of success in developing techniques to address these issues. HDL researchers have already started to tackle the reuse problem by applying standard software programming language techniques such as meta-programming, polymorphism, and abstract data types [4, 11]. Evidently HDL researchers stand to benefit greatly from the lessons learned by their software language counterparts.

This paper identifies three problem domains that lie at the intersection of programming languages and hardware: language design, compiler infrastructure, and formal methods. For those unfamiliar with the hardware design process or HDLs, Section 2 covers the essential concepts required to understand the intellectual challenges discussed in Section 3. Section 4 presents a vision of a multi-language system for constructing hardware that is designed to address these intellectual challenges. With this impending golden age of HDLs, it is an exciting time to be interested in programming languages and hardware.

2 Background

A *hardware description language* (HDL) is an instance of a programming language that has been designed to provide abstractions for describing circuits. This paper will focus on the discussion of digital hardware, where circuits are described as logic operating on discrete, binary signals. Digital hardware can be further divided into two categories: synchronous and asynchronous. In a synchronous circuit, state changes are synchronized by a clock signal. In contrast, asynchronous circuits can contain state elements that change at any time. A majority of modern digital designs are synchronous, but the increasing demand for efficiency has renewed interest in asynchronous designs or hybrid models such as globally asynchronous, locally synchronous [58]. A digital HDL is defined to be *expressively complete* if it can be used to express both synchronous and asynchronous designs.

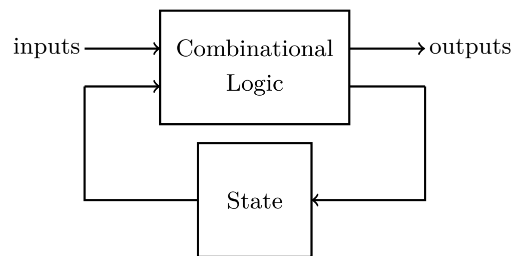


Figure 1 An abstract depiction of a sequential logic circuit constructed as the composition of combinational logic with state. Notice that the outputs *could* depend on the inputs, implying that the circuit *could* describe a Mealey machine [43]. Depending on the mechanism chosen for storing state, the circuit could be synchronous or asynchronous. See Section 2.1 for more details.

2.1 Digital Design

In digital circuit theory, *combinational logic* refers to circuits where the output is a pure and total function of the inputs. In contrast, *sequential logic* refers circuits where the outputs are dependent on the sequence of past inputs. Figure 1 depicts the canonical design pattern for using combinational logic circuits composed with state to construct sequential logic circuits. Sequential logic circuits are used to implement *finite-state machines* (FSMs), a fundamental component of building digital systems.

When discussing FSMs in the context of hardware design, it is important to recognize the distinction between Mealey [43] and Moore [45] machines. For a Mealey machine, the output of the circuit is a function of the state and the inputs, while for a Moore machine, the output of the circuit is purely dependent on the state. The differences between these classes of FSMs are more pronounced in hardware than in software because they exhibit different timing characteristics. When using registers to store state, a Moore machine can be viewed as a purely synchronous entity where changes to the state and output values are triggered by a clock, while a Mealey machine can exhibit asynchronous behavior where output values immediately respond to changes to input values. An expressively complete HDL will be able to describe and compose both Moore and Mealey machines.

2.2 Verilog

The *Verilog* language is the dominant HDL used in practice today [23]. The language was originally developed as a commercial verification and simulation product [24] and was later adopted as a basis for logic synthesis. As a result, the semantics of the language are defined in terms of a hardware simulation being executed as a software program [57]. The design of the language is directly inspired by C, exhibiting many of the same features including a preprocessor, control flow, and operators. Like C, Verilog has become the *lingua franca* of the HDL ecosystem and is used as the common interchange format for design tools.

The core of Verilog's semantics is based on a module abstraction which shares many similarities to function abstraction from software languages. A *module* has an interface and a definition. An *interface* is a set of typed ports. A *port* is similar to a function argument or return value and represents a named entity that is used to consume or produce data. A *definition* is a list of statements that describe the module behavior using various language features such as the wiring and module instancing operators. Verilog designs are comprised of hierarchically composed modules that are simulated using a dataflow execution model. Figure 2 shows an edge detector FSM written in Verilog.

```
1  module edge_detector(input in, output out, input clk);
2      localparam A=0, B=1, C=2;
3
4      reg [1:0] state,      // Current state
5                nextState; // Next state
6
7      always @(posedge clk) begin
8          if (reset) begin
9              state <= A; // Initial state
10         end else begin
11             state <= nextState;
12         end
13     end
14
15     always @(*) begin
16         nextState = state;
17         out = 0;
18         case (state)
19             A : if (in) nextState = C;
20                 else nextState = B;
21             B : if (in) begin
22                     out = 1;
23                     nextState = C;
24                 end
25             C : if (~in) begin
26                     out = 1;
27                     nextState = B;
28                 end
29             default : begin
30                 out = 1'bX;
31                 nextState = 3'bX;
32             end
33         endcase
34     end
35 endmodule
```

Figure 2 Verilog implementation of an edge detector FSM adapted from the University of Washington CSE370 course materials [20]. The circuit has two inputs and one output and is designed as a Mealey machine where the output is 1 if the current value of `in` is the inverse of the previous value of `in` (i.e., the input is changing from 1 to 0 or from 0 to 1). Line 1 declares the module name and interface. The ports have an implicit width of 1 bit and are qualified with a direction `input` or `output`. Line 2 declares a set of constants that are used to abstract the encoding of the FSM states. Lines 4 and 5 declare variables to hold the current and next state. Lines 7-13 describe the state update logic inside a Verilog `always` block. This block of code defines a procedure to run when a `posedge clk` event occurs. That is, on a positive clock edge, update or reset the `state` variable. Lines 15-34 define another `always` block that is sensitive to changes to any input signal, denoted by the `@(*)`. This means that if any input value changes, this block of code will fire. The block encodes the combinational logic for computing the output and next state values as a function of the input and current state values. Because the second `always` block is sensitive to any input change, the semantics are defined asynchronously. Contrast this with the first `always` block which enforces the state updates to be synchronous by only executing on the positive edge of the clock. On lines 30 and 31, the values of `out` and `nextState` are assigned the value `X` to explicitly indicate they are undefined and can be any value. See Section 2.2 for more details.

```
acc1 :: Stream Word -> Stream Word
acc1 in = out
     where
         out = (delay out 0) + in

-- input -> current state -> (new state, output)
acc2 :: Word -> Word -> (Word, Word)
acc2 in s = (s', out)
     where
         out = s + in
         s' = out
```

Figure 3 An example of two mechanisms for encoding state in a Haskell embedded HDL adapted from the Clash documentation [39]. Both functions describe an accumulator architecture that stores a running count of the input values over time. The function `acc1` shows the first approach which is based on a `Stream` data structure with a `delay` operator. The `delay` operator returns the input stream with the values shifted by one cycle. The second argument to `delay` is used to specify the first value of the stream. The function `acc2` uses a different approach where the current state is passed as an argument and the next state is returned as an output. See Section 2.3 for more details.

2.3 Functional HDLs

HDL development has a long tradition in the functional languages community [46]. Functional HDLs leverage the idea that a pure function can be used to model combinational logic. The fundamental problem these languages face is integrating the concepts of time and state in order to enable the description of sequential logic.

μFP [55] and Daisy [35] both introduced a technique based on reactive programming where a stream data structure is used to describe circuits where the output can depend on the history of the inputs. μFP extends the FP language with a recursively defined μ operator which takes a function and produces a new function with internal state. The essence of the μ operator is that it supplies the current value of the state as an input to the function, and it uses an output of the function to set the next value of the state. Daisy uses a different approach by modeling sequential logic using recursive equations. Both these approaches required the development of a new language in order to implement their ideas. The designers of Clash [39] recognized that Haskell's lazy evaluation can be used to construct infinite streams, indicating that it could serve as a host for an embedded HDL. Figure 3 shows how Haskell's semantics enable the description of sequential logic circuits.

One interesting technique applied to functional HDLs is the use of combinators to describe circuit structure. Hydra [50] showed that a recursive, stream-based abstraction enabled the use of higher-order functions to capture structural patterns. Lava [11] extended the use of recursive data types with the ability to describe general circuit networks rather than just tree-like structures. This technique provides powerful facilities for code reuse by enabling the description of circuits as a regular pattern of components. In practice, this approach has proved particularly useful when applied to the problem of circuit layout [56].

2.4 Term Rewriting Systems

Another lineage of work [32] has explored the application of term rewriting systems (TRS) [52] to the description of hardware. In these systems, circuits are described as a set of rewrite rules which are applied to the inputs and state values to produce the outputs and next

```
module mkCounter(Counter);
    Reg#(Bit#(8)) value <- mkReg(0);

    method Bit#(8) read();
        return value;
    endmethod

    method Action load(Bit#(8) newval);
        value <= newval;
    endmethod

    method Action increment();
        value <= value + 1;
    endmethod
endmodule
```

Figure 4 Implementation of a synchronous counter adapted from the Bluespec tutorial [12] with the module interface specification omitted. Line 2 declares an 8-bit register named `value`. Lines 4-14 define the implementation of the `read`, `load`, and `increment` methods which define the behavior of the module. Notice that the compiler must handle the data race between `load` and `increment` on `value`. See Section 2.4 for more details.

state values. An important quality of TRS is that they model the non-determinism and concurrency that are intrinsically present in hardware. For example, a conflict could occur when two rules match the same input data and try to update the same state element. The development of schemes for detecting and arbitrating conflicts represents the main intellectual challenge for these systems. Figure 4 shows a synchronous counter written in Bluespec [47], an established HDL based on TRS.

2.5 High-level Synthesis

High-level synthesis (HLS) is a technique that is broadly defined as compiling general software programs to hardware [65]. This paper will eschew the use of the term HLS due to the ambiguity of what is considered high-level. For example, a recent survey evaluated HLS tools using benchmarks written in C [46]. However, the PL community would consider C to be a low-level language. Instead, this paper will use the concept of the virtual machine abstraction to encompass the languages used as input to HLS systems. Languages based on a virtual machine abstraction provide some notion of unbounded resources such as an infinite register space. Section 3.1.3 discusses this in more detail.

A hardware compiler for a general purpose programming language relies on a strategy for mapping a program that may be unbounded in time and space into a finite set of resources. Typically this involves exploring the trade-offs between scheduling computation in space or time. If the compiler can determine parallelism in some computation, this logic can be mapped into concurrently executing hardware modules. However, data dependencies, finite resources, and suboptimal cost models complicate the task for larger applications. The compiler must use heuristics to schedule computation into the time dimension and insert the requisite logic to orchestrate the sequencing of the computation. Figure 5 shows a synchronous counter implemented in SystemC [22], a subset of the C language used for HLS.

```
1  SC_MODULE (counter) {
2      sc_in_clk clock;
3      sc_in<bool> reset;
4      sc_out<sc_uint<4> > counter_out;
5
6      sc_uint<4> count;
7
8      void incr_count () {
9          if (reset.read() == 1) {
10             count = 0;
11             counter_out.write(count);
12         } else {
13             count = count + 1;
14             counter_out.write(count);
15         }
16     }
17
18     SC_CTOR(counter) {
19         SC_METHOD(incr_count);
20         sensitive << reset;
21         sensitive << clock.pos();
22     }
23 };
```

Figure 5 Example of a 4-bit counter defined in SystemC adapted from EDAplayground [1]. Lines 2-4 declare the interface of the module. Line 6 declares an internal state variable. Lines 8-16 define a method `incr_count` which implements the behavior of the counter using the SystemC data types. Notice that input ports are read using the `read` method and outputs are written using the `write` method. The rest of the body of the definition is interpreted as normal C code. Lines 18-22 define a constructor for the counter object and is mainly responsible for defining the sensitivity of the module to the `reset` input as well as the positive edge of the `clock` input. See Section 2.5 for more details.

3 Intellectual Challenges

This section divides the concerns of HDL research into three intellectual domains: language design, compiler infrastructure, and formal methods. Each of these domains represents a subset of a more general research area that is of interest to the broader PL community.

3.1 Language Design

The general discipline of programming language design revolves around the development of abstractions. A language designer will employ abstractions to enable the user to ignore certain details about a program. Well-designed abstractions make the development and maintenance of programs easier. In some cases, such as in domain-specific languages (DSLs), abstractions also serve as a basis for the development of compiler optimizations. In this impending golden age, the main challenge for HDL designers will be devising and composing abstractions that enable code reuse, improve correctness of programs, and that can be used to construct designs that produce high quality results from a compiler.

There are three major levels of abstractions employed in modern HDL design. The lowest level is the *circuit* abstraction where hardware is modeled as a graph of connected components. The next level is the *register-transfer* abstraction where hardware is described

as the computation on data flowing between registers. The highest level is the *virtual machine* abstraction where hardware is modeled as a set of instructions for an abstract machine. Many HDLs incorporate abstractions from multiple levels. For example, the Verilog language is based on the circuit abstraction but also provides various facilities for describing hardware using the register-transfer abstraction.

3.1.1 Circuit Abstraction

In the *circuit* abstraction, hardware is described as a graph of connected components. The abstraction consists of three primitive concepts: circuits, ports, and wires. A *circuit* has an interface and a definition. An *interface* consists of a set of ports. A *port* is a named entity that is used to consume or produce data. A *definition* contains a set of circuit instances and wires. A *wire* connects two ports. In the hardware community, language features based on the circuit abstraction are described as *structural*. For example, a design written in structural Verilog will only use the language features for defining, instancing and wiring up Verilog modules. Purely structural designs are called *netlists*.

The dominant structural HDL in use today is Verilog. Chisel [4] and Magma [27] are examples of an emerging subclass of structural languages called *Hardware Construction Languages*. These languages embed the circuit abstraction into a general purpose programming language which provides a mechanism for meta-programming circuit definitions. This approach exhibits a distinct advantage over Verilog; moving features related to parametrization and code generation to the host language simplifies the precise specification of the HDL.

In theory, a purely structural HDL is expressive enough to capture any real world digital hardware design. This is argued by the fact that the physical result from manufacturing hardware is always a component that is composed of connected sub-components, recursing all the way down to the transistors. Based on this fact, we posit that the circuit abstraction is the fundamental primitive upon which all other HDL abstractions can be constructed. Remark that the abstraction is agnostic as to whether the behavior of the circuit is synchronous or asynchronous which indicates that it is expressively complete.

The main challenge for the circuit abstraction is determining whether the connection between two ports is semantically correct with respect to the intended behavior of the design. Most HDLs attach a notion of direction to ports which enables the use of a type system to check that only an output can be connected to an input. An interesting research direction moving forward will be increasing the expressiveness of the types used for circuit ports. Ideally these types are able to capture the semantics of the protocols used to communicate between two components. Section 3.3.2 provides a more detailed discussion on how session types might be used to address this issue.

An interesting quality of the circuit abstraction is that, while it is based on the low-level details of hardware design, it can be used to compose black box modules at any level in the design hierarchy. This makes it a compelling basis for the development of hardware libraries. As discussed in Section 2.3, functional HDLs with a circuit abstraction can leverage combinator patterns to construct reusable circuit structures. Further research on language facilities that enable the construction of hardware libraries based on the circuit abstraction will be an essential component of this new golden age of HDLs.

3.1.2 Register-Transfer Abstraction

The *register-transfer* abstraction models hardware as computation on data flowing between registers. Registers are defined as primitive data storage elements that update their values based on a clock signal. Because register semantics are intrinsically tied to a clock, this

abstraction is concerned with the description synchronous digital circuits. However, it important to note that this abstraction could be composed with other abstractions for describing asynchronous logic. In the hardware community, languages using the register-transfer abstraction are described as *register-transfer level* (RTL) languages.

A structural HDL that includes a notion of a register provides a register-transfer abstraction; the computation on data flowing between registers is described using circuit instances and connections. However, the register-transfer level of abstraction encompasses a broader set of concepts such as functions and operators. In practice, most HDLs combine the register-transfer abstraction with the circuit abstraction by extending the concept of a circuit definition to include constructs such as expressions, statements, and procedures. This technique raises the level of abstraction by removing the need to explicitly define, instance, and wire up register circuits. Instead registers are treated as language primitives that behave similarly to variables in a standard software programming language.

The fundamental issue in RTL language design is the precise choice of semantics for abstracting the concept of a register. For example, the Verilog `always` block provides a procedural abstraction for describing the simulation behavior of a component. To model a register, the designer uses variables to store data across clock events. The Verilog specification is explicit in stating that a variable does not imply a hardware register [57], instead it is left to the synthesis tool to determine how the simulation behavior of an `always` block can be mapped into an implementation using concrete hardware registers. This design choice raises the level of abstraction by enabling the user to ignore details about how the program is concretely implemented in hardware. However, it also removes the ability for the user to explicitly specify the registers used in the synthesized design. In practice, this choice can result in a mismatch between the results of register synthesis and the designer's intent. To remedy this, Verilog design teams enforce style guidelines that restrict the usage of `always` blocks such that the synthesis results are transparent. An alternative design could use qualifiers to explicitly declare variables that should be hardware registers.

Choosing how to map the concept of a variable to a register is a fundamental design issue for all imperative RTL HDLs. A related issue is reconciling the synchronous update semantics of registers with the asynchronous update semantics of standard variables. For example, given standard imperative evaluation semantics, writing to two different variables in a procedure would happen at different steps in the evaluation. However, if both variables are mapped to hardware registers, they would be updated at the same time in the synthesized hardware. One design choice would be to explicitly model the synchronized temporal update semantics of a register variable in the evaluation semantics of the language, ensuring that the user's model of the computation exactly matches the behavior of the synthesized hardware.

Verilog's non-blocking assignments provide the capability to explicitly model synchronous storage. In Verilog, a blocking assignment is executed before the subsequent statements in a block of sequential code. In contrast, a non-blocking assignment does not block procedural flow and is performed near the end of a time-step. When combined with clock events, the non-blocking assignment can be used to model the synchronous update semantics of hardware registers by delaying variable updates until the end of a clock period.

The interplay between these two forms of assignments comprise a major component of the complexity of the Verilog specification. One might think that the semantics of non-blocking assignment could be simply to delay the evaluation until all blocking assignments have been completed. However, the evaluation of a non-blocking assignment will trigger an event on the variable being assigned, which in turn may trigger an event involving a blocking assignment. To handle this, Verilog's semantics include a loop for each time step that moves

between blocking and non-blocking assignments until there are no events left to process. The complexity of these evaluation semantics can make it difficult to reason about a Verilog design involving both forms of assignment. In practice, Verilog design teams will follow style guides that enforce the usage of assignments in a reasonable to understand manner.

The functional HDLs described in Section 2.3 demonstrate two more techniques for abstracting the concept of a register. One approach is to encode the state in the interface of a function. This is done by describing a circuit as a function that consumes the current state as one of its inputs and produces the next state as one of its outputs. A function of this form describes the transition function of a finite-state transducer (FST), providing a basis for a simple hardware synthesis algorithm where the current state is stored in registers. This technique necessarily implies a synchronized state update because the next values of the state are produced at the end of the evaluation of a function. There is no means to specify a state update at any other time.

The second approach uses a stream-based abstraction to encode state. The inputs and outputs of a function are a `Stream` data structure with a special `delay` operator that allows the user to look into the past values of a `Stream`. Given this operator, the user may describe a circuit where the values of an output stream depend on the values of an input stream at a prior clock cycle. The simplest compilation algorithm for this approach will insert registers to implement the behavior specified by the `delay` operator. This approach provides a convenient abstraction for working with the past values of the input, but prevents the user from explicitly managing state. For example, the output of a circuit could depend on some computation on a window values of the input. In this case, it may be most efficient to store a partial computation on the input value as the state, but the stream-based abstraction forces the user to describe the computation as a function of the delayed input stream values. The compiler is then responsible for discovering the fact that the intermediate computation can be stored as opposed to storing just the stream values and redoing computation.

Term rewriting systems for hardware [32] abstract registers by mapping terms to synchronous storage elements and rewrite rules to combinational logic. This approach shares many similarities to the functional HDLs that encode state in the interface of a function. However, TRS faces a unique challenge because the technique introduces the possibility of conflicts during state updates. Two rules may fire and try to update the same register which means the compiler must be sophisticated enough to detect conflicts and insert arbitration logic when possible. This issue is compounded when considering the modular composition of rules. The compiler could schedule the rules for each module separately, or it could lift the rules into a single top-level module which is then scheduled as a single unit. Prior work has shown that both approaches could be viable depending on the input design [36]. Minimizing the overhead of the compiler generated logic for scheduling rewrite rules is the key challenge for applying term rewriting systems as a register-transfer abstraction.

Devising abstractions for FSMs is another essential design problem for RTL languages. For example, Verilog provides abstractions that enable logic synthesis to generate optimized implementations of FSMs. A key issue for FSM synthesis is choosing the best representation for state. Consider a design where the state of an FSM is being consumed by a circuit performing an arithmetic operation. In this case, using a non-binary state encoding would require the insertion of decode logic. On the other hand, if the target platform is an FPGA, a one-hot state encoding often maps more efficiently to a lookup table architecture. As shown in Figure 2, Verilog designers can use *parameters* to abstract the encoding of the state. Control logic dispatches on the abstract parameters, and the concrete parameter values can be easily changed or selected by an automated tool.

Related to the abstraction of state encoding is the synthesis of control logic for the FSM. The canonical pattern for describing a Verilog FSM, shown in Figure 2, uses a `case` statement that dispatches on a state variable. The semantics of the Verilog `case` statement introduces complexity for the synthesis tool because cases are not necessarily mutually exclusive. Furthermore, the tool must also synthesize logic to handle behavior for cases that have not been listed. For example, consider a binary encoded FSM with 12 states. The state will be stored in a 4-bit quantity that is used to dispatch a `case` statement. Without guidance, the synthesis tool must be sophisticated enough to prove that there are only 12 possible values of the 4-bit quantity, otherwise it must insert extra logic to handle the illegal values. The SystemVerilog language avoids this issue by introducing the `unique` qualifier for indicating that all legal cases have been listed and are mutually exclusive.

One major issue in the canonical design pattern for Verilog FSMs is that for large FSMs with complex transitions, the description exhibits a serious lack of structure. For example, a simple SDRAM controller in Verilog contains 25 states with the entire FSM transition behavior defined in a single `case` statement [21]. Understanding the code requires the reader to follow large jumps between arbitrary cases. This design pattern incurs a significant cognitive load on the designer who must manage a large amount of complex temporal behavior to read the code. There is a direct relation between the use of large case statements to the use of `goto` statements, and we consider this design pattern to be similarly harmful [19].

It is important to remark that this program structuring issue is not restricted to the description of hardware FSMs, but is in fact an instance of a more general problem for imperative languages. Fortunately, the software community has found a promising solution based on coroutines [10]. Recent work is investigating the application of this technique to hardware FSMs by restricting the semantics of a coroutine so it can be precisely compiled to a circuit [60]. The main challenge is restricting the coroutine semantics to be only able to describe FSMs while still enabling the use of coroutine composition.

The essence of this technique is to augment the semantics of the Verilog `always` block to describe a coroutine. The designer may suspend the procedure in arbitrary locations to incorporate more structure in the code. For example, the sequencing of two states can be achieved by separating the logic with a `yield` statement, and the looping of a state can be described using a `while` loop containing a `yield`. Compare this to the `case` statement pattern where the structure of sequences and loops are not explicit in the flat list of cases.

A related issue is the description of the sequential composition of FSMs. Using just the circuit abstraction composed with a basic RTL abstraction, the sequential composition of distinct FSM circuits is achieved by using wires between the two circuits. These wires are used to relay signals indicating that an FSM should start or that an FSM has ended. In order to abstract away these wires and the accompanying control logic, the creators of Lava developed the Pace [13] language. Blarney, a modern variant Lava, provides a similar concept in the form of *Recipes* [44]. The developers of Bluespec also created similar language called STMTFSM [48]. Underlying all these languages is a notion of modular, sequential composition of program fragments. Early work on the Silica language [60] is exploring the use of coroutine composition as another abstraction for modular, sequential composition.

3.1.3 Virtual Machine Abstraction

The *virtual machine* abstraction models hardware as a set of instructions for an abstract machine. This technique hides certain details found in lower levels of abstraction such as the finiteness of resources. For example, the C language provides an abstraction over a virtual machine that can store an infinite number of variables. In order to synthesize a

hardware implementation of a C program, the compiler must perform a variant of register allocation that maps variables to registers depending on a set of constraints provided by the user. Historically, this approach has been mostly applied to the synthesis of hardware from traditional software languages such as C, but an emerging body of work is exploring the application of this technique to software DSLs.

The major advantage of this approach is that it greatly improves the productivity of the user by enabling them to design hardware as if they were developing software. However, in practice, the user is required to have deep knowledge of the hardware they are trying to generate in order to achieve the desired performance [46]. This negates much of the advantage of using a traditional software language because instead of simply reasoning about a software program, the user must break the virtual machine abstraction and reason about how the program will be mapped to hardware. The central challenge for designing languages based on the virtual machine abstraction is to find what aspects of hardware can be abstracted in order to improve productivity without weakening the performance of the compiler.

The problem of compiling a general software program into hardware shares many of qualities found in the problem of automatically parallelizing a general software program. Fortunately, recent work has mirrored the domain of parallel computing by leveraging DSLs to facilitate better compiler mappings to hardware. DSLs are able to provide productivity and performance by leveraging domain-specific abstractions.

For example, recent work on Halide, a DSL for high performance array and image processing code, extended the compiler to support hardware synthesis by introducing directives for hardware-specific optimizations [53]. This approach maintains the virtual machine abstraction for describing image processing algorithms while giving the user a level control over how the compiler synthesizes hardware. While this technique still requires the user to think about the hardware they are designing, it avoids changing the original source input in order to effect change in the compiler output. The main advantage of this approach is that the user can achieve the desired synthesis results by guiding the compiler rather than relying on optimizations based on heuristics.

The Spatial [38] DSL takes another approach to simplifying the problem by introducing a virtual machine abstraction with an alternative design for memory. Instead of using the uniformly accessible address space abstraction presented by standard CPUs, Spatial programs explicitly interact with the memory hierarchy. This design choice is motivated by the fact that a major challenge for compiling a general software program to hardware is determining an optimal memory architecture. Spatial allows the user to explicitly set a memory architecture using a template while still leveraging the compiler to schedule other aspects of the computation. Spatial is an example of discarding the traditional virtual machine abstraction used by most modern software languages and replacing it with a new abstraction that is tailored to the problem of hardware compilation.

Underlying both the Halide and Spatial approaches is a technique that involves identifying a key problem for the compiler and developing a language abstraction to simplify this problem. Moving forward, researchers interested in developing HDLs based on a virtual machine abstraction should explore techniques that balance the productivity of the user with the quality of the hardware synthesized by the compiler.

3.2 Compiler Infrastructure

The proliferation of software languages based on LLVM [40] demonstrates the value of shared compiler infrastructure for both industrial and academic purposes. For researchers, LLVM provides a means for rapidly prototyping languages without having to implement

standard compiler passes or create backends for standard architectures. Efforts to develop common infrastructure for hardware compilers are underway and have elucidated key issues when compared to their software compiler counterparts [34, 15]. There is a clear need for the development of hardware-specific compiler passes. Optimization passes are of critical importance because new ideas in HDLs will see no practical use unless they can be compiled into high performance implementations.

While some standard compiler passes, such as constant folding, can be directly applied to HDLs, there exists an entire class of passes that are specific to hardware. As an example, hardware computations are always predicated in the sense that if a computation is mapped to physical components, those components will be continuously executing. Conditional logic is implemented using multiplexers on the flow of data. In order to simplify the lowering of conditional logic, programs can be rewritten into single-static assignment form. In software, leaving it in this form would incur a cost because instructions in a non-traversed branch would always be executed. However, this is already the case for hardware, so leaving the program description in this form incurs no cost. In fact, this simplifies the synthesis stage of the compiler by enabling a one-to-one mapping from phi nodes to multiplexers. The key challenge facing researchers interested in developing HDL compiler infrastructure will be devising reusable, hardware-specific analysis and transformation passes.

Another impediment to design productivity is the development of software compilers that target a novel architecture. For example, developing an extension to the RISC-V ISA [3, 64] would require extending an existing compiler backend to target the new instructions. This means that hardware design teams must include compiler experts, which in turn indicates a need for the ability to automatically synthesize compiler backends for new hardware architectures. The Tensilica processor generator [25] demonstrated the feasibility of automatically generating a compiler that targets new instructions. However, this capability required that the user conform to a fixed processor architecture. Future work should explore extending this technique to support the extension of a broader class of architectures.

Given that many software compilers have converged on the ISA abstraction for backend targets, it is essential that the HDL community converge on an ISA specification language that is machine readable. Convergence would allow researchers to experiment with automatically synthesizing compiler backends for a new ISA described using a standard input format. ISP [7] is an older example of a processor specification language that could describe ISAs. Early stage research on the Peak language [28] is exploring the use of `smt-lib` [9] to develop a modern variant of ISP with formal semantics.

Finally, a critical issue facing hardware compiler developers is performance of the compiler itself. For example, a recent paper [37] touting a new methodology for high productivity hardware design reported that compiling their RTL design to an integrated circuit layout took *only* 12 hours. This is an obvious bottleneck in the design space exploration process. Their technique for reducing the runtime of the compiler was based on reducing the complexity of place and route, a stage in the compiler where logical components of the design are placed into physical space. Optimizing the place and route phase of hardware compilers is just one opportunity for researchers interested in improving the runtime performance of HDL compilers.

3.3 Formal Methods

Programming languages have long enjoyed an abundance of elegant theories that form the basis of useful formal methods. In the style of Grothendieck [42], researchers working on foundational theories have created a sea of techniques for developing practical solutions to

difficult problems. The recent development and proliferation of WebAssembly [26] demonstrates the utility of designing a new language with formal specification in mind. Rather than face the challenge of retrofitting formal methods to old language designs, researchers in this new golden age of HDLs should leverage the opportunity to develop new languages specifically designed for the application of advanced formal methods.

3.3.1 Execution Semantics

Formalizing the execution semantics of an HDL is essential requirement for the integration with formal tools such as model checkers [14]. The major challenge is capturing the intrinsic concurrency and parallelism in hardware. Process calculi [5], specifically with a notion of time [6] present one approach. Real and discrete time could be used to describe the semantics of analog and digital circuits respectively. One issue is the integration of real and discrete time for the modeling of mixed-signal circuits. A similar issue is the modeling of synchronous and asynchronous digital logic. Communicating sequential processes [31] are one technique that have been applied to modeling of asynchronous circuits [62].

The existence of the circuit abstraction as an expressively complete primitive is reminiscent of function abstraction from the domain of software languages. This raises the question as to whether a core calculus can be constructed that captures the execution semantics of the circuit abstraction in the same way that the lambda calculus [8] captures the semantics of function abstraction. While function abstraction presents a compelling basis for the development of this calculus, there are two key issues that must be addressed: circuits can hold state and must have finite size. Contrast this with the basic definition of function abstraction which can be used to describe infinite computation through recursion. A type system can be used to enforce the finiteness of computation [49], which leaves the issue of managing state. Section 2.3 discusses two techniques for encoding state in a functional HDL. One of the key challenges with using these techniques is that it restricts the language to describing synchronous circuits. An essential contribution to the community will be the development of a state encoding mechanism that can capture both synchronous and asynchronous logic.

3.3.2 Type Systems

The fact that Verilog is the dominant HDL indicates that the hardware community has not enjoyed the benefits of the latest advances in type systems. The consequences of this is demonstrated by the fact that the ARM Advanced Peripheral Bus (APB) interface [41] uses special prefixes in port names to indicate that they are part of the protocol. This requires users to manage interface connections using name matching, which is considerably less safe than what embedding this in the type system could offer.

One major issue is that the Verilog type system does not provide a concept of algebraic data types. Introducing the concept of a finite size product type would enable the APB interface specification to be defined as a tuple or record type rather than using a naming convention. The use of a product type offers the same benefits to HDL designers as it does to software developers. They are also an example of an *abstraction without overhead* [61] because they can be compiled out of a design by flattening the types into their leaf elements. Compare this to sum types which would require inserting extra logic into the generated design to distinguish between variants. Despite this cost, sum types still provide the same useful static guarantees as they do for software. They also provide a mechanism for abstracting away the details of the control logic, which creates an opportunity for the compiler to synthesize an optimized implementation.

Another interesting avenue of research is the application of behavioral types [2], specifically session types [33], to hardware interfaces. Hardware communication protocols exhibit many of the same characteristics as the software protocols that session types have already been applied to. The core problem will be weaving the session type semantics into HDL execution semantics. Researchers interested in this problem should consider how the domain of hardware protocols differ from more general software protocols, with the intention of finding opportunities to make the problem simpler. One crucial aspect of hardware protocols is the use of *magic* bit patterns to encode portions of the protocol. More generally, hardware protocols can involve data-dependent communication. This reveals an opportunity for the application of dependent type techniques to specify properties on the values of data moving through an interface. While this is a generally difficult problem, restricting the domain to hardware protocols might provide opportunities for practical applications of these ideas.

4 Vision: A Multi-Language System for Hardware Construction

A golden age of HDLs presents an opportunity to experiment with alternative HDL designs. This section presents a vision for a multi-language system where a meta-programmed host language is used to implement embedded DSLs for hardware construction. The multi-language approach is directly inspired by Lua/Terra [17, 16, 18], a two-language system that integrates a statically typed, low-level programming language with a dynamically typed, high-level language. Much like software development, hardware design involves components written in multiple languages. For example, a software model of a specific component could be implemented in C and used by a test written in Verilog. Furthermore, the hardware implementation of the module may be defined using Verilog generated by a Perl meta-program.

Rather than treat an HDL as a standalone language like Verilog, this vision adopts the approach of embedding an HDL in a general purpose programming language. This vision is based on a core structural embedded DSL that is used as a common compilation target for a sea of DSLs each targeting various aspects of the hardware design process. Unifying all aspects of the hardware design process into subsets of the same language reduces the cognitive load on the hardware designer. They are only required to learn a single syntax and integration via embedding enables DSLs to be composed without requiring glue code.

4.1 Meta-programmed Host Language

The vision of this multi-language system is based on a meta-programmed host language. The extent to which it may be meta-programmed must enable the implementation of rich DSLs. For example, Magma [27] uses Python's metaclass features to embed a circuit abstraction, and Silica [60] inspects the Python AST to compile coroutines to hardware finite-state machines. The implementation of these DSLs require language support for meta-programming that is much richer than simple preprocessing. A side-effect of this requirement is that the meta-programming features of the host language will become meta-programming features for the embedded DSLs.

A standard multi-stage programming approach is sufficient for supporting the flexible code generation required for implementing hardware generators [51]. A hardware *generator* is a program that consumes a set of parameters and produces an instance of a hardware design [54]. Embedding an HDL in a meta-programmed host language enables hardware generators to be implemented using standard meta-programming techniques. Applying the technique of multi-stage programming to hardware generators is somewhat easier than in the software domain because, in practice, the interaction between the meta-language and

the HDL is one directional. Compare this to Lua/Terra where control can be transferred between both languages. In hardware generators, meta-programs construct fragments of HDL programs, but the generated HDL code does not typically invoke code in the host language. In theory this might be possible given a reconfigurable hardware system, but in practice this is limited by the slow performance of hardware compilers. Improving compiler performance could enable the construction of JIT compiler systems for hardware, which could then leverage two way interaction between generator code and generated hardware.

One key issue is whether the host language is statically or dynamically typed. The major trade-off is between productivity and correctness. A statically typed host language would provide increased safety, which could be viable for large, complex systems. However, a dynamically typed language would promote rapid design space exploration and provide the flexibility required for more complex hardware generators. For example, generators in dynamically typed languages can employ dynamically constructed types. Another important issue is the cognitive load placed on the user by the type system. Hardware designers are not experts in software engineering and therefore stand to benefit greatly from a type system that is simple and easy to understand. Furthermore, the correctness of the generated hardware design is of greater importance than the correctness of the generators used to construct the design fragments. These requirements suggest that a system based on a dynamically typed host language such as Python composed with a statically typed embedded DSL presents a compelling solution that balances productivity and correctness. A ubiquitous language like Python has the added benefit that many hardware engineers are likely to have already encountered it for scripting purposes. Compare this to a language like Scala; while it provides many compelling language features for building DSLs, it is highly unlikely that a hardware engineer has encountered the language in their schooling or professional work.

4.2 The Core Structural DSL

After the host language, the second essential ingredient of the vision is an embedded DSL that provides a structural circuit abstraction. The definition of this core DSL should be simple and precise because many of the complexities of a traditional HDL will be offloaded to other DSLs or the host language. As discussed in Section 3.1.1, a circuit abstraction is expressively complete, which means that this structural DSL can serve as a common compiler target for all other languages in the system. This design enables the structural abstraction to be used to compose modules defined in different DSLs. This is achieved by performing staged execution where in the final stage, all program fragments have been compiled to the core DSL. During this final phase of execution, the user defines a program to structurally compose the various components. Magma [27] and Chisel [4] are concrete examples of embedded structural languages that could serve this purpose. An important requirement is that this core language be formally specified, which then provides a consistent basis for the formal specification of other DSLs in the system.

4.3 A Sea of Hardware DSLs

The combination of a host language with an embedded core structural HDL serves as the basis for the research and development of other DSLs to address the intellectual challenges discussed in Section 3. For example, recent work has used Magma [27] and Python as the basis for developing the Peak language [28] for specifying processing elements. Peak supports compilation to Magma, which can then be composed with other components written in other DSLs such as a Silica [60], a DSL for describing hardware finite-state machines using

coroutines. A major theme in this design is *separation of concerns through separation of languages*. That is, the description of different hardware components may benefit from being described using a different set of abstractions. If this is the case, these components can be implemented using different DSLs and composed through a well-defined structural interface.

The aforementioned DSLs address issues specific to the design of concrete hardware. This vision includes another class of DSLs that target accelerating specific applications. For example, a DSL based on Numpy [63] could be used to compile numerical computation algorithms into hardware circuits. While these languages should be designed primarily to provide application specific abstractions to the user, they should also be designed to interoperate with the core structural DSL. This would enable code written in application oriented DSLs to be integrated with libraries that generate harnesses for application accelerators. In this case, a library routine could instance the compiled version of the algorithm and wire it up to other components using the core structural DSL.

4.4 Verification

Because the host language is a general purpose programming language, it provides the necessary facilities for performing verification tasks. Underlying this will be a connection between general purpose code in the host language and circuits defined in the core structural DSL. Recent work on *fault* [59] has explored solutions to this by developing an embedded DSL that allows users to interact with circuits through a set of actions. A key advantage of this approach is that it enables verification components, such as random number generation, to be implemented as libraries in the host language. Also, the tests can be meta-programmed in the same fashion as the hardware, which reduces verification cost through more flexible testing infrastructure. Compare this approach to SystemVerilog, where the core language for describing hardware was extended with abstractions specifically for verification such as a class system and string data type. Overtime, this has resulted in feature creep and complexity in the SystemVerilog specification. This is another example of the vision's fundamental design pattern based on decoupling features that are not hardware specific from the HDL.

5 Conclusion

The PL community stands on the critical path to a new golden age of computer architecture. Fortunately, there is an abundance of intellectual challenges that indicate that we are on the cusp of a new golden age of HDLs. This paper develops a vision for a multi-language system for hardware construction that will provide the productivity gains required to induce this new golden age of computer architecture. This is an exciting time to be a researcher interested in PL and hardware.

References

1. *A 4 bit up-counter with synchronous active high reset*, 2009 (accessed April 5, 2019). URL: `https://www.edaplayground.com/x/3cf`.
2. Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral types in programming languages. *Foundations and Trends® in Programming Languages*, 3(2-3):95–230, 2016.
3. Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

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

5 Jos CM Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

6 Josephus Cornelis Maria Baeten and Cornelis Adam Middelburg. Process algebra with timing: real time and discrete time. In *Handbook of process algebra*, pages 627–684. Elsevier, 2001.

7 M Barbacci, C Gordon Bell, and Allen Newell. *ISP: A language to describe instruction sets and other register transfer systems*. Citeseer, 1972.

8 H.P. BARENDREGT. Chapter 1 - Introduction. In H.P. BARENDREGT, editor, *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*, pages 3–21. Elsevier, 1984. doi:10.1016/B978-0-444-87508-2.50009-5.

9 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

10 Eli Bendersky. Co-routines as an alternative to state machines. https://eli.thegreenplace.net/2009/08/29/co-routines-as-an-alternative-to-state-machines, 2009.

11 Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *ACM SIGPLAN Notices*, volume 34(1), pages 174–184. ACM, 1998.

12 Inc. Bluespec. *BSV 101: DESIGNING A COUNTER Using the Bluespec Development Workstation*, 2009 (accessed April 5, 2019). URL: http://wiki.bluespec.com/Home/Getting-Started/Tutorials.

13 K Claessen and M Sheeran. A slightly revised tutorial on lava: A hardware description and verification system, 2007.

14 Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

15 Ross Daly, Lenny Truong, and Pat Hanrahn. Invoking and Linking Generators from Multiple Hardware Languages using CoreIR. In *Proceedings of the 1st Workshop on Open-Source EDA Technology*, 2018.

16 Zachary DeVito. *Terra: Simplifying High-performance Programming Using Multi-stage Programming*. PhD thesis, Stanford University, 2014.

17 Zachary DeVito and Pat Hanrahan. The Design of Terra: Harnessing the best features of high-level and low-level languages. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

18 Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48(6), pages 105–116. ACM, 2013.

19 Edgar Dijkstra. *Edgar Dijkstra: Go To Statement Considered Harmful*, 1968 (accessed February 6, 2019). URL: https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf.

20 Carl Ebeling. *CSE370 - XV - Verilog for Finite State Machines (Spring 2010)*, 2010 (accessed April 5, 2019). URL: https://courses.cs.washington.edu/courses/cse370/10sp/pdfs/lectures/15-VerilogIIPrint.pdf.

21 Mike Field. *Simple SDRAM Controller (Verilog Memory controller v0.1)*, 2014 (accessed April 5, 2019). URL: http://hamsterworks.co.nz/mediawiki/index.php/File:Verilog_Memory_controller_v0.1.zip.

22 Frank Ghenassia et al. *Transaction-level modeling with SystemC*, volume 2. Springer, 2005.

23 Steve Golson and Leah Clark. Language Wars in the 21st Century: Verilog versus VHDL–Revisited. In *Synopsys Users Group (SNUG)*, 2016.

24 Steve Golson, Gardner Hendrie, and Philip Moorby. *Moorby, Phil (Philip Raymond) oral history*, 2013 (accessed April 5, 2019). URL: `https://www.computerhistory.org/collections/catalog/102746653`.

25 Ricardo E Gonzalez. Xtensa: A configurable and extensible processor. *IEEE micro*, 20(2):60–70, 2000.

26 Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Notices*, volume 52(6), pages 185–200. ACM, 2017.

27 Pat Hanrahan. *magma*, 2019 (accessed February 6, 2019). URL: `https://github.com/phanrahan/magma`.

28 Pat Hanrahan. *peak*, 2019 (accessed February 6, 2019). URL: `https://github.com/phanrahan/peak`.

29 John Hennessy and David Patterson. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–29, June 2018. `doi:10.1109/ISCA.2018.00011`.

30 John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

31 Charles Antony Richard Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.

32 James C Hoe et al. Hardware synthesis from term rewriting systems. In *VLSI: Systems on a chip*, pages 595–619. Springer, 2000.

33 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

34 Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 209–216. IEEE Press, 2017.

35 Steven Dexter Johnson. *Synthesis of digital designs from recursion equations.* PhD thesis, Indiana University, 1983.

36 Michal Karczmarek et al. *Synthesis of multi-cycle circuits from guarded atomic actions.* PhD thesis, Massachusetts Institute of Technology, 2011.

37 Brucek Khailany, Evgeni Khmer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam (Likun) Xi, Yanqing Zhang, and Brian Zimmer. A Modular Digital VLSI Flow for High-productivity SoC Design. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 72:1–72:6, New York, NY, USA, 2018. ACM. `doi:10.1145/3195970.3199846`.

38 David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: a language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311. ACM, 2018.

39 M. Kooijman. Haskell as a higher order structural hardware description language, December 2009. URL: `http://essay.utwente.nl/59381/`.

40 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

41 Arm Limited. *Arm AMBA (Advanced Microcontroller Bus Architecture) Protocols*, 2019 (accessed February 6, 2019). URL: `https://developer.arm.com/products/architecture/system-architectures/amba`.

42 Colin McLarty. The Rising Sea: Grothendieck on simplicity and generality, 2007.

43 G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, September 1955. `doi:10.1002/j.1538-7305.1955.tb03788.x`.

44 mn416. *Blarney – Example 7: Recipes*, 2019 (accessed April 6, 2019). URL: `https://github.com/mn416/blarney#example-7-recipes`.

45 Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

46 Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

47 Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

48 Rishiyur S Nikhil and Kathy R Czeck. *BSV by Example.* Createspace Independent Publishing Platform, 2010.

49 Russell O'Connor. Simplicity: a new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120. ACM, 2017.

50 John O'Donnell. Hydra: hardware description in a functional language using recursion equations and high order combining forms. *The Fusion of Hardware Design and Verification*, pages 309–328, 1988.

51 Gordon J Pace and Christian Tabone. Multi-Stage Languages in Hardware Design, 2008.

52 Gordon D Plotkin. A structural approach to operational semantics, 1981.

53 Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):26, 2017.

54 Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, et al. Rethinking digital design: Why design must change. *IEEE micro*, 30(6):9–24, 2010.

55 Mary Sheeran. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 104–112. ACM, 1984.

56 Satnam Singh and Philip James-Roxby. Lava and JBits: From HDL to bitstream in seconds. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pages 91–100. IEEE, 2001.

57 IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, February 2018. `doi:10.1109/IEEESTD.2018.8299595`.

58 Paul Teehan, Mark Greenstreet, and Guy Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design & Test of Computers*, 24(5), 2007.

59 Lenny Truong. *fautl*, 2019 (accessed February 6, 2019). URL: `https://github.com/leonardt/fault`.

60 Lenny Truong. *silica*, 2019 (accessed February 6, 2019). URL: `https://github.com/leonardt/silica`.

61 Aaron Turon. *Abstraction without overhead: traits in Rust*, 2015 (accessed April 5, 2019). URL: `https://blog.rust-lang.org/2015/05/11/traits.html`.

62 Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of the conference on European design automation*, pages 384–389. IEEE Computer Society Press, 1991.

63 S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. doi:10.1109/MCSE.2011.37.

64 Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 2014.

65 Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*, pages 99–112. Springer, 2008.

Version Control Is for Your Data Too

Gowtham Kaki
Purdue University, West Lafayette, USA
https://gowthamk.github.io
gkaki@purdue.edu

KC Sivaramakrishnan
IIT Madras, Chennai, India
kcsrk@iitm.ac.in

Suresh Jagannathan
Purdue University, West Lafayette, USA
suresh@cs.purdue.edu

Abstract

Programmers regularly use distributed version control systems (DVCS) such as Git to facilitate collaborative software development. The primary purpose of a DVCS is to maintain integrity of source code in the presence of concurrent, possibly conflicting edits from collaborators. In addition to safely merging concurrent non-conflicting edits, a DVCS extensively tracks source code provenance to help programmers contextualize and resolve conflicts. Provenance also facilitates debugging by letting programmers see diffs between versions and quickly find those edits that introduced the offending conflict (e.g., via `git blame`).

In this paper, we posit that analogous workflows to collaborative software development also arise in distributed software execution; we argue that the characteristics that make a DVCS an ideal fit for the former also make it an ideal fit for the latter. Building on this observation, we propose a distributed programming model, called CARMOT that views distributed shared state as an entity evolving in time, manifested as a sequence of persistent versions, and relies on an explicitly defined *merge semantics* to reconcile concurrent conflicting versions. We show examples demonstrating how CARMOT simplifies distributed programming, while also enabling novel workflows integral to modern applications such as blockchains. We also describe a prototype implementation of CARMOT that we use to evaluate its practicality.

2012 ACM Subject Classification Computing methodologies → Distributed programming languages; Software and its engineering → Software configuration management and version control systems; Software and its engineering → API languages

Keywords and phrases replication, distributed systems, version control

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.8

Acknowledgements We would like to thank Thomas Gazagnaire, Romain Calascibetta, and Zack Shipko for patiently assisting us with Irmin and OCaml-Git libraries. This material is based upon work supported by the National Science Foundation under Grant No. CCF-SHF 1717741 and the Air Force Research Lab under Grant No. FA8750-17-1-0006.

© Gowtham Kaki, KC Sivaramakrishan, and Suresh Jagannathan;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 8; pp. 8:1–8:18

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Building distributed applications is hard. The crux of the problem is the management of concurrent updates to distributed shared state that maintain user-level invariants and properties. The problem is especially pronounced in the context of modern data-intensive applications, which replicate large amounts of data across geographically diverse locations to enable trust decentralization, guarantee low latency access to state, and provide high availability even in the face of node and network failures. In general, these systems allow each replica instance of a distributed application to concurrently accept updates to shared data, which could potentially conflict with other updates, and consequently violate data integrity. In addition, transient faults in the underlying network, such as network partitions, message reorderings etc., can yield counter-intuitive anomalous executions that are hard to predict and even harder to preempt [9, 25, 15]. While conventional concurrency control criteria, such as linearizability and serializability, are designed to preclude such executions, applications are reluctant to impose them given their prohibitive cost in a distributed setting (an observation succinctly captured in the CAP theorem [12]). Instead, applications mostly operate within the weak guarantees provided by an asynchronous state replication model, occasionally resorting to stronger forms of concurrency control for "risky" operations, using *ad hoc* and error-prone reasoning to distinguish between them. Another available alternative is to restructure applications around a library of distributed data structures that are carefully designed by experts and proven to be correct under an asynchronous distributed setting[1]. Such re-engineering, however, may not always be feasible since applications often use bespoke data structures to serve specific needs. Consequently, the cognitive overhead in building distributed applications remains high, limiting program development to distributed system experts.

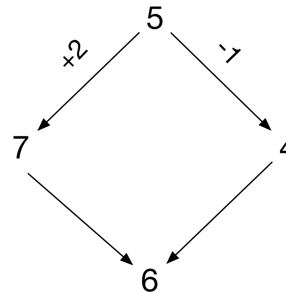
It may therefore come as a surprise to note that programmers, as humans, are exposed to the complex realities of distributed computing almost on a daily basis, and they seem to be doing just fine. Almost all programmers these days use some form of a distributed version control system (DVCS), such as Git or Mercurial, and when doing so, emulate the logic of a distributed application consciously or otherwise. A DVCS lets a programmer safely *merge* concurrent *versions* of source code created by her collaborators working independently and concurrently, fork-off her own *branch* from any existing version to work in isolation, group together multiple related changes as a single *commit* to be pushed to a *remote*, look at the provenance information to understand how the code has evolved across multiple versions, and track which collaborator is responsible for which piece of code. The overwhelming adoption of DVCS as a paradigm for distributed and collaborative software development is indicative of the utility of the model it supports. Much of its attractiveness stems from the intuitive mental picture it offers the developer to reason about the integrity of source code as it evolves in the face of concurrent modifications.

In this paper, we ask whether the benefits of DVCS can be transplanted to manage *data* in addition to source code. In particular, we propose the thesis that building safe distributed applications would be dramatically simpler if concurrent conflicting updates to application state are explicitly recognized and resolved, rather than preempted. Supporting this thesis requires addressing a number of challenging questions: (a) How can the provenance of data be systematically exploited to automate the resolution of merge conflicts? (b) Would the

[1] Analogous, for example, to carefully designed lock-free data structures in a concurrent programming language [24, 13].

```
module Counter: sig
  type t
  val zero: t
  val add: int -> t -> t
  val sub: int -> t -> t
  val read: t -> int
end = struct
  type t = int
  let zero = 0
  let add x v = v + x
  let sub  x v = v - x
  let read v = v
end
```

(a) `Counter` data type in OCaml.



(b) `Counter` merge visualized.

Figure 1 The `Counter` example.

ability to fork-off a version of the state, and group together multiple changes to the state as a single commit give useful transactional semantics? (c) Can we generate sufficiently high-level provenance data to serve as a transaction ledger and satisfy the auditing requirements of emerging applications such as those built on distributed blockchains ledgers? It is to explore the answers to these questions that we conceived CARMOT- a distributed programming model build around the same concepts as distributed version control systems.

At the core of CARMOT is the principle that any data structure that ascribes a well-defined merge semantics to merge its concurrent versions becomes a distributed data structure. Our experience with DVCS informs us that provenance information greatly helps in contextualizing merges and resolving conflicts. True to that spirit, CARMOT allows a data structure to define a merge semantics for its concurrent versions in the context of their *lowest common ancestor* (LCA) version, resulting in a three-way merge. Thus, any ordinary data structure equipped with a three-way merge function becomes eligible to be a distributed data structure. As we describe in Sec. 2, the simplicity of this criterion lets us build bespoke distributed data structures and develop applications around such data structures with a relative ease. We subsequently demonstrate how CARMOT can build on the branch-and-merge model of DVCS to define a transactional semantics with a well-defined isolation model at no additional cost. Lastly, we show that extensive provenance tracking, similar to a DVCS, helps CARMOT naturally express blockchain applications, which otherwise have to rely and specialized shim layer, such as the Hyperledger Fabric [2]. However, all the aforementioned benefits would amount to naught if CARMOT as a programming model cannot be realized in an asynchronous distributed setting. In Sec. 3 we describe a prototype implementation of CARMOT that sits atop Git that can actually run CARMOT distributed applications seamlessly with low overheads.

2 An Overview of Programming with Version Controlled Data

2.1 Version Control-Inspired Replication

Fig. 1a shows a simple counter data type in OCaml that admits additions and subtractions. Suppose Alice wants to use the counter in a distributed setting, meaning she wants to replicate the counter state across various machines, allow the state to be updated concurrently on each replica, and let the updates be propagated asynchronously to other (remote) replicas. One way she could achieve this is by maintaining a log of operations performed at each replica,

periodically flushing the log to other replicas either on demand or by default. A replica receiving a remote operation has to apply it on the local state to keep it consistent with the remote state. Operations may be received and applied at different replicas in different orders, but since additions and subtractions commute, the resultant counter state would *eventually* guaranteed to be the same on all the replicas. Indeed, this is how asynchronous replication is often implemented in distributed applications [11, 23, 22, 16, 25].

An alternative take on replication would be one where Alice views the counter state as data being managed by a version control system, and sees various replicas as her collaborators creating concurrent versions of the counter state. In order to fetch her collaborators' updates, she would then be obligated to write a `merge` function that reconciles concurrent versions of the counter in the context of their lowest common ancestor (LCA). An example merge scenario is shown in Fig. 1b, where counter versions (with values) 7 and 4 have evolved concurrently from the original version 5. Alice could observe that the concurrent versions represent an addition of 2 and a subtraction of 1 on the ancestor state, and hence may choose to reconcile them as a single operation performing an addition of 1 on 5, to compute the merged version as 6. Alice's merge logic could be generalized as the following `merge` function:

```
let merge lca v1 v2 =
   lca + (v1 - lca) + (v2 - lca)
```

Indeed, such `merge` function is what CARMOT requires to promote a counter data type to the status of a distributed data type.

The version control-inspired view of replication really stands out when Alice decides to add a `mult` operation to multiply the value of the counter. Perhaps Alice wants to use the counter to count the account balance in a banking application, and she needs `mult` to compute the interest on the balance. She defines `mult` straightforwardly:

```
let mult x v = x * v
```

The `mult` operation serves Alice well as long as she uses her counters on a single machine. In a distributed setting however, under the conventional model of replication, Alice may see unexpected results as `mult` doesn't commute with `add` and `sub`, thus yielding different counter states on different machines. In other words, Alice's counter implementation, which is still correct in a single machine setting, is no longer correct in a distributed setting. She is now forced to abandon her recent additions to `Counter`, and restructure her banking application to express interest addition in different terms, perhaps using `add` to specify the increase in balance. Version control-based replication on the other hand lets Alice continue using her latest counter implementation (with `mult`) in a distributed setting as the counter's merge semantics already capture the effect of interest computation in terms of an increase in balance.

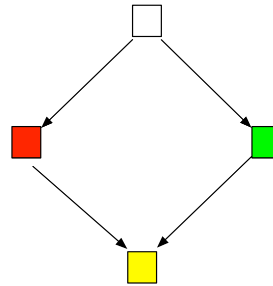
2.2 Application-specific Bespoke Merges

We provide further motivation using a more colorful data type - a pixel. We might write a pixel data type in OCaml as a triple with three fields (Fig. 2a), each standing for a red, blue and green color components respectively (following the rgb coloring scheme). `Pixel` supports a single operation - `set_color` that sets the color of the pixel to the given rgb values. A `get_color` function returns the color triple of the pixel.

Alice originally defined the pixel data type to use in her drawing board application she calls `Canvas`. She now wants to add collaborative drawing features to `Canvas`, and hence is interested in replicating the state of a pixel. Following the conventional model of replication, Alice sets up the pixel data type to asynchronously propagate `set_color` operations across

```
module Pixel: sig
  type t
  val white: t
  val set_color: (int*int*int) -> t -> t
  val get_color: t -> (int*int*int)
end = struct
  type t = {r:int; g:int; b:int}
  let white= {r=255; g=255; b=255}
  let set_color (x,y,z) _ = {r=x;g=y;b=z}
  let get_color {r;g;b} = (r,g,b)
end
```

(a) A `Pixel` data type in OCaml.



(b) `Pixel` merge visualized.

Figure 2 The `Pixel` example.

```
let merge lca ({r=r1;g=g1;b=b1} as v1) ({r=r2;g=g2;b=b2} as v2) =
  let mix (x,y) = min (x+y) 255 in
  if lca = v1 then v2
  else if lca = v2 then v1
  else {r=mix(r1,r2); g=mix(g1,g2); b=mix(b1,b2)}
```

Figure 3 Pixel merge function.

replicas, but quickly discovers that this leads to diverging pixel states across replicas. For instance, when Bob colors the pixel green on one replica, and Alice colors it red on a different replica, each's `set_color` operation may overwrite the other's on the remote replica, leading to diverging states. Unfortunately, unlike the previous example, Alice doesn't know how to redefine the `set_color` operation or restructure the `Canvas` application to solve the problem of diverging states. Nor can she find an appropriate consistency model [9, 26] weaker than linearizability that preempts such anomalous executions even if it comes at some expense. Alice is therefore stuck.

With version control-based replication, however, Alice starts with the assumption that her collaborators could create concurrent versions of the pixel state, and specifies the logic to merge such versions under the context of their LCA version (Fig. 3). She could, for example, reconcile the concurrent updates to the pixel color by using an additive color mixing scheme to mix the colors (Fig. 2b). When there are no concurrent updates, i.e., when at least one of the two versions is same as the ancestor version (thus causally preceding the other version), the successor version trivially becomes the result of the merge.

2.3 Transactional Semantics

Because version control-based replication lets Alice promote her pixel type to a distributed data type, she moves ahead with the development of her collaborative drawing app - `Canvas`. She defines a canvas type as a two-dimensional composition of pixels:

```
type canvas = Pixel.t list list
```

Alice initially emulates free-hand drawing by coloring individual pixels via `Pixel.set_color`, but soon realizes that it would be nice to have a few basic shapes, such as a circle or a rectangle, that she could draw in a single stroke. The functionality requires several pixels to be colored atomically and in isolation; atomically because Alice's collaborators should only see her draw a full circle or a rectangle rather than coloring an assortment of pixels, and isolation because Alice would like to draw the circle completely before she deals with

conflicting writes from her collaborators[2]. In other words, the application needs transactional semantics. With version control-based replication , Alice gets that for free. An atomic action, such as drawing a basic shape, can be performed on Alice's local version of canvas as a series of `set_color` operations on pixels making up the basic shape. Only the resulting canvas version is committed and pushed to Alice's collaborators, effectively enforcing the atomicity of writes. Furthermore, Alice wouldn't "pull" her collaborator's updates while her basic shape is in progress, thus guaranteeing the isolation of basic shape drawing operation.

2.4 Distributed Ledger

An important benefit of the CARMOT programming model is its inherent support for provenance. Such support is critical in applications where some form of consensus is required. In this applications, divergent states found on different replicas can be merged to a convergent (consensus) value based on application semantics, A particularly noteworthy instance of such applications is a distributed ledger.

As typically conceived, a distributed ledger is a safety-critical distributed data type that requires every node to maintain an untamperable ledger of operations that were ever performed on the shared state it represents. The ledger is colloquially called a blockchain due to its organization in terms of a sequence of blocks, where each block refers to its predecessor in the sequence. The untamperability of blocks is ensured by making the blocks content-addressable, i.e., by making the address of the block depend on its contents (e.g., an address could be the block's SHA1 hash as explained in the following section). Thus, tampering with a block results in the construction of a new block with a different address that does not belong to the chain, hence leaving the chain unchanged. Provenance information available to programmers in a version control-based replication model allows us to build untamperable distributed ledgers (blockchains) effortlessly, as we shall demonstrate through an extensive example.

2.4.1 Blockchain Preliminaries

At a high level, a blockchain represents a distributed bank-like application involving multiple *peers*, each maintaining a replica of a shared ledger of transactions. The ledger represents some form of consensus among the peers about the set of valid transactions performed thus far (since the beginning of time), and their relative order. Note that the distributed ledger is the *only* source of truth in a blockchain application; all relevant information (e.g., the account balances (in a banking application) is computed by reading the contents of the ledger. A transaction in a blockchain application transfers something of value between anonymous users. The thing of value could be a Bitcoin, but we simply refer to it as money in this discussion. If a transaction makes its way into the ledger, it is said to have been *confirmed*, meaning that there is a consensus that the transaction is valid, i.e., it does not engage in illegitimate behaviors, such as double spending available money. A transaction, when submitted, is initially unconfirmed, and the application maintains a set of such yet-unconfirmed transactions, which is also replicated across peers. Simply put, the task of a peer is to pick a yet-unconfirmed transaction, validate it, and if it is deemed to be valid, confirm it by adding to the distributed ledger. However, doing so in an uncontrolled fashion in a

[2] Alice could, for example, define a `merge` function for canvas that removes or retains an entire basic shape in case of a conflicting write. For pixels not part of a basic shape, she could default to the `Pixel.merge` function.

```
type txn = {timestamp: float;
            sender: pub_key;
            receiver: pub_key;
            amount: int}
```

```
type block = {txns: txn set;
              timestamp: float;
              proof:int64}

type t = {txns: txn set;
          chain: block list}
```

Figure 4 Blockchain type definitions in OCaml.

large system of peers (e.g., Bitcoin) leads to the divergence of the ledger state across replicas, resulting in a disagreement among peers about the contents of the ledger, and the consequent confirmation of invalid transactions. To prevent this, blockchain applications define a *soft* consensus protocol that limits the rate at which transactions can be appended to the ledger, and specifies how to resolve conflicts in case there are competing versions of the ledger. Most public blockchains, such as Bitcoin and Ethereum, employ *proof-of-work* as the rate limiting mechanism, where a peer has to solve a computationally hard problem to earn the right to append to the ledger (and a financial reward for *mining* the solution). The solution to this hard problem is also appended to the ledger to let other peers verify the solution. Unlike computing proof-of-work, verification is expected to be easy, i.e., the computational problem should ideally be NP-complete. In practice, various kinds of problems are used, whose description is out of scope for this paper. The problem we choose for this example is explained below. To ameliorate transaction confirmation latency, a peer which earns the right to append to the ledger is allowed to append a *block* of transactions, confirming them all in a single action. The new block links to the previous block, thus making the ledger a blockchain.

Note that proof-of-work is only a rate limiting mechanism; it makes concurrent appends to the ledger unlikely, but not impossible. Occasionally, when two peers simultaneously compute a proof-of-work, they both append their respective blocks to the ledger, resulting in a fork. Peers aware of only one of the two forked versions continue to mine and append blocks to their respective versions. At some point, if a peer becomes aware of two competing versions, it chooses one version over the other based on a predefined set of heuristics. For instance, if there are two competing chains of Bitcoin ledgers, then the longer chain is chosen as it represents more work confirming large number of transactions than the shorter one. The transactions of the shorter chain that do not belong to the longer chain are re-added to the pool of unconfirmed transactions and need to be confirmed again by a mining peer[3].

2.4.2 A Blockchain Application in OCaml

Having introduced a blockchain's conceptual underpinnings, we now describe how we can support its necessary functionality in a version control-based programming model. Fig. 4 shows the type definitions needed to build a simple blockchain application in OCaml. A transaction is simply a record documenting the transfer of a given amount of money from a sender to a receiver, both identified only by their public key. As mentioned earlier, a blockchain ledger is a chain of blocks, where each block consists of a set of transactions being confirmed, and a proof-of-work justifying the block's presence in the chain. Application state is defined via type `t`, which is a record containing a set `txns` of as-of-yet-unconfirmed transactions, and a list of blocks named `chain`, which is the blockchain.

[3] It is thus possible for a confirmed transaction to become unconfirmed again. Bitcoin therefore defines *number of confirmations* of a transaction based on how many blocks deep the transaction is inside the ledger. The greater the number of confirmations, the deeper the transaction sits inside the ledger, and the less likely it is to become unconfirmed again.

```
let mine_block my_key t =
  let valid_txns =
    filter_valid_txns t.txns in
  let last_block = hd t.chain in
  let last_proof =
      last_block.proof in
  let proof = proof_of_work
                last_proof in
  let ts = gettimeofday () in
  let reward_txn =
        {timestamp=ts;
         sender=genesis_key;
         receiver=my_key;
         amount=25;} in
  let block =
      {txns=Set.add reward_txn
                valid_txns;
       timestamp=ts;
       proof=proof} in
  let txns' = Set.diff t.txns
              valid_txns in
  let chain' = block::t.chain in
  {txns=txns'; chain=chain'}

let valid_txn chain txn = ...
```

```
let filter_valid_txns =
  Set.filter
    (valid_txn t.chain)

let valid_proof last_proof proof =
  let str1 = Int64.to_string
               last_proof in
  let str2 = Int64.to_string
                 proof in
  let str = str1^str2 in
  let hex = SHA1.to_hex @@
      SHA1.digest_string str in
  String.sub hex 0 3 = "000"

let proof_of_work last_proof =
  let rec loop_iter i =
    if i >= Int64.max_int
    then failwith "No proof!"
    else
      if valid_proof last_proof i
      then i
      else loop_iter (i + 1) in
  loop_iter 0
```

Figure 5 `mine_block` - a function that creates a new block, mines a proof-of-work, and adds it to the chain. Other relevant functions are also shown.

Operations can be defined that map one application state to another. The `new_txn` operation, for instance, creates a new transaction with the given user keys and the amount, and adds it to the pool of unconfirmed transactions. Its type is as shown below:

```
val new_txn: pub_key -> pub_key -> int -> t -> t
```

The second, and most important operation, is `mine_block` that composes a new block using the available pool of unconfirmed transactions, mines a proof-of-work, and adds the block to the chain. Fig. 5 shows the (abridged) code. The function first filters the set of valid transactions from the available pool of unconfirmed transactions using the function `filter_valid_txns` (shown on the right), which in turn uses `valid_txn` predicate of type shown below (definition elided in Fig. 5):

```
val valid_txn: block list -> txn -> bool
```

Given a blockchain ledger and a transaction, the function returns true if and only if the transaction is legitimate under the ledger, i.e., if and only if the account balance of the sender, computed from the ledger, is enough to carry out the transaction. Once the set of valid transactions is computed, `mine_block` proceeds to confirm them by computing the proof-of-work necessary to create a new block. In this example, we define proof-of-work as a solution to p_{i+1} in the following equation, where p_i denotes the proof-of-work of the previous block (`last_block.proof` in Fig. 5), and $\cdot$ denotes the (string) concatenation operation:

$$\texttt{sha1}(p_i \cdot p_{i+1}) < 2^{148}$$

The function `valid_proof` implements the above check in a slightly different form, given a p_{i+1} (`proof`) and p_i (`last_proof`). Instead of checking if the hash is less than 2^{148}, it checks if the leading 3 hex digits of the 40-digit SHA1 hash are zero. Given that it is in

```
let merge old v1 v2 =
  let oldc = old.chain in
  let v1c = v1.chain in
  let v2c = v2.chain in
  let x = valid_extension
            oldc v1c in
  let y = valid_extension
            oldc v2c in
  match x,y with
  | None, Some new2 -> v2
  | Some _, None -> v1
  | Some _, Some _
    when v1=v2 -> v1
  | Some _, Some new2
    when len v1c > len v2c ->
    add (union new2 v2.txns) ~to:v1
  | Some new1, Some _
    when len v1c < len v2c ->
    add (union new1 v1.txns) ~to:v2
  | Some new1, Some new2 ->
    let add_prf a b = a+b.proof in
    let prf_sum1 =
      fold_left add_prf 0 new1 in
    let prf_sum2 =
      fold_left add_prf 0 new2 in
    if prf_sum1 > prf_sum2
    then add (union new2 v2.txns)
            ~to:v1
    else add (union new1 v1.txns)
            ~to:v1
  | None, None -> error()
```

```
let valid_extension oldc newc =
  try
    let newbs = get_prefix newc
                  ~suffix:oldc in
    let _ = fold_right
      (fun b chain ->
        if valid_block chain b
        then b::chain
        else raise Invalid_arg)
      newbs oldc in
    Some newbs
  with Invalid_arg _ -> None

let rec get_prefix v ~suffix:s =
  match v with
  | v when v = s -> []
  | x::xs when xs = s -> [x]
  | x::xs -> x::(get_prefix xs s)
  | [] -> raise (Invalid_arg)

let add (txns:txn set) ~to:t =
  Set.fold
    (fun txn t' ->
      let chn = t'.chain
      if confirmed_txn chn txn
      then t'
      else {t' with txns=
              Set.add txn t'.txns})
    txns t

let confirmed_txn chain txn = ...
```

Figure 6 `merge` function for a blockchain. Other relevant functions also shown.

general impossible to invert a SHA1 hash in polynomial time, we solve the above equation by painstakingly iterating through all possible values of p_{i+1}, which in this case is every 64-bit integer, until we find the solution. The corresponding logic is implemented by the function `proof_of_work`.

Once the proof-of-work is computed, `mine_block` has everything it needs to compose a new block and add it to the chain. To incentivize mining, blockchain protocols allow mining peers to add a special transaction rewarding themselves a fixed pre-determined value. The `reward_txn` in Fig. 5 denotes such a transaction, which uses a special "genesis user" as the source of such rewards. The transaction is added to the set of valid transactions, following which the new block is composed and added to the chain. The transactions that could not be validated (`txns'`) are left in the pool of unconfirmed transactions.

Beyond `new_txn` and `mine_block`, a third `init` operation need also be defined to initialize the blockchain with a special "genesis block" that bootstraps the ledger with initial money (subsequent to genesis, the only way to add money to the system is through mining rewards). We elide discussion of `init` to focus on other aspects relevant to this paper.

2.4.3 Merge function

Blockchain applications are, by definition, distributed with the state of the ledger replicated across all the participating peers. To make our blockchain application distributed under a version control-based replication model, we provide a three-way merge function that merges the competing versions of blockchains given their provenance information in form of their

lowest common ancestor (LCA) version. Fig. 6 shows the merge function. The merge strategy is based on the heuristic employed in Bitcoin to pick the longest valid chain among the competing chains. The merge function first checks that the two competing chains are indeed valid chains through the function `valid_extension`. The function ensures that the old chain (the LCA) is a suffix of the competing chain `newc`, and that each block in the newly added prefix is valid. Validity of a block is checked through the predicate `valid_block` (not shown), which in turn checks the `valid_txn` predicate for each transaction in the block. The valid prefix is then returned. If the chain `newc` is not a valid extension of the LCA, then `None` is returned. Note that `valid_prefix` uses the `get_prefix` function on lists (definition shown), which compares the lists for structural equality (e.g., `v = s`). While determining structural equality is in general expensive, it can be resolved in constant time for versions obtained through CARMOT; we elaborate on this point in Sec. 3.

A notable aspect of `valid_extension` is that it uses provenance information available as the LCA version to determine if the chain has been tampered with. If the new version has tampered with the chain, for example, by sliding a new transaction in to an older block, then the LCA version `oldc` is no longer the suffix of the new version, leading `valid_extension` to return `None`.

In the `merge` function, if `valid_extension` returns `None` for one of the two chains, then the other (valid) chain is returned as the result of the merge. If both chains are valid but are not equal, then the longer one is picked. In such case, the transactions newly confirmed in the shorted chain are no longer considered confirmed, and need to be re-added to the unconfirmed pool (along with the transactions in the unconfirmed pool accompanying the shorter chain). This is done via the `add` function (shown) that adds the new transactions from the shorter chain and the corresponding unconfirmed pool, that are not confirmed by the longer chain, to the unconfirmed pool accompanying the longer chain. The predicate `confirmed_txn` (definition elided) returns true if and only if the transaction `txn` is listed in the chain `chain`. If both the competing chains are of equal length, then `merge` picks the chain whose extension w.r.t `oldc` was "harder" to compute, where hardness is assumed to be proportional to the value of the proof. Finally, if both chains are invalid, then `merge` throws an error, but this case never occurs in practice as one the two chains is a local chain which is guaranteed to be valid by `mine_block`.

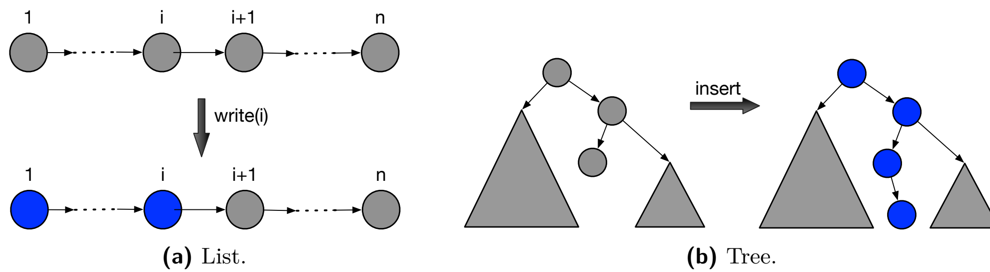
We have thus far presented various examples of how version control-based replication lets one build non-trivial distributed applications by defining merge semantics in the convenient form of a three-way merge function. To make such applications operational, however, we need a programming interface that lets developers take advantage of the version control model to define and compose distributed computations around applications. CARMOT is such a programming model, whose details are presented in the next section.

3 Realizing The CARMOT Programming Model

We have realized the CARMOT programming model on top a basic Git programming abstraction in OCaml [21]. In the following, we discuss the salient aspects of CARMOT's implementation, describe the API it exposes, and demonstrate how one can use CARMOT to orchestrate complex distributed computations.

3.1 Content Addressability & Sharing

One of the key aspects of CARMOT is its reliance on a content-addressable file system/memory to store arbitrary objects. We call it the CARMOT *store*. Like any other store, CARMOT lets one write an object, or a collection of linked objects to the store. However, unlike other stores,



(a) List. **(b)** Tree.

Figure 7 Linked data structures composed of content-addressable objects.

the address of an object in CARMOT store is its own SHA1 hash, which means mutating the object results in a new (version of) object being written to the store at a different address. This property lets CARMOT store multiple versions of linked data structures succinctly by sharing common objects, while simultaneously highlighting the *diff* between such versions. Fig. 7 illustrates.

Fig. 7a shows a linked list structure laid out on a content-addressable store. Each node is an object that stores some data and a link to the next object in the list, which is simply the latter's hash. For instance, a list object (call it A) could like the following:

```
{data = 24;
 next = "2aae6c35c94fcfb415dbe95f408b9ce91ee846ed"}
```

The value "2aa...6ed" is the SHA1 hash of the next object (call it B) in the list. If, for some reason, B's `data` is mutated, its hash value changes, making it necessary to update A's `next` field, which changing A's hash, and the update cascades. This scenario is depicted in Fig. 7a, where updating the i'th node in the list effectively creates a new version of the list with new objects for nodes 1 through i. The two versions of the list however share the common suffix containing nodes $i + 1$ through n. The diff and sharing between the two versions is thus clearly highlighted in this representation. Fig. 7b describes similar scenario for a tree structure, where inserting an element creates a new version of the tree that only slightly differs from the previous version in the store (The diff is highlighted). Such succinct representation of diff, and its easy computation, lets CARMOT efficiently support replicated data structures over a network. Moreover, content addressability lets CARMOT support constant-time structural equality checks by simply comparing hashes instead of iterating through data structures.

Readers familiar with functional programming may find Fig. 7 reminiscent of the persistence and sharing aspects of functional data structures. Indeed, hash-linked data structures (as described above) and functional data structures are similar in that respect. However, one notable difference is that the sharing in hash-linked data structures is based on the primitive notion of *common content*, rather than data dependence, or other similar programmatic notions. Thus, one could create OCaml lists `[3;2;1]` and `[4;2;1]` independent of one another, whereas they share nothing on the OCaml heap, they nonetheless share the objects corresponding to the common suffix `[2;1]` on a content-addressable store. This property is crucially relied on by CARMOT to support distributed applications composed of high-level (OCaml) data structures, as described below.

3.2 The CARMOT API

CARMOT hides the full complexity of a version control system behind a monadic abstraction called a Versioned State (VST), and exposes just the right level of detail for programmers to reap the benefits of version control-based replication. The API comprising the CARMOT

```
module type VST = sig
  type ('a, 'b) t
  val return : 'b -> ('a, 'b) t
  val bind : ('a, 'b) t -> ('b -> ('a, 'c) t) -> ('a, 'c) t
  val get_current_version: unit -> ('a, 'a) t
  val with_init_version_do: 'a -> ('a, 'b) t -> 'b
  val with_forked_version_do: string -> ('a,'b) t -> 'b
  val fork_version: (string -> ('a,'b) t) -> ('a,string) t
  val sync_next_version: 'a -> string list -> ('a, 'a) t
end
```

Figure 8 The CARMOT API.

programming model is shown in Fig. 8. The VST monad couches a versioned state of type `'a`. For instance, `'a` can be a `Counter.t`, a `Pixel.t`, or even a composite data type such as `Counter.t list`. The representation of these types in the underlying CARMOT store is as described in the previous section, and not exposed by the `VST` interface. Instead the interface orchestrates computations around the versioned state, translating between the high-level and low-level representations as needed. Computations on the monad read the latest version, commit a new version, or pull and merge concurrent versions. The type `('a, 'b) t` represents a monadic computation on the version state `'a` that returns a result of type `'b`. Operation `get_current_version` returns the high-level representation of the latest version of the state behind the monad. A programmer can initiate a computation against an explicitly provided initial version of the state using `with_init_version_do` API. Alternatively, computation can be run against an initial version forked from a remote using `with_forked_version_do` API. The string argument to the API is the URL of the remote. To fork off a new version of the state and run a (local) concurrent computation against it, `VST` provides the `fork_version` API. The argument to the function is the computation to run concurrently. The computation can expect the URL of the parent to be given as a string argument. The return value of `fork_version` is also a URL string that identifies the fork in the same terms as a remote. The underlying thread library is LWT [27]. Lastly, `sync_next_version` API (simply called `sync`) does two things. First, it commits the given `'a` argument as the new local version. Next, it pulls the latest versions from the given list of remotes (`string list`), and successively merges them to the latest local version, creating a later version each time. The latest version at the end of the merge sequence is returned. Note that if some of the remotes are unreachable during the operation, they are merely skipped. Consequently, `sync` is only guaranteed to sync with a subset of replicas. Functions `return` and `bind` are the usual monadic glue. Following the convention, we use the infix operator $\gg=$ to denote `bind`.

3.3 CARMOT Examples

To understand how the CARMOT API helps orchestrate distributed computations, let us reconsider the `Canvas` drawing application from the earlier section. Say Alice finished building `Canvas`, and would now like to use it to collaborate with her friends Bob and Cheryl. She could do that conveniently via the CARMOT API. A sample drawing session between the three collaborators is shown in Fig. 10 A possible execution of the session is visualized in Fig. 9. Assume that Alice starts her session on a 5×5 blank canvas, as shown below:

```
with_init_version_do (Canvas.new_blank 5 5) alice_f
```

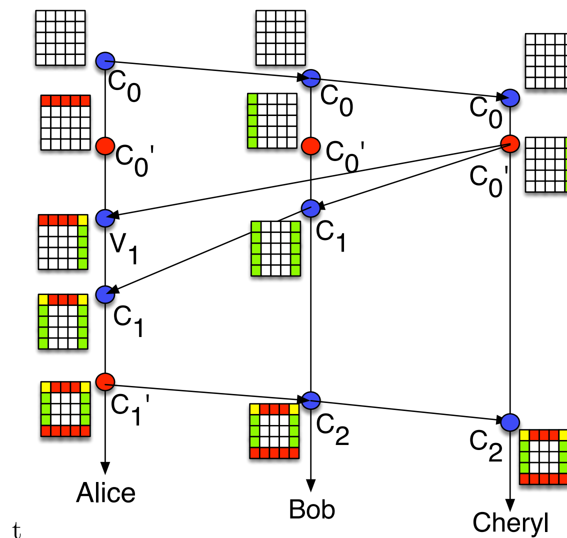


Figure 9 Collaborative drawing session visualized.

Bob and Cheryl, on the other hand, start their sessions with a version forked from Alice's initial version as shown below (Bob's shown; Cheryl's is similar):

```
with_forked_version_do "alice" bob_f
```

Assume `Canvas.new_blank` returns a blank canvas of a given dimension, and the function `Canvas.draw_line` draws a line between the given pair of points (and returns the new canvas). Alice starts by reading the current version of the canvas, which is blank. Alice draws a red horizontal line from $(0, 0)$ (top-left) to $(4, 0)$ (top-right) using `Canvas.draw_line`. Meanwhile, Bob draws a green vertical line from $(0, 0)$ to $(0, 4)$, and Cheryl draws a similar line from $(4, 0)$ to $(4, 4)$. All three of them call `sync` to commit their latest versions (C'_0). While any partial ordering of concurrent `syncs` is valid, we consider a linear order where Cheryl's `sync` happens first, followed by Bob's and then Alice's. Cheryl's `sync` does not find any concurrent versions, hence installs the proposed version (C'_0) as the next version at Cheryl's end. Bob's `sync` finds Cheryl's C'_0 as a concurrent version, and merges it with its proposal to produce the next version C_1. Next, Alice's `sync` finds Cheryl's C'_0 and Bob's C_1 as concurrent versions, and merges them successively with Alice's latest version creating new versions V_1 and C_1. The latest version C_1 is returned. Next, Alice draws a red horizontal line from $(0, 4)$ to $(4, 4)$, and commits the version C'_1 via `sync`. Since there are no concurrent versions, C'_1 becomes the latest version on Alice's end. The subsequent `sync` operations from Bob and Cheryl simply install Alice's C'_1 as the latest version.

Computations at Blockchain peers can be similarly defined using the CARMOT API. The peer that initiates the blockchain with a genesis block (using the `init` function of blockchain) also starts the computation using CARMOT's `with_init_version_do` (let `init_user_url` stand for the initializing peer's public key):

```
with_init_version_do
  {txns=Set.empty; chain=init init_user_pkey}
  (peer_f init_user_pkey)
```

Other peers start their computations by forking off the init user (let `peer_key` denote the peer's public key):

```
with_forked_version_do init_user_url (peer_f peer_key)
```

```
let alice_f : (Canvas.t,unit) VST.t =
  get_current_version () >>= fun c0 ->
  let c0' = Canvas.draw_line c0
              {x=0;y=0} {x=4;y=0} in
  sync_next_version c0'
              ["bob"; "cheryl"] >>= fun c1 ->
  let c1' = Canvas.draw_line c1
              {x=0;y=4} {x=4;y=4} in
  sync_next_version c1'
              ["bob"; "cheryl"] >>= fun c2 ->
  return ()

let bob_f : (Canvas.t,unit) VST.t =
  get_current_version () >>= fun c0 ->
  let c0' = Canvas.draw_line c0
              {x=0;y=0} {x=0;y=4} in
  sync_next_version c0'
              ["alice"; "cheryl"] >>= fun c1 ->
  sync_next_version c1
              ["alice"; "cheryl"] >>= fun c2 ->
  return ()

let cheryl_f : (Canvas.t,unit) VST.t =
  get_current_version () >>= fun c0 ->
  let c0' = Canvas.draw_line c0
              {x=4;y=0} {x=4;y=0} in
  sync_next_version c0'
              ["alice"; "bob"] >>= fun c1 ->
  sync_next_version c1
              ["alice"; "bob"] >>= fun c2 ->
  return ()
```

Figure 10 A collaborative drawing session between Alice, Bob, and Cheryl via the Canvas app.

Once initiated, the computation that runs on each peer is the same, and is defined by the `peer_f` function. Each peer runs in an eternal loop, concurrently serving new transactions and mining blocks, and periodically synchronizing with other (available) peers. operation. A illustrative definition of `peer_f` is shown in Fig. 11. The peer initially forks two threads - one for mining new blocks (`miner_f`) and other to serve incoming transaction requests (`server_f`). Next, it enters a loop where it first synchronizes with the local miner and server threads, and then synchronizes with other peers in the blockchain network, thereby acting as a mediator between its miner and server, and also between the local threads and remote peers. The loop repeats every 5ms. The `lift_lwt` API, which was not listed in Fig. 8, is a helper function that lets an LWT computation [27] be treated as a CARMOT computation. The mining thread (`miner_f`) repeatedly mines a new block and synchronizes with the parent. Similarly, the server thread (`server_f`) repeatedly reads a new transaction request (blocking until one is available), creates and adds a new transaction to the pool of unconfirmed transactions, and subsequently synchronizes with the parent. This computation is repeated on every peer, and all such peers together makeup the blockchain network.

4 Related Work

Our idea of versioning state bears resemblance to Concurrent Revisions [7], a programming abstraction that provides deterministic concurrent execution. The idea of using revisions as a means to programming eventually consistent distributed systems was further developed in [8]. The CARMOT programming model, however, differs from a concurrent revisions model because

```
let rec miner_f peer_key parent_url =
  get_current_version () >>= fun t ->
  let t' = mine_block peer_key t in
  sync_next_version t' [parent_url] >>= fun _ ->
  miner_f peer_key parent_url

let rec server_f parent_url =
  get_current_version () >>= fun t ->
  get_txn_request () >>= fun req ->
  let t' = new_txn req.s_key req.r_key req.amt t in
  sync_next_version t' [parent_url] >>= fun _ ->
  server_f parent_url

let peer_f peer_key =
  fork_version (miner_f peer_key) >>= fun miner_url ->
  fork_version server_f >>= fun server_url ->
  let rec loop () =
    get_current_version () -> fun t ->
    sync_next_version t [miner_url; server_url] -> fun t' ->
    sync_next_version t' peer_urls
    lift_lwt (Lwt_unix.sleep 0.005) >>= fun _ ->
    loop () in
  loop ()
```

Figure 11 Computation at a blockchain peer expressed using CARMOT API.

it imposes no distinction between *servers*, machines that hold global state, and *clients*, devices that operate on local, potentially stale, data. Any computation executing in a distributed environment is free to fork new versions, and synchronize against other replicated state, i.e., the operation is fully decentralized, which lets CARMOT express unconventional applications such as Blockchains. Just as significantly, CARMOT allows applications to customize join semantics with programmable merge operations. Indeed, the integration of a version-based mechanism within OCaml allows a degree of type safety, composability, and profitable use of polymorphism not available in related systems.

[10] also presents an operational model of a replicated data store that is based on the abstract system model presented in [9]; their design is similar to the model described in [25]. In both approaches, coordination among replicas involves transmitting operations on replicated objects that are performed locally on each replica. In contrast, CARMOT allows programmers to use familiar state-based and functional abstractions when developing distributed applications.

Modern distributed systems are often equipped with only parsimonious data models (e.g., key-value model) and poorly understood low-level consistency guarantees that complicate program reasoning, and make it hard to enforce application integrity. Some authors [4] have demonstrated that it is possible to *bolt on* high-level consistency guarantees (e.g., causal consistency) [20, 6] as a *shim layer* service over existing stores without losing availability. Version control-based replication model in CARMOT is causally consistent by construction, and does not require any additional reasoning about consistency on behalf of programmers.

A number of verification techniques, programming abstractions, and tools have been proposed to reason about program behavior in a geo-replicated weakly consistent environment. These techniques treat replicated storage as a black box with a fixed pre-defined consistency model [3, 1, 14, 18, 19, 5]. On the other hand, compositional proof techniques and mechanized verification frameworks have been developed to rigorously reason about various components of a distributed data store [28, 17]. CARMOT seeks to provide a rich high-level programming

model, built on rigorous foundations, that can facilitate program reasoning and verification. An important by-product of the programming model is that it does not require algorithmic restructuring to transplant a sequential or concurrent program to a distributed, replicated setting; the only additional burden imposed on the developer is the need to provide a merge operator, a function that can be often easily written for many common data types.

CARMOT shares some resemblance to conflict-free replicated data types (CRDT) [24]. CRDTs define abstract data types such as counters, sets, etc., with commutative operations such that the state of the data type always converges. Unlike CRDTs, the operations on data types in CARMOT need not commute and the reconciliation protocol is defined by user-defined merge functions. CARMOT uses 3-way merges using the lowest common ancestor, which is critical for all of our user-defined merges. However, CRDTs do not have the benefit of lowest common ancestor for merges and are only presented with the two concurrent versions. If a 3-way merge is desired, then the causal history has to be explicitly encoded in the data type. As a result, constructing even simple data types like counters are more complicated using CRDTs [24] compared to their implementation in CARMOT.

CARMOT uses 3-way merges using the lowest common ancestor, which is critical for all of our user-defined merges. However, CRDTs do not have the benefit of lowest common ancestor for merges and are only presented with the two concurrent versions. If a 3-way merge is desired, then the causal history has to be explicitly encoded in the data type. As a result, constructing even simple data types like counters are more complicated using CRDTs [24] compared to their implementation in CARMOT. CRDTs also tend to be implemented directly over the network protocols. Hence, low-level concerns such as duplicate delivery, lost messages, message reordering are explicitly handled in the data type definition. Such low-level details are abstracted away by CARMOT, which relies on the version control backend to implement a high-level branch-consistent distributed store that handles fault tolerance and network errors behind the screens.

References

1. Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.
2. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM. doi:10.1145/3190508.3190538.
3. Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014. doi:10.14778/2735508.2735509.
4. Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM. doi:10.1145/2463676.2465279.
5. Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. Putting the Consistency back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer System*, EuroSys '15, Bordeaux, France, 2015. URL: http://lip6.fr/Marc.Shapiro/papers/putting-consistency-back-EuroSys-2015.pdf.

6 Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On Verifying Causal Consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 626–638, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009888.

7 Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent Programming with Revisions and Isolation Types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 691–707, New York, NY, USA, 2010. ACM. doi:10.1145/1869459.1869515.

8 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_14.

9 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535848.

10 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *Proceedings of the 29th European Conference on Object-Oriented Programming*, ECOOP '15, Prague, Czech Republic, 2015. URL: http://research.microsoft.com/pubs/240462/gsp-tr-2015-2.pdf.

11 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. doi:10.1145/1294261.1294281.

12 Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601.

13 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, October 2017. doi:10.1145/3133933.

14 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 371–384, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837625.

15 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe Replication Through Bounded Concurrency Verification. *Proc. ACM Program. Lang.*, 2(OOPSLA):164:1–164:27, October 2018. doi:10.1145/3276534.

16 Martin Kleppmann and Alastair R. Beresford. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems*, PP, August 2016. doi:10.1109/TPDS.2017.2697382.

17 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 357–370, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837622.

18 Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2643634.2643664.

19 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. URL: `http://dl.acm.org/citation.cfm?id=2387880.2387906`.

20 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. `doi:10.1145/2043556.2043593`.

21 2019. OCaml Git Library. URL: `https://opam.ocaml.org/packages/git`.

22 Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society. `doi:10.1109/ICDCS.2009.20`.

23 2019. CRDTs in Riak. URL: `https://docs.basho.com/riak/kv/2.0.1/learn/concepts/crdts/`.

24 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011. `doi:10.1007/978-3-642-24550-3_29`.

25 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM. `doi:10.1145/2737924.2737981`.

26 Paolo Viotti and Marko Vukolic. Consistency in Non-Transactional Distributed Storage Systems. *CoRR*, abs/1512.00168, 2015. `arXiv:1512.00168`.

27 Jérôme Vouillon. Lwt: A Cooperative Thread Library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 3–12, New York, NY, USA, 2008. ACM. `doi:10.1145/1411304.1411307`.

28 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM. `doi:10.1145/2737924.2737958`.

The Next 700 Semantics: A Research Challenge

Shriram Krishnamurthi
Brown University, Providence, RI, USA
sk@cs.brown.edu

Benjamin S. Lerner
Northeastern University, Boston, MA, USA
blerner@ccs.neu.edu

Liam Elberty
Unaffiliated

Abstract

Modern systems consist of large numbers of languages, frameworks, libraries, APIs, and more. Each has characteristic behavior and data. Capturing these in semantics is valuable not only for understanding them but also essential for formal treatment (such as proofs). Unfortunately, most of these systems are defined primarily through implementations, which means the semantics needs to be *learned*. We describe the problem of learning a semantics, provide a structuring process that is of potential value, and also outline our failed attempts at achieving this so far.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Language features; Software and its engineering → Semantics; Software and its engineering → Formal language definitions

Keywords and phrases Programming languages, desugaring, semantics, testing

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.9

Funding This work was partially supported by the US National Science Foundation and Brown University while all authors were at Brown University.

Acknowledgements The authors thank Eugene Charniak and Kevin Knight for useful conversations. The reviewers provided useful feedback that improved the presentation.

© Shriram Krishnamurthi, Benjamin S. Lerner, and Liam Elberty;
licensed under Creative Commons License CC-BY

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 9; pp. 9:1–9:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Motivation

Semantics is central to the trade of programming language researchers and practitioners. A useful semantics for a system helps us understand it better, guides us in building tools, and provides us a basis for proving valuable properties such as soundness and completeness.

However, there are far fewer semantics than there are objects that need them. A few languages (notably Scheme [5] and Standard ML [16]) are accompanied by a reasonably thorough semantic description, but even these rarely cover the behavior of the language that users use (such as libraries). Nevertheless, these semantics are valuable in that they provide users a standpoint from which to evaluate an implementation: if it diverges from the semantics, they can argue that the fault definitively lies with the implementation, since the semantics was given as part of the process of defining the very language.

However, the need for and value of a semantics extends well beyond what we would conventionally call a "language". Consider, for instance, a Web program: there is value to having a semantics for every layer in the stack, from the hardware's instruction set to the operating system's API to the browser's APIs to the JavaScript implemented by the browser to the numerous frameworks that sit atop JavaScript and effectively create their own ontologies (for page traversal, page transformation, etc.) to the libraries that sit atop frameworks. Even if we draw a line at, say, the browser, we still need meaningful semantics for the upper layers to make sense of what a program does. Thus, reasoning about the security of a Web program effectively demands a "full-stack semantics", as it were.

Unfortunately, virtually none of these levels enjoy the benefits of originating in a formal semantics. They are instead defined by an ad hoc collection of implementations, documents, and test suites. The task of the semanticist then becomes one of *reverse engineering* a semantics in a way that ensures its *conformance* with reality: after all, if there is a divergence between the semantics and the implementation, the implementors would not agree that they are to blame and accordingly "fix" their systems. This is essentially a bottom-up discipline akin to that practiced in the natural sciences.

2 Tested Semantics and a Small Core

Over the past decade, several researchers have begun to create so-called *tested semantics*: one where the semantics shows conformance with the real-world artifact [1, 2, 3, 4, 7, 8, 10, 11, 12, 13, 15, 17, 18, 19]. A notable line of work has addressed the semantics of "scripting" languages, which are ostensibly defined by trying to provide a meaning to as many utterances as possible, as opposed to the less libertine behavior of more academic languages. Because of their enormous set of behaviors, many of which are stumbled into by programmers and exploited by attackers, it becomes especially important that a semantics conform to reality rather than to a researcher's mental idea of how the language *ought* to be.

However, not all tested semantics are equal in value. Some semantics come in the form of hundreds of rules. In contrast, other projects (such as [11, 18, 19]) have followed the pattern shown in Figure 1. In this style, the semantics is broken into two components. The *surface* language P is reduced to an essential *core* version, λ_P, and all P programs are mapped to λ_P using a *desugaring* function. We use the term "desugaring" to evoke the idea that most of the surface language can be easily rexpressed (i.e., in a structured, recursive, compositional manner) as idioms of the core language, and can therefore be dismissed as ergonomic convenience rather than as novel features requiring analysis. (Strictly speaking, this term is slightly misleading, since the core language λ_P might be a *different* language rather than a subset of P itself; nevertheless, this term captures our design intent better

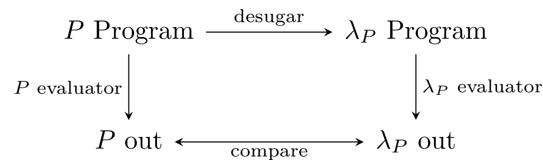


Figure 1 Testing Strategy for λ_P.

than the more general "compilation".) It is usually very straightforward to implement an interpreter for λ_P (an appropriately-sized exercise for undergraduates), and composing this implementation with desugaring gives us a new implementation of P. Of course, P already has existing implementations. We can therefore then compare the results of running the two implementations on a suitably large suite of programs, tweaking desugaring and λ_P until the result is in harmony with the existing behavior of P. Since existing P implementations may have values with opaque internal structure, we may not be able to translate them directly into λ_P values; instead we may have to use coarser notions of equivalence (such as textually comparing the output of two programs), or even more abstract ones, to check that our desugaring preserves intended behavior.

This style has significant advantages over the brute-force generation of hundreds of rules. First, it helps us reduce the complexity of the overall problem. Second, it forces us to find general patterns in the language, which reduces the likelihood of overfitting to a particular set of programs against which we run tests. Third, it gives us λ_P, which is a valuable artifact in its own right. From the perspective of constructing proofs, λ_P – which might have five to twenty constructs, in the typical lambda calculus fashion – is a far more accessible target than having hundreds of rules. Additionally, λ_P reduces P to an essence (there may, of course, be multiple λ_P's that provide different essential views of the language), thus accomplishing one of the jobs of a semanticist: to give *insight*, not merely brutally reconstruct reality. In that light, the desugaring function too is a useful artifact, and we want it to be both readable and terse. An unreadable desugaring function yields no insight into the semantics of constructs in P, and in fact casts suspicion on whether all behaviors have been faithfully captured. Likewise, a readable function that nevertheless has inordinately complicated behavior indicates that the proposed λ_P does a poor job of modeling the intrinsically "interesting" parts of P.

The problem remains: how to create these artifacts? The first author has been involved in several tested semantics efforts, and their labor demands are daunting: for instance, that for Python [19] took approximately 72 person-months, including recruiting participants from a MOOC. This simply does not scale. Can we instead *learn* these semantics?

3 Learning a Semantics

Before we apply machine learning techniques, it helps to agree on the appropriate role for human versus machine. It may be tempting to simply let machine-learning techniques run wild and reconstruct a set of rules that match an existing implementation, but as we have argued in Section 2, this would be counter-productive. Instead, we claim Figure 1 provides us with a clear guide. The place for an expert is in the design of the core λ_P, using an understanding of (a) the surface P itself, (b) general design principles in the design of semantics, and (c) eventual goals for the use of the semantics. In contrast, the act of writing desugaring is where the grunt work lies, and this is the part that would best be automated. In addition, in our experience, engineering λ_P is both interesting and does not consume effort that is not insightful, whereas constructing desugaring can take months and is only very rarely insightful.

In our ideal workflow, then, the semantics engineer (who presumably understands P well) proposes an initial design for λ_P, alongside an implementation for it (which, depending on how λ_P is written, may even be obtained automatically). They also provide a conformance suite, such as existing test suites for widely-used P implementations.[1] A learning system then attempts to construct the desugaring function.

The space of desugaring functions is of course unbounded. We believe a useful constraint is given by Felleisen's expressiveness framework [9], which (loosely) says that if a feature can be expressed by local macro transformations (i.e. structural, compositional changes), it is not "expressive". We believe that (unless a semantics engineer explicitly chooses otherwise) desugaring should *not* be expressive: if it is, a non-trivial feature of the language is not captured by λ_P.

So long as the search for desugaring succeeds, all is well. But when the search has proceeded for too long, it seems reasonable to present the input for which no reasonable desugaring could be found: the semanticist is then in a position either to realize that λ_P must grow, or to guide the desugaring system with a hint, or even to provide the desugaring rule directly. Based on our experience manually constructing semantics in this style, these interruptions grow far less frequent over time, and the rare situations where a desugaring can't easily be found are usually instructive about the nature of P.

Problem Definition

The essential problem, then, is as follows. Assuming we can obtain a parser for P (usually to be found inside P implementations) and similarly for λ_P (which should be straightforward), the goal is to *learn a translation from P trees to λ_P trees* – a tree transducer [6, chapter 6] – such that Figure 1 ensues. This problem is not new: it lies at the heart of natural language translation (NLT).

A critical difference, however, is in the setup of the problem. In NLT, one is typically translating, say, French to German. To do this, the learning system is given millions of pairs of the "same" sentences in both languages: obtained from proceedings of national bodies, book translations, and so on. From this large corpus of examples, an algorithm infers a transducer. However, there is no oracle for ground truth; some use has been made of crowdsourcing for this purpose [20], but it may not be straightforward to handle millions of new proposed translations this way, and it would anyway be slow and somewhat unreliable.

In programming languages, we have the dual situation. The semantics engineer will not have the patience to write more than a few dozen examples of P-to-λ_P translations. However, we *do* have ground truth! A system can easily generate new instances of P programs and run them; and also pass them through the candidate desugarer, compute their answer under the semantics, and compare the two answers. If this succeeds, we have gained further trust in the desugaring. If it fails, we have a P program for which our desugaring must be revised. The system can run this loop automatically, correctly, and at scale (with extreme parallelization).

Unfortunately, there seems to be little literature on this problem in the NLT literature. This is perhaps unsurprising: there is little use in natural languages in creating techniques that assume perfect, mechanized ground truth. Thus, our dual problem is not only valuable but also technically interesting.

[1] Of course, different implementations may differ slightly. This only further reinforces the need for a learning-based approach, because innocent programmers might stumble on these differences while adversaries exploit them.

$$
\begin{array}{rcl}
\texttt{x}, \texttt{y} \in & \textit{SIdentifiers} & \\
n \in & \textit{Number} & \\
s \in & \textit{Str} & \\
\texttt{o} \in & \textit{SOp} & ::= \texttt{0-} \mid \texttt{not} \mid \texttt{+} \mid \texttt{-} \mid \texttt{and} \mid \texttt{or} \mid \texttt{<} \mid \texttt{>} \\
\texttt{e} \in & \textit{SExp} & ::= \texttt{SVar(x)} \mid \texttt{SPrim(o, }\vec{\texttt{e}}\texttt{)} \mid \texttt{SBetween(e}_1\texttt{, e}_2\texttt{, e}_3\texttt{)} \\
& & \mid \texttt{SNum(}n\texttt{)} \mid \texttt{SStr(}s\texttt{)} \mid \texttt{STrue} \mid \texttt{SFalse} \\
& & \mid \texttt{SIf(e}_1\texttt{, e}_2\texttt{, e}_3\texttt{)} \mid \texttt{SLet(x, e}_1\texttt{, e}_2\texttt{)} \mid \texttt{SLetRec(x, e}_1\texttt{, e}_2\texttt{)} \\
& & \mid \texttt{SLam(}\vec{\texttt{x}}\texttt{, e)} \mid \texttt{SApp(e, }\vec{\texttt{e}}\texttt{)} \mid \texttt{SAssign(x, e)} \\
& & \mid \texttt{SList(}\vec{\texttt{e}}\texttt{)} \mid \texttt{SListCase(e}_1\texttt{, e}_2\texttt{, e}_3\texttt{)} \mid \texttt{SFor(e}_1\texttt{, }\vec{\texttt{b}}\texttt{, e}_2\texttt{)} \\
\texttt{b} \in & \textit{SForBind} & ::= \texttt{SFBind(x, e)}
\end{array}
$$

$$
\begin{array}{rcl}
\texttt{x}, \texttt{y} \in & \textit{CIdentifiers} & \\
n \in & \textit{Number} & \\
s \in & \textit{Str} & \\
b \in & \textit{Bool} & \\
\texttt{o} \in & \textit{COp} & ::= \texttt{0-} \mid \texttt{not} \mid \texttt{+} \mid \texttt{-} \mid \texttt{and} \mid \texttt{or} \mid \texttt{<} \mid \texttt{>} \\
\texttt{e} \in & \textit{CExp} & ::= \texttt{CVar(x)} \mid \texttt{CPrim1(o, e)} \mid \texttt{CPrim2(o, e}_1\texttt{, e}_2\texttt{)} \\
& & \mid \texttt{CNum(}n\texttt{)} \mid \texttt{CStr(}s\texttt{)} \mid \texttt{CBool(}b\texttt{)} \\
& & \mid \texttt{CIf(e}_1\texttt{, e}_2\texttt{, e}_3\texttt{)} \mid \texttt{CLet(x, e}_1\texttt{, e}_2\texttt{)} \mid \texttt{CLetRec(x, e}_1\texttt{, e}_2\texttt{)} \\
& & \mid \texttt{CLam(}\vec{\texttt{x}}\texttt{, e)} \mid \texttt{CApp(e, }\vec{\texttt{e}}\texttt{)} \mid \texttt{CAssign(x, e)} \\
& & \mid \texttt{CList(}\vec{\texttt{e}}\texttt{)} \mid \texttt{CListCase(e}_1\texttt{, e}_2\texttt{, e}_3\texttt{)}
\end{array}
$$

Figure 2 Simple Source and Target Core languages.

4 Initial Non-Progress

In addition to formulating the problem, to date we have tried four different existing techniques, with no real success. The rest of this paper documents what we have learned so far, but we caution that, since we are not experts in this area, one should not read too much into our failure.

4.1 Notation

We denote `source programs` (previously P) in red monospaced text, `core programs` (previously λ_P) in blue monospaced text, and the desugarers themselves in black sans-serif font. Additionally, we prefix the names of source AST nodes with a `S`, and likewise prefix core AST node names with a `C`. We use • to denote a hole in a tree, into which another tree may be merged; when defining a desugarer, subscripts are used to indicate corresponding holes in source and core terms. Finally, for expressions with arbitrary arity, we use $\cdots$ to indicate the repetition of the preceding term (much as ... are used in Racket macros).

4.2 Example Languages

Consider the languages in Figure 2. We have tried to learn a desugaring from the source to the core. Most expression forms are straightforward to translate, and there are some characteristic "impedance mismatches" between the two languages. The construction `SBetween(lo, mid, hi)` is intended to be the chained comparison $lo \leq mid \leq hi$, where each expression is evaluated at most once. Primitive operations are represented in the source via an

$$
\begin{aligned}
\mathsf{ds}(\texttt{SVar(}\bullet\texttt{)}) &= \texttt{CVar(}\bullet\texttt{)} \\
\mathsf{ds}(\texttt{SPrim(}\bullet_1\texttt{, [}\bullet_2\texttt{])}) &= \texttt{CPrim1(}\bullet_1\texttt{, }\bullet_2\texttt{)} \\
\mathsf{ds}(\texttt{SPrim(}\bullet_1\texttt{, [}\bullet_2\texttt{, }\bullet_3\texttt{])}) &= \texttt{CPrim2(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)} \\
\mathsf{ds}(\texttt{SBetween(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CLet(\%t1, }\bullet_1\texttt{, CLet(\%t2, }\bullet_2\texttt{, CLet(\%t3, }\bullet_3\texttt{,} \\
&\qquad \texttt{CPrim2(and, CPrim2(<, \%t1, \%t2),} \\
&\qquad\qquad\qquad \texttt{CPrim2(<, \%t2, \%t3)))))} \\
\mathsf{ds}(\texttt{SNum(}\bullet\texttt{)}) &= \texttt{CNum(}\bullet\texttt{)} \\
\mathsf{ds}(\texttt{SStr(}\bullet\texttt{)}) &= \texttt{CStr(}\bullet\texttt{)} \\
\mathsf{ds}(\texttt{STrue}) &= \texttt{CBool(true)} \\
\mathsf{ds}(\texttt{SFalse}) &= \texttt{CBool(false)} \\
\mathsf{ds}(\texttt{SIf(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CIf(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)} \\
\mathsf{ds}(\texttt{SLet(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CLet(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)} \\
\mathsf{ds}(\texttt{SLetRec(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CLetRec(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)} \\
\mathsf{ds}(\texttt{SLam([}\bullet_1\cdots\texttt{], }\bullet_2\texttt{)}) &= \texttt{CLam([}\bullet_1\cdots\texttt{], }\bullet_2\texttt{)} \\
\mathsf{ds}(\texttt{SApp(}\bullet_1\texttt{, [}\bullet_2\cdots\texttt{])}) &= \texttt{CApp(}\bullet_1\texttt{, [}\bullet_2\cdots\texttt{])} \\
\mathsf{ds}(\texttt{SAssign(}\bullet_1\texttt{, }\bullet_2\texttt{)}) &= \texttt{CAssign(}\bullet_1\texttt{, }\bullet_2\texttt{)} \\
\mathsf{ds}(\texttt{SList(}\bullet\ \cdots\texttt{)}) &= \texttt{CList(}\bullet\ \cdots\texttt{)} \\
\mathsf{ds}(\texttt{SListCase(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) &= \texttt{CListCase(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)} \\
\mathsf{ds}(\texttt{SFor(}\bullet_1\texttt{, [SFBind(}\bullet_2\texttt{, }\bullet_3\texttt{)}\cdots\texttt{], }\bullet_4\texttt{)}) &= \texttt{CApp(}\bullet_1\texttt{, SLam([}\bullet_2\cdots\texttt{], }\bullet_4\texttt{), [}\bullet_3\cdots\texttt{])}
\end{aligned}
$$

Figure 3 Intended ground-truth translation from Simple Source to Target Core. `%t1` etc are fresh generated names for temporary variables.

arbitrary-arity `SPrim` constructor, and must be translated to either unary `CPrim1` or binary `CPrim2` constructions. Booleans `STrue` and `SFalse` must be translated to actual booleans inside a `CBool` constructor, and other primitive values are simply carried across. The `SFor` construction is unique to the source language, and must be translated into simpler terms in the core. Our intended translation is given in Figure 3.

4.3 First attempt: Naïve Tree Matching

4.3.1 Essential Ideas

As an initial guess, we posit that translating the AST of one language into a corresponding tree in another language will result in a tree "of roughly the same shape": that is, a source node with N children will desugar to a tree with N disjoint descendants that correspond to the source node's children, respectively. For example, `SPrim(+, [`$\bullet_1$`, `$\bullet_2$`])` might correspond to `CLet(%t1, `$\bullet_1$`, CLet(%t2, `$\bullet_2$`, CPrim2(+, %t1, %t2)))`.

With this as a guide, our initial attempt is to learn a tree transducer that transforms source nodes into core subtrees in a uniform, top-down manner. Learning this transducer from a corpus of tests amounts to learning an explanation for how each test output was produced from its input. We can simplify the problem further under our assumption that tree shape is roughly preserved.

4.3.2 Worked Example

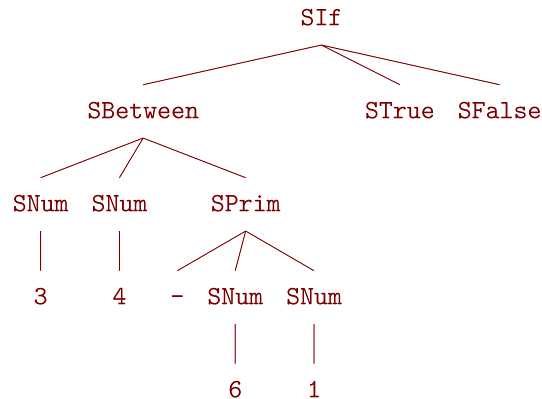
Consider the source program

```
SIf(SBetween(SNum 3, SNum 4, SPrim(-, [SNum 6, SNum 1])), STrue, SFalse)
```

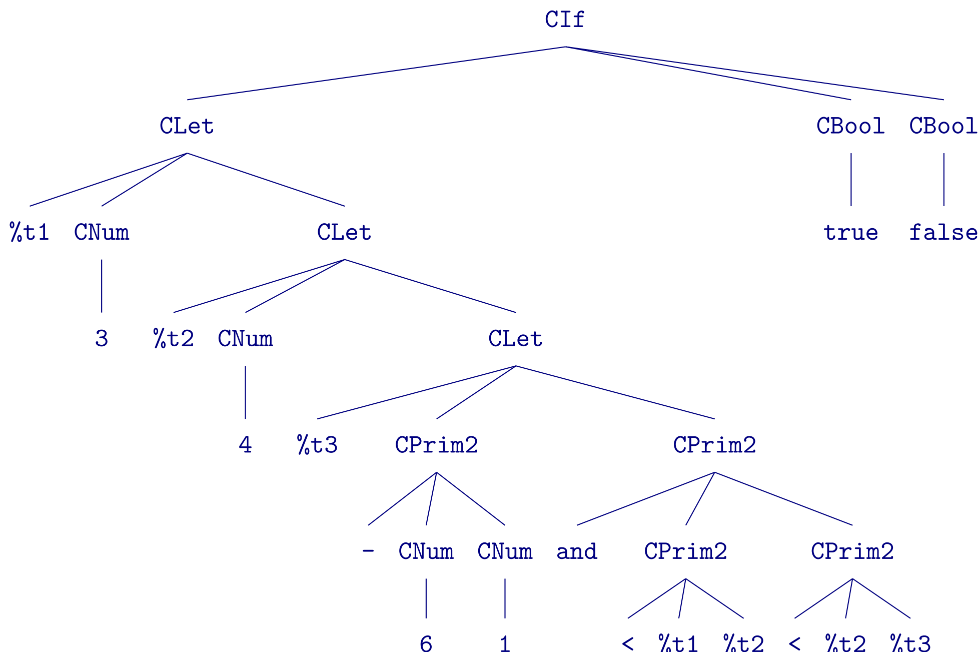
Our ground-truth desugaring results in the desugared expression

```
CIf(CLet(%t1, CNum 3, CLet(%t2, CNum 4, CLet(%t3, CPrim2(-, CNum 6, CNum 1),
          CPrim2(and, CPrim2(<, %t1, %t2), CPrim2(<, %t2, %t3))))),
     CBool(true), CBool(false))
```

Graphically, these two trees are:

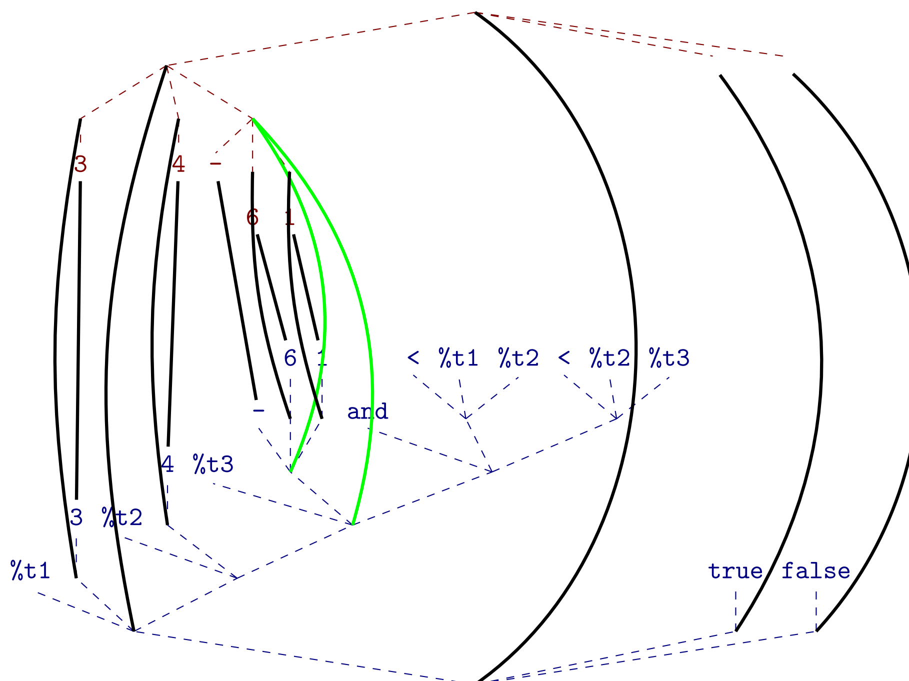


and



Our goal is to compute all possible alignments between these two trees, ignoring the labels on the internal nodes. (We draw the core tree inverted, to make drawing the alignments easier to read.) An alignment is simply a mapping of nodes in one tree to nodes in the other

that preserves ancestry: if a source node `S` maps to a core node `C`, then all descendants of `S` must map to descendants of `C`.



Note that in this example, one node (the root of `SPrim(-, [SNum 6, SNum 1])`) has two possible alignments (shown in green). This leads to two possible desugarings for `SBetween` and `SPrim` nodes: the correct one presented earlier, and the incorrect

$$\mathsf{ds}(\texttt{SBetween(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)}) \Rightarrow \texttt{CLet(\%t1, }\bullet_1\texttt{, CLet(\%t2, }\bullet_2\texttt{, }\bullet_3\texttt{))}$$

$$\mathsf{ds}(\texttt{SPrim(}\bullet_1\texttt{, [}\bullet_2\texttt{, }\bullet_3\texttt{])}) \Rightarrow \texttt{CLet(\%t3, CPrim2(}\bullet_1\texttt{, }\bullet_2\texttt{, }\bullet_3\texttt{)} \\ \texttt{CPrim2(and,} \\ \texttt{CPrim2(<, \%t1, \%t2),} \\ \texttt{CPrim2(<, \%t2, \%t3)))}$$

Such a mistake is possible in this test case, because only one instance of either `SBetween` or `SPrim` is seen. With a larger test corpus, this mistaken desugaring could only appear as part of a non-deterministic desugarer, and so would eventually be discarded by the algorithm.

4.3.3 Requirements on Developer

Developers must provide a corpus of tests that adequately covers the language semantics, and also manually write the desugared ground-truth versions of the tests. This is potentially as tedious as writing the desugaring itself, and not a viable long-term improvement.

4.3.4 Shortcomings

Unfortunately, of course, this algorithm is potentially absurdly expensive: if the `core` tree is larger than the `source` (as it most likely will be), there are many possible alignments for each node, and therefore exponentially many possible alignments in total.

Additionally, this algorithm cannot learn the intended desugaring for our full language. The `SLam` and `SApp` expressions are variable-arity, which means our desugaring would require an infinite corpus to learn all the desugarings for every possible arity – when clearly the desugaring is uniformly defined. Further, the desugaring of `SFor` expressions examines the grand-children of the `SFor` node, and moreover does not preserve the ordering of children. This is simply beyond the expressive power of this algorithm.

4.4 Second Attempt: Learning a Tree Transducer by Gibbs Sampling

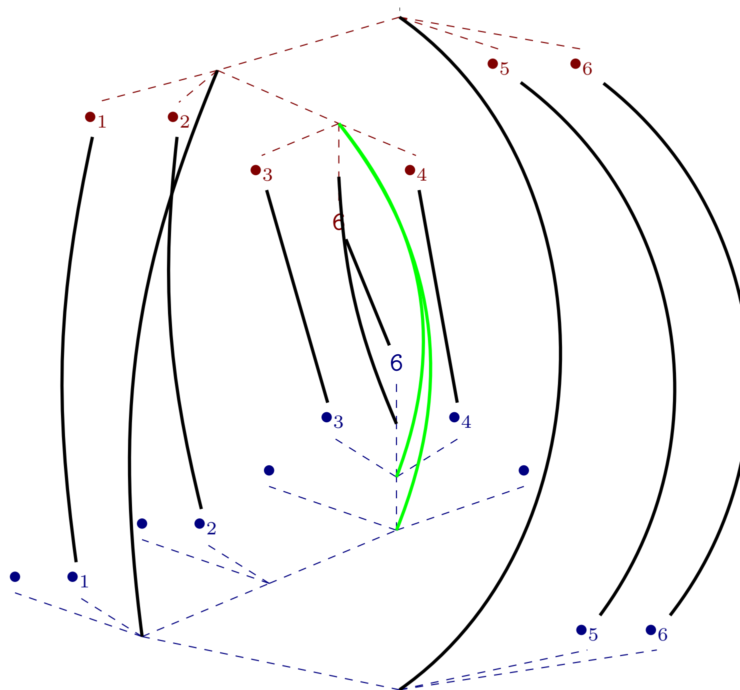
4.4.1 Essential Ideas

To constrain the search-space explosion of the simple algorithm above, we try a sampling approach instead: we relax our requirement that a desugarer be deterministic, and start with *one* alignment, which induces some desugarer. We then perturb that desugarer by picking some path in the `source tree`, from root to leaf, and attempting to "re-explain" it, hoping to find an explanation that is more deterministic than the current one. We then iterate this process until it converges on a deterministic desugarer that can explain the full test corpus. Specifically:

1. As before, we start with a corpus of `source` and ground-truth `core` programs.
2. For each test, we compute an inclusion from `source` to `core`. Combining all these inclusions produces a non-deterministic desugarer, where each translation rule is weighted by how often it was used in the current explanations of the tests. Each translation from `source` to `core` yields a *derivation tree*, whose nodes are labeled by desugarer rules and whose shape is the same as the `source` tree. Moreover, these weights induce a probability that describes how likely this derivation tree was to have occurred.
3. Repeat until convergence:
   a. Pick a derivation tree of lowest probability. Pick a path within that tree of lowest probability.
   b. "Truncate" the corresponding `source` and `core` trees by removing all side branches from that derivation path, and all `core` subtrees corresponding to the removed `source` nodes. This produces two narrow trees.
   c. Compute all possible inclusions of the truncated `source` into the truncated `core` tree. The roots must obviously correspond, as must the sole remaining leaf and its counterpart; the only nodes whose alignments could change are therefore the nodes along the `source` path. This greatly limits the combinatorial explosion to a manageable level.
   d. Compute the probability of each of these new alignments, using weights that have been adjusted to remove the existing alignment's contribution, and choose a new alignment based on these probabilities. This is essentially assigning a Dirichlet prior to the rules, and it will tend to reuse explanations that have been used many times in other tests, and tend to avoid explanations that have not been useful in many other cases.
   e. Update the weights of all the relevant rules.

4.4.2 Worked Example

Consider again the example expression in Section 4.3.2, and suppose by step 2 we had computed the mistaken desugaring shown there. In step 3.a, we would pick a path of lowest probability: in this example, we choose the path leading to the leaf `6`. We then truncate both the `source` and `core` trees and prune away all branches not relevant to that path. (We give the pruned nodes distinct labels, so we don't have to worry about mis-aligning them.)



Now the two possible alignments are very easy to compute; in fact, they are the only two possible alignments of these two trees (compared to potentially exponentially many alignments in the original, non-truncated trees). We then compute the probability of either alignment occurring given the frequencies with which the relevant desugaring rules have appeared in other test cases. The correct alignment is seen more frequently in other tests, leading to a higher probability here, which in turn reinforces the algorithm's confidence in the correctness of that alignment.

4.4.3 Requirements on Developer

Just like the naïve algorithm, this one requires the developer to supply a sufficient test corpus and its ground-truth accompaniment. However, it does not take quite so long to run.

4.4.4 Shortcomings

There are four key failings with this approach. First, just as with the naïve approach, this algorithm makes the key assumption that `source` trees will map to `core` trees of a similar shape, and `SFor` breaks that assumption.

Second, as above, the algorithm cannot handle arbitrary-arity nodes such as `SLam`. (We could shoehorn languages into fitting a fixed-arity model since no single program will ever have unbounded-arity source expressions, but even so it is subjectively wrong to provide completely disparate desugaring rules for binary functions from ternary ones, etc. Moreover, while any individual program is of fixed arity, the desugaring must handle all possible programs: the correctness of a single uniform rule is supported by all the programs of all the arities that it translates correctly, while a set of arity-specific rules necessarily are each supported by much smaller, independent sets of sample programs.)

Third, the algorithm cannot handle unbounded state during the desugaring process. In particular, it is quite common to need to generate new names (i.e., gensym) for use in the translated program. Nothing in the tree-transducer approach – regardless of how the transducer is constructed – can support such a construction.

Fourth, translation might sometimes need to inspect "deeply" into the source tree, i.e., examine more than just the root node and its immediate children, in order to translate it correctly (as, for example, with SFor). An *extended* tree transducer can handle this case, but the inference problem for extended tree transducers is known to be hard, and the algorithms above have no representation for such deep rules.

4.5 Third attempt: Genetic Programming

4.5.1 Essential Ideas

We aim to fix the first, second, and fourth problems above, while still remaining tractably expensive. Our main idea is to develop a domain specific language for describing translations. Our desugarers are now explicit programs in this language, rather than implicit in the mappings between two sets of trees. Because they are now explicit, we can mutate them in various ways, and this leads us to a genetic algorithm for evolving a correct translator. Our key observation is that rather than relying on extensive programmer effort to supply the ground truth, we can instead use the evaluators to compute ground truth.

4.5.2 Requirements on Developer

Unlike the algorithms above, we no longer require the programmer to supply a comprehensive corpus of source or core programs. Instead we can systematically generate source programs, or generate new source programs from a small number of exemplars via fuzzing. Additionally, the developer need *never* manually desugar a complete program.

Instead, our algorithm may ask the developer:

1. To say whether a given desugaring rule is at fault, or else provide a smaller source test program that still translates incorrectly, and
2. To provide the correct desugaring of a particularly troublesome source language construct, or suggested mutations to try when searching for such desugarings.

This latter point is *precisely where the programmer's attention should be focused* – on constructs whose semantics are not readily captured by "natural" tree transformations.

4.5.3 Shortcomings

Unfortunately, we were unable to make this scale well enough to find our desugaring. In general, because it is a randomized algorithm that depends heavily on the history of a particular run, convergence is troublesome, and like all genetic algorithms depends strongly on the fitness function we use. Since correctness is a boolean property with no gradient against which to optimize, our attempts at "smoothing" this into a useful fitness function amounted to counting what fraction of programs were translated correctly. However at the small scales we could attempt, this function was still too discrete to work well.

4.6 Fourth attempt: Synthesis

The first two approaches tried to exploit the tree-structure of the source and destination languages. The third approach tried to incorporate the fact that desugaring should be *a computable function*, and therefore a program in its own right. Perhaps techniques from program synthesis might help generate this desugaring? We know of only one attempt in this direction so far [14]. We highlight both the achievements and simplifications of this work, to emphasize what makes this problem quite so challenging.

The starting observation is that, because desugaring should not be expressive, and hence essentially be a tree homomorphism, the skeleton of any such desugarer is a pattern match across the source-language AST, followed by recursive desugaring calls to the subtrees, and finally some core-language constructor(s) to connect the resulting pieces. To make the problem tractable, the authors deeply embed the correctness criterion from Figure 1 as the specification of a correct synthesis result. To derive a desugarer such that the results of evaluation match up, when the authors encounter a subexpression of the outer term they leave it marked as a symbolic input, so that when attempting to interpret the resulting translated program, they replace the unknown output with the interpreted original source program. This "inductive decomposition" provides an order-of-magnitude improvement in performance.

However, the approach there takes some crucial simplifying steps. They rely on a shared value space for the interpreted answers for both languages; a more accurate representation of Figure 1 would account for the translation of values across the languages, which defeats the inductive decomposition property. Additionally, even in the sample language in Figure 3, the desugaring of `SFor` is not a trivial tree homomorphism: the translation needs to reach more deeply into the tree to rearrange binders and bound values. This form of manipulation is commonplace for languages, and should be part of a full solution here. Finally, the authors note that incorporating stateful semantics into their interpreters exploded the runtime cost of their approach, yet state is a necessary component of nearly all language designs.

5 Conclusion

Programming language research is caught between two extremes. On the one hand, we aspire to have pure, clean-slate designs that are designed well, accompanied by rich specifications, and implemented faithfully. On the other hand we have the messy world of contemporary systems, which have evolved haphazardly and democratically, and are encumbered with awkward edge cases and legacy constraints.

While we applaud those who work to make the former world a reality (and hope to make our own modest contributions to this effort), we want to call attention to the latter, which is at least our present reality. In this world, errors and attacks abound. The rich toolkit of programming languages can do much to defend against these problems, but only if we can bring these systems within our ambit. A semantics seems a necessary and important step in that direction. We believe the research program outlined here, with an emphasis on core languages and on a great deal of accidental complexity swept away by desugaring, is a smart way to proceed.

Earlier, we referred to (informal) semantics being defined in terms of "implementations, documents, and test suites". Our work shows how we can leverage implementations and test suites to reverse-engineer the formal semantics. It is natural to wonder what documents can contribute to this process. Do they make things worse, or do they actually make them better? Might we be able to curtail the combinatorial searches demanded by this paper by using information in prose? We do not know the answer, but hopefully others far more qualified can answer that authoritatively.

References

1. 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)*, pages 87–100, New York, NY, USA, 2014. ACM Press. doi:10.1145/2535838.2535876.
2. Martin Bodin, Tomás Diaz, and Éric Tanter. A Trustworthy Mechanized Formalization of R. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, pages 13–24, New York, NY, USA, 2018. ACM. doi:10.1145/3276945.3276946.
3. Denis Bogdanas and Grigore Roşu. K-Java: A Complete Semantics of Java. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 445–456, New York, NY, USA, 2015. ACM. doi:10.1145/2676726.2676982.
4. Arthur Charguéraud, Alan Schmitt, and Thomas Wood. JSExplain: A double debugger for JavaScript. In *The Web Conference 2018*, pages 1–9, Lyon, France, April 2018. doi:10.1145/3184558.3185969.
5. William Clinger and Jonathan Rees. The Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.
6. Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. http://tata.gforge.inria.fr/, 2007.
7. Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 2015. ACM Press.
8. Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 533–544, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103719.
9. Matthias Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, 17:35–75, 1991.
10. Daniele Filaretti and Sergio Maffeis. An Executable Formal Semantics of PHP. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 567–592, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-662-44202-9_23.
11. Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.
12. Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 336–345, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737979.
13. Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, July 2018. doi:10.1109/CSF.2018.00022.
14. Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. Synthesis of Recursive ADT Transformations from Reusable Templates. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 247–263, 2017.
15. Ali Kheradmand and Grigore Roşu. P4K: A formal semantics of P4 and applications. Technical report, University of Illinois at Urbana-Champaign, April 2018. arXiv:1804.01468.
16. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
17. 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)*, pages 346–356, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737991.

18 Joe Gibbs Politz, Matt Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Dynamic Languages Symposium*, 2012.

19 Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The Full Monty: A tested semantics for the Python programming language. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2013.

20 Omar Zaidan and Chris Callison-Burch. Crowdsourcing Translation: Professional Quality from Non-Professionals. In *Association for Computer Linguistics*, pages 1220–1229, 2011.

Toward Domain-Specific Solvers for Distributed Consistency

Lindsey Kuper
University of California, Santa Cruz, USA
lkuper@ucsc.edu

Peter Alvaro
University of California, Santa Cruz, USA
palvaro@ucsc.edu

Abstract

To guard against machine failures, modern internet services store multiple *replicas* of the same application data within and across data centers, which introduces the problem of keeping geo-distributed replicas *consistent* with one another in the face of network partitions and unpredictable message latency. To avoid costly and conservative synchronization protocols, many real-world systems provide only *weak* consistency guarantees (e.g., eventual, causal, or PRAM consistency), which permit certain kinds of disagreement among replicas.

There has been much recent interest in language support for specifying and verifying such consistency properties. Although these properties are usually beyond the scope of what traditional type checkers or compiler analyses can guarantee, *solver-aided languages* are up to the task. Inspired by systems like Liquid Haskell [43] and Rosette [42], we believe that close integration between a language and a solver is the right path to consistent-by-construction distributed applications. Unfortunately, verifying distributed consistency properties requires reasoning about transitive relations (e.g., causality or happens-before), partial orders (e.g., the lattice of replica states under a convergent merge operation), and properties relevant to message processing or API invocation (e.g., commutativity and idempotence) that cannot be easily or efficiently carried out by general-purpose SMT solvers that lack *native* support for this kind of reasoning.

We argue that domain-specific SMT-based tools that exploit the mathematical foundations of distributed consistency would enable both more efficient verification and improved ease of use for domain experts. The principle of *exploiting domain knowledge for efficiency and expressivity* that has borne fruit elsewhere – such as in the development of high-performance domain-specific languages that trade off generality to gain both performance and productivity – also applies here. Languages augmented with domain-specific, *consistency-aware* solvers would support the rapid implementation of formally verified programming abstractions that guarantee distributed consistency. In the long run, we aim to democratize the development of such domain-specific solvers by creating a *framework for domain-specific solver development* that brings new theory solver implementation within the reach of programmers who are not necessarily SMT solver internals experts.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Consistency; Software and its engineering → Software verification

Keywords and phrases distributed consistency, SMT solving, theory solvers

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.10

© Lindsey Kuper and Peter Alvaro;
licensed under Creative Commons License CC-BY
3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 10; pp. 10:1–10:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Modern internet services store multiple *replicas* of the same application data within and across data centers. Replication aids fault tolerance and data locality: if one replica fails or is unreachable due to network partitions or congestion, another will be available in its place. In addressing those problems, though, replication introduces a new problem: the problem of keeping geo-distributed replicas consistent with one another.

In a replicated system that enforces *strong consistency*, clients cannot observe that the data has been replicated at all – but strong consistency must come at the expense of *availability*, the guarantee that every request from a client receives a meaningful response. Consider a banking application in which a user's account balance is stored in a replicated object, and where the application must maintain the invariant that a user's account balance is greater than zero. When the user withdraws from the account, to maintain the $balance \geq 0$ application invariant, the replica processing the withdrawal must make sure that any concurrent withdrawals at other replicas have also been applied to its local state before it allows a new withdraw operation to proceed. In other words, the withdraw operation requires replicas to *synchronize*.

However, not every operation against a distributed data store requires strong consistency. For instance, in our banking application, replicas need not synchronize with each other for a deposit operation. Instead, the replica that processes the deposit can report success to the user immediately and remain available to process more operations, while asynchronously updating other replicas with the new balance at some point in the future.

Although it might seem like a good thing that at least some operations can proceed without synchronization, the differing synchronization requirements for different operations hugely complicate the application programmer's task. In general, different operations on the same data may require drastically different amounts of synchronization in order to maintain application-level invariants.

1.1 Language-level tools for taming the consistency zoo

To make it easier for application developers to navigate this "consistency zoo", a number of lines of research on language-level abstractions and tools for programming against replicated data have emerged. For instance:

- *Replicated data types* (RDTs), such as conflict-free replicated data types (CRDTs) [38], replicated abstract data types [36], Cloud Types [9], and replicated lists [4], are data structures designed for replication, with an interface that limits the permissible operations to those that will ensure convergence of replicated state despite message reordering or duplication.
- *Mixed-consistency programming models* augment existing languages with sophisticated type systems, contract systems, or analyses for specifying and verifying various combinations of consistency properties. Recent representative examples of this line of work include the MixT domain-specific language for mixed-consistency transactions [32], Consistency Types [20], and the Quelea contract system [39].
- *Fault injection infrastructures* for distributed systems, such as lineage-driven fault injection (LDFI) [3], systematically inject faults, including machine crashes and network partitions, to test whether (ostensibly) fault-tolerant replicated systems maintain their claimed consistency guarantees under these contingencies. In particular, to use LDFI for implementing a distributed protocol, the programmer specifies the protocol as a program in a Datalog-like distributed programming language; the LDFI system then simulates execution of the protocol in the presence of faults and tries to determine the smallest (or most likely) set of faults that reveal bugs in the program.

All of these approaches try to lift the question of whether a given program upholds a particular application-level correctness property to the level of the programming language. Language-level tools for ensuring program correctness most often manifest as type systems or program analyses that statically (and conservatively) rule out badly-behaved programs. But traditional type systems and analyses are usually not expressive enough to statically rule out programs that violate consistency properties. The desire to be able to enforce program properties that are richer than those that can be enforced by traditional type checkers or compiler analyses has led to a proliferation of work that relies on extending programming languages with SAT or SMT solvers such as Z3 [15], CVC4 [5] or MathSAT [8]. Indeed, Quelea [39] and LDFI [3] both work by augmenting languages (Quelea's contract language and LDFI's protocol specification language, respectively) with solvers.

However, verifying distributed consistency properties requires reasoning about transitive relations (e.g., causality or happens-before), partial orders (e.g., the lattice of replica states under a convergent merge operation), and properties relevant to message processing or API invocation (e.g., commutativity and idempotence) that cannot be easily or efficiently carried out by general-purpose SMT solvers that lack *native* support for this kind of reasoning. For example, ensuring the correctness of certain flavors of CRDT implementations involves showing that replica states constitute a join-semilattice and the replica merge operation computes a least upper bound over this lattice. Existing general-purpose SMT solvers lack native support for reasoning efficiently about such order-theoretic properties [18]. Consequently, proving these properties in an SMT-aided language like Liquid Haskell is harder than it needs to be. Likewise, solver-aided tools like Quelea and LDFI must make use of simplifying assumptions, compromises, or hacks in order to be able to use the solver to reason about distributed programs.

1.2 Toward consistency-aware solvers and consistency-aware solver-aided languages

We see an opportunity to address all of these shortcomings while unifying existing lines of work on programming language support for consistency, replicated data types and SMT-based tools for mixed-consistency programming. In doing so, we follow the lead of long traditions of work on *high-performance domain-specific theory solvers* [22, 23] and *high-performance domain-specific languages* [12]. Specifically, we advocate the development of domain-specific SMT-based tools that bake in support for the mathematical foundations of consistency to support the implementation of language-level abstractions and tools for ensuring the correctness of distributed programs.

We aim to make the implementation and use of these tools accessible not only to systems programmers who would ordinarily implement replicated data storage systems, but to application programmers – the people who are usually most familiar with the application-level invariants that their operations on replicated data will ultimately need to satisfy. Domain-specific solvers for distributed consistency should be a double win, enabling both improved ease of use by domain experts (because the constraints to be discharged to the solver can be encoded in a more familiar way) *and* efficiency advantages over general-purpose, off-the-shelf solvers (because the solver will be able to reason about those constraints at a higher level).

An especially exciting way to make use of the considerable power of SAT and SMT solvers is by means of *solver-aided languages*[1] such as Liquid Haskell [43], which augments Haskell's type system with *refinement types* [37] which are compiled to equivalent constraints that can

[1] The term "solver-aided languages" was coined by Torlak and Bodik in their work on Rosette [41].

be discharged by an SMT solver. With the help of the external solver, Liquid Haskell can check refinement types at compile time. We believe that close integration between a language and a solver, as pioneered by systems like Rosette [42] and Liquid Haskell, is the right path to consistent-by-construction distributed applications. Consistency-aware theory solvers would be usable from existing solver-aided languages like Liquid Haskell, and they would dovetail with Rosette's support for building new solver-aided DSLs. New domain-specific solvers call for new domain-specific solver-aided languages, and we hypothesize that building consistency-aware languages on top of our proposed consistency-aware solvers would be an ideal application of Rosette.

Finally, we hope to use the development of consistency-aware solvers as a jumping-off point for a broader research agenda. Today, new theory solver implementation is considered the territory of SMT internals experts. Even though the architecture of modern SMT solvers appears to lend itself to a modular style of development in which theory solvers could be developed independently, in practice it would seem that SMT solvers are monolithic and SMT internals expertise is required for theory solver development. We aim to create a theory solver development framework, inspired by existing frameworks for rapid development of high-performance DSLs [12], to democratize the implementation of domain-specific theory solvers. Our goal is to make it possible for domain experts – including and especially distributed systems programmers – to implement their own domain-specific theory solvers that modularly extend existing SMT solvers.

The rest of this paper is organized as follows. In Section 2, we give a brief tour of the zoo of distributed consistency models and the guarantees that they do (and don't) provide, in the context of our example banking application with its $balance \geq 0$ invariant. In Section 3, we dig into three example use cases for a consistency-aware solver: efficient verification of CRDTs with Liquid Haskell, reasoning about message reorderings in LDFI, and precisely reasoning about the transitive closure of relations in the Quelea contract language. Finally, in Section 4, we consider SMT and theory solver implementation, and the broader problem of how to democratize the implementation of domain-specific theory solvers like the consistency-aware solvers we aim to build.

2 Consistency anomalies: a brief tour of the zoo

The desire for applications to provide a responsive, "always-on" [16] experience to users has motivated much work on systems that trade strong consistency for *eventual consistency* [40, 46] and high availability. Under eventual consistency, updates may arrive at each replica in arbitrary order, and replicas may diverge for an unbounded amount of time and only eventually come to agree.

Between the extremes of eventual consistency and strong consistency are a bewildering variety of intermediate consistency options [40, 30, 1, 31, 29, 34]. For instance, under *causal consistency* [1, 27], if message m_1 is sent before message m_2 (in the sense of Lamport's happens-before relation [27]), then m_1 must also be received before m_2. Among other things, this means that in our banking application, if, say, replica A processes a deposit of \$25 followed by a withdrawal of \$30, and sends a message to replica B for each operation in that order, then replica B must apply them in that order, as well. The idea is that because the deposit of \$25 happened before the \$30 withdrawal on replica A, the deposit *potentially caused* the withdrawal, and so the withdrawal must not be allowed to happen on replica B until the deposit has happened there first. This guarantee – that operations will occur in

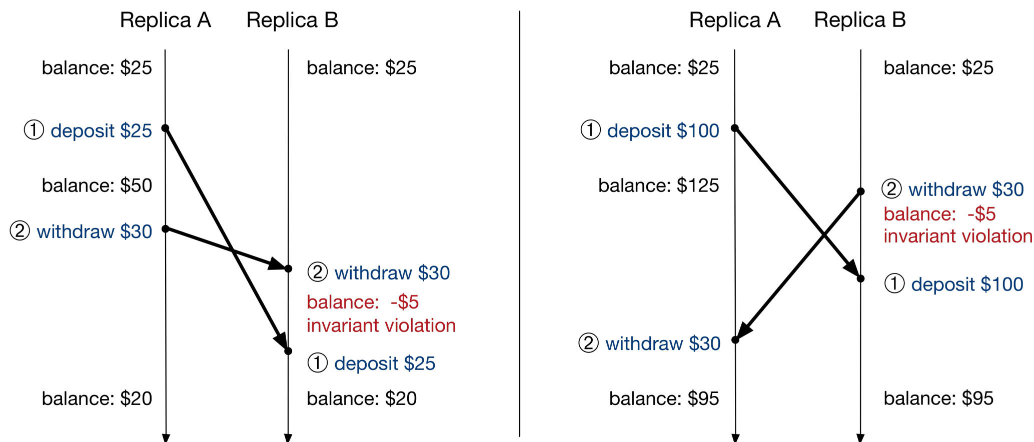


Figure 1 Examples of invariant-violating executions that are disallowed (left) and allowed (right) under causal consistency. In the execution on the left, event 2 follows event 1 in the causal order, and so the execution shown violates causal consistency. The execution on the right exhibits a different violation of the same application invariant, but in this case, one that causal consistency does *not* rule out: events 1 and 2 have no causal order, but the $balance \geq 0$ invariant is still violated.

causal order – is enough to rule out a number of consistency-related anomalies, and suffices for many applications. The execution shown in Figure 1 (left) shows a violation of the $balance \geq 0$ invariant in our bank application that would be ruled out by causal consistency.

For many applications, though, causal consistency is not enough in general. For our banking application, if a deposit takes place on replica A and a *concurrent* withdrawal takes place on replica B, then causal consistency says nothing about the order of the two operations. Enforcing the $balance \geq 0$ application invariant demands a stronger consistency guarantee. Figure 1 (right) shows such an execution, which respects causal consistency but nevertheless violates the $balance \geq 0$ invariant.

The execution in Figure 1 (left) in fact also violates a slightly weaker form of consistency, known as PRAM or FIFO consistency [30]. Under PRAM consistency, if two operations take place in a particular order on a given replica, then on any other replica *on which both operations take place*, the first operation must take place first. Figure 2 shows an execution allowed under PRAM, but ruled out by causal consistency. A withdrawal originating on replica B is delivered with no problem at replica A, but at replica C, it causes an invariant violation because a causally earlier deposit has not yet arrived. On the other hand, even PRAM consistency may be overkill for the deposit operation: if two deposits are delivered in different orders on different replicas, then we have violated PRAM (and causal) consistency, but no application invariant is violated, and we have saved the potentially substantial cost of having to synchronize between replicas.

Clearly, choosing the consistency guarantee for each operation that will enforce exactly as much synchronization as is necessary between replicas – but no more than that – can be a daunting task for programmers, even when the data store or stores offer only two or three consistency options to choose from [32, 39]. When there are more choices, the job only gets harder. For example, *read-your-writes* consistency [40] occupies a space between PRAM and weak consistency and is incomparable with eventual consistency. A recent survey paper [45] catalogues over 50 distinct notions of consistency from the literature, ordering them by their semantic strength. Figure 3 illustrates a small slice of this consistency hierarchy.

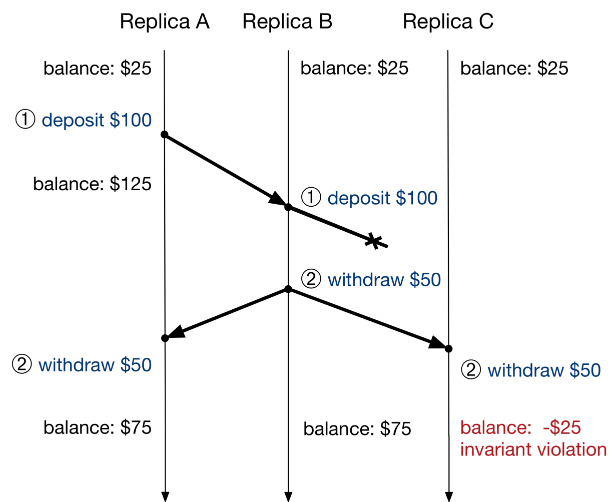


Figure 2 An invariant-violating execution allowed by PRAM consistency, but disallowed by causal consistency. Here, replica B attempts to send event 1 to replica C, but the message is lost or delayed in transit. Meanwhile, the (causally later) event 2 arrives at replica C.

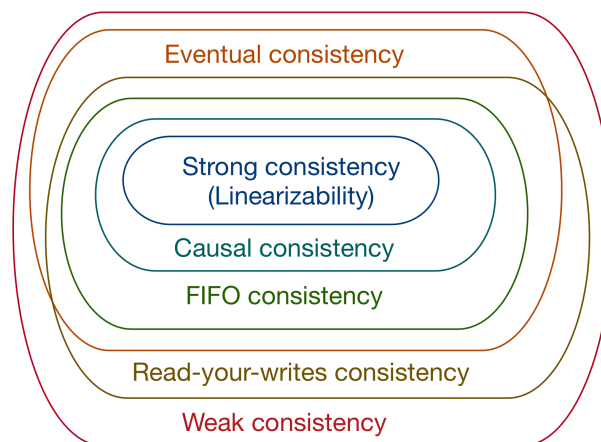


Figure 3 Some popular models of distributed consistency, ordered by strength: smaller regions admit fewer executions.

2.1 Discussion

The current state of the art of language-level support for distributed consistency discussed in Section 1.1 can help to address some of the difficulties that arise when navigating the consistency zoo. For example, if we programmed our banking application with a mixed-consistency programming system like Quelea, we might be able to obtain assurance that deposit operations may proceed without synchronization, and a warning that even casual consistency is not sufficient to ensure that concurrent withdrawal operations are safe. Alternatively, we might use RDTs to ensure that replicas eventually converge to the same (possibly negative) balance after all operations are applied, and we could use LDFI to ensure that the effects of these operations remain durable in the face of message loss and machine crashes.

In principle, we could even combine these approaches in the same system. For instance, a single solver-aided language could both identify which pairs of operations (e.g., deposits and withdrawals that witnessed them) must be causally ordered, as in the Quelea contract language, *and* ensure that a particular implementation of an RDT upholds its convergence guarantee (e.g., for concurrent withdrawals), by means of rich type specifications like those expressible in Liquid Haskell. The underlying solver for such a language would need to reason efficiently about the causality relation and partial orders, respectively. Unfortunately, general-purpose solvers are not necessarily well suited for such reasoning, as we elaborate on in the next section.

3 The case for consistency-aware solvers

What could we do if we had a solver capable of natively reasoning about distributed consistency? In this section, we motivate the need for consistency-aware solvers with three example use cases, based both on our own experiences [3] and on our reading of the literature on solver-aided language verification tools [39, 43, 44].

3.1 Efficient verification of CRDTs with Liquid Haskell

Liquid Haskell [43] is a tool that augments Haskell's type system with refinement types, calling off to an SMT solver (Z3 by default) under the hood for type checking and inference. Refinement type checking and inference is undecidable in general, but Liquid Haskell gets around this issue by employing *liquid types* [37], which require one to specify up front a fixed set of *logical qualifiers* from which refinement predicates can be built. The solver can then search over that set, returning the problem to the realm of decidability.

Liquid Haskell can be used to verify the commutativity, associativity, and idempotence of functions – all properties that one would want to guarantee about operations that modify replicated data structures. These are, in fact, exactly the properties that *must* hold of the merge operation in CRDTs. Ensuring that an RDT implementation is correct amounts to showing that certain order-theoretic properties (e.g., that replica states constitute a join-semilattice and the replica merge operation computes a least upper bound over this lattice) hold over program state. Unfortunately, these properties are difficult to verify in practice. Even just specifying the desired RDT behavior is a hard problem in itself [10, 6], and to our knowledge, the only *mechanized* proofs of correctness of RDT implementations have been done in the context of a theorem prover [19, 48], rather than in a solver-aided language like Liquid Haskell that might be usable for real implementations.

In order to verify such properties in Liquid Haskell, though, one would need to make use of its recently added *refinement reflection* mechanism [44], which goes far beyond what traditional refinement types can express and which is only well understood by a handful of experts. Furthermore, checking that the ordering laws hold for concurrent sets, for example, took 40 seconds and hundreds of SMT queries [44]. What if, by hooking up Liquid Haskell to a domain-specific solver with built-in knowledge of ordering constraints, we could get that 40 seconds down to 4 seconds or 0.4 seconds? Doing so would allow for fast, interactive verification in the REPL, in the way that Haskell programmers are used to interacting with the type checker.

3.2 Reasoning about message reorderings in a lineage-driven fault injection tool

Lineage-driven fault injection (LDFI) [3] is a methodology for testing distributed systems for fault tolerance by systematically causing certain messages to be dropped. LDFI makes use of the concept of *data lineage* – that is, the combination of data and messages that led to a particular successful execution outcome – to make decisions about which message omissions would be most likely to reveal bugs.

However, the existing LDFI implementation [3] does not consider message reorderings at all. Rather, it assumes a fixed, deterministic message ordering for a successful execution and then uses a SAT solver to exhaustively produce possible ways to make the execution fail. Yet many real-world bugs in distributed systems [47, 28] are triggered by a combination of message omission faults with *message races*, in which (for example) messages delivered in a unexpected order compromise a system's response to a fault event. In order to bound the (already astronomically large, due to the combinatorial explosion of possible faults) space of possible executions to consider, LDFI must assume that messages are delivered in a fixed, deterministic order. Hence bugs that are triggered by combinations of faults and message races could be missed.

A naive solution that, for each combination of faults selected by LDFI to inject, exercised all possible message delivery orders would be intractable for anything but small systems. Of course, in a great many cases, the order in which messages are delivered to a particular replica has no effect on the final state or behavior of that replica, because those messages *commute* with respect to the downstream message-processing logic. A possible solution to this problem would be to reason directly about the mathematical properties of the message-processing program logic. This resembles the problem of verifying the properties of the CRDT merge operation. With the help of a solver capable of reasoning about commutativity, we might be able to prove that particular pairs of messages commute, and pay the cost of exploring a larger state space only for those pairs of operations for which we cannot find such a proof.

3.3 Precise reasoning about transitive closure in a language of consistency contracts

The Quelea programming model [39] offers an innovative approach to programming against replicated data stores that offer a mix of consistency guarantees. The idea is that programmers annotate functions that operate on replicated data with contracts that describe application-level invariants, expressed in a small contract language. For example, to enforce the $\mathit{balance} \geq 0$ invariant from our example banking application, the programmer can annotate the withdraw operation with a contract specifying that calls to it must synchronize with other withdraws or deposits. A given operation might be executed against several different backing stores,

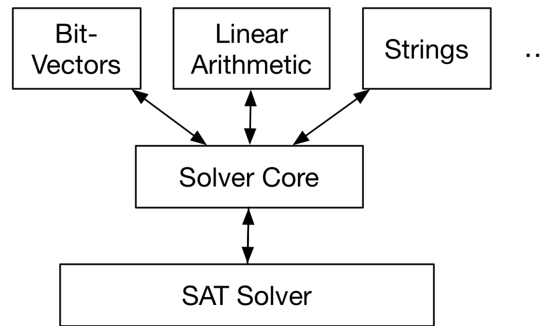


Figure 4 A lazy SMT architecture.

each of which has its own consistency guarantee, specified using the same contract language. Quelea then calls off to an SMT solver, which performs *contract classification* to determine the weakest consistency level at which each operation must be run (and therefore the backing store against which it can run with the least synchronization). Under this approach, the application programmer is able to think in terms of the application-level invariants that their code needs to satisfy, instead of having to think about the whole zoo of consistency options.

The Quelea contract language had to be carefully crafted to make contract classification decidable. For example, causal consistency is defined in terms of a transitive happens-before relation: if event 1 happens before event 2 and event 2 happens before event 3, then event 1 happens before event 3. However, transitive closure is not expressible in first-order logic, so the developers of Quelea have to make do with a relation that expresses a *superset* of transitive closure. As a result, in Quelea's notion of happens-before, some events that are actually independent are instead considered to have a happens-before relationship, leading to more synchronization than is strictly necessary to enforce causal consistency. A custom consistency-aware solver could make it possible to avoid such compromises.

4 Building a consistency-aware solver

SMT solvers allow us to check that an implementation satisfies a specification by encoding both as a formula understood by the solver, where satisfiability (or unsatisfiability) of the formula corresponds to the truth of the property we want to verify. The SMT problem is a generalization of the famous SAT problem of determining whether a Boolean formula is satisfiable. With SMT, formulas may additionally contain expressions that come from various *theories* – the "T" in "SMT". For instance, in the theory of linear real arithmetic, formulas can contain real-valued variables, addition, subtraction, and multiplication operations, and equalities or inequalities over the real numbers. Modern solvers such as Z3 come with a variety of built-in theories, such as linear arithmetic, bit-vectors, strings, and so on.

SMT solvers may operate in either an *eager* or an *lazy* manner. In the eager approach, the solver takes the SMT formula provided as input and essentially compiles it down to a Boolean SAT formula, which it can then hand off to a SAT solver. This is possible to do as long as the theory of the input formula is decidable, but in the process of compiling to the Boolean SAT formula, one may lose the high-level structure of the problem, and with it the ability to efficiently apply domain knowledge from the original theory. Modern SMT solvers therefore tend to use the lazy approach, which also involves an underlying SAT solver, but additionally involves a collection of *theory solvers* that are each specific to a certain theory. Theory solvers reason at a higher level of abstraction than the underlying SAT solver, with which they communicate via a *solver core*, as shown in Figure 4.

The efficiency advantage of the lazy approach to SMT solving is an example of the more general principle of *exploiting domain knowledge for efficiency.* For example, high-performance embedded domain-specific languages (DSLs) such as those built using the Delite DSL framework [33, 12, 7] trade off generality to gain both expressivity and efficiency; a high-level representation of programmer intent enables the compiler to do optimizations that it would not have enough information to do otherwise.

One particularly compelling recent example of a domain-specific SMT solver is the Reluplex SMT solver for verifying properties of neural networks [22, 23]. In SMT-based neural network verification, one encodes a description of a trained network and a property to be proved about it as a formula that the solver can determine the (un)satisfiability of. The tractability of the verification task depends on the ability of the solver to reason efficiently about *activation functions*, which allow networks to learn potentially extremely complex non-linear functions (indeed, without them, a network could only compute a linear combination of its inputs). The Reluplex solver enables efficient reasoning about a particular kind of activation function, the rectified linear unit (ReLU), by extending an existing theory solver for linear real arithmetic to additionally handle a new ReLU primitive and allowing SMT constraints representing ReLUs to be lazily split into disjunctions. This "lazy ReLU splitting" approach changes many verification problems from intractable to tractable and has made possible the verification of networks one to two orders of magnitude larger than the previous state of the art could handle [35].

Although we are interested in a different domain than Reluplex, we are inspired by how it illustrates the power of domain-specific solvers to bring about significant improvements in solving efficiency. It might even be possible to apply the lessons learned from the Reluplex work to our own domain of reasoning about distributed consistency. However, it is worth pointing out that a consistency-aware solver might not necessarily extend an existing theory solver with new primitives as Reluplex does. Instead, the best efficiency improvement might come from taking functionality *away* from existing theory solvers. For instance, Ge et al. [18] developed an custom solver for reasoning about ordering constraints in concurrent programs by starting with an existing theory of integer difference logic and then customizing it to remove unwanted functionality that was irrelevant to the problem at hand.

Regardless of the approach taken, a consistency-aware theory solver would need to reason about *partial orders*, in both strict (irreflexive) and non-strict (reflexive) varieties. Partial orders are essential for specifying CRDTs [38] and notions of consistency such as causal consistency [27, 1], and have been a recurring theme in previous work on concurrent and distributed programming models [2, 13, 24, 26, 25]. However, no off-the-shelf solver that we are aware of provides a built-in theory of partial orders. Indeed, off-the-shelf solvers have difficulty handling transitive relations, forcing systems to implement conservative workarounds [39]. Partial orders are ubiquitous in computer science, in areas as diverse as static analysis [14], information-flow security [17], and the semantics of inheritance in object-oriented programming [11], and we anticipate that a solver capable of efficient native reasoning about partial orders would have applications beyond our intended domain of distributed consistency.

Finally, as part of our proposed research agenda of building consistency-aware solvers, we want to consider the broader problem of theory solver development. The architecture of a lazy SMT solver, as shown in Figure 4, appears to lend itself very well to a modular style of development in which theory solvers can be developed independently. Unfortunately, in practice it would seem that SMT solvers are monolithic, and SMT internals expertise is

required for implementing new theory solvers.[2] Although much research in programming languages (and distributed systems) now makes use of SMT solvers, the solver itself is generally treated as a black box, and theory solver implementation is considered the territory of the same SMT internals experts who implement the solver core.

We argue that *programmers should not have to be SMT internals experts in order to implement theory solvers for their domain of interest*. We propose to evaluate that claim by developing a framework for implementing custom, efficient domain-specific solvers. In doing so, we hope to democratize theory solver development and make it accessible to programmers who are not SMT internals experts, in the same way that Delite aimed to democratize DSL implementation and make it accessible to programmers who are not compiler experts.

References

1 Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995. doi:10.1007/BF01784241.

2 Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: A CALM and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011.

3 Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 331–346, New York, NY, USA, 2015. ACM. doi:10.1145/2723372.2723711.

4 Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and Complexity of Collaborative Text Editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 259–268, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933090.

5 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2032305.2032319.

6 Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying Eventual Consistency of Optimistic Replication Systems. *SIGPLAN Not.*, 49(1):285–296, January 2014. doi:10.1145/2578855.2535877.

7 Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/PACT.2011.15.

8 Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT Solver. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-70545-1_28.

9 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud Types for Eventual Consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_14.

[2] The architecture of the Reluplex solver is a case in point: the implementation was originally based on an off-the-shelf linear programming (LP) solver, but required invasively modifying the existing LP solver instead of building on top of it in a modular fashion, and the published work based on the off-the-shelf LP solver was later discarded in favor of building a new solver from scratch [21].

10 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535848.

11 Luca Cardelli. A Semantics of Multiple Inheritance. In *Information and Computation*, pages 51–67. Springer-Verlag, 1988.

12 Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A Domain-specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM. doi:10.1145/1941553.1941561.

13 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, New York, NY, USA, 2012. ACM. doi:10.1145/2391229.2391230.

14 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi:10.1145/512950.512973.

15 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

16 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. doi:10.1145/1294261.1294281.

17 Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, May 1976. doi:10.1145/360051.360056.

18 Cunjing Ge, Feifei Ma, Jeff Huang, and Jian Zhang. SMT solving for the theory of ordering constraints. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC 2015, pages 287–302, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-319-29778-1_18.

19 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, October 2017. doi:10.1145/3133933.

20 Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 279–293, New York, NY, USA, 2016. ACM. doi:10.1145/2987550.2987559.

21 Guy Katz and Clark Barrett. Personal communication, 2017.

22 Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 97–117, 2017. doi:10.1007/978-3-319-63387-9_5.

23 Lindsey Kuper, Guy Katz, Justin Gottschlich, Kyle Julian, Clark Barrett, and Mykel J. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks. *CoRR*, abs/1801.05950, 2018. arXiv:1801.05950.

24 Lindsey Kuper and Ryan R. Newton. LVars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. ACM. doi:10.1145/2502323.2502326.

25 Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 2–14, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594312.

26 Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze After Writing: Quasi-deterministic Parallel Programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 257–270, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535842.

27 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

28 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 517–530, New York, NY, USA, 2016. ACM. doi:10.1145/2872362.2872374.

29 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2387880.2387906.

30 R.J. Lipton and J.S. Sandberg. PRAM: A scalable shared memory. Technical Report no. 180, Princeton University, Department of Computer Science, 1988.

31 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. doi:10.1145/2043556.2043593.

32 Mae Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 226–241, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192375.

33 Kunle Olukotun. High Performance Embedded Domain Specific Languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 139–140, New York, NY, USA, 2012. ACM. doi:10.1145/2364527.2364548.

34 Matthieu Perrin, Achour Mostefaoui, and Claude Jard. Causal Consistency: Beyond Memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 26:1–26:12, New York, NY, USA, 2016. ACM. doi:10.1145/2851141.2851170.

35 Luca Pulina and Armando Tacchella. An Abstraction-refinement Approach to Verification of Artificial Neural Networks. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, pages 243–257, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-14295-6_24.

36 Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, 2011. doi:10.1016/j.jpdc.2010.12.006.

37 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. doi:10.1145/1375581.1375602.

38 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2050613.2050642.

39 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737981.

40 Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=645792.668302.

41 Emina Torlak and Rastislav Bodik. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509586.

42 Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594340.

43 Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 39–51, New York, NY, USA, 2014. ACM. doi:10.1145/2633357.2633366.

44 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, December 2017. doi:10.1145/3158141.

45 Paolo Viotti and Marko Vukolić. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, June 2016. doi:10.1145/2926965.

46 Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009. doi:10.1145/1435417.1435432.

47 Pamela Zave. Using Lightweight Modeling to Understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012. doi:10.1145/2185376.2185383.

48 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal Specification and Verification of CRDTs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

A Tour of Gallifrey, a Language for Geodistributed Programming

Mae Milano

Cornell University, Ithaca, NY, USA
`https://www.cs.cornell.edu/~milano`
milano@cs.cornell.edu

Rolph Recto

Cornell University, Ithaca, NY, USA
`https://www.cs.cornell.edu/~rector`
rector@cs.cornell.edu

Tom Magrino

Cornell University, Ithaca, NY, USA
`https://www.cs.cornell.edu/~tmagrino`
tmagrino@cs.cornell.edu

Andrew C. Myers

Cornell University, Ithaca, NY, USA
`https://www.cs.cornell.edu/andru`
andru@cs.cornell.edu

Abstract

Programming efficient distributed, concurrent systems requires new abstractions that go beyond traditional sequential programming. But programmers already have trouble getting sequential code right, so simplicity is essential. The core problem is that low-latency, high-availability access to data requires replication of mutable state. Keeping replicas fully consistent is expensive, so the question is how to expose asynchronously replicated objects to programmers in a way that allows them to reason simply about their code. We propose an answer to this question in our ongoing work designing a new language, Gallifrey, which provides orthogonal replication through *restrictions* with *merge strategies*, *contingencies* for conflicts arising from concurrency, and *branches*, a novel concurrency control construct inspired by version control, to contain provisional behavior.

2012 ACM Subject Classification Software and its engineering → Cooperating communicating processes; Software and its engineering → Massively parallel systems; Software and its engineering → Distributed programming languages

Keywords and phrases programming languages, distributed systems, weak consistency, linear types

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.11

Funding This work was supported by NSF grants 1717554 and 1704788.
Mae Milano: This research was conducted with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, Grant number 32 CFR 168a.

Acknowledgements We would like to thank Fabian Muehlboeck, Andrew Hirsch, the members of the Applied Programming Languages Group at Cornell University, and our anonymous SNAPL reviewers for their helpful feedback on drafts of this paper.

© Mae Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers;
licensed under Creative Commons License CC-BY

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 11; pp. 11:1–11:19

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The modern internet landscape is filled with *geodistributed* programs: single logical applications split among thousands of machines across the globe. These programs present the illusion of a single available object – be it a Twitter feed, a Facebook timeline, or a Gmail inbox – which is implemented as a constellation of copies, loosely synchronized across perhaps dozens of data centers. This weakly consistent replication became popular due to its performance benefits, but at a significant cost: where objects were once stored on databases offering strong consistency, consistency must now be recovered through the careful effort of application programmers.

Needless to say, it is hard to correctly synchronize replicated objects in this setting. And while past work (Section 6) has created an excellent foundation, existing solutions lack modularity and compositionality. Typically, they either fail to provide whole-program guarantees or rigidly constrain what can be replicated and how it should be replicated. Few systems provide consistency guarantees without forcing the entire program into a single consistency model.

This paper proposes Gallifrey, a general-purpose language for distributed programming, whose guiding principles are extensibility, modularity, and flexible consistency. Gallifrey's design encourages extensibility and modularity through the principle of *orthogonal replication.*[1] Under orthogonal replication, the conflict-handling strategy for a replicated object is separated from the implementation of the object itself. Any object can be replicated, yet no object must be replicated.

Gallifrey embodies this principle through a language mechanism, *restrictions*. Restrictions refine the interface of a sequential object and provide a *merge function* to resolve concurrent use of allowed methods. Crucially, objects are not tied to a single restriction: programmers may implement many restrictions for a given interface, and may use these restrictions on any object which satisfies this interface. Further, the restrictions on an object may change over time.

Gallifrey combines restrictions with a strong type system to ensure *strong consistency* and *race freedom* by default. Objects in Gallifrey are subject to an ownership-based linear type system to ensure that at most a single thread has access to any given object at a time, and that fields of replicated objects can only be accessed via a correct restriction. Further, restrictions are statically checked to ensure all permitted operations commute, allowing programs to safely operate against replicated state asynchronously, without needing to coordinate during normal execution.

But strong consistency without coordination does not constitute a sufficiently powerful programming model. Gallifrey goes further by introducing the idea that restrictions can specify *provisional operations* that are not required to commute and are therefore, in general, unsafe to use without coordination. Provisional operations can be used only from within explicit *branches*, a new primitive inspired by distributed version control. Branches represent explicit forking of state and serve as the basis for threads, transactions, and speculative execution. Branches and provisional operations combine to allow speculative execution; provisional methods executed within a branch remain isolated in that branch until it is explicitly merged, either synchronously or asynchronously. When merged synchronously, branches have the semantics of optimistic transactions, and thus sacrifice no consistency; when merged asynchronously, branches have a weakly consistent semantics, as provisional

[1] The name is inspired by orthogonal persistence [6].

```
1  interface Library {
2    int numItems();
3
4    unique Set[Item] getItems(unique String col)
5      requires collection(col);
6
7    void addCollection(unique String col)
8      ensures collection(col);
9
10   void addItem(unique Item i, local String col)
11     requires collection(col)
12     ensures next(numItems()) >= numItems();
13
14   // also moves items in col to a default collection
15   void removeCollection(local String col)
16     ensures !collection(col) && (next(numItems()) == numItems());
17 }
```

Figure 1 Library interface. `requires` and `ensures` refer to pre- and postconditions respectively. `collection` is an abstract predicate indicating the presence of a collection in the library.

operations contained within a branch may conflict with other concurrent operations. To compensate for such conflicts, programmers provide a callback as a *contingency* to be executed if a conflict does occur.

We are working toward a formalization and implementation of Gallifrey, and we hope that it adds to the recent resurgence in designing language abstractions for distributed systems.

2 A running example

To better understand the difficulties of programming with replicated objects and how Gallifrey makes this task easier, we introduce a running example. Consider a "library" object (Figure 1) that maintains a set of items grouped by collections – for example, a set of books collected under "Programming Languages" might include *Structure and Interpretation of Computer Programs* [1] and *Types and Programming Languages* [46]. Alice and Bob use this library object to keep citations for a paper they are writing together. Like many academics, Alice and Bob find themselves frequently traveling to conferences, working on their bibliography on the go – including in places with limited internet connectivity. Their bibliography application must allow them to continue working while disconnected. Now, suppose Alice adds a book to the collection, *How to Design Programs* [27], while at the same time Bob removes the "Programming Languages" collection itself, adding its orphaned contents to a default collection. To what state of the library should Alice and Bob's devices both eventually converge?

There are two strategies for responding to such irreconcilable conflicts. One is *prevention*: restrict concurrent execution of operations that might conflict. For example, Alice and Bob might agree to not remove collections from the library so that either of them can add books safely. The second is *restoration*: provide a way to safely merge conflicting operations.[2] Alice and Bob can agree on a restorative strategy by allowing book additions and *provisionally*

[2] Indigo [10] makes a similar distinction between *conflict avoidance* and *conflict resolution*.

```
1  class RemoveCollectionLost {
2    local String collection;
3    RemoveCollectionLost(unique String col) { collection = col; }
4  }
5
6  restriction AddOnly for Library {
7    allows addItem;
8  }
9
10 restriction AddWins for Library {
11   allows addItem;
12   allows removeCollection contingent RemoveCollectionLost;
13   test sizeAtLeast(int n) { return numItems() >= n; }
14
15   merge (op1, op2)
16   where op1 = addItem(_,c) && op2 = removeCollection(c) {
17     delete op2 with RemoveCollectionLost(c);
18   }
19 }
```

Figure 2 Restrictions for `Library` interface.

allowing collection removals (with contents moved to a default collection), understanding that in the case of a concurrent addition and removal, the removal will be invalidated. If Bob removes a collection under this restorative strategy, he needs to understand that his removal could be invalidated, and likely wants to be notified if the invalidation happens.

Now suppose Alice gets on a plane and wants to see what books are in the library. Without being connected to Bob, Alice can't be sure that the list of books she's seeing contains all the books in the library; after all, Bob could have added more books while Alice wasn't looking. Alice might be fine with this. She might just want an estimate of the state of the library – with the option to receive a notification later if her estimate was inaccurate. Or perhaps she was only interested in checking if the library was at least a certain size. This she can do safely even without Bob, since Alice and Bob both agreed not to remove items from the library.

Gallifrey's programming model is designed for this challenging setting.

3 Restrictions for shared objects

The primary purpose of Gallifrey– safely sharing objects via asynchronous replication – is enabled by *restrictions*. Restrictions represent the conflict-handling strategies for replicated objects. Restrictions are a part of the type of a replicated object, and Gallifrey uses them at compile time to ensure that all replicas agree on a conflict-handling strategy. Syntactically, an object declared with type `shared[R] T` is of class `T` and is shared under a restriction `R`.

Restrictions are defined against a specific interface. For example, Figure 2 shows two possible restrictions for library objects: `AddOnly`, which only allows `addItem` operations, and `AddWins`, which allows `addItem` and `removeCollection` but invalidates `removeCollection` in case of conflicts. These correspond to the two conflict-handling strategies in Section 2. A restriction consists of the following parts:

Interface refinements. Restrictions specify exactly which operations of an interface are allowed under them. Any operation not specified in a restriction cannot be executed under it, thus allowing for preventative conflict-handling strategies. For example, in Figure 2 `AddOnly` prohibits collection removals. Allowed operations in a restriction can also be marked `contingent`, indicating that they are a *provisional* operation; this operation may need to be rolled back due to conflicts. Client code that executes such provisional actions can register *contingency callbacks*[3] for such cases of invalidation (Section 5.1). For example, in Figure 2, the `AddWins` restriction allows for both adding items and removing collections; the latter is provisional and can be later invalidated with a `RemoveCollectionLost` contingency. In the scenario from Section 2, this contingency could be used to send Bob a message if he attempted to remove a collection while Alice was concurrently adding items to it.

Merge functions. Restrictions include *merge functions* to handle any conflicts that may arise when two operations execute concurrently, thus allowing for restorative conflict-handling strategies. Merge functions pattern-match over pairs of operations and their arguments, and then dictate whether to edit the operations, delete them, or synthesize new ones. Merge functions can also call contingencies of provisional operations as needed. For example, in Figure 2, the merge function for `AddWins` invalidates collection-removal operations concurrent with operations that add an item to the same collection, and then calls the `RemoveCollectionLost` contingency of the invalidated remove operations.

Monotonic tests. Because updates to replicated objects can be reordered, reads of the object's state before convergence can vary across replicas. Thus, reading a replicated object's state directly is usually eschewed: instead, a special class of reads, found in programming models such as LVars (*threshold reads*) and Lasp (*monotonic reads*), is defined [36, 41]. Restrictions provide a similar functionality with *monotonic tests*: boolean expressions whose value is guaranteed to remain true once it becomes true, no matter what further operations are received by the replica. With this property, monotonic tests can be used for *triggers*, code whose execution is blocked until a monotonic test becomes true. For example, in Figure 2 the `AddWins` restriction has a `sizeAtLeast` test that returns whether the number of items in the library has passed some threshold. If Alice (from Section 2) is worried that the library is getting too big, then this test can be used to inform her that the library is bigger than some threshold size. This test cannot be invalidated because collection removals do not remove items, but rather move them to a default collection.

3.1 Safety guarantees

Importantly, restrictions are intended to offer the following type-safety guarantees:

- No object can perform an operation forbidden by the restriction under which it is shared.
- Merge functions are *exhaustive*: all possible conflicts between operations allowed under a restriction are handled by a merge function declared in the restriction.
- Monotonic tests cannot be invalidated: once their value is true, their value will always be true afterward *until replicas explicitly coordinate*.

Taken together, these three guarantees provide a strong safety result: a program with correct merge functions, correct precondition and postcondition annotations, and no contingencies, always enjoys strong consistency.

[3] Helland and Campbell call these "apologies." [34].

To support these guarantees, we take inspiration from Indigo [10] and annotate interfaces of shared objects with pre- and postconditions, which are written as logical formulas over abstract predicates and read operations defined in the interface. Abstract predicates do not have a concrete definition; they are asserted directly in pre- or postconditions in order to describe the assumptions and effects of an operation over an object's state. Including read operations in the language of postconditions allows us to connect these postconditions with the state of the object, describing how subsequent reads will be affected by an operation. These annotations allow the detection of conflicts that arise from concurrent operations – for example, when the postcondition of one operation violates the precondition of another, or when two operations have conflicting postconditions. Thus the type checker can determine whether all such conflicts are handled by the merge function. The annotations can also be used to determine whether operations can violate the monotonicity of tests. Like Indigo, we plan to use an SMT solver to verify that the pre- and postcondition annotations on operations are consistent with our desired safety guarantees [10].

Consider the annotated `Library` interface in Figure 1 and its restrictions in Figure 2. Here, `addItem` adds an item to an existing collection if it is not already in the collection, so its postcondition says that the return value of `numItems` after invocation of `addItem` (i.e. `next(numItems())`) is at least the return value of `numItems` before invocation – the number of items in the library remains the same or it increases by one. Meanwhile, `removeCollection` removes a collection from the library without removing the items in it from the library, instead moving orphaned items (those not in any other collection) to a default collection. Since the postcondition of `removeCollection` violates the precondition of `addItem` when their arguments reference the same collection, the concurrent operations conflict, which is handled by the merge function for `AddWins` – otherwise, if the merge function does not handle this conflict, `AddWins` will be rejected at compile time because its merge function is not exhaustive. Note that the `sizeAtLeast` test in `AddWins` is verified to be monotonic at compile time because the allowable operations under `AddWins` have postconditions that do not decrease the value of `numItems()`.

3.2 Transitioning between restrictions

One might find shared object restrictions *too* restrictive: since they are essentially static contracts, they might appear to ban certain operations throughout the entire lifetime of a shared object. Prior work [40, 48, 57] has shown that loosely synchronized replicas can eschew coordination for most operations, and then coordinate only to safely change established invariants. Taking a cue from this work, we propose the ability to *transition* shared objects across restrictions. The strict separation of object implementations and conflict resolution strategies allows programs to dynamically transition between restrictions, changing the conflict-handling strategy of shared objects over time. Coordination between replicas during transition points ensures that replicas always agree on the conflict-handling strategy for an object. We introduce new language constructs to support this feature.

Union restrictions. First, we introduce a new kind of restriction, a *union restriction*, which is composed of a set of restrictions. Replicated objects shared under a union restriction always are associated with a concrete restriction, which must be a member of the union, at runtime. Like regular restrictions, union restrictions are part of the type of a shared object: syntactically, an object declared with type `shared[U] T` is of class `T` and is shared under a union restriction `U`. For example, Figure 3 defines a union restriction `Threshold` defined for the `Library` interface at line 5, ranging over the `AddWins` and `ReadOnly` restrictions, and the field `library` is declared as a `shared[Threshold] Library`.

```
restriction ReadOnly for Library  {
  allows getItems;
}

restriction Threshold = AddWins | ReadOnly

class LibraryClient {
  shared[Threshold] Library library;
  shared[Messaging] User user;

  public LibraryClient(shared[Threshold] Library lib,
                       shared[Messaging] User u) {
    library = lib;
    user = u;
    match_restriction library with
    | shared[AddWins] Library awlib {
      changeRestriction(awlib);
    }
    | shared[ReadOnly] Library rolib { }
  }

  void addItem(unique Item item, unique String collection) {
    match_restriction library with
    | shared[AddWins] Library awlib {
      awlib.addItem(item, collection);
    }
    | shared[ReadOnly] Library rolib {
      throw ClientException("Library is read only!");
    }
  }

  void removeCollection(unique String collection) {
    match_restriction library with
    | shared[AddWins] Library awlib {
      provisionallyRemove(awlib, collection);
    }
    | shared[ReadOnly] Library rolib {
      throw ClientException("Library is read only!");
    }
  }

  unique Set[Item] getItems(unique String collection) {
    match_restriction library with
    | shared[AddWins] Library awlib {
      throw ClientException("Library must be read only!");
    }
    | shared[ReadOnly] Library rolib {
      return rolib.getItems(collection);
    }
  }

  void changeRestriction(shared[AddWins] Library awlib) {...}

  void provisionallyRemove(shared[AddWins] Library awlib,
                           unique String collection){...}
}
```

Figure 3 Client that uses a shared library object.

```
1  void changeRestriction(shared[AddWins] Library awlib){
2    thread (awlib, user) {
3      when (awlib.sizeAtLeast(100)) {
4        user.sendMessage("Library is too big!");
5        transition(awlib, ReadOnly);
6      }
7    }
8  }
```

Figure 4 Using a trigger to transition restrictions.

Matching restrictions. Objects shared under union restrictions can at run time be in any of the restrictions specified; but all replicas must agree on *which* concrete restriction they are under. To determine the current restriction of an object shared under a union restriction, Gallifrey provides a `match_restriction` construct, which allows the programmer to exhaustively match over all the constituent restrictions of the union restriction. For example, at Figure 3 line 23, the `addItem` method uses this construct to test whether the library currently allows modification. Gallifrey may synchronize before this `match_restriction` to ensure that all replicas of the shared object agree on the shared object's current restriction.

Transitioning restrictions. Next, we introduce the ability to *transition* between restrictions. The primitive operation `transition()` creates a *request* to transition an object shared under a union restriction to one of its constituent restrictions. After any replica requests a transition, Gallifrey's runtime asynchronously initiates a transition at all replicas. A reference shared with a union restriction allows transitioning only among its constituent restrictions, ensuring a statically known bound on the possible restrictions on referenced shared objects. A transition induces coordination among nodes that hold a reference to the shared object to establish consensus on the new restriction for the shared object. This process is asynchronous; the transition does not necessarily take place immediately, so to use the shared object under the new restriction, one must match over the union restriction. When a transition is in progress, `match_restriction` may block until it is complete, after which the arm for the new restriction is executed. Note that `transition()` is a request: it does not guarantee that the transition occurs, since it can fail for various reasons (e.g., coordination times out, or there is a concurrent transition to another restriction that overrides the request). Thus, `match_restriction` is needed to check if the transition actually succeeds.

For an example of transitions between restrictions, consider Figure 4. The `LibraryClient` constructor calls `changeRestriction`, which creates a thread with a new replica of the library object that adds a trigger to transition its library object to `ReadOnly` when the library reaches a certain size using the `sizeAtLeast` test defined in `AddWins`. The `addItem` and `removeCollection` methods match on the current restriction of the library to ensure it is `AddWins`; otherwise the methods throw an exception. The `getItems` method does something similar for the `ReadOnly` restriction.

4 Tracking aliasing and replication

Restrictions are an answer to how objects are shared – but not all objects need to be shared, and we do not want to pay the cost and complexity of sharing unnecessarily. Therefore Gallifrey must support both *replicated* and *non-replicated* objects. When these replicated and non-replicated objects interact, Gallifrey needs to guarantee *restriction safety*: all fields of a replicated object, if not explicitly shared under their own restriction, can only be accessed via the object's restriction.

To see why this rule is important, consider the following example: an implementation of the `Library` interface (Figure 1) which internally uses a `Set` to store its contents. When an instance of this `Library` is shared under some restriction, Gallifrey relies on the fact that all mutations to the internal state of this shared library occur via operations on its restricted interface. If the internal `Set` is accessed via an alias outside of the shared library's restriction, then there is no guarantee any mutations made via this outside alias adhere to the restriction's requirements.

Gallifrey captures the interactions between replicated and non-replicated objects via three reference qualifications: `shared`, `unique`, and `local`. The `shared` qualifier indicates that objects of the qualified type are replicated. When a `shared` reference is passed to a new branch or thread (Section 5.1), it implicitly constructs a new replica owned by that thread. These `shared` references can be created by combining an existing `unique` reference with a restriction, after which the original `unique` reference is destroyed. Due to the implications of orthogonal replication, the implementation of a shared object need not know it is shared; an object sealed under a restriction and `shared` reference still has unrestricted access to itself. We see `shared` references in use in Figure 3.

Handling non-replicated references is somewhat more complex. Our type system for `unique` and `local` (collectively, *unshared*) references must provide two core guarantees: that at most a single thread has access to an unshared reference at a time (race freedom) and that no references to the unshared fields of shared objects exist outside of those shared objects (restriction safety). To enforce these guarantees, we propose to combine a linear type system with a notion of ownership (as has been done previously [29]). Unshared references in Gallifrey are always ultimately owned by a thread or shared object. We treat `unique` references linearly, while `local` references, which can be aliased *within* an owner, must be externally reachable via only their owner.

The `unique` reference qualification denotes *transferable* ownership. A unique reference is an affine resource; its use is tracked by the type system and it cannot be aliased. A `unique` reference *dominates* its object graph: all references transitively reachable via a `unique` reference are reachable via no other external references. This provides an isolation guarantee: a `unique` reference is the only way to access the object graph to which it refers. This guarantee in turn allows Gallifrey to send `unique` references across concurrency boundaries without inviting race conditions or requiring costly run-time techniques. Similar unique references have received a great deal of support in recent language designs, including Rust [49], C++-11 [18], Wyvern [43], and Pony [22]. Unlike many of these languages, Gallifrey does not propose to use linearity to track memory usage, but rather only to prevent concurrent access. Because of their ability to cross concurrency boundaries, `unique` references are the correct reference to use when sending messages to shared objects, as we see in Figure 3.

The `local` reference qualification denotes *non-transferable* ownership. A `local` reference statically knows its direct owner but is *not* linearly tracked. Direct owners of `local` references are inferred at creation time based on the context in which the `local` reference is created; for example, a `local` reference created in a constructor which received only `unique` references as parameters is directly owned by the object under construction. We see a use of `local` references in Figure 2. Local references cannot escape their owner without destroying it; in exchange for this restriction, local references enjoy relaxed aliasing rules. Local references can be freely aliased so long as all aliases share the same owner. For example, a set of `local` references owned by a single object are allowed to form cycles to each other. Like `unique` references, `local` references are inspired by a long history of language design [3, 14, 15, 21].

```
1  void provisionallyRemove(shared[AddWins] Library awlib,
2                           unique String collection){
3    Branch tok = branch(awlib, collection) {
4      awlib.removeCollection(collection);
5    };
6    tok.pull(RemoveCollectionLost rclost => {
7      user.sendMsg("Cannot remove collection " + rclost.collection);
8    }, Success succ => {
9      user.sendMsg("Removed collection " + succ.collection);
10   });
11 }
```

Figure 5 Using branches for a provisional operation with contingency.

5 Revisiting provisionality: branches and contingencies

Section 3 discussed Gallifrey's use of restrictions to guarantee strong consistency and whole-program convergence in the absence of provisional methods. But without provisional methods, only very limited sets of operations may appear in a restriction – for example, commutative writes and tests, or exclusively reads. These limitations are impractical for many common programs; sometimes programs may need to read *and* write a shared object, without stopping for consensus between operations.

This is exactly the role of *provisional* methods. Provisional methods leave open the possibility of conflicts; in exchange, there are no limitations on what a provisional method can do. These methods are executed optimistically, allowing users to continue operating against replicated state without stopping for consensus.

But one cannot simply execute potentially conflicting actions without acknowledging the significant inconsistency invited by doing so. To partially recover from this, Gallifrey pairs every provisional method with a *contingency*: a named callback intended to recover from – or at least apologize for – any consistency error resulting from using a provisional method. Contingencies are invoked directly from the merge function for the associated restriction, and so can receive any necessary information from the merge. As a simple example of the use of these features, recall the running example introduced in Section 2. In this example, we considered allowing Bob to *provisionally* remove a collection from the library, while leaving open the possibility that a merge function would reject this operation. To compensate, Bob registers a contingency callback, which sends an error message indicating the removal did not take place (Figure 5, line 7).

5.1 Branches

Using provisional methods and contingencies raises important semantic questions. After the invocation of a provisional method on a shared object, are all subsequent uses of this object also provisional? If a provisional observation from an object flows to other values in the program, should those values also be considered provisional? What if that flow reaches different, unrelated shared objects? And precisely where is the right place to register a contingency callback – close to the provisional invocation, or close to the eventual visible use of its result?

In Gallifrey, the key to answering all these questions is a new mechanism called *branches*. Branches exist to contain provisionality; like their namesake in the world of version control, every branch possesses its own fork of state, isolated from external mutations until it is *merged*

back into its parent. Programmers may enter and exit ("check out") in-progress branches, spawn sub-branches, and freely choose to discard or merge branches. When a provisional operation occurs within a branch, then the *entire branch* is considered provisional; any code that executes after a provisional operation may be tainted by that provisional operation, and so inherits all of its potential for conflict. Helpfully, branches also allow deferring the point at which contingencies are required. Because branches are strongly isolated from the remainder of the system, any potential conflicts are safely contained within the branch; the only point at which these conflicts become visible is when the branch attempts to merge with the outside world. It is precisely this point at which we require programmers to supply contingencies.

Syntactically, branches are created by the syntax `branch(args...){body}`, as in Figure 5. The `args...` is a list of `shared` or `unique` objects that the branch now owns, and which are available within the branch's body. When a `unique` object passes into a branch, its ownership is moved; thus past references to these objects are no longer valid. When a shared reference first passes into a branch, a new replica is made for the branch. The branch's body is executed immediately, after which control proceeds with the statement immediately following the branch. The `branch` construct also returns a token via which programmers can interact with the branch. This token is linear; it cannot be aliased, and it must be either merged or aborted before it goes out of scope.

A typical use of the branch's token, as seen in Figure 5 on line 6, is to optimistically merge the branch into the calling context via `pull`. This construct immediately merges everything in the branch with `pull`'s calling context, leaving open the potential for conflict with other, as-yet-unmerged branches. Because of this potential for future conflict, users must provide a set of contingency callbacks covering all provisional behavior that occurred on this branch. These callbacks are intended as a means to repair any damage done in the case of a consistency violation caused by any conflict.

To avoid the possibility of conflict, Gallifrey's branches also support a synchronous `commit` operation. `commit` blocks until a consensus can be reached among replicas, deciding which provisional operations are consistent with global state, and which, having been found in conflict with already accepted operations, should be rejected by the system. After commit, all effects from within the branch become visible to the wider system; operations rejected due to conflict are re-executed against consistent state, with the new results replacing the old. With `commit`, branches become a generalization of transactions. Branches operate on an isolated snapshot of state, apply the effect of all their operations, verify that their snapshot remains consistent with the system at large, and re-execute their operations if not.

This token can be used for more advanced features as well. With `*token*.abort()`, programmers can explicitly abandon the branch. With `*token*.peek`, programmers can steal a reference to the branch's state *without* first merging the branch, and *without* needing to supply contingencies so long as the result of `peek` does not influence any visible actions outside the branch.

These features are illustrated in Figure 6. This figure introduces the example of withdrawing from an ATM. The method takes a shared bank account which supports provisional `withdraw()` and provisional `balance()`, with contingencies `Overdraft` and `UpdatedBalance` respectively. The withdrawal is allowed to proceed provisionally if the chance for overdraft is low; if the chance of overdraft is high then it instead chooses to synchronously `commit` the withdrawal.

```
1  void atm_withdraw(shared[All] Account acct, unique Integer amnt) {
2    Branch tok = branch(acct,amnt){
3      unique Integer withdrawn_amnt = acct.withdraw(amnt);
4      unique Double percent = withdrawn_amnt / acct.balance();
5    };
6    if (tok.peek[percent] <= 0.25){
7      tok.pull(Overdraft amnt => { charge_overdraft(acct) },
8               UpdatedBalance => {/* ignore */});
9    } else tok.commit();
10 }
```

Figure 6 More advanced features of branches.

5.2 Information flow in branches

Branches in general – and `peek` in particular – require fine-grained tracking of provisionality. Gallifrey tracks provisionality using an information flow type system [50].

In an *information flow* type system, values are associated with labels drawn from a lattice. Our lattice contains elements that are sets of provisional methods, ordered by subset inclusion. Each value is labeled with the set of provisional methods which have influenced it. Values labeled with the empty set (indicating they have not been influenced by provisional behavior) live at the bottom of the lattice ($\bot$), while values which have been influenced by all possible provisional behavior live at the top ($\top$). To prevent computation from depending on provisional observations, information should be influenced only by information whose label is a subset of that of the influenced information. Information flow handles both direct influence, like assignment, and indirect influence, like control flow.

Every reference and variable in Gallifrey – including `unique`, `local`, `shared`, and even branch tokens – is associated with one of these provisional labels. Branch tokens are somewhat special; branches contain computation, and so their labels indicate the set of provisional methods that have been called within them. Similarly, in order to call a provisional method on a shared reference it must be possible to type that reference with an appropriate provisional label – which in turn means it must reside within a branch that can be typed with the appropriate label.

Provisional labels define precisely where the effects of provisional behavior may be visible, enabling the safe use of `peek`. With `*token*.peek[*ref*]`, users can read a `unique` value from a branch and use this value outside of the branch's scope. This value's label contains the provisional operations from the branch which have influenced it. For example, a user can use a `peek`ed value to decide whether its branch should be synchronously `commit`ed or asynchronously `pull`ed (Figure 6 line 6).

Our information-flow types also delay the point at which contingency callbacks for peeked values must be supplied. This is because contingencies are only necessary when provisional operations have influenced some visible action; in Gallifrey, visible actions can only be influenced by values with the empty provisional label ($\bot$). Thus our information-flow type system will prevent a `peek`ed value from influencing a visible action unless the peeked value is *endorsed*, an operation which downgrades its label so that it can be used in contexts that do not allow influence from provisional operations. It is at this point of endorsement that the user must provide contingency callbacks. For example, a user might `peek` a value from a branch, transform it, and print the result; it is at the point of printing the result that the user must endorse the peek, making it easy to supply callbacks which apologize for the observed effect of the `peek` – the printed value.

6 Related work

Handling conflicts in concurrent operations. A recent trend is to treat conflict-handling strategies as part of a shared object's implementation, as seen in the literature on conflict-free replicated data types [52] (CRDTs) and as seen in programming models such as Bloom [4], Cloud Types [17], Lasp [41], and others [35, 36, 53]. Earlier systems – like Bayou [56], Dynamo [25], and others [23, 51] – often specify conflict handling separately from an object's implementation. But these systems do not ensure that conflict handling is sensible: they leave the job of merging inconsistent state entirely to the user, inviting errors by allowing partial, incorrect, or even inconsistent merge functions. Gallifrey takes this second approach, since restrictions are defined separately from interfaces, and can be defined without access to implementation internals. However, it aims to provide stronger guarantees that these systems by making restrictions part of a shared object's type, allowing unsafe use of the shared object to be rejected at compile time (e.g. when prohibited operations are used, when a merge function is not exhaustive, when the monotonicity of a test can be violated by an allowed operation).

Speculative operations. To provide higher availability in a geo-replicated setting, some systems expose speculative operations in their programming model. Correctables [32] provides a mechanism to speculate on preliminary values returned by weakly consistent operations. If the final values returned by strongly consistent operations do not match the preliminary values, then Correctables allows programmers to recompute or discard the effects of the initial speculation. PLANET [45] provides callbacks that fire depending on what stage a transaction is in before a specified timeout, also allowing users to specify a stage when it *speculatively commits* (i.e., will commit "if all goes well", with some explicit probability that all will go well). It also provides callbacks that fire when the final status of the transaction is known, allowing users to execute *apologies* [34] when it was speculated to have committed but was ultimately aborted. In a different setting, Concurrent Revisions [16] provides an intuitive programming model for parallel programs by allowing "revisions" to fork off the state of objects and then to join revisions back into their parents by merge functions specified using *revision types*. Gallifrey takes a similar approach, allowing programmers to speculate at the language level within explicit *branches* (Section 5.1) that fork off the state of shared objects. Branches can be used without coordination among replicas, in which case Gallifrey requires our own notion of "apologies" via contingency callbacks; with coordination, branches enjoy strong consistency – no apologies needed.

Coordination avoidance. Work on coordination avoidance in distributed databases has shown that nodes need to coordinate only when they would otherwise execute operations which violate specified invariants [7, 10, 40, 47, 48, 57]. Gallifrey's *restrictions* (Section 3) embody this principle by refining the interface of a shared object such that only specific operations are available at every replica. Restrictions are a *type-safe* mechanism for coordination avoidance, rejecting programs that violate invariants at compile time. In particular, Indigo presents a framework for users to develop replicated objects which allow commutative operations [10]. Indigo allows the programmer to specify pre- and postconditions, used to statically determine which pairs of operations may conflict. When operations are determined to conflict, Indigo's compiler inserts appropriate code to use *reservations* [47] in a way that is analogous to Gallifrey's restrictions. We similarly use pre- and postcondition annotations to determine when operations conflict in checking for the exhaustiveness of merge functions. Additionally, we use these annotations to check that the monotonicity of tests are not violated by allowed

operations in the restriction. Unlike Gallifrey Indigo does not support orthogonal replication; its analysis is performed on the object interface, while ours is performed on the restrictions.

Linear and ownership types. Linear and ownership type systems have been long studied as mechanisms to avoid races in concurrent code. Linear type systems were identified as a mechanism by which ownership, or alias restriction, could be tracked at least as early as Clarke's work in 1998 [21]. Using ownership and linearity to allow for safe concurrency has been explored several times, but was first investigated by Flanagan and Abadi [28]. Ownership types without linearity have been used to avoid races in several previous works [12–15, 20, 24, 33, 54]; linear type systems for concurrency safety have been similarly well-studied [26, 30, 31]. By ensuring that all owners are linear, Gallifrey can combine the concurrency protections of linear references and ownership references. This combination is reminiscent of linear regions [29].

7 Future work

We now give a high-level description of open questions, potential challenges, and possible solutions as we flesh out Gallifrey's design and implementation.

7.1 Extensions to the language design

Bootstrapping. Our language as described to this point works well for objects which require symmetric replication across a potentially unbounded group of nodes. It is mute on the question of bootstrapping: how does a newly-started node initially receive a replicated object to use? For this, we take inspiration from Fabric [38] and provide syntax by which a program can name a global variable located on some other Gallifrey node. Concretely, we plan to support the syntax `gal://hostname.tld/TypeName/Restriction/instance_name` to name the global object `instance_name` of type `shared[Restriction] TypeName` located on the machine at `hostname.tld`.

Typestate and reopening branches. Earlier in this paper, we mentioned that Gallifrey programmers can enter and exit in-progress branches.
With the syntax *`token`*`.open(args...){body}` programmers can re-enter a branch, passing it new references to own and giving it a new body to execute. The `body` here has access to all the objects the branch already owns in addition to the ones newly passed in via `open`. To further refine our information-flow type information and to enable the *`token`*`.open` feature, Gallifrey employs *typestate* on branch tokens and `unique` references. With typestate, linear items can acquire additional labels on their types as the program evolves. Combining information flow with typestate yields a novel variant of *statically tracked*, *flow-sensitive* information flow. For *`token`*`.open`, this means that provisional behavior introduced during `open`'s body does not require a provisional label on the token *before* the point of `open`. Using this we can also extend `abort` and `pull`, allowing programmers to recover (via `peek`) `unique` objects owned by branches even after they have completed.

Actors. Gallifrey's replicated objects are best suited to a setting where all replicas are peers; we cannot comfortably capture concepts like "all nodes may perform some operations and a designated owner node may perform some additional operations". To support explicitly centralized objects, Gallifrey should include a native notion of actors [2]. We have not yet explored how actors fit into the design of Gallifrey.

Subtyping on restrictions. We desire subtyping on restrictions for two reasons. First, we would like to make it easy for users to write parametric code. It should not be an error to pass a more permissive restriction (i.e., more operations allowed) to a function that expects a less permissive restriction. The second is for encapsulation: a programmer may wish to expose a reference to a shared object via a restriction that permits fewer operations on that object, retaining the more permissive restriction for themselves. To implement subtyping, we plan to view restrictions as records of their allowed operations (with contingencies) and use standard width subtyping on records.

Borrowing unshared values. Gallifrey's type system guarantees race-freedom in the presence of replicated objects, but relies on a strong set of linear types in order to do so. Using these types is restrictive; writing something as simple as a `print` function involves one `print` that takes and returns `unique` values, and another `print` which only works with `local` values. Most linear or ownership type systems avoid this problem with an explicit notion of borrowing – taking ownership of a resource temporarily, and returning it to the user afterwards. We need to create a notion of borrowing that works with both `local` and `unique` objects.

Extensions to monotonic tests. Monotonic tests are used with `when` blocks to set up a trigger. When the condition in the `when` block becomes true, then the body of the block executes. We believe the language of conditions within the `when` block's condition could be enriched. In general, it should be safe to use any function on `tests` as part of a `when`'s condition so long as those functions are monotonic with respect to boolean ordering. For example, conjunctions of monotonic tests is also monotonic: it becomes true when its conjuncts become true, and since its conjuncts never become subsequently false, it never becomes subsequently false either. We hope to take advantage of recent work by Clancy and Miller [19] to statically prove such functions monotonic and thus safe for use in triggers.

7.2 Implementation considerations

We envision Gallifrey as supporting the next generation of wide-area replicated applications. It requires an efficient, correct implementation of a compiler and a runtime system.

Run-time consistency guarantees. In order to give our branches a fighting chance of merging (without firing their contingencies) and to cut down on the number of distinct merge events in the system, we expect to provide a baseline of at least causal+ consistency [39] or even prefix consistency [55] for all objects within the system. As a result, there is a natural tree-like ordering on all events for a given replicated object – in contrast to much of the related work [5, 36, 41], which do not assume causality.

Replicated object state. We expect to represent shared objects as a log of events, containing both mutative operations (which determine the state of a shared object) and read or test operations (whose results must be respected by provisional mutations). A common performance concern for systems that maintain and merge histories for replicated objects is *compaction* – when can the system safely drop a prefix of the history that is guaranteed to be stable? One possible solution is to use a vector clock to track the latest committed update known by each replica of an object [42]. Prefixes of the history that have been committed to all replicas, as determined by minimum of the vector clock values, can be safely garbage collected by the system, avoiding extreme memory or storage overheads for long-living objects. Other potential solutions include using existing designs for consistent replicated logs that perform compaction [8, 9] or enforcing a more centralized approach to common global history [17].

Tracking active replicas and restrictions. In order to safely and consistently commit potentially conflicting updates to a replicated object or transition the restrictions of shared objects, Gallifrey must contact all other replicas and ensure they will behave consistently with respect to the update. But in order to do this, the system must know who holds replicas of shared objects and what restrictions are guaranteed by references to the replicas. Gallifrey applications are intended to operate in settings with large numbers of nodes that go through periods of disconnection, making it difficult to determine if a disconnected replica intends to reconnect and continue making progress, has failed, or is simply no longer replicating a given object or referencing it under a particular restriction. Existing systems solve this either by running an external membership service or having replicas manage the membership themselves as part of the protocol.

Consistent synchronous branch merges. As mentioned in Section 5.1 branches with provisional operations can be synchronously committed without risking provisional conflicts, giving programmers access to the strong and expressive semantics of traditional transactions. We must strive to make this *transactional commit* operation usable. In particular it must be typically fast, for otherwise programmers will be tempted to fall back to asynchronous `pull`s, inviting more provisional behavior than they may truly require. This can be solved with an appropriate choice of an efficient commit protocol such as two-phase commit (2PC) [11] or a consensus protocol such as Paxos [37] or Raft [44]. A key challenge introduced by Gallifrey is its tendency toward disconnection; it will be necessary to carry out these commits with high probability even in the presence of intermittent disconnection.

Efficient restriction matching and transitions. To ensure that matching does not require blocking and coordinating on every use, the system can provide mechanisms for nodes to acquire and reuse guarantees that an object will be operating under a specific restriction. Thus, after coordinating and identifying the current restriction once, the restriction can be reliably matched later in the application without coordinating again. Transitions, meanwhile, need to perform a consistent commit to update the allowed restrictions for references to an object. We believe this will be solved using a commit protocol, similar to merging branches.

Exposing flexibility to the user. There are many difficult tradeoffs and design decisions to be made in Gallifrey's runtime. These tradeoffs are necessarily influenced by the particular Gallifrey deployment in question: is the application running across data centers, or across phones? Whatever mechanisms we ultimately create, we must always provide the Gallifrey user with choices to better match Gallifrey's runtime characteristics to the user's deployment domain.

8 Conclusion

Our ideas for Gallifrey represent a new vision for handling concurrent, distributed programming. With *restrictions*, Gallifrey separates *what* can be replicated from *how* it is shared, and provides a statically enforced mechanism for ensuring consistent access to replicated objects. With *branches*, Gallifrey unifies threads, transactions, and replicas into a single intuitive construct. With *contingencies*, Gallifrey provides some sanity to working with weakly consistent state, allowing explicitly scoped violations of isolation and consistency.

Taken together, these features represent a compelling answer to the question of how to write distributed, concurrent, programs with replicated data. While we do not yet have an implementation of or formal results for this language, we hope that its ideas prove stimulating to readers.

References

1 Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, July 1996.

2 Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

3 Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *17th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 311–330, 2002.

4 Peter Alvaro, Peter Bailis, Neil Conway, and Joseph M. Hellerstein. Consistency without borders. In *ACM Symp. on Cloud Computing (SoCC)*, pages 23:1–23:10, 2013.

5 Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR (Conference on Innovative Data Systems Research)*, pages 249–260, 2011.

6 Malcolm Atkinson and Ronald Morrison. Orthogonally Persistent Object Systems. *The VLDB Journal*, 4(3):319–402, July 1995.

7 Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, 2014.

8 Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 1–14, San Jose, CA, 2012.

9 Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 24th ACM Symp. on Operating System Principles (SOSP), 2013.

10 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, page 6, 2015.

11 Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

12 Adrian Birka and Michael D. Ernst. A Practical Type System and Language for Reference Immutability. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 35–49, New York, NY, USA, 2004.

13 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002.

14 Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 213–223, New York, NY, USA, 2003.

15 Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *16th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Tampa Bay, FL, October 2001.

16 Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent Programming with Revisions and Isolation Types. In *25th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, OOPSLA '10, pages 691–707, New York, NY, USA, 2010.

17 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*, pages 283–307. Springer, 2012.

18 Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, 2011.

19 Kevin Clancy and Heather Miller. Monotonicity Types for Distributed Dataflow. In *Proceedings of the 2nd Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '17, 2017.

20 Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *Asian Symposium on Programming Languages and Systems*, pages 139–154. Springer, 2008.

21 David G Clarke, John M Potter, and James Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, volume 33(10), pages 48–64, 1998.

22 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *5th Int'l Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*, pages 1–12, 2015.

23 Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. Tardis: A branch-and-merge approach to weak consistency. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 1615–1628, 2016.

24 David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universes for race safety. *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 20–51, 2007.

25 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key–Value Store. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

26 Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2002.

27 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* The MIT Press, February 2001.

28 Cormac Flanagan and Martin Abadi. Types for safe locking. In *European Symposium on Programming*, pages 91–108. Springer, 1999.

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

30 Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Notices*, volume 47(10), pages 21–40, 2012.

31 Dan Grossman. Type-Safe Multithreading in Cyclone. In *ACM SIGPLAN Int'l Workshop on Types in Languages Design and Implementation (TLDI)*, pages 13–25, 2003.

32 Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 169–184, 2016.

33 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *European Conference on Object-Oriented Programming*, pages 354–378. Springer, 2010.

34 Pat Helland and Dave Campbell. Building on Quicksand. *CIDR (Conference on Innovative Data Systems Research)*, 2009.

35 Farzin Houshmand and Mohsen Lesani. Hamsaz: replication coordination analysis and synthesis. *ACM on Programming Languages (PACM)*, 3(POPL):74, 2019.

36 Lindsey Kuper and Ryan R Newton. LVars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84, 2013.

37 Leslie Lamport. The Part-Time Parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, May 1998.

38 Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A Platform For Secure Distributed Computation and Storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.

39 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symp. on Operating System Principles (SOSP)*, 2011.

40 Tom Magrino, Jed Liu, Nate Foster, Johannes Gehrke, and Andrew C. Myers. Efficient, Consistent Distributed Computation with Predictive Treaties. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, March 2019.

41 Christopher Meiklejohn and Peter Van Roy. Lasp, a language for distributed, coordination-free programming. In *Int'l Symp. on Principles and Practice of Declarative Programming*, pages 184–195, 2015.

42 Mae Milano and Andrew C Myers. MixT: a language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 226–241, 2018.

43 Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *5th Workshop on Mechanisms for Specialization, Generalization and Inheritance.*, July 2013.

44 Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014.

45 Gene Pang, Tim Kraska, Michael J. Franklin, and Alan Fekete. PLANET: making progress with commit processing in unpredictable environments. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 3–14, 2014.

46 Benjamin C. Pierce. *Types and Programming Languages (The MIT Press)*. The MIT Press, February 2002.

47 Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003.

48 Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 1311–1326, 2015.

49 Rust programming language. http://doc.rust-lang.org/0.11.0/rust.html, 2014.

50 Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

51 Hans-Jürgen Schönig. *PostgreSQL Replication*. Packt Publishing Ltd, 2015.

52 Marc Shapiro. A Comprehensive Study of Convergent and Commutative Replicated Data Types. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1–5. Springer New York, New York, NY, 2017.

53 Krishnamoorthy C Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *ACM SIGPLAN Notices*, volume 50(6), pages 413–424, 2015.

54 Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *European Conference on Object-Oriented Programming*, pages 104–128. Springer, 2008.

55 Doug Terry. Replicated Data Consistency Explained Through Baseball. *Commun. ACM*, 56(12):82–89, December 2013.

56 Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Mike J. Spreitzer. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *15th ACM Symp. on Operating System Principles (SOSP)*, pages 172–183, December 1995.

57 Michael Whittaker and Joseph M Hellerstein. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment*, 12(1):14–27, 2018.

Formal Verification vs. Quantum Uncertainty

Robert Rand
University of Maryland, College Park, USA
http://www.cs.umd.edu/~rrand/
rrand@cs.umd.edu

Kesha Hietala
University of Maryland, College Park, USA
https://www.cs.umd.edu/people/khietala
kesha@cs.umd.edu

Michael Hicks
University of Maryland, College Park, USA
http://www.cs.umd.edu/~mwh/
mwh@cs.umd.edu

Abstract

Quantum programming is hard: Quantum programs are necessarily probabilistic and impossible to examine without disrupting the execution of a program. In response to this challenge, we and a number of other researchers have written tools to verify quantum programs against their intended semantics. *This is not enough.* Verifying an idealized semantics against a real world quantum program doesn't allow you to confidently predict the program's output. In order to have verification that works, you need both an error semantics related to the hardware at hand (this is necessarily low level) and certified compilation to the that same hardware. Once we have these two things, we can talk about an approach to quantum programming where we start by writing and verifying programs at a high level, attempt to verify properties of the compiled code, and repeat as necessary.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Hardware → Quantum error correction and fault tolerance; Hardware → Circuit optimization

Keywords and phrases Formal Verification, Quantum Computing, Programming Languages, Quantum Error Correction, Certified Compilation, NISQ

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2019.12

Funding All authors are funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040.
Robert Rand: Also partly funded by a Victor Basili Postdoctoral Fellowship.

Acknowledgements We would like to acknowledge our co-authors on work reviewed here, including Jennifer Paykin, Dong-Ho Lee, Steve Zdancewic, Shih-Han Hung, Shaopeng Zhu, Xiaodi Wu, and Mingsheng Ying. We also thank the anonymous referees for their helpful feedback.

© Robert Rand, Kesha Hietala, and Michael Hicks;
licensed under Creative Commons License CC-BY

3rd Summit on Advances in Programming Languages (SNAPL 2019).
Editors: Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi; Article No. 12; pp. 12:1–12:11

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Writing quantum programs is hard. Fundamentally, a quantum program corresponds to applying a limited set of operations to vectors of complex numbers, with the goal of producing a vector with the majority of its weight in a few meaningful indices. *Measuring* this vector then returns an index with a probability corresponding to the weight at that index (we go into more detail in Section 2). For instance, if you are trying to factor 77, you might want the 7th or 11th entry to contain a number close to 1 while the other indices contain numbers close to 0, maximizing the probability that 7 or 11 is obtained. As a result, designing a quantum program requires a fair amount of effort and mathematical sophistication. This difficulty is compounded by the fact that quantum programs are very difficult to test or debug.

Consider two standard techniques for debugging programs: breakpoints and print statements. In a quantum program, printing the value of a quantum bit entails measuring it (an effectful operation) and printing the returned value. This is akin to randomly and irreversibly coercing a floating point number to a nearby integer – it will give a weakly informative answer and corrupt the rest of the program. Unit tests are similarly of limited value when your program is probabilistic; repeatedly running unit tests on a quantum computer is likely to be prohibitively expensive. Simulating quantum programs on a classical computer holds some promise (and simulators are bundled with most quantum software packages) but it requires resources exponential in the number of qubits being simulated, so simulation can't help in the general case.

Where standard software assurance techniques fail us, formal verification thrives. When we reason about a quantum program, we are reasoning exclusively about vectors, and vectors don't collapse when we analyze them. Formal verification is parametric in its inputs: Instead of reasoning about a 128-qubit implementation of an algorithm, we can prove properties of that algorithm for arbitrary arities given as arguments. Using techniques like induction, algebraic reasoning and equational rewriting we can verify the correctness of a broad range of quantum programs, as we previously showed [29] using the $\mathcal{Q}$WIRE programming language and verification tool.

Unfortunately, the challenges of measurement and simulation complexity are only the tip of the iceberg when it comes to near-term quantum computing.

For the foreseeable future, useful quantum computing will face a broad range of obstacles. The two major competing architectures for quantum computers are the superconducting qubit model used by IBM, Google, and Rigetti, and the trapped-ion model of IonQ and a number of academic labs. To varying extents (and the variance matters), each of these models suffers from the following issues:

1. Coherence times: Qubits can only maintain their state for a certain amount of time before they *decay*, effectively resetting themselves.
2. Gate errors: Quantum gates introduce errors and these errors vary with the gates being applied.
3. Connectivity: In general, you can't apply an operation to two or more qubits unless those qubits are physically adjacent to one another. We can use quantum gates to swap the values of qubits, and thereby bring two or more qubits together, but these operations take time and introduce additional errors.

These limitations aren't uniform within a given machine, let alone across machines. On IBM's largest publicly available quantum computer, Melbourne, phase coherence times range from 22.1 to 106.5 microseconds depending on the qubit in question, and gate errors similarly vary by qubit [20].

Given the substantial challenges facing quantum computing, is formal verification even useful? We argue that it can be.

To tackle the limitations of near-term quantum computers, we will need to tailor our verification efforts to the lowest level of the quantum software stack. We will need to incorporate information about the connectivity and error rates of a given machine in order to verify that a program can be run on that machine, and to bound the error of such an execution. Such verification will be messy: It will have to take a lot of variables into account, including the ideal semantics of a given program, qubit by qubit decoherence and error rates, and specifications that may differ substantially by platform. However, we argue that this verification is necessary, and that we can provide the tools necessary to perform it.

To make this case, we begin by introducing quantum computing and the semantics of error-free quantum programs (Section 2). In Section 3, we survey the literature on formally verified quantum computing and in Section 3.2 we address certified optimizing compilers, both in an idealized, error-free setting. In Section 4 we consider the additional challenges faced by quantum programming in the near term, and how we can address them. We devote most of this discussion to the issues of errors (Section 4.1) and architectural limitations (Section 4.2). Finally, in Section 5 we reflect on how this discussion can inform the nascent field of quantum programming languages.

2 Logical Qubits

The approach to quantum computation taken by most textbooks, as well as quantum complexity theorists and (most) quantum algorithms designers assumes the existence of *logical qubits*, quantum bits which are not error-prone and behave according to a strict mathematical model. To be precise, a qubit corresponds to a two element vector of the form $\langle \alpha, \beta \rangle$ for complex numbers α and β, subject to the constraint that $|\alpha|^2 + |\beta|^2 = 1$.

Logical qubits obey strict rules but they are not deterministic. If we *measure* the qubit above, we will obtain one of the two *basis qubits* $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$ with probability $|\alpha|^2$ and $|\beta|^2$, respectively.

We can combine two qubits by taking their tensor product, where

$$\langle \alpha, \beta \rangle \otimes \langle \gamma, \delta \rangle = \langle \alpha\gamma, \alpha\delta, \beta\gamma, \beta\delta \rangle.$$

Measuring the quantum system above will yield one of four basis vectors, with probabilities corresponding to the given entries. With probability $|\alpha\gamma|^2$ we will obtain $\langle 1, 0, 0, 0 \rangle$; with probability $|\alpha\delta|^2$ we will obtain $\langle 0, 1, 0, 0 \rangle$; and so forth.

Besides measuring (systems of) qubits, we can modify them by applying *unitary gates* which correspond to multiplication on the left by a restricted set of matrices called *unitaries*, which preserve the property that the sum-of-squares adds up to one. It's worth noting that if we apply certain gates (such as the controlled-not gate) to two or more qubits, it may no longer be possible to represent the outcome as the tensor product of multiple vectors. This state of affairs is called *entanglement* and is analogous to probabilistic dependence.

Let's give a simple example of entanglement in action. Imagine we start we with the simple two qubit state $\langle 1, 0 \rangle \otimes \langle 1, 0 \rangle$. We then apply the Hadamard unitary $\frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$ to the first qubit, obtaining $\langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \rangle \otimes \langle 1, 0 \rangle$, which we can also write as $\langle \frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}, 0 \rangle$. The controlled-not unitary exchanges the third and fourth elements of the vector, yielding $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}} \rangle$. This entangled vector cannot be decomposed into the tensor product of two smaller vectors, and is known as a *Bell pair*.

(a) A circuit to produce a Bell pair.

(b) The density matrix for our Bell pair.

Figure 1 An example Bell pair.

What happens if we measure our Bell pair? As we've seen, we will obtain either $\langle 1,0,0,0\rangle = \langle 1,0\rangle \otimes \langle 1,0\rangle$ or $\langle 0,0,0,1\rangle = \langle 0,1\rangle \otimes \langle 0,1\rangle$, each with probability $\left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$. After measurement, our two qubits are no longer entangled but their outcomes are correlated: Either both are $\langle 1,0\rangle$ or both are $\langle 0,1\rangle$. This correlation is an effect of entanglement, and one of the features that gives quantum computing its power.

Generally speaking, we will represent quantum programs as *circuits*. For instance, the circuit for constructing a Bell pair is shown in Figure 1, where 0 represents the vector $\langle 1,0\rangle$, H is the Hadamard matrix and the structure bridging the two wires is the controlled-not. The circuit model is standard for quantum computing: Quantum programming languages like Quipper [14], Scaffold [21] and $\mathcal{Q}$WIRE are all circuit description languages; Microsoft's recent Q# tries to move away from this model by adding some abstractions, but Q# programs can easily be read as describing circuits as well.

Conveniently, quantum circuits have a straightforward denotational semantics: They correspond precisely to functions over complex vectors. We can also represent them as functions over *density matrices* which in turn correspond to distributions over complex vectors. A density matrix for our Bell pair, obtained by multiplying $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}\rangle$ by its transpose, is given in Figure 1. The $\frac{1}{2}$s in the first and fourth positions along the diagonal represent the probability of measuring both qubits as $\langle 1,0\rangle$ and $\langle 0,1\rangle$, respectively. Embedding probabilities inside density matrices saves us from having to include probabilistic transitions in our denotational semantics. Once we have a denotational semantics for quantum circuits, we can begin to prove things about them.

3 Verification Under Ideal Conditions

One of the simplest things to verify about a quantum program is that it doesn't attempt to duplicate qubits, which would violate the *no cloning* theorem of quantum mechanics. Non-duplication can be enforced by a linear type system, like that employed by the quantum lambda calculus [35], Proto-Quipper [33] or our $\mathcal{Q}$WIRE [27] language. A linear type system treats a function type $A \multimap B$ as something that *consumes* an A (precisely once) and produces a B (that may itself be used precisely once). Hence, it ensures that once we've done an operation on a qubit, the original qubit can no longer be used.

It's rather more complicated to ensure that a quantum program uses *ancillae* safely. Ancillae are spare qubits that are used in the computation of some result and then returned to their original state and discarded. They can be thought of as scratch space for intermediate quantum computations, but they have to be regularly garbage collected. Since qubits can be entangled with one another, operating on improperly discarded ancillae can corrupt the rest of our computation. Ancillae are a common feature of quantum algorithms, and hence appear in quantum programming languages like Quipper [14] and Q# [36]. Unfortunately,

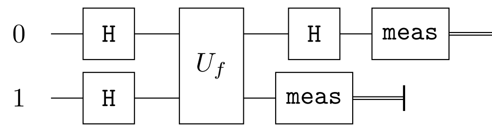


Figure 2 A circuit implementing Deutsch's algorithm.

it's quite difficult to guarantee that ancilla qubits are properly garbage collected. Inspired by the ReVerC [3] compiler for reversible programs, $\mathcal{Q}$Wire allows the programmer to prove that ancillae are discarded correctly, while providing syntactic conditions for cases where this is trivially true [30]. In the general case, though, proving that we've appropriately disposed of ancillae requires us to reason about the behavior of complete quantum programs.

Doing whole-program analysis substantially increases the scope for formal verification, which inspired us to use $\mathcal{Q}$Wire as a general-purpose verification tool [31]. One thing we may want to do is verify the correctness of a complete program, like Deutsch's algorithm [8, 11]. Deutsch's algorithm, shown in Figure 2, takes in an unknown function f, represented as a quantum gate U_f, and returns the 0 qubit if and only if the function is constant. Using $\mathcal{Q}$Wire, we can express the correctness of this algorithm as follows:

```
Lemma deutsch_constant : ∀ f, constant f →
  ⟦deutsch (fun_to_gate f)⟧ I1 = |0⟩⟨0|.

Lemma deutsch_balanced : ∀ f, balanced f →
  ⟦deutsch (fun_to_gate f)⟧ I1 = |1⟩⟨1|.
```

Here $|0\rangle\langle 0|$ represents a 0 qubit in density matrix form, and similarly for $|1\rangle\langle 1|$. I_1 is a 1×1 identity matrix, representing that the circuit has no input qubits (akin to `unit` or `void`). Deutsch's algorithm is one of the easier algorithms to prove correct since we can simply compute the output matrix. In practice we can verify a broad range of algorithms where this isn't possible, from families of quantum coin tossing circuits to quantum programs over arbitrary input unitaries [29].

$\mathcal{Q}$Wire isn't the only tool that attempts to guarantee the correctness quantum programs. Amy [1] uses Feynman path integrals to check the correctness of a variety of concrete quantum circuits, including a quantum Fourier transform [10] using up to 31 qubits. A number of authors have also introduced Hoare-style logics for quantum programs and used them to verify Grover's algorithm [15, 40] and a quantum one-time pad [5, 38]. More specialized tools allow us to verify the security of quantum protocols [39] and rewrite quantum programs expressed in the ZX calculus [9, 22].

3.1 The Verification-Programming Loop

So far, we've treated verification as a task that is subsequent to programming, which guarantees that the program behaves as expected. However, in our experience, verification also plays a major role in quantum programming itself. Even reproducing well-known quantum algorithms proves to be difficult, given the low level at which they are written. (For examples, see Huang and Martonosi's [18] recent exploration of bugs in the implementations of common quantum algorithms.)

Following Dijkstra's vision for formal verification in *A Discipline of Programming* [12], we expect quantum verification to assist in writing quantum programs. In practice, we often find ourselves writing quantum programs, attempting to prove their correctness, failing, and revising the original program. Often the failures can be quite informative: If the Coq proof

assistant asks us to prove that $\frac{1}{\sqrt{2}} = 1$, it's a sure sign that we've either left out a Hadamard gate or put in one too many (since $\frac{1}{\sqrt{2}}$ appears throughout the Hadamard matrix). We can then go back to the original program, insert the missing Hadamard gate, and return to our verification attempt, repeating as necessary.

3.2 Compilation and Optimization

Not all errors are caused by mistakes in the high-level program. Following the success of the CompCert C compiler [23], another promising target for formal verification is the compiler that translates a high-level quantum program to a circuit capable of being run on a given architecture. This compiler should preserve the semantics of the source program and should also perform optimizations to reduce resource usage and running time. Given the difficulty of testing quantum programs and the expense involved in running them, it is especially important to verify these compilers.

A number of optimizing compilers have been designed for quantum programs to reduce resource usage [2, 16, 26]. These optimizations are often mathematically sophisticated, and thus vulnerable to programmer error. For example, in discussions with Nam et al. we learned that, while developing their optimizer, they found several bugs in their own implementation and also in the implementations that they compared against [26].

The Quantomatic tool [22] can apply verified transformations to quantum computation expressed in terms of the ZX-calculus [9, 4], a diagrammatic approach to quantum computation. Unfortunately, not every ZX diagram corresponds to a valid quantum circuit, and an optimization in ZX may not optimize a corresponding circuit. Recent work [13] optimizes a restricted subset of ZX diagrams that do represent circuits, but these are limited to a subset of quantum circuits known as Clifford circuits.

We are currently developing a new intermediate representation for quantum circuits, called sQIRe [17], to help us go further. sQIRe allows us to perform verified optimizations on circuits, with the goal of reducing the total circuit size. So far we have verified simple optimizations, like skip elimination and canceling repeated X (negation) gates.

4 Verification in the Real Quantum World

So far we have an interesting story and perhaps even a compelling one: Quantum programs are hard to write and debug and hence provide an excellent target for the techniques of formal verification, from program logics to proof assistants. However, these logics will be of limited use for the near future. The quantum computers that exist today and are likely to exist over the next ten to twenty years will be incapable of running arbitrary quantum circuits and will be very failure prone. Hence, for formal verification to be useful in the near term, it will need to be tied to the machines we expect, not those we hope for.

In a recent keynote address [28], John Preskill coined the term Noisy Intermediate-Scale Quantum Computing (NISQ) to refer to quantum computing over the coming five or ten years. Preskill, like many in the field, suspects that quantum computing will soon have its first practical applications on computers with under 1000 physical qubits. On the other hand, it will take hundreds or thousands of physical qubits to construct one error-corrected logical qubit, and many thousands of logical qubits to beat classical computers at factoring numbers or performing a range of other tasks. In the near term, quantum programs will have to be aimed at problems for which they are uniquely well-suited (like modeling quantum systems in physics [7] and chemistry [32]) and tailored to the limitations of the available computers.

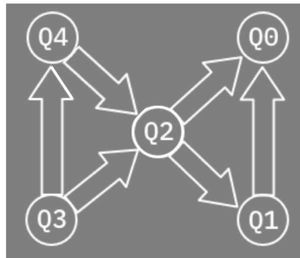


Figure 3 Two-qubit gate connections on IBM's Tenerife machine. Taken from `https://github.com/Qiskit/ibmq-device-information/blob/master/backends/tenerife/V1/version_log.md`.

What do these limitations look like? One hard limitation is the number of qubits on the machine. Another limitation is error rates. There are multiple sources of errors in quantum circuits, stemming from *decoherence* (qubits tend to revert to their basis states with time) and errors in gate application.[1] Today's machines also have a number of architectural limitations, related to the connectivity of qubits: Instead of being complete undirected graphs, they tend to resemble sparse directed graphs. For example, consider the diagram of IBM's 5-qubit Tenerife machine shown in Figure 3. In this architecture, if you want to apply a controlled-not gate from program qubit q_1 to program qubit q_2, then you need to map those program qubits to adjacent physical qubits in the machine (e.g. $Q4$ and $Q2$). This mapping may need to be updated over the course of the program by adjusting the physical locations of program qubits. This adjustment is both computationally expensive and error prone.

If formal verification is to guarantee the correctness of quantum programs in practice, it will need to account for these crucial limitations of NISQ devices.

4.1 Verification in the Presence of Errors

Let us sketch out some possible approaches to dealing with the errors that are sure to arise when we run our quantum computers.

One straightforward approach to verification in the presence of errors is to simply aggregate errors along a quantum circuit's wire. Instead of the Hadamard gate having the type `Qubit` ⊸ `Qubit`, it can have the type ∀n, (`Qubit`,n) ⊸ (`Qubit`,n+1), meaning that the gate adds a single error to its wires. We could equally well include error probabilities along the wires, though those would assume we knew the error rate for each gate and they were consistent across qubits. Multiple-qubit gates are a bit trickier, as they take in and produce multiple wires: We can either output the sum of all error terms along each output wire plus the additional error introduced by the gate, take the max of the two wires, or (particularly in the case of probabilities) use a more complex function over multiple inputs. A circuit then, would likewise have an error term corresponding to the aggregated output of its wires. For a simple example, depending on whether U_f takes the max or sum of its inputs' errors, Deutsch's algorithm (Figure 2) would produce 2 or 3 errors, plus the errors introduced by U_f, assuming measurement doesn't introduce errors itself.

[1] The presence of these gate errors actually allows us to comfortably ignore a more fundamental issue in quantum computing. The so-called "universal gate sets" implemented by general purpose quantum computers are not actually universal in the sense that NAND gates are universal for classical computation: They only allow us to *approximate* arbitrary quantum operations. Unfortunately, in the near term, all gates will only loosely approximate their specified behaviors.

An advantage of this approach is that it's very easy to implement and sufficiently general that we can interpret the output in a number of different ways, depending on the setting. We can also automate it, allowing a built-in type inference algorithm to calculate the errors that occur in a given circuit.[2] It is limited, though, in that errors only increase as the circuit size grows. This makes it difficult to analyze programs that include mechanisms for error mitigation, which will be important in near-term applications [24]. (In principle, we could have error-correcting gates that shrink the error, but error correction tends not to be so simple.)

We can also take ideas from our recent robustness logic [19], which draws on Carbin et al.'s Rely language for approximate classical computing [6] and provides bounds for the errors in a quantum program. While powerful, this logic suffers from the same limitations as our error wire semantics – errors only increase throughout a program. It is also high level: It uses the same quantum while language as QHL [40], which doesn't directly describe circuits that can run on near-term machines.

Ideally, the semantics for a quantum program would contain error terms, corresponding to possible failures. This would bring the advantage of allowing us to reason about and reduce error terms, through majority voting or the like. Unfortunately, it's still hard to know what the denotational model should be, without reference to the specific hardware. Hence, while doing high-level reasoning we want to leave the error term abstract (as in our robustness logic), and instantiate it for specific hardware models.

4.2 Verified Compilation to Restricted Architectures

As we've seen, in order to make effective use of near-term quantum devices, we cannot entirely abstract away low-level architectural details. However, this is not to say that the programmer needs to consider every low-level detail when writing programs. In some cases, as in classical computing, the complexity of running on a particular architecture can be handled by the compiler.

For example, we have already discussed the challenge of limited connectivity on near-term machines. There has been significant work on developing automated transformations that map arbitrary quantum circuits into circuits that satisfy a particular machine's connectivity constraints [41]. Some of this work has even looked at how to perform this mapping in a way that reduces the error of the resulting circuit [25, 37].

Another way that near-term compilers for quantum programs can help is by performing optimizations that reduce resource usage, as discussed in Section 3.2. Reducing resource usage is critical because near-term devices have access to a limited number of qubits and can support few operations before decoherence undoes the effect of any useful computation.

We would like to go further. A verified compiler for quantum circuits should take the following three things into account:

1. The connectivity of the target machine;
2. the error rates for each qubit; and
3. the fidelity of individual gates applications.

It should use this information to compile an arbitrary quantum circuit to an equivalent circuit that can run on the target machine in a way that minimizes errors. This is difficult: Each of our desirata imposes substantial constraint on the compiler and a burden on the verifier. However, each is necessary for our compiler to be both useful and reliable.

[2] It's worth noting that checking linearity, though harder, is an orthogonal problem, so we can infer the error terms and check linearity separately. This is somewhat surprising, since without linearity we would have to be concerned about adding errors to terms without using them up.

Our new sQIRe language [17] can be used to verify transformations that guarantee structural properties of circuit. For instance, the `map_to_lnn` function compiles to a linear nearest neighbor architectures, guaranteeing that every controlled-not gate is applied to adjacent qubits. However, at present, this transformation is applied manually, rather than being part of a compilation toolchain. Also, the linear nearest neighbor architecture is a toy example: The connectivity of a quantum computer can be represented by any graph, directed or undirected. Verified compilation to such machines remains a significant challenge.

5 Looking Forward

As we've seen, verification has an important role to play in quantum programming, both in the short and long term. Quantum programs are buggy and difficult to test, so if we want to have any confidence in our programs' correctness, we had better verify them. There is already substantial progress on this front, through tools like $\mathcal{Q}$WIRE [27, 29] and QHL [40].

However, in the short term these tools aren't sufficient. We need an approach to verification that deals with errors, like our recent quantum robustness logic [19]. Moreover, we need to verify error-prone programs with respect to the hardware we intend to run them on. The coherence times and error rates of quantum computers vary widely and a good error model will need to be machine specific.

Machines aren't only constrained by their error rates, but also by the number and connectivity of their qubits. This results in additional constraints that a compiler must satisfy, and satisfaction of these constraints must also be verified. The sQIRe intermediate representation for quantum circuits is a step towards verified compilation, but much work remains to be done.

In an ideal world, we would not think about circuit, let alone machine architectures, when writing quantum programs. Indeed, another ambitious project for quantum computing is developing useful abstractions for programmers. Some steps in this direction include quantum control flow [34] and amplitude amplification as a subroutine [36]. These efforts look towards a future where we have scalable quantum computers with many error-free qubits.

By contrast, we are focused on the quantum devices available today and likely to be available over the next decade. Our goal is to develop tools that will assist in developing efficient algorithms with correctness guarantees and precisely bounded errors. We aim to execute those algorithms on fundamentally limited machines, with low qubit counts and high error rates. To that end, we have to provide information about everything down to the individual qubit decoherence to the programmer, so they can handle that decoherence. Providing these tools will allow us to do more with near-term quantum devices than we could possibly do today.

References

1 Matthew Amy. Towards Large-scale Functional Verification of Universal Quantum Circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018*, June 2018.

2 Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33, 2013.

3 Matthew Amy, Martin Roetteler, and Krysta M. Svore. Verified compilation of space-efficient reversible circuits. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2017)*. Springer, July 2017. URL: `https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/`.

4 Miriam Backens. The ZX-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics*, 16(9):093021, 2014.

5 P Oscar Boykin and Vwani Roychowdhury. Optimal encryption of quantum bits. *Physical review A*, 67(4):042317, 2003.

6 Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 33–52, 2013. doi:10.1145/2509136.2509546.

7 Andrew M. Childs, Dmitri Maslov, Yunseong Nam, Neil J. Ross, and Yuan Su. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences of the United States of America*, 115 38:9456–9461, 2018.

8 Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 454, pages 339–354. The Royal Society, 1998.

9 Bob Coecke and Ross Duncan. Interacting quantum observables. In *International Colloquium on Automata, Languages, and Programming*, pages 298–310. Springer, 2008.

10 Don Coppersmith. An Approximate Fourier Transform Useful in Quantum Factoring. *arXiv preprint*, 1994. arXiv:quant-ph/0201067.

11 David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 400, pages 97–117. The Royal Society, 1985.

12 Edsger Wybe Dijkstra. *A Discipline of Programming*, volume 1. Prentice-Hall, 1976.

13 Andrew Fagan and Ross Duncan. Optimising Clifford Circuits with Quantomatic. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*, 2018.

14 Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013, pages 333–342, 2013.

15 Lov K. Grover. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM. doi:10.1145/237814.237866.

16 Luke Heyfron and Earl T. Campbell. An Efficient Quantum Compiler that reduces T count. *Quantum Science and Technology*, 4, 2017.

17 Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. Verified Optimization in a Quantum Intermediate Representation. *arXiv e-prints*, page arXiv:1904.06319, April 2019. arXiv:1904.06319.

18 Yipeng Huang and Margaret Martonosi. QDB: from quantum algorithms towards correct quantum programs. *arXiv preprint*, 2018. arXiv:1811.05447.

19 Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. Quantitative Robustness Analysis of Quantum Programs. *Proc. ACM Program. Lang.*, 3(POPL):31:1–31:29, January 2019. doi:10.1145/3290344.

20 IBM. IBM quantum experience, 2017. URL: https://quantumexperience.ng.bluemix.net/qx/devices.

21 Ali Javadi-Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold: Quantum programming language. Technical report, PRINCETON UNIV NJ DEPT OF COMPUTER SCIENCE, 2012.

22 Aleks Kissinger. *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing.* PhD thesis, University of Oxford, 2011.

23 Xavier Leroy et al. The CompCert verified compiler. *Development available at http://compcert.inria. fr*, 2009, 2004.

24 Sam McArdle, Xiao Yuan, and Simon Benjamin. Error mitigated digital quantum simulation. *arXiv preprint*, 2018. `arXiv:1807.02467`.

25 Prakash Murali, Jonathan M Baker, Ali Javadi Abhari, Frederic T Chong, and Margaret Martonosi. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. *arXiv preprint*, 2019. `arXiv:1901.11054`.

26 Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, 2018.

27 Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 846–858, New York, NY, USA, 2017. ACM. `doi:10.1145/3009837.3009894`.

28 John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. `doi:10.22331/q-2018-08-06-79`.

29 Robert Rand. *Formally Verified Quantum Programming*. PhD thesis, University of Pennsylvania, 2018.

30 Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. ReQWIRE: Reasoning about Reversible Quantum Circuits. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*, 2018.

31 Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in Coq. In *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, pages 119–132, 2017. `doi:10.4204/EPTCS.266.8`.

32 Markus Reiher, Nathan Wiebe, Krysta Marie Svore, Dave Wecker, and Matthias Troyer. Elucidating reaction mechanisms on quantum computers. *Proceedings of the National Academy of Sciences of the United States of America*, 114 29:7555–7560, 2017.

33 Neil J. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie University, 2015.

34 Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. From Symmetric Pattern-Matching to Quantum Control. In *International Conference on Foundations of Software Science and Computation Structures*, pages 348–364. Springer, 2018.

35 Peter Selinger and Benoît Valiron. Quantum lambda calculus. In Simon Gay and Ian Mackie, editors, *Semantic Techniques in Quantum Computation*, pages 135–172. Cambridge University Press, 2009.

36 Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, page 7. ACM, 2018.

37 Swamit S. Tannu and Moinuddin K. Qureshi. A Case for Variability-Aware Policies for NISQ-Era Quantum Computers. *arXiv e-prints*, page arXiv:1805.10224, May 2018. `arXiv:1805.10224`.

38 Dominique Unruh. Quantum Hoare Logic with Ghost Variables. *arXiv preprint*, 2019. `arXiv:1902.00325`.

39 Dominique Unruh. Quantum Relational Hoare Logic. *Proc. ACM Program. Lang.*, 3(POPL):33:1–33:31, January 2019. `doi:10.1145/3290346`.

40 Mingsheng Ying. Floyd-Hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19, 2011.

41 Alwin Zulehner, Alexandru Paler, and Robert Wille. Efficient mapping of quantum circuits to the IBM QX architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 1135–1138, 2018. `doi:10.23919/DATE.2018.8342181`.