Meta-metaprogramming

by

James Koppel
B.S. in Computer Science, Carnegie Mellon University (2012)
B.S. in Mathematics, Carnegie Mellon University (2012)
S.M. in Electrical Engineering and Computer Science,
Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author . . . .
Department of Electrical Engineering and Computer Science
August 27, 2021

Certified by . . . .
Armando Solar-Lezama
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Meta-metaprogramming

by

James Koppel

B.S. in Computer Science, Carnegie Mellon University (2012)

B.S. in Mathematics, Carnegie Mellon University (2012)

S.M. in Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science on August 27, 2021, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Programming languages researchers have developed many advanced tools that promise to greatly ease software engineering. Yet even conceptually simple tools are expensive to implement fully due to the complexity of the target language, and standard techniques tie an implementation to a particular target language. In order to make the development of advanced programming tools economical, these problems demand new techniques for decomposing the development of tools and automating portions of their construction, which I collectively dub "meta-metaprogramming."

In this thesis, I present three new meta-metaprogramming techniques reducing the work needed to build programming tools, each applicable to the specific problem of sharing implementation code between similar tools for different languages. These techniques respectively allow a single implementation of a transformation to losslessly rewrite code in many languages, automatically generate a family of programming tools from a language's semantics, and develop a new representation for sets of programs which is applicable to a variety of languages and synthesis tasks.

Thesis Supervisor: Armando Solar-Lezama
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I remember my first conversation with Stanford's Alex Aiken in July of 2012, where I told him I wanted to build commercial developer tools for software maintenance, and he warned me that no prior attempt had overcome what I now call the heterogeneity problem. Not even a year and a half later, my program-repair startup Tarski Technologies had grown and died, and his words began to crystallize in my mind into a new research program. With some detours, I've stuck to that plan unusually well in the 8 years since. Thanks to him, and to all the others who helped me get my start: to Claire le Goues and Ethan Fast and Jacob Steinhardt, to the many senior researchers willing to entertain an excitable 21-year old showing up alone at conferences, and, of course, to my incredibly kind and smart undergraduate advisor, Jonathan Aldrich.

When I think of a word to describe my advisor Armando, the first word that comes to mind is “supportive," a quality he takes to an extreme I never thought possible. Grad school for me had a rough start. I immediately chose a problem that was too difficult in a field too immature, and then personal trauma left me unable to focus for months. Armando was a rock in my life during this period. His support for me was so unwavering that I cannot conceive of him judging a student, even subtly. I must also thank Martin Rinard, who supported me even in the times I felt unproductive, and continues to be among the first people I go to every time I'm shocked by a paper rejection. I am also thankful to Caroline Uhler and Stephen Chong, who took me in for exciting side-projects during this period. With some shame, after I regained the ability to focus on a single project, I disappeared on them as many undergrads have done to me. Perhaps one day I'll go back and complete my projects with each of them.

Beyond my advisor Armando, I am honored to have a committee composed of people each of whom I've worked independently with. Spending a summer with Ira Baxter at Semantic Designs was one of the best decisions I made in grad school. That summer, I became a sponge for information about industrial whole-program transformation, and gained first exposure to how much of practical use the field of term rewriting had accomplished beyond the basics, techniques which form a core part of this thesis. And because I had the pleasure of actually living with Ira, at least one night a week we'd be up for hours talking about language engineering and tools research. When I speak about software maintenance, I'm often asked how I know so many stories, and the answer is usually "I got them from Ira Baxter."

As my blog and business about software design were taking off, I was truly blessed to be sitting just a few doors down from Daniel Jackson, who was able to immediately understand my ideas and draw connections with his vast knowledge of prior writing. After he pitched me the problem of giving a general definition of dependence in software engineering, we soon realized it was a problem involving counterfactuals, which I was uniquely suited to solve because of my first year grappling with causality in programs — that year had not been wasted after all. Sometimes life just comes around like that.

I am thankful to my collaborators. My time collaborating with officemates Xin Zhang and Zenna Tavares were a blast, when we'd often switch between writing and playing `surviv.io`, joined sometimes in the latter by Max Nye and Evan Pu. And among my happiest days in grad school were those spent with Michael A. Specter reversing the "Voatz" voting app, followed by the thrill as Andrew Sellars and Daniel Weitzner guided us in disclosing to the DHS and then the New York Times.

I would like to thank my master's student Varot "Pond" Premtoon, who created YOGO and in doing decisively demonstrated the advantages of CUBIX for building multi-language tools. And I would like to thank the many other undergraduates I've worked with: Zoe Anderson, Jasper Haag, Angel Huang, Jackson Kearl, Diana Molodan, Shirlyn Prabahar, Elijah Rivera, Kliment Serafimov, and Arman Talkar.

I am thankful to the many other faculty and researchers who have offered ideas and support in ways big and small. I particularly thank Adam Chlipala, who is always a source of biting insight, and who (re)connected me to Gabriel Scherer, who became a mentor and then a coauthor when I dipped my feet into computational effects.

I am pleased to give another thank you to the "most thanked man in computer science" Olivier Danvy, for inventing an entire field used as prior work for MANDATE, and for making my visit to Singapore truly unforgettable. And to the other faculty who have hosted me as I toured universities: Işil Dillig, Mooly Sagiv, Eran Yahav, Bor-Yuh Evan Chang, Sasa Misailovic, Zhiqiang Zuo, Li-Shiuan Peh, Cyrus Omar, Jeremy Siek, Ryan Newton, Benjamin Delaware, and Ravi Chugh.

I would like to give thanks to those who were there for me as I began to broaden my academic horizons. To Gopal Sharma, for believing in me when I was a teenager, and then a decade later giving me my first opportunity to co-organize a workshop. To Cody Roux for helping with the workshop and answering my StackExchange questions. To Xiangen Hu and Andrew Olney, for their guidance as I began to explore a distantly-related research area.

I am thankful to the MIT Educational Studies Program student group. ESP accepted me with open arms when I arrived at MIT mid-year without realizing the social ramifications, and has continually offered me a place at MIT that feels like home as part of the ESPhamily, and a place to nurture my skills as a teacher. And I am thankful to the members of MIT Gymnastics, for their constant non-judgmental support, and for making it so that most of the flips and flops in my life were of my own volition.

I would like to thank the other members of my lab for helping keep my life full of programming languages: Rishabh Singh, Jean Yang, Xiaokang Qiu, Kuat Yessenov, Zhilei Xu, Rohit Singh, Nadia Polikarpova, Shachar Itzhaky, Sicun Gao, Ivan Kuraj, Kevin Ellis, Maxwell Nye, Jack Feser, Jeevana Priya Inala, Osbert Bastani, Nathan Hunt, Kavi Gupta, Matthew Bowers, Leonardo Cano, and Ria Das. And I would like to thank other members of my cohort for great times and interesting conversations: Tej Chajed, Jiasi Shen, Sara Achour, Thomas Bourgeat, Aviv Adler, Vikas Garg, and Jason Gross. And, outside of MIT but on a similar note, Cosmin Radoi.

Finally, I must thank those connections and companions who have been sources of support throughout these sometimes exciting and often difficult years: Jon Paulson, Amy Quispe, Jack Gurev, Emerson Studt, Ryan Alweiss, Nancy Hua, Melody Guan, Josh Alman, Jessica Su, Ariel Shih, Regan Dvoskin, Xinyu Wu, and Jing Yu; my parents Ralph and Suellen; and my sister Julie and brother-in-law Josh.

Contents

List of Figures

List of Tables

Glossary of Notation

Standard Notation

$(\rightarrow)$ A function ("arrow") type. Also used in Chapter 4 for abstract machine transitions. (Section 4.3.4)

$(\rightharpoonup)$ A partial function type. (Finite) partial functions can be thought of as associative map data structures.

$[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$ A partial function, defined $[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k](x_i) = v_i$. Also called a *map* or *environment*.

$\sigma[x \rightarrow e]$ If σ is a partial function, then constructs a new ("updated") partial function defined $\sigma[x \rightarrow e](x) = e$ and $\sigma[y \rightarrow e](x) = \sigma(y)$ for $y \neq x$. When σ represents an environment, this constructs a new environment updated for a variable assignment.

$\mathbb{P}(S)$ The powerset of S.

$\varnothing$ The empty set. Also used for the empty environment (empty map).

σ A metasyntactic variable used for environments/states (maps of variables to values).

$P \vdash Q$ Entailment. Indicates that Q may be inferred from P by a series of logical rules.

$x : \tau$ Denotes that x has type τ.

$[v/x]e$ The substitution of v for x in e, i.e.: the object constructed by taking e and replacing all occurrences of variable x with v.

γ Used to denote the concretization function of an abstraction relation. Used for a specific such concretization function in Section 4.5.1.

$\widehat{\cdot}$ Used to denote some abstraction of the underlying object; several specific instantiations in Chapter 4.

$[\![\cdot]\!]$ The denotation of an object (of an ECTA, ECTA node or edge, etc).

$::=, |$ Used to give BNF grammars of an inductive type.

$\square$ A hole; an empty term or context to be later filled with a value.

$e[v]$ Plugs in v for the unique hole in e.

$\overline{\cdot}$ Denotes that the thing under the line is a list, be it a list of variables (e.g.: $\overline{F_i}$) or a list of facts or assertions (e.g.: $\overline{a_i < b_i}$).

$i_i.i_2 \ldots i_k$ **(e.g.:** 1.3.4.2**)** A path. The example denotes starting at a given node, then descending down through child 1, then child 3, then child 4, then child 2.

$f\big|_s$**,** $t\big|_p$ The notation $f\big|_s$ denotes the restriction of function (or partial function) f to the set s, where $s \subseteq \mathrm{dom}(f)$. $t\big|_p$ denotes the value of term t at path p, and is also used in Chapter 3 for the sub-ECTA at a path. Which meaning is intended should be clear from context.

Notation used in Chapter 3

$\mathcal{T}(\Sigma, \mathcal{X})$ The set of terms constructible from the signature Σ and the variable set $\mathcal{X}$. (Section 3.2.1)

$\mathcal{T}(\Sigma)$ The set of closed terms constructible from signature Σ. Equivalent to $\mathcal{T}(\Sigma, \varnothing)$. (Section 3.2.1)

$\{p_1 = \cdots = p_k\}$ **(e.g.:** $\{1.1 = 2.1\}$ A path constraint set (PEC) requiring that the terms at the given paths must be equal. (Section 3.2.2)

$e\big|_p^{\cap m}$ Intersection at a path: For ECTA e, constructs the ECTA given by replacing each node at path p with the intersection of said node and m. (Section 3.3.2)

Notation used in Chapter 4

mt A metasyntactic variable for a match type. The available match types are $\mathsf{Val}, \mathsf{NonVal}$ and All.

$(\leadsto)$ A transition in the SOS (structural/"straightened" operational semantics) of a language. (Section 4.3.2)

$(\uparrow), (\downarrow), (\updownarrow)$ $\uparrow$ and $\downarrow$ are symbols denoting the "returning" and "evaluating" phase of a phased abstract machine. $\updownarrow$ (also seen with subscripts, e.g.: $\updownarrow_1$) is a metasyntactic variable denoting either $\uparrow$ or $\downarrow$.

$\langle c \mid K \rangle\updownarrow$ A phased abstract machine state, where c is the configuration, K the context, and $\updownarrow$ the phase.

$(\hookrightarrow)$ A phased abstract machine transition. (Section 4.3.3)

$\langle c \mid K \rangle$ An abstract machine stat, where c is the configuration and K the context. (Section 4.3.4)

$(\longrightarrow)$ An unfused abstract machine transition. (Section 4.3.4)

$(\rightarrow)$ A abstract machine transition. (Section 4.3.4) Also used for the standard function ("arrow") type.

$\leadsto_l, \hookrightarrow_l, \longrightarrow_l, \rightarrow_l$ The l subscript is used when the language of the transition is unclear.

$\star_{\mathsf{Val}}, \star_{\mathsf{NonVal}}, \star_{\mathsf{All}}$ Abstract nodes matching all values, all non-values, and all nodes, respectively. (Section 4.5.1)

β Metasyntactic variable reserved for a *base abstraction.* (Section 4.5.2)

α Metasyntactic variable reserved for a *machine abstraction.* (Section 4.5.3)

$(\underset{\beta}{\widehat{\rightarrow}})$ The abstract reduction relation, parameterized by base abstraction β. (Section 4.5.2)

($\rightsquigarrow$) The narrowing relation. (Section 4.6)

($\underset{\alpha}{\widehat{\rightsquigarrow}}$) The abstract narrowing relation / narrowing on abstract terms, with machine abstraction α. (Section 4.6, Appendix B.3)

Chapter 1

Introduction

The world runs on software, and any boost in the efficiency of software development and maintenance will have an outsized downstream impact on the world. Tools researchers have answered with a plethora of programming tools, each promising great efficiency gains on their targeted tasks. Recent achievements include better techniques for finding nontermination bugs [187] and bugs in multi-threaded programs [25], automated mitigation of hardware-based security holes [170], automatically generating database queries [165] and accurate floating-point libraries [109], improvements in automated bug-fixing [154], and interactive evaluation of program analysis queries [158] and choices of complex literal values [128] — and these are only from 2021 alone!

Yet in spite of the vast economic potential of making software development more efficient and the availability of this research cornucopia to claim it, there are few commercial programming tools available today based on program analysis, transformation, and synthesis technology. Consequently, programmer needs go unaddressed. According to a 2010 survey by LaToza and Myers [104, 105], of the questions programmers ask during development: 34% of the time, some commercial tool existed for that problem for some language; 25% of the time, a research tool existed; and 41% of the time, no tool existed at all. And, of course, this excludes the questions they didn't think to ask.

A typical tools paper today commonly presents a prototype for something less than a full industrial language, often a just-large-enough subset of an industrial language to obtain an impressive result. Conventional wisdom is that scaling these research prototypes into a viable tool is simply a matter of adding labor. In fact, as we shall argue, obtaining a real tool economically is not “just engineering." After solving the research challenge of how to build some advanced programming tool at all, there are more research challenges in how to build it effectively. There are systematic reasons why tools of all stripes require disproportionate effort to build for the general use-case. Here are two:

- The **linguistic complexity problem**: that the difficulty of building a tool stems more from the linguistic complexity of the supported language than from the essential complexity of the tool.
- The **heterogeneity problem**: that developers are split across a long tail of distinct programming languages — and current techniques make it highly difficult to share code between similar tools for different languages.

The complex language problem increases the cost of building a tool. The heterogeneity problem decreases the benefits. Combined, these can decrease the ROI of the tool to the point that it is not economical to build. Duplicating that effect across all potential tools, and none get built. We believe these two technological barriers are major causes of the *missing market* in advanced programming tools. And, unlike the sociological and purely economic causes, these barriers can be decisively dismantled with a technological solution.

In this thesis, we attack both these problems through techniques which permit **splitting the information of a language and tool** and **programming both at a higher level**. We introduce a new way of decomposing programming languages called **incremental parametric syntax**, and use it in our Cubix framework to implement program transformation tools simultaneously for 5 industrial languages. We develop new algorithms for manipulating programming language semantics and a new method to analyze them called **abstract rewriting**, and implement both in our Mandate tool to create the world’s first control-flow graph generator generator. Finally, we make two new contributions furthering the ability for general-purpose automated-reasoning tools to solve programming problems. We create a fusion of equality saturation and dataflow abstraction enabling a single code-search query can match many variants of an idiom across several languages, and implement it in our YOGO tool atop CUBIX, with support for 3 languages. And we develop a new representation called **equality-constrained tree automata** which can express a new class of programming problems amenable to “constrained equational reasoning" and solve them with a generic solver, implemented in our ECTA library.

All of these are techniques which add new capabilities to how people can build programming tools. They can be deemed part of the existing field of *software language engineering* (SLE), though SLE places higher emphasis on how code is first input (e.g.: parsing, structure editors) and on streamlining existing techniques for building tools (e.g.: tool DSLs, language workbenches). Instead, we consider our contributions to be part of the newly-dubbed field of **meta-metaprogramming**, which we define as: techniques aimed at making program analysis (broadly: code into information), transformation (code into code), and synthesis (information into code) tools easier to build and more general and which substantially alter the way these tools are built.

1.1 Fundamental Challenges, Scientific Solutions

Previously, we credited the twin problems of linguistic complexity and heterogeneity for the high cost of building tools. We now flesh out our arguments that these are both foundational problems which merit technological solutions beyond “just trying harder," and explain more how the meta-metaprogramming techniques in this thesis mitigate them.

Linguistic Complexity There is no controversy that modern programming languages are very complicated. The C++ standard [159] exceeds 1000 pages. Drawing on Brookes’s concept of accidental vs. essential complexity [22], we argue that **the complexity of building most advanced programming tools stems more from the complexity of the language than the essential complexity of the idea**

for the tool and that **this *linguistic complexity* is *accidental complexity***, as it is **partially an artifact of conventional techniques for tool development**.

Why would it not be inevitable that more complex languages need more complex tools? Let us introspect on how linguistic complexity is transmuted into tool complexity. Program analysis, transformation, and synthesis tools all require some partial prediction about the behavior of code in order to be correct. Thence, a tool's complexity increases along with the number of distinct language constructs that may affect its predictions. This manifests in low-level forms like a **case** statement over a language's syntax, and in higher-level forms like the need to consider aliasing for languages containing certain features. We thus suggest that, to interrupt the flow of linguistic complexity into tool complexity, one must improve the isolation and reuse of the components that produce information about the program and those that consume it. As expert can use their knowledge of a single language in multiple ways to develop multiple tools, so too can more general tools or tool-generator use encoded knowledge of a language in several ways.

Information about a language can be used in two ways, either exposed to a *derivation* or hidden behind an *abstraction*. Using language information derivationally means that facts about that language are exposed in some declarative form, so that a general tool can adapt itself to that language. MANDATE does this in a major fashion, turning a full language semantics into a tool, but so does YOGO, taking as input both a language-general and a language-specific library of rewrite rules. Outside of our work, all solver-aided tools work in this manner; consider, for instance, Pasket's encoding of Java method resolution into SAT [84], or Cassius's encoding of the CSS standard [131]. Our own ECTA library is a new option for such encoding, which we apply both to the typing rules of polymorphic function application and the staging constraints of a database DSL. CUBIX also has some small interfaces by which a tool may directly query facts about a language, such as its treatment of short-circuiting operators by a generic interface to get the strictness of an expression, so that it can properly break apart expressions like `f() && g()` without language-specific special treatment (see Figure 2-20). These encodings and interfaces are mostly partial and specific to a target tool, yet they in turn could conceptually be drawn from a comprehensive central description of the language, as Cassius is attempting with CSS.

Information about a language can also be used by building tools against some higher-level abstract operation which has implementations specific to each language. The design of CUBIX's language fragments, for instance, such as the reexpression of variable declarations into a single generic node with language-specific children, can be seen as creating a general interface to the idea of a variable declaration. A more direct example is its control-flow-based inserter (Section 2.7), a higher-level rewriting primitive which uses the control semantics of a language to decide where to put certain statements, so that one can insert a statement “before" a loop-condition, and have it appear before the loop, at the end of the loop, and before every `continue` statement. The dataflow abstractions of YOGO, in which e.g. many kinds of loops can be translated into a higher-level “loops over a sequence" construct, can also be seen as presenting an abstracted interface to specific language features.

Heterogeneity Even across similar languages, it is difficult-to-impossible to share work between similar tools for different languages. (Specifically the analysis, synthesis, and transformation components of these tools.) Here, it is easier to give clear reasons: it is because of **closedness** and, for transformation tools, **type invariance** and **bidirectionalization**.

A portion of a program P is *closed* in some collection C if no addition can be made to C without a change in P. For example, in a rich-text editing program, any subset of the program which enumerates or cases over the available colors would be closed in the collection of available colors. In contrast: open sums are *open* because new summands can be added without changes to either existing producers or consumers.

In the context of tools, closedness means that some part of the implementation makes an assumption about the totality of constructs and features in the target language. Though closed-world assumptions must often be introduced at some point so that the tool can reason about what a language cannot do, supporting multiple languages requires that as much of the implementation as possible only make assumptions about small portions of the target language, open to many possible extensions. But, to the trained observer, nonessential closedness is pervasive in common ways of implementing programming tools.

The most apparent form of closedness is in recursive traversals over a program, which embed assumptions about a large portion of the syntax of the language, even for operations focused only on a single corner of the syntax. The key idea permitting operations to be written for single nodes without reference to the whole language is to represent syntax using an *unfixed data type*[1], a well-known technique taken to a new extreme in CUBIX. But a more subtle common expression of closedness is when an analyzer assumes certain effects cannot occur in a given step, e.g.: that calling a function cannot alter the local variables of the caller — readily assumed in a Java analyzer, but false in a language with call-by-reference or unsafe memory access. We counter this in CUBIX through the aforementioned constant use of interfaces to query facts about a language, and through a style of programming where each operation states everything it assumes about the language it runs on. Our solution is still imperfect, however; it cannot elegantly express assumptions about what is not in a language, in large part due to lack of negation in Haskell's type system.

Closedness can also be countered by shifting to an entirely different programming paradigm. Both MANDATE and YOGO take something akin to a set of rewrite rules as input, with each rule being independently meaningful in the presence or absence of any other set of rules. With MANDATE one can freely alter the set of rules in an open fashion, allowing MANDATE to automate all code generation after closing the set of rules, yielding a control-flow graph generator tuned to the set of possible control-flow transitions that MANDATE discovers. YOGO's rules are similarly open, so that several languages can share a common set of generic rules, though YOGO lacks a closed phase where it can make decisions based on what features are not in a language.

[1]"Unfixing" a data type is a folkloric term for defining a data type initialy without recursion, but then later applying an explicit type-level fixpoint operator. It is similar to writing a recursive function explicitly with the Y Combinator. The datatypes à la carte data construction explained in Section 2.3.1 is one way of implementing unfixed data types.

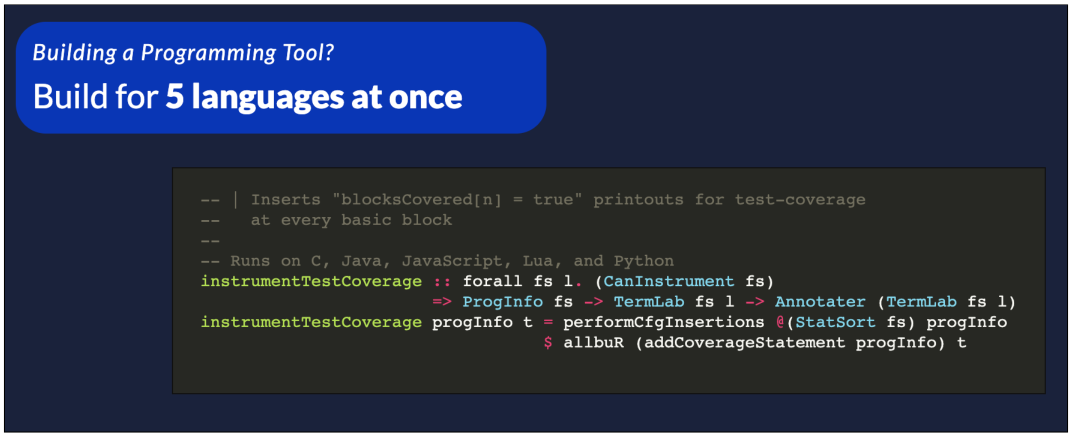
One important form of closedness is the *type invariance* of a program transformation operation. In type theory, the theory of subtyping describes when a function can be applied to many different kinds of data. Because it is based on facts about the assumptions made by different components of a program on their data, this theory can be applied even in the absence of a formal type system. Program transformations traditionally have type $L \to L$, where L is the type of terms in the target programming language. This type is invariant in L, meaning that such a function necessarily cannot be used on any restrictions or extensions of L. Or more informally: A tool intended to consume code in a language L only works for L and its subset. A tool intended to produce code in a language L only works for L and its supersets. Therefore, a tool which transforms (i.e.: both consumes and produces) code in L only works for L exactly, and therefore breaks for any variant of L. We discuss invariance more in Chapter 2. Overall, this clear account helps explain why, in the general case, source-to-source transformation is much harder than analysis or synthesis alone. Cubix solves this decisively by instead giving program transformations a parametrically polymorphic type akin to $\forall l.\mathsf{DesiredProperties}(l) \Rightarrow l \to l$. Functions of this type must be open in the language l, and may make no assumptions about it save that it satisfied the chosen $\mathsf{DesiredProperties}$.

Finally, *bidirectionalization* is the problem of maintaining a correspondence between two views of some artifact that can be transformed into each other. For source-to-source transformation tools, the two views in question are the program source code and the tool's internal representation. Any such tool, if it is to produce output that will be read by a human programmer, must produce code as similar as possible to the original while still performing the intended change. Two common failures are accidentally turning for-loops into while-loops (losing syntactic information) and unintentionally altering the original program's comments and whitespace (losing lexical information). It is again tempting to think of preserving these as another pesky engineering problem to be solved by some careful tracking and extra casework within the implementation. In fact, they are both challenging problems made far more challenging by the attempt to support multiple languages, for solving them requires "passing a camel through the eye of a needle" in simplifying the input program greatly to be easy for a tool to process, and then magically reconstituting a large amount of information that was dropped. CUBIX solves part of the problem of losing syntactic information, by making it feasible to build multi-language tools without normalizing many languages into some common form. But on the whole, bidirectionalization in source-to-source transformation remains an open problem. Later, we will discuss how our work on ECTAs was originally motivated to serve as a component in a larger idea for solving the bidirectionalization problem.

A coordinated attack Meta-metaprogramming seeks to overcome major obstacles in how programming tools are built. These challenges may be fundamental, but solving them is possible.It should be possible to overcome linguistic complexity because all this complexity is in some way duplicated across different tools for the same language. It should be possible to overcome heterogeneity because languages have high similarities even as they are peppered with differences.

Though it may seem that the meta-metaprogramming techniques in this thesis are disparate ideas, each dependent on a distinct stroke of insight extending a different body of work, there are common themes running throughout. If these techniques are the technology of meta-metaprogramming, then the science is in deeply understanding how languages are defined, how algorithms for processing code are conceived, and how tools are built.

Though different parts of this thesis draw on different fields, from type theory to tree automata, one field stands out connecting all of them. The four projects described in this thesis — CUBIX, YOGO, MANDATE, and ECTA — all draw on the theory of rewriting in different ways. Rewriting is the source of the "sum-of-signatures" approach and the signature/node distinction underlying CUBIX's modularity; the mechanism used by YOGO to discover syntactically-distinct yet semantically-identical code; the generic substrate upon which MANDATE is able to describe and transform semantics; and the target of ECTA's representational innovations. Because so much of the study of both languages and transformations is elegantly expressed in rewriting,



```haskell
-- | Inserts "blocksCovered[n] = true" printouts for test-coverage
--   at every basic block
--
-- Runs on C, Java, JavaScript, Lua, and Python
instrumentTestCoverage :: forall fs l. (CanInstrument fs)
                       => ProgInfo fs -> TermLab fs l -> Annotater (TermLab fs l)
instrumentTestCoverage progInfo t = performCfgInsertions @(StatSort fs) progInfo
                                    $ allbuR (addCoverageStatement progInfo) t
```

Figure 1-1: Promotional banner from CUBIX website

we can say that rewriting is the theory of tools.

Rewriting, particularly its main subfield of term rewriting, is a well established field, taught by textbooks such as Baader and Nipkow [7]. Concepts from term rewriting used in this thesis include the basic theory of terms, algebras, and signatures; narrowing and unification; the concept of linearity; and termination orders, particularly the multiset order.

We now give a more detailed overview of the individual projects in this thesis.

1.2 The CUBIX Framework: One Tool, Many Languages (like YOGO)

The traditional approach to building multi-language tools is to translate multiple languages to some common intermediate representation. While this works acceptably for static analysis and code generation tools, for source-to-source transformation, it's a choice between "mutilating the code" in the process of normalizing it (e.g.: turning all for-loops into while-loops), or not normalizing at all and attaining no actual sharing.

As an alternate to IRs, there are several approaches for modular syntax, where languages are decomposed into several fragments. However, doing so requires reconstructing the entire language out of reusable components, and that these components are identical across languages. As a result, previous attempts to use modular syntax

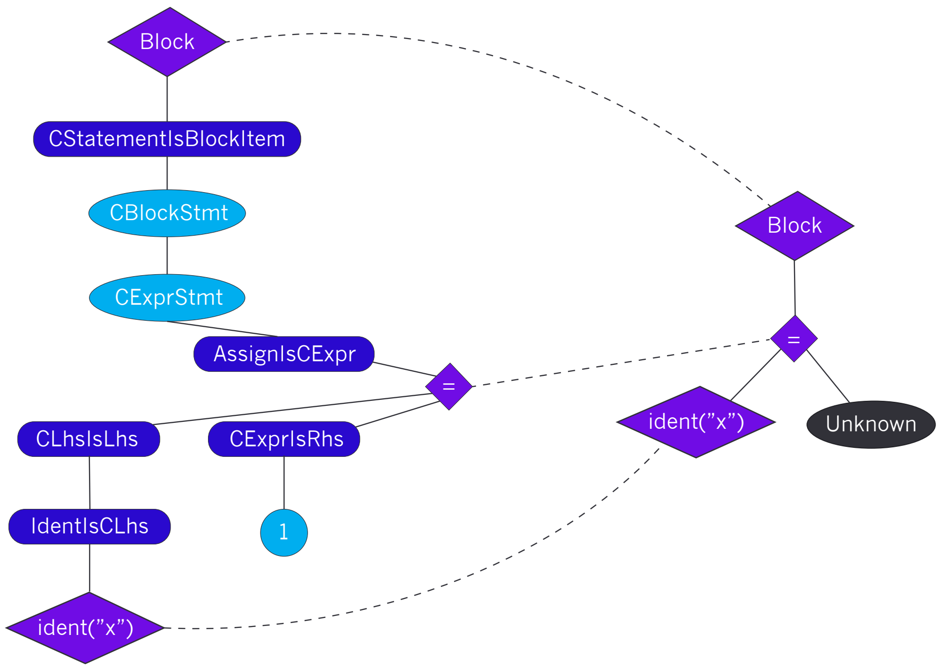


Figure 1-2: (Left) Representing a C program as a mixed tree with language-specific (light blue ellipse) and generic (purple rhombus) parts, with sort injection nodes (dark blue rounded rectangle) as their glue (Right) The conceptual interface of a “blown-down" tree presented to a language-parametric transformation

have only been for DSLs and toy languages,

Cubix makes modular syntax scale by adding **sort injections**. A sort injection is a modular component declaring one sort to be a subsort of another. Using these, Cubix represents programs as a mixture of language-specific and generic parts. Transformations can then be written atop the generic parts, and parameterized on language-specific operations. Figure 1-2 depicts such a "mixed tree" and the restricted view offered to transformations.

We have implemented support in Cubix for 5 languages: C, Java, JavaScript, Lua, and Python. We implement 3 semantics-preserving source-to source transformations in Cubix, and achieved a 100% pass rate on applicable compiler tests. Our "Turing test" human study shows that code transformed by our Cubix transformations are no less readable than code manually transformed by a human. And we created a prototype of a transformation that Facebook and Dropbox engineers spent thousands of hours performing manually, implemented simultaneously for all 5 languages.

On top of Cubix, we built Yogo, a semantic search tool with the world's most expressive matching language, as both an application of Cubix and an interesting multi-language tool in its own right. With Yogo, one can write a single query for a high-level operation such as an array frequency count, and match all the implementations in Figure 1-3, which includes many variations in both Python and Java. It does this by translating these loopy programs into the purely-function *program expression graph* (PEG) representation, using a declarative rule set to build an *e-graph* to represent an exponential number of equivalent programs, and using *dataflow patterns* to recognize ever-higher-level operations. Figure 1-4 depicts Yogo recognizing that a concrete loop is an implementation of the abstract operation of iterating through a sequence (iterV), so that higher-level rules may be written atop it.

Yogo's ability to handle multiple languages comes primarily from its use of declarative rules, but it relies on Cubix to actually support multiple languages, as its translator from program to PEGs is implemented in Cubix, and supports 3 languages. Though Yogo is not a source-to-source transformation tool, the primary focus of Cubix, it nonetheless benefits in the greater ease of sharing code between its support for those 3 languages.

By using YOGO to search for a buggy pattern, we were able to find a bug in a 1.2 million line codebase that had been missed by a handwritten static analyzer designed to catch that exact kind of bug. And, showing its power beyond code search, we have used YOGO to prune equivalent programs from a meta-learning benchmark set.

YOGO was primarily the work of Master's student Varot Premtoon under supervision of this thesis's author, and so it is given abridged coverage in this thesis.

1.3 ECTAs: Program Synthesis with Dependencies

The number of tiny programs is enormous. This is the challenge faced by enumerative program synthesizers, which try to discover programs by searching some restricted space.

The solution is in shared substructure. Just as a graph contains perfect information about exponentially-many paths, there are compact data structures representing exponentially-many programs. Just as the number of paths in a graph is easy to count, though prohibitive to enumerate, there are facts about an entire space of programs which are easy to compute, and can be used to select the "best" one.

Version space algebras (VSAs) [135, 107] are the most popular such data structure. They rest on the ability to efficiently represent cross products of program spaces: if P1, P2, and P3 are the three programs in the space which compute a, and R1, R2, and R3 the three programs which compute b, then $f_{\bowtie}(\{P1, P2, P3\}, \{R1, R2, R3\})$ is the space of 9 programs which compute $f(a, b)$ — if the P and the R can be chosen independently. Choices of subterms are not independent in many domains, which hinders VSAs from being used for many domains. Another more-recently-popular data structure, e-graphs, are equivalent in expressiveness and have the same problem. In fact, we find that they are identical to each other except in the manner of construction, and are both identical to special cases of another such data structure, *tree automata*, all of which struggle with said domains.

One such domain is in polymorphic function application. The `map` function has

```
count = 0
for a in cart:
  if a == item:
    count += 1
use(count)
```

(a)

```
count = 0
for i in range(len(arr)):
  if itm != arr[i]:
    continue
  count += 1
use(count)
```

(b)

```
count = 0
for i in cart:
  if debug:
    print(cart[i])
  if cart[i] == item:
    count += 1
use(count)
```

(c)

```
count = 0
i = 0
while i < len(cart):
  if cart[i] == k:
    count += 1
  i += 1
use(count)
```

(d)

```
use(cart.count(item))
```

(e)

```
int count = 0;
for (Item x : list)
  if (x == k)
    count += 1;
use(count);
```

(f)

```
int count = 0;
for (i = 0; i < list.size(); i++)
  if (list.get(i) == k)
    count += 1;
use(count);
```

(g)

```
int count = Collections.frequency(list, k);
use(count);
```

(h)

Figure 1-3: Python (a-e) and Java (f-h) variations of array frequency count

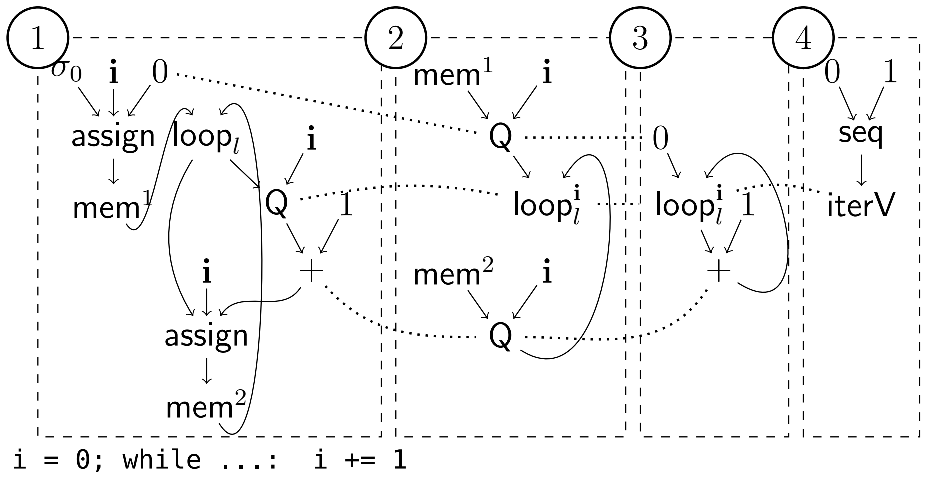
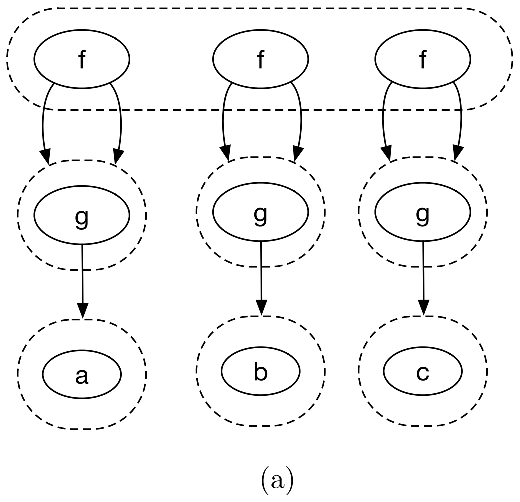


Figure 1-4: E-PEG for `i` = 0; while ...: `i` += 1. Some nodes duplicated for clarity.

type $\forall a.(a \to b) \to [a] \to [b]$. The space of valid programs of the form `map f l` is not given by the product of the set of available choices for f and the set of available choices for l.

Our solution is **equality-constrained tree automata** (ECTAs). ECTAs are like ordinary tree automata (and hence VSAs), except that they may also contain constraints that certain paths of terms in the space must have subpaths equal to each other. This simple constraint language is nonetheless sufficient to encode many useful spaces of programs, such as well-typed terms built out of polymorphic function applications. We designed a fast enumeration algorithm for ECTAs, and implemented it into our ECTA library ECTA. Applying the fast enumeration algorithm to an ECTA representing a space of well-typed terms gives a synthesizer. For the polymorphic function domain, on some benchmarks, ECTA can find all well-typed terms over 100x faster than a specialized best-in-class synthesizer, HOOGLE+, can find a single term.

Our initial motivation for this work was as part of a much larger problem extending CUBIX in a deeper attack against the bidirectionalization problem. With CUBIX, a source-to-source transformation need not duplicate work between the similar constructs across languages. But it still may need to do so for similar constructs within a language, such as the many types of loops. This is the problem of *resugaring*. The general setup of resugaring is to create desugaring rules like `x += a;` $\to$ `x = x + a;` to



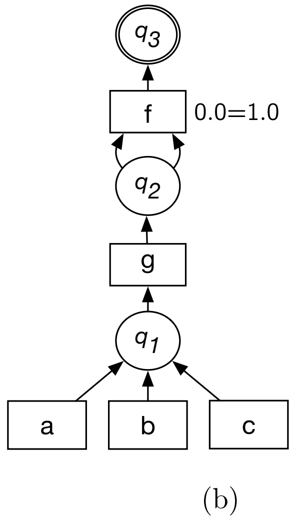


Figure 1-5: (a) E-graph representing the set of terms of the form $f(g(X), g(X))$ where $X \in \{a, b, c\}$ (b) ECTA for the same set. Note how the E-graph gets no sharing benefit because it cannot represent the dependence between the two copies of X, whereas the ECTA can.

produce a simplified program, and to later run these rules backwards. The choices of possible backwards rewrites create a large space of possible resugared programs, which can then be scored for selection. We sought to use the existing technique of version-space algebras to represent this space, but hit a barrier: version spaces perform poorly when different subterms cannot be chosen independently, as in the two occurrences of `x` in `x = x + a`. We suspended the resugaring project to solve this more general problem in program synthesis, for which we devised ECTAs. Now that we have ECTAs, finishing the original project to better solve the bidirectionalization problem is once again viable future work.

1.4 Mandate: A CFG Generator Generator

The ultimate solution to the linguistic complexity problem would be to write down the semantics of each programming language once, and then automatically generate all desired tools from it. As a first step towards that dream, we developed a control-flow-graph generator generator.

Building a CFG generator generator is interesting theoretically. For the tools that have been generated from a semantics, such as an interpreter, the correspondence is usually obvious. Yet a control-flow-graph generator looks nothing like an operational semantics. The ability to turning semantics into a CFG generator is hence a great milestone toward the dream of generating all tools.

But it is also interesting practically. Though CFGs are simple and usually taken for granted, there are actually several common variants and many more uncommon ones, based on the granularity and the attributes tracked. With a CFG generator generator, it is easy to obtain generators for all of them. Our initial impetus for this project came from an abandoned project in static analysis, attempting to combine analyzers which partition a program in incompatible ways.

But first: what is a control-flow graph? It turns out that, for all the talk of “a control-flow graph is an abstraction of control flow," CFGs have never been formally defined with respect to program semantics. Without such a definition, one cannot even show that a CFG-generator generator produces correct output.

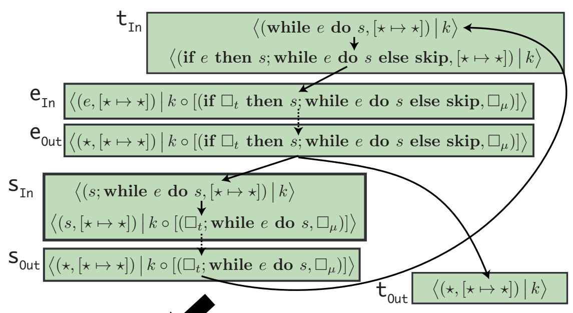
We develop an *executable theory* of control-flow graphs from first principles, implemented in our Mandate tool. We first develop an algorithm to convert a structural operational semantics into a form with a clear notion of “current execution point" and prove its correctness. We then introduce the technique of **abstract rewriting**, applying rewrite rules to symbolic terms, and use it both to state exactly how a CFG is an abstraction of control-flow and to convert the semantics into a CFG-generator, overapproximating the effect of each semantic rule on the set of all possible terms.

We used Mandate to generate multiple varieties of CFG-generators each for MITScript and Tiger, two languages used in compilers courses. On these languages, Mandate produces concise output similar to a human-written CFG-generator, even for a looping construct with multiple layers of syntactic sugar. We then wrote two static analyzers atop these generated CFGs, proving their utility.

$$\textbf{while } e \textbf{ do } s \rightsquigarrow$$
$$\textbf{if } e \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}$$

$$\textbf{if true then } s_1 \textbf{ else } s_2 \rightsquigarrow s_1$$

$$\textbf{if false then } s_1 \textbf{ else } s_2 \rightsquigarrow s_2$$



t_{In}
$\langle(\textbf{while } e \textbf{ do } s, [\star \mapsto \star]) \mid k\rangle$
$\langle(\textbf{if } e \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}, [\star \mapsto \star]) \mid k\rangle$
e_{In}
$\langle(e, [\star \mapsto \star]) \mid k \circ [(\textbf{if } \square_t \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}, \square_\mu)]\rangle$
e_{Out}
$\langle(\star, [\star \mapsto \star]) \mid k \circ [(\textbf{if } \square_t \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}, \square_\mu)]\rangle$
s_{In}
$\langle(s; \textbf{while } e \textbf{ do } s, [\star \mapsto \star]) \mid k\rangle$
$\langle(s, [\star \mapsto \star]) \mid k \circ [(\square_t; \textbf{while } e \textbf{ do } s, \square_\mu)]\rangle$
s_{Out}
$\langle(\star, [\star \mapsto \star]) \mid k \circ [(\square_t; \textbf{while } e \textbf{ do } s, \square_\mu)]\rangle$
t_{Out}
$\langle(\star, [\star \mapsto \star]) \mid k\rangle$

```
genCfg t@(While e s) = do
  (tIn, tOut) <- makeInOut t
  (eIn, eOut) <- genCfg e
  (sIn, sOut) <- genCfg s
  connect tIn eIn
  connect eOut sIn
  connect eOut tOut
  connect sOut tIn
  return (tIn, tOut)
```

Figure 1-6: A portion of a sample run of Mandate. (Top left) SOS rules for loops and conditionals (Top right) A graph-pattern generated from these rules, describing the control-flow of all while-loops (Bottom) Generated CFG-generation code

1.5 Overview and Works Covered

Cubix is covered in Chapter 2; Yogo is covered therein as an application of Cubix. Ecta and Mandate are covered in Chapters 3 and 4 respectively. Related work for all chapters is collected into Chapter 5. Finally, the conclusion (Chapter 6) sums up our thoughts on this thesis's impact on the problems of meta-metaprogramming.

This thesis contains material previously published in the following papers and tech reports:

- One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax [96]
- One CFG-Generator to Rule them All [95]
- Multi-Language Code Search [140]
- Semantic Code Search via Equational Reasoning [141]

- Automatically Deriving Control-Flow Graph Generators from Operational Semantics [94]

The work on equality-constrained tree-automata is currently unpublished.

These papers represent joint work with Varot Premtoon, Jackson Kearl, John K. Feser, and Armando Solar-Lezama. Additionally, a very small portion of this thesis was first published in "A Large-Scale Benchmark for Few-Shot Program Induction and Synthesis" [5], joint work with Ferran Alet, Javier Lopez-Contreras, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Josh Tenenbauhm,

Chapter 2

Cubix: One Tool, Many Languages

2.1 The Problem of Language-Parametric Tools

In 2014, Dropbox had a massive refactoring to do. They wanted to let users log in with both a personal and corporate account on the same computer, but they had built the client assuming users only had one account. To change this, they needed to pass along information about which account each operation was for, and thread an `Account` parameter through tens of thousands of functions. Many program transformation experts could have readily built a tool for this, though it would have been a quite expensive task for one use-case, and Dropbox opted not to hire one. And so, in a company of over 100 engineers, the top project of the year was to tediously add parameters to functions.

Back in 2010, Facebook had a similar problem. All sorts of privacy bugs were being exposed by the media, like weird combinations of settings that would let someone view another user's private photos. Facebook assembled a crack team; they needed this problem fixed quickly, and made to never return. The privacy checks were too haphazard: a bunch of conditionals every place where photos may be displayed. They needed to move these all to one place: private photos would never be fetched from the database in the first place. To do this, they needed to add a `ViewerContext` parameter to tens of thousands of functions. And so, for several weeks, every waking moment of several dozen of their top engineers was spent adding parameters to functions.

One might think that a clairvoyant entrepreneur in 2010 could have built a tool for this problem, and sold it to both Facebook and Dropbox. Alas, no, for Facebook's codebase was in PHP, while Dropbox's was in Python. And, with today's methods, building a similar program transformation tool for different languages requires **building it separately for each language**.

We are not the first to notice the *language-parametric transformation* problem of building a single transformation that can run on multiple languages. Intuitively, this should be possible: languages have a lot of similarities, and humans can readily apply the same refactoring in many different languages. The challenge then is to find some way to capture the similarities across languages, while being flexible enough to express their differences.

The obvious approach is to convert many languages into a single *intermediate representation.* Unfortunately, doing so inevitably loses information. While this is fine for code-generation or analysis, it fails for source-to-source transformations, which must produce an output similar to the input. Instead, IR-based tools are known to "mutilate" the program, such as by converting all for-loops into while-loops.

There is another line of work that promises the kind of flexible representations needed: instead of building one representation to represent all languages, having a different representation for each language, but letting them share common fragments. This is the approach taken by previous work on modular syntax [9, 186], along with its cousin work on modular interpreters [108] and modular semantics [47, 120]. In principle, these techniques could be used to do language-parametric transformation, but the previous work does not scale to real languages. All these approaches assume that the entire language is built from these generic fragments. Hence, one would have to do huge amounts of up-front work to define fragments capable of representing all variations of each feature of modern programming languages, and assemble them into representations for each language. The difficulty of developing language-parametric infrastructure has meant that previous work in this space, such as the work funded by the Dutch program on language-parametric program restructuring [98, 76], has all been for DSLs, toy languages, and language subsets.

This chapter presents the first work that builds source-to-source transformations that run on multiple real languages. Our key insight is a new representation called **incremental parametric syntax** (IPS). In incremental parametric syntax, languages are represented using a mixture of language-specific and generic parts. Like previous work on modular syntax, transformations deal only with the generic fragments. Unlike previous work, the implementer starts with a pre-existing normal syntax definition, and only does enough up-front work to redefine a small fraction of a language in terms of these generic fragments. Rather, they can *incrementally* convert more of a language to generic fragments, as needed by new transformations. Best of all, since IPSs are defined as a "diff" to an existing syntax definition, implementations can re-use third-party language frontends.

We have implemented incremental parametric syntax in a Haskell framework called CUBIX, and implemented support for 5 languages: C, Java, JavaScript, Lua, and Python. To evaluate CUBIX, we built several program transformations that each run on multiple of those languages. We show transformations built in this style can have readable output, unlike IR-based approaches: our "Turing test" human study (Section 2.6) shows their output is no less readable than hand-transformed code. In this process, we also developed a new multi-language benchmark suite for semantics-preserving program transformations, the RWUS (Real World, Unchanged Semantics) suite (Section 2.6.1). We show transformations built in this style can handle language corner-cases: the example transformations pass 100% of compiler test suites, excluding some self-referential tests that should not pass ("assert function `foo` is declared on line 37") and tests that break the third-party parsers and pretty-printers (Section 2.5.3). Finally, using CUBIX, we created a prototype tool for threading variables throughout chains of function calls (Section 2.5.1), as in Dropbox and Facebook's problem, and implemented it for all 5 language simultaneously (including Python, but not yet PHP). We then used CUBIX in another project, creating YOGO (Section 2.8), the world's most advanced semantic code search tool, implemented simultaneously for multiple languages.

2.1.1 Why IRs Don't Solve Multi-Language Transformation

An old idea for building multi-language tools is to translate each language into some *intermediate representation*. This works for writing analyses and code-generators, but is a poor fit for source-to-source transformation, which must preserve information.

Conceptually, the IR approach to analysis is to provide a family of `lower` functions of type `C` → `IR`, `Java` → `IR`, etc, which transform each language into the IR, along with an `analyze` function of type `IR` → `AnalysisResult`. Similarly, the IR approach to code generation provides a term of type `IR`, along with `lift` functions of type `IR` → `C`, `IR` → `Java`, `IR` → `Python`, etc. The natural extension to transformation is to implement a `transform` function of type `IR` → `IR`, and compose it with the `lower` and `lift` functions to get language-specific transforms of type `Java` → `Java`, `C` → `C`, etc. But this makes a promise which is too good to be true: one can compose the `lower` and `lift` functions to get a translation from any language to any other!

The catch is that tools that implement this approach "mutilate" the program. Most commonly, the IR will be some kind of least common denominator of the supported languages, seen in frameworks like SAIL [50] and BAP [24], and bytecodes such as LLVM [106] and the JVM [110]. If the IR only supports `while`-loops, then any transformation through this `IR` will convert all loops into `while`-loops, even if the transformation has nothing to do with loops. Information about the original program has been lost. The alternative is for the IR to be a union of all concepts of the languages. The Clang AST, for instance, contains separate node types for both Objective-C and C++ exception-handling. This approach essentially still requires the user to write a transformation separately for each language: it can use the same node to represent similar constructs in different languages *only if* they are exactly identical. And, among its many other drawbacks, it still loses information about what's *not* in the program (e.g.: Java contains no pointer arithmetic, which simplifies analysis).

The end result is: because of these problems with conventional approaches, at time of writing, we are aware of no previous framework that allows the user to define a single program transformation, run it on programs from multiple languages, and obtain output suitable for humans.

2.1.2 Incremental Parametric Syntax

So, one-size-fits-all IRs don't work. Our solution is to find a way to apply parametric polymorphism to program transformations. The high-level idea of *incremental parametric syntax* is to build transformations with the following functions:

$$
\begin{array}{ll}
\texttt{decompose}_J & :: \texttt{Java} \rightarrow \texttt{Generic} \bowtie \texttt{Remainder}_J \\
\texttt{decompose}_C & :: \texttt{C} \rightarrow \texttt{Generic} \bowtie \texttt{Remainder}_C \\
\texttt{transform} & :: \forall \texttt{x} .\ \texttt{Generic} \bowtie \texttt{x} \rightarrow \texttt{Generic} \bowtie \texttt{x} \\
\texttt{recompose}_J & :: \texttt{Generic} \bowtie \texttt{Remainder}_J \rightarrow \texttt{Java} \\
\texttt{recompose}_C & :: \texttt{Generic} \bowtie \texttt{Remainder}_C \rightarrow \texttt{C}
\end{array}
$$

Here, languages are decomposed into generic and language-specific parts. Then a transformation can be run on the generic parts, while preserving the rest of the program so that high-quality source code may be reconstructed. Unlike the common IR approach, these type signatures guarantee that a transformation cannot modify the language-specific parts, and the `decompose` and `recompose` functions cannot be used to translate one language into another. And rather than construct the generic/language-specific decomposition up-front, IPS allows a programmer to begin with a third-party frontend for each language, and incrementally shift pieces of the language into the generic fragment as needed for new transformations. Hence, developers can add support for a new language in less than two days of work — and much of this time is spent looking at the language spec to understand how to model it in terms of generic components.

The composition $X \bowtie Y$ is done using an approach known in term-rewriting as "sum of signatures" and known in the functional-programming community as "data types à la carte" [160]. This approach can modularly define node types and mix-and-match them between languages, but does not let these nodes differ between languages: it cannot use the same notion of variable declarations to model both C declarations (which have types) and JavaScript ones (which do not). Similarly, in this approach, a generic assignment node cannot be used for both C/Java (where assignments are expressions) and in Lua/Python (where they are statements). We solve many of these problems with the new idea of **sort injections**. Sort injections are deceptively simple: just add an `AssignIsExpression` node to C and an `AssignIsStatement` node to Python. Yet they complete sum-of-signatures by modularly specifying what *edges* may be in an AST, and, in their general form, they solve many of the limitations of sum-of-signatures. Thanks to these sort injections, CUBIX can take a pre-existing syntax definition for a language, and generate a new representation of the language which is fully isomorphic to the original, but replaces portions of the AST with generic nodes.

With each language expressed as an IPS, we can write a transformation parameterized on the nodes and sort injections it deals with. It can then be run on any language that has these nodes and sort injections, but will give a compiler error when used on one that does not. These transformations can be further parameterized on language-specific operations such as symbol resolution, allowing us to build sophisticated multi-language transformations that can still handle many language-specific corner-cases.

2.2 Overview

In this section, we show how our approach allows constructing language-parametric transformations, and the work required to add support for a new language. In Section 2.2.1, we explain the construction of a transformation called “declaration hoisting,” and how it is configured to run on several languages. Section 2.2.2 then explains how to create an incremental parametric syntax for C. In the language of Section 2.1.2, Section 2.2.1 defines `transform`, while Section 2.2.2 defines $\texttt{Remainder}_C$, $\texttt{decompose}_C$, and $\texttt{recompose}_C$.

```c
int f(int a,int b,int s) {
  int t1 = 0, t2 = 1;
  if (s) {
    int r1 = t1*a+t2*b;
    return r1;
  }
  int r2 = t2*a+t1*b;
  return r2;
}
```

```c
int f(int a,int b,int s) {
  int t1, t2; int r2;
  t1 = 0; t2 = 1;
  if (s) {
    int r1;
    r1 = t1*a+t2*b;
    return r1;
  }
  r2 = t2*a+t1*b;
  return r2;
}
```

Figure 2-1: An example of hoisting a C program

2.2.1 An Elementary Hoisting Transformation

In this section, we describe the construction of a simplified transformation for *declaration hoisting*, and how with a small amount of configuration, we can apply it to C, Java, and JavaScript. This transformation showcases the versatility of our approach: although it totals only 27 lines for the transformation plus 30 lines for the language-specific code, it handles multiple language corner-cases, and achieves a high pass rate on the compiler validation test suites. Full code for the general portion is given below in Figure 2-2 and explained at a high level; a more detailed explanation is in Section 2.4.3.

The declaration hoisting transformation moves all variable declarations to the top of the scope, using normal assignments to initialize them. The end result is similar to how C89 requires programs to be written. Figure 2-1 gives an example C program and its hoisted version. The elementary hoisting transformation of this section is a simplified version of the hoisting transformation in our benchmarks (2.5.2), which also supports Lua and handles shadowing. Neither supports Python because Python lacks variable declarations.

Setting the syntactic constraints The user first writes a type signature declaring the general syntactic constructs a language must have to use this transformation:

```
declToAssign :: (CanHoist f)
          ⇒ Term f MultiVarDeclAttrsL → Term f VarDeclL → [Term f BlockItemL]
declToAssign mattrs (VarDecl′ lattrs b optInit) = case optInit of
  NoVarInit′        → []
  JustVarInit′ init → [injF (Assign′ (varDeclBinderToLhs b) AssignOpEquals′
                                          (varInitToRhs mattrs b lattrs init))]

removeInit :: (CanHoist f) ⇒ Term f VarDeclL → Term f VarDeclL
removeInit (VarDecl′ a n _) = VarDecl′ a n NoVarInit′

splitDecl :: (CanHoist f)
          ⇒ Term f BlockItemL → ([Term f BlockItemL],[Term f BlockItemL])
splitDecl (projF → (Just (MultiVarDecl′ attrs decls)))
          = ([injF (MultiVarDecl′ attrs (mapF removeInit decls))]
            , concat (map (declToAssign attrs) (extractF decls)))
splitDecl t = ([], [t])

hoistBlockItems :: (CanHoist f) ⇒ [Term f BlockItemL] → [Term f BlockItemL]
hoistBlockItems bs = concat decls ++ concat stmts
  where (decls, stmts) = unzip (map splitDecl bs)

elementaryHoist :: (CanHoist f) ⇒ Term f l → Term f l
elementaryHoist t = transform inner t
  where
    inner :: (CanHoist f) ⇒ Term f l → Term f l
    inner (project → (Just (Block bs e))) = Block′ (liftF hoistBlockItems bs) e
    inner t                               = t
```

Figure 2-2: Implementation of the elementary hoist transformation

variable declarations, assignments, blocks, and identifiers. The type signature also requires that assignments and variable declarations must be valid members of blocks — these are *sort injections*, as described in Section 2.2.2. These constraints are all combined into the overall constraint `CanHoist`, as seen in Figure 2-2. We give full code for these constraints in Section 2.4.

Language-specific operations Variable initializations and assignment RHSs can be different. The Java array initialization code `int [] x = {1,2,3};` is transformed into `x = new int []{1,2,3};` — variable initializers in Java are a strict superset of Java expressions. C variable declarators have different abstract syntax from C lvalues. To deal with these, the transformation takes as a parameter two language-specific operations, `varInitToRhs` and `varDeclBinderToLhs`, invoked on lines 5 and 6 of Figure 2-2.

Writing the transformation The transformation traverses every block in the program. At each block, it checks if each block item is a variable declaration (line 13). If so, it splits the declaration into one without initialization, and into a sequence of zero or more assignments (lines 14—15). The extracted assignments are inserted where the variable declarations lay previously, while the extracted variable declarations are prepended to the front of the block.

Dealing with language subtleties The hoisting transformation deals with several subtleties through the language-specific operations, but we give another one here: In JavaScript, directives such as `"use strict"`; must be placed at the top of a block to have effect; hoisting something above it can break the code. Perusing the spec, we saw directives are essentially treated as a separate kind of syntax, so we modified the representation of JavaScript to store them separately. This is seen in the `e` variable on line 26 of Figure 2-2, which stores extra data associated with a block — possible pragmas for JavaScript, and nil for most languages. Implementing this design decision fixed bugs in multiple transformations. More examples are given in the figures in Section 2.5.2.

While simple, the elementary hoisting transformation in this section runs on three languages, deals with multiple language subtleties, and has a 98.4% pass rate on compiler test suites (compared to the 100% pass rate of the real version). Overall, these techniques allow transformations for different languages to share code to the extent that the two languages are syntactically similar. Later in this chapter, we give more interesting transformations that also make use of static analysis and control-flow information.

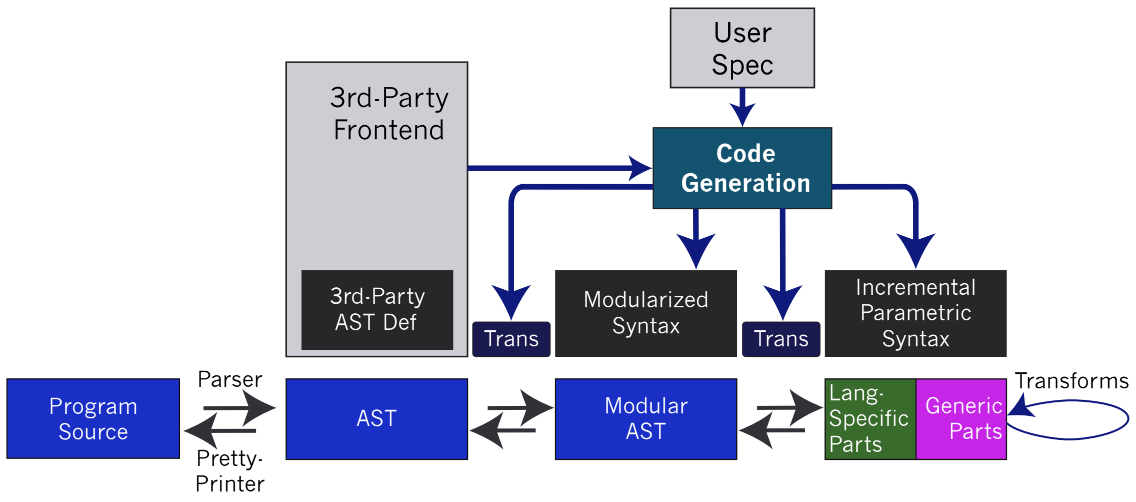


Figure 2-3: Architecture of Cubix

2.2.2 Modularizing C

In Section 2.2.1, we outlined how to build a hoisting transformation which works on any language that contains some common notion of variable declarations, assignments, and blocks. We now show how to construct an incremental parametric syntax for C, in which parts of the language are recast in terms of these common components.

Our approach gives a language three representations. The starting point is some already-existing syntax definition of the language from a third-party library, with its accompanying parser and pretty-printer. For C, we use Haskell's `language-c` library [77], which defines C's abstract syntax as a set of mutually recursive algebraic data types like `CExpression` and `CAssemblyStatement`. Next is the "modularized" representation, which gives the exact same set of data types, but as independent signatures that do not reference each other. The sum of these signatures is isomorphic to the `language-c` abstract syntax definition. This makes it easy to sum together a different set of signatures, replacing some of the C-specific data types with generic ones, yielding the third representation, the incremental parametric syntax. These three representations are mutually isomorphic, and translations between them are derived mostly automatically: the user only writes code for the node types which have been supplanted by generic equivalents. Figure 2-3 depicts how the representations and translations are generated, and how a program is transformed through each of them at runtime.

Modularized representation For each algebraic data type in the C abstract syntax, the user must generate a new data type representing nodes of that sort inside an arbitrary AST (a **signature** for that node). Combining these give a new representation identical to the original, but made of independent components. The user generates these definitions completely automatically, using the Template Haskell code-generation engine. Section 2.3.1 explains how we represent and combine signatures, while Section 2.3.4 explains the data type transformation in more detail.

Incremental parametric syntax: Nodes The hoisting transformation is built on general components for assignments, variable declarations, and blocks. The user will need to replace these components of C, but no others, with their corresponding generic components, yielding the incremental parametric syntax. This is incremental because the user will revisit this step as more components of C need to be genericized to support new transformations.

To genericize these components, the user first compares their definitions in the C specification to the specification of the generic components, making sure the latter can model the former. To customize the generic `VarDecl` node to C, the user must create a new node of sort `VarDeclAttrsL` containing the C-specific components of a variable declaration (type and storage specifiers, assembly name, and attributes). The user does similarly for a couple other C constructs.

IPS: Sort injections The user now finishes customizing the generic components to C by specifying where they fit in the C syntax. The user indicates that generic assignments may be used as C expressions, while C expressions form the RHS of assignments. The user does this by e.g.: generating a `AssignIsCExpr` node. This establishes an injection from terms of sort Assign to terms of sort CExpr, which we call a **sort injection**. The user does similar to place the other generic nodes within the C syntax. CUBIX generates nodes witnessing these sort injections

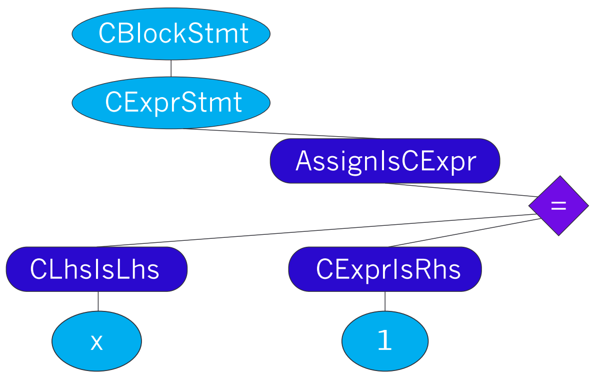


Figure 2-4: A term in the incremental parametric syntax for C. The ellipses (light blue) represent language-specific nodes; rhombi (purple) represent generic nodes; rounded rectangles (dark blue) represent sort injection nodes.

IPS: Putting it together The user now defines the incremental parametric syntax for C by writing a couple lines of Template Haskell that takes the list of signatures in the modularized syntax, subtracts the replaced nodes, and adds the generic components and sort injection nodes. This code is given in Figure 2-14 in Section 2.4.1. The sum of these signatures is the signature for the IPS for C, and the terms of this signature are given by its type-level fixpoint. These terms resemble Figure 2-4, showing the mixture of C-specific and generic nodes, with sort injections between them.

IPS: Translations The user writes instances of the `trans` and `untrans` operators between the nodes that have been removed from C, and the generic ones that replaced them. Generic programming deals with the nodes shared between the IPS and the modularized syntax, giving translation functions between the two representations. Our actual implementation of these translations for C totals 130 lines of Haskell code, about 40 of which are boilerplate.

2.3 Core Ideas

In this section, we explain the core new ideas that make our language-parametric transformations possible. Section 2.3.1 gives background on modular syntax, used in the rest of this section. Section 2.3.2 presents the terminology and goals of incremental

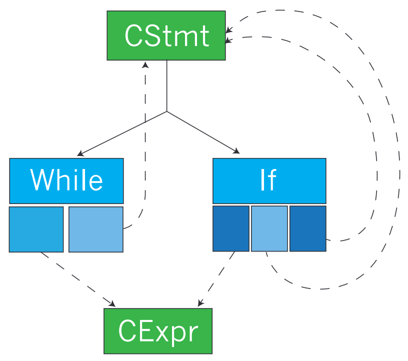


Figure 2-5: Fragment of a typical representation of C. The solid arrows represent the instance-of relationship; dotted represent containment.

parametric syntax. We achieve this through the concepts in the following sections: Section 2.3.3 presents sort injections, and Section 2.3.4 explains the translation of a syntax into its modularized version.

2.3.1 Background: Data Types à la Carte

The basic idea of modular syntax is simple: languages should be defined by a set of nodes, and the same node can appear in many languages. So, a transformation to swap the two branches of an if-statement should be runnable on any language with if-statements.

Unfortunately, in common representations of syntax, whether as an algebraic data type (ADT) like the fragment in Figure 2-5, or as a set of classes, this is not possible. The problem is mutual recursion between types. A C if-statement contains C expressions, which can contain C statements. So, the node for C if-statements is tied to definitions for all other C statements. The structure of code follows the structure of data, and so a traversal written over this type will also be coupled to all C statements.

Even without the mutual recursion, trouble arises as soon as one uses a fixed type like $C \to C$ or Java $\to$ Java for a program transformation. The reason goes back to the basic theory of subtyping. Producer functions of type $A \to C$ are *covariant* in C, meaning new cases can be added to C without changing the function. Consumer functions of type $C \to A$ are *contravariant* in C, meaning cases can be removed from C without changing the function. But functions of type $C \to C$ are *invariant*, meaning the code will break if any cases are added or removed from the language. Techniques such as the visitor pattern can help, but introduce new limitations (discussed by [100]), and do not allow for a multi-language transformation so long as the types are tied together. Switching to a dynamically-typed language also does not help; removing the types does not remove constraints over the data.

What does help is removing the recursion from the syntax definitions, and using parametric polymorphism for the types. Mathematically, an ADT is defined in three stages: first, data is tupled into a constructor; then many constructors are summed into a signature, a list of node types with unspecified children; and then a fixpoint is taken over the signature, yielding recursive trees. The idea of the sum-of-signatures representation, known in the functional programming community as data types à la carte (DLC) [160], is to defer the fixpoint operation. The programmer instead programs against signatures, which are not recursive, and can be modularly combined.

In DLC, a signature takes the form of a data type similar to conventional abstract syntax, but where all recursive terms have been replaced with a type variable, so that the type of children may be specified later. Each signature may represent an independent fragment of a language; these signatures may be freely summed into a signature for an entire language, and then closed recursively, as depicted in Figure 2-7. Figure 2-6 shows an example of a term written in DLC, taken from [160].

Data types à la carte generalizes easily to multiple sorts: have a type variable for terms of sort `Stmt`, another for terms of sort `Exp`, etc. This unfortunately makes it difficult to add new sorts, or to have languages with different numbers of sorts. The insight of [185] is to merge these into a single higher-order type variable t. Subscripting t with various labels gives the type of terms of a certain sort: $t_{\texttt{Stmt}}$ represents terms of sort `Stmt`, $t_{\texttt{Exp}}$ represents terms of sort `Exp`, etc. But `t` itself is a single variable, representing terms of all sorts. Figure 2-11 shows signatures following this pattern.

```
data Add e = Add e e
data Val e = Val Int
data (f :+: g) e = Inl (f e) | Inr (g e)
data Term f = Term (f (Term f))

type ExpSig = Add :+: Const
type Exp = Term ExpSig

addExample :: Exp
addExample = Term (Inl (Add (Term (Inr (Val 118))) (Term (Inr (Val 1219)))))
```

Figure 2-6: Using data types à la carte to present the expression 118+1219, with addition and constant nodes defined in separate fragments. Note how the `Add` and `Val` fragments do not reference each other, but instead use the type variable `e`, which is later filled in to contain both fragments.

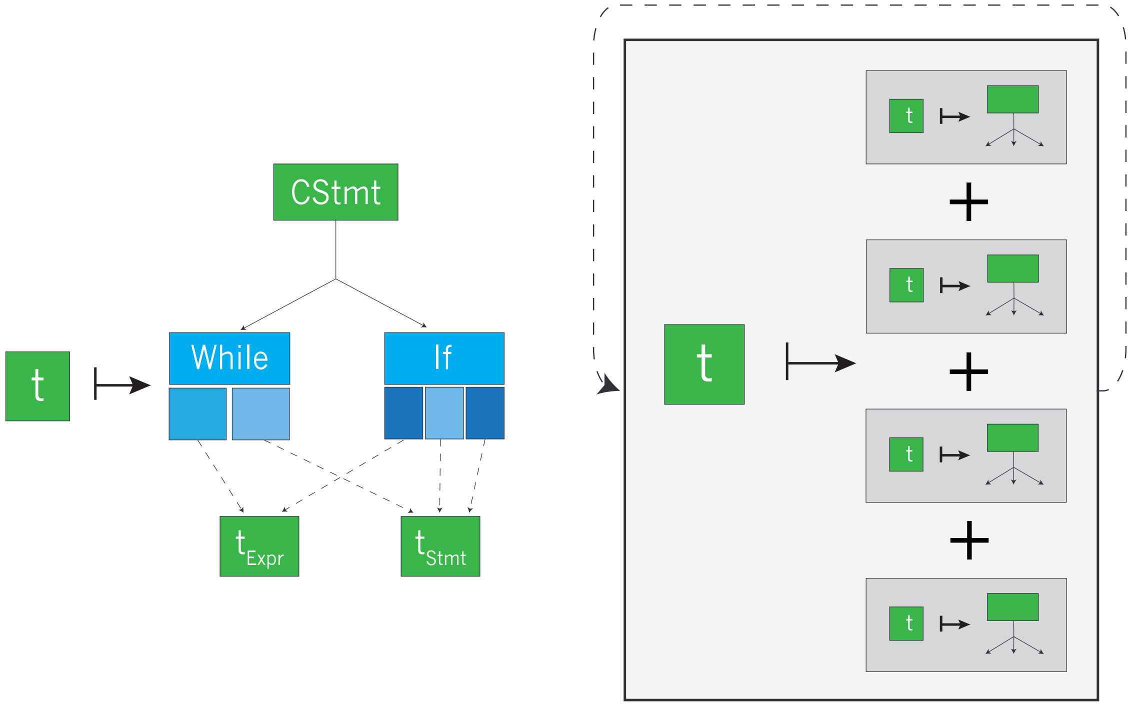


Figure 2-7: In DLC, a language is represented by a list of subsignatures like the one on the left. Each signature has a type variable for subterms, in lieu of self-reference. The subsignatures are combined into a signature for the whole language, which is then closed by specifying that allowed subterms of terms of this signature are other terms of this signature (right).

2.3.2 Incremental Parametric Syntax

As explained above, functions of type $C \to C$ have a type which is *invariant* in C. That is, in general, the code for any function that consumes and produces a value of type C will break when the definition of C is modified. So, instead of using a fixed type, the way to write a function that can transform many data types is with parametric polymorphism. For instance, the sort function of type $\forall x.\text{Ord x} \Rightarrow [x] \to [x]$ works over lists of any data type that supports comparison, and, after inlining, is just as efficient as a sort function written for each data type. Our goal is to bring this combination of generality and specialization to program transformation.

Let $F_1, \ldots, F_n$ be fragments that may be contained in many languages (i.e.: generic parts of languages). We define a **parametric syntax** $\mathcal{S}$ for a language as any representation that supports an operation $\prec$ such that a transformation over any language containing $\overline{F_i}$ may be written $\forall x.\overline{F_i \prec x} \Rightarrow x \to x$. This gives a name to previous work: any language written in DLC is a parametric syntax.

But the drawback of previous incarnations of DLC and other forms of modular syntax is that language definitions in those styles are what we term a **fully parametric syntax**, meaning that the syntax must be written entirely in terms of generic fragments.

More formally, a fully parametric syntax is any representation satisfying:

1. There is some combination operator $\bowtie$ which merges fragments. The $\bowtie$ operation must satisfy the property: if $F \prec G$ or $F \prec H$, then $F \prec (G \bowtie H)$.

2. Each syntax definition is built entirely by combining generic fragments. That is, $\mathcal{S}$ is a fully-parametric syntax if it can be written $\mathcal{S} \doteq G_1 \bowtie \ldots \bowtie G_m$, where each $G_i \in \{F_1, \ldots, F_n\}$.

Defining a fully parametric syntax for a language requires a large amount of up-front labor. Incremental parametric syntax lowers this initial barrier.

We say that $\mathcal{S}$ is an **incremental parametric syntax** if there is a non-parametric syntax $\mathcal{T}$ and a "fragment removal" operator $\setminus$ such that $\mathcal{S}$ may be expressed:

$$\mathcal{S} \doteq (\mathcal{T} \setminus F_1 \setminus \cdots \setminus F_m) \bowtie G_1 \bowtie \ldots \bowtie G_n$$

An incremental parametric syntax allows the user to start with a pre-existing syntax definition, replace some components with their generic equivalents, and then write transformations against the generic components. Given the complexity of a production language, this approach is necessary for getting a language-parametric transformation running on real languages in a reasonable amount of time.

In our instantiation of incremental parametric syntax, we use the signature subsumption and sums from data types à la carte to provide the fragment subsumption ($\prec$) and $\bowtie$ operators. We use new ideas for the $\setminus$ operator: convert the existing syntax into a sum of language-specific signatures (Section 2.3.4), and then use signature subtraction. Additionally, to add generic fragments, one must also add new nodes to reshape the grammar to accept them (Section 2.3.3),

Parametric syntax closely relates to the Expression Problem [176], which concerns being able to separately extend a language with new terms and new operations. Any incremental parametric syntax is also a solution to the Expression Problem, as it allows a language to be extended with new terms and operations. However, a solution to the Expression Problem need not allow for expressing multiple languages. As parametric syntax is our name for a family of existing approaches, discussion of how parametric syntax solves the Expression Problem can be found in the DLC paper [160].

2.3.3 Sort Injections

Using data types à la carte, one can modularly specify which nodes may be in a language, and replace them with generic ones. However, similar nodes in different languages may interact differently with the rest of the language. Assignments are expressions in C/Java and statements in Lua/Python. Most languages have various

```
data AssignIsCExpr t l where
  AssignIsCExpr :: t AssignL → AssignIsCExpr t CExprL

instance (AssignIsCExpr :≺: f) ⇒ InjF (Term f) AssignL CExprL where
  injF = AssignIsCExpr
  projF x = case project x of
    Just (AssignIsCExpr x) → Just x
    _                      → Nothing
```

Figure 2-8: Sort injection node and its associated sort injection

assignment operators like +=, but Lua does not. Rarely will a generic node be an exact fit for a construct already in a language. Instead, it must be customized for that language.

We solve this with sort injections. A **sort injection** from A to B is an injective function from terms of sort A to terms of sort B, together with its partial inverse. C and Java have a sort injection from `Assign` to `CExpr` and `JavaExpr` respectively, while Lua and Python have ones from `Assign` to their respective statement sorts. And all languages except Lua have a sort-injection from their respective language-specific assignment-operation sorts to the generic assignment-operation sort.

The most straightforward way to provide such a sort injection is via a **sort injection node**, an unary production of sort B with a single child of sort A. Figure 2-8 gives an example sort injection and node from generic assignments to C expressions.

So, while DLC modularizes which nodes may be in a languages, sort-injections modularize the edges. Adding the `AssignIsCExpr` node from Figure 2-8 to a syntax definition is equivalent to allowing a parent-child edge from anything that contains C expressions to assignments.

Sort injections also serve an additional purpose: abstracting over intermediate nodes. In all supported languages, assignments may be used as top-level items in blocks. However, this occurs through a chain of language-specific nodes. Block statements in C may not be assignments directly, but they can be ordinary state-

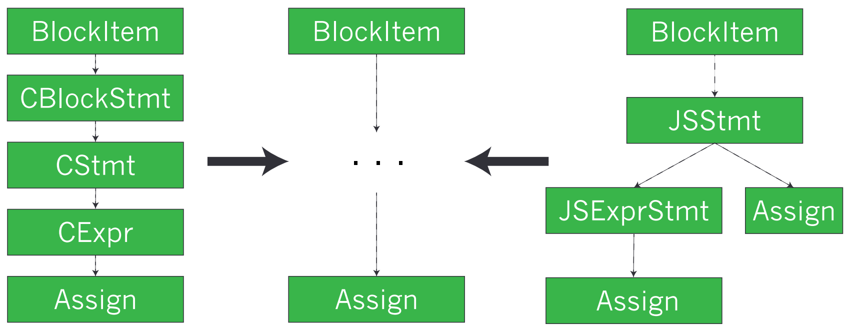


Figure 2-9: Sort injections from `Assign` to `BlockItem`

ments, which may be expression statements, which may be assignments. And, in the third-party JavaScript frontend used by Cubix, there are actually two (semantically-equivalent) ways that assignments may be statements. All this can be abstracted into the constraint: there is a sort injection from assignments to block items. Figure 2-9 illustrates this example.

In a transformation such as Hoist that works on languages with a sort injection from assignments to block items, the transformation has the ability to place assignments into blocks, and to check if a block item is an assignment. So one can think of this transformation as working not on the original tree but on a "blown-down" tree, which only contains these generic nodes. Figure 2-10 shows an example of a blown-down tree. This is similar to the theory views seen in Maude [31], and to the homeomorphic embedding in term rewriting [7].

2.3.4 Modularizing a Syntax Definition

The preceding sections gave some of the techniques of incremental parametric syntax; we now show how to convert an existing syntax definition so that it can be incrementally generalized. The key idea is to transform an existing syntax definition $\mathcal{T}$ into the combination $F_1 \bowtie \ldots \bowtie F_m$. This provides the final component of incremental parametric syntax, as $\mathcal{T} \setminus F_i$ can be defined by simply removing F_i from the combination.

The ADT modularization transformation is most easily explained by an example:

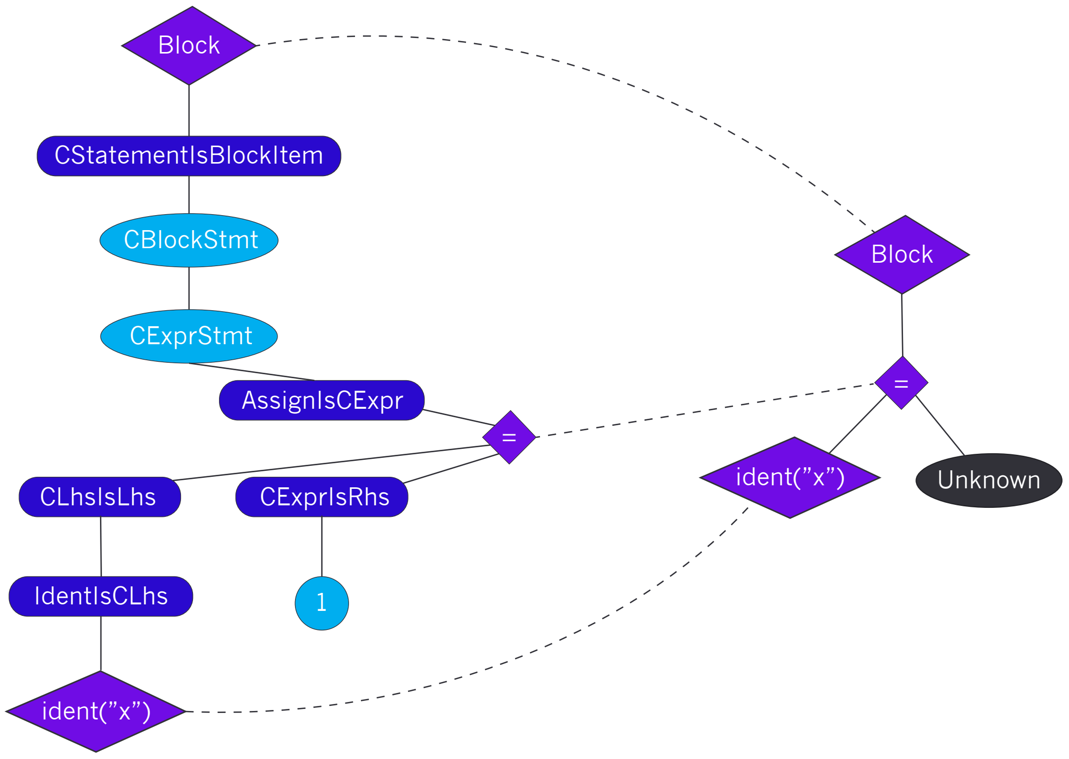


Figure 2-10: Blowing down a tree

it transforms the ADTs on the left side of Figure 2-11 into the generalized algebraic data types (GADTs) on the right. The GADTs stand independently, with no recursion between them. Instead of a recursive reference to the `Arith` type, for example, the type `t ArithL` can be read "Terms of sort `ArithL`, which will be specified later." But when those terms are specified, and the independent types are combined back together, the result type `Term (Arith :+: Atom :+: Lit) ArithL` is isomorphic to `Arith`. Figure 2-12 shows how these types are combined.

In our instantiation of incremental parametric syntax, this means converting a syntax definition into DLC. For syntax definitions given as mutually recursive algebraic data types, this is quite easy to do. The recursive knot is already tied in a separate step in the metatheory; the ADT modularization transformation just puts that in code as well.

The transformation generalizes easily from this example. We give a full formal definition in Appendix A, and implement it in our `comptrans` tool.

```
data Arith = Add Atom Atom

data Atom  = Var String
           | Const Lit
           | Parens Arith

data Lit = Lit Int
```

```
data ArithL
data AtomL
data LitL

data Arith t l where
  Add :: t AtomL → t AtomL
                 → Arith t ArithL
data Atom t l where
  Var    :: String   → Atom t AtomL
  Const  :: t LitL   → Atom t AtomL
  Parens :: t ArithL → Atom t AtomL
data Lit (t :: * → *) l where
  Lit :: Int → Lit t LitL
```

Figure 2-11: Example input (left) and output (right) of `comptrans`.

```
data (:+:) f g t l = Inl (f t l) | Inr (g t l)
data Term f l = Term (f (Term f)) l
type LangSig = Arith :+: Atom :+: Lit
type LangTerm = Term LangSig
```

Figure 2-12: Combining the fragments of Figure 2-11

The modularized representation has several other benefits, even when writing transformations for only one language. For instance, a sort-preserving rewrite that can be applied to to terms of any sort can be given type `Term Sig l` → `Term Sig l`, and it also enables many generic-programming techniques. See Bahr and Hvitved [9] for a full discussion.

2.4 Implementation

We have implemented our approach in the Cubix system, named for a fictional robot composed of modular parts that can be reassembled for many purposes. Cubix is organized as a collection of libraries which assist in building incremental parametric syntaxes and language-parametric transformations. Our implementation totals approximately 29,000 lines of Haskell, providing support for five languages and several transformations. We build heavily on our own fork of Bahr and Hvitved's `compdata` library [9] for modular syntax, and extend it with support for sort injections and the `comptrans` library for converting a third-party syntax definition into modular syntax. We provide generic language components, and modules for labeled terms, control-flow graphs, and higher-order tree traversals.

The code is split between approximately 5400 lines in our language implementations, 2300 in our transformations, 3400 in our general multi-language machinery and generic language fragments, 1100 in `comptrans`, 5000 in our fork of `compdata`, 4000 in our tests, 3900 in our other libraries, and the rest in our driver, miscellaneous code, and minor extensions to third-party libraries. Note that our language implementations do contain a lot of code clones, due to the limits of metaprogramming in Haskell. Cubix as presented in the original paper [96] only contained 13,000 lines; the chief edition since then are the tests and the inclusion of the fork of `compdata`.

The rest of this section discusses Cubix in more detail. Section 2.4.1 describes implementing an IPS in Cubix. Section 2.4.2 describes how to implement transformations, and Section 2.4.3 gives example code. Section 2.4.4 discusses how to generalize Cubix beyond Haskell and the 5 target languages.

Cubix's treatment of control-flow graphs is sufficiently interesting that it merits its own section, Section 2.7.1

2.4.1 Languages

As shown in Figure 2-3, to add support for a language, the users selects a third-party frontend, and then constructs two derived representations.

Creating the modularized syntax There are three steps to creating the modularized syntax. Because the modularized syntax is identical to the original, but in a different form, the user only need enter a few boilerplate commands in Template Haskell, and then this representation is generated automatically.

```
data MultiVarDeclAttrsL
data VarInitL
data MultiVarDeclL

data OptVarInitL
data VarDeclAttrsL
data VarDeclL
data AssignOpL
data AssignL
data LhsL
data RhsL

data OptVarInit t l where
  JustVarInit :: t VarInitL → OptVarInit t OptLocalVarInitL
  NoVarInit   :: OptVarInit t OptVarInitL

data VarDecl t l where
  VarDecl :: t VarDeclAttrsL → t VarDeclBinderL
          → t OptVarInitL → VarDecl t VarDeclL

data MultiVarDecl t l where
  MultiVarDecl :: t MultiVarDeclAttrsL →  t [VarDeclL]
               → MultiVarDecl t MultiVarDeclL

data Assign t l where
  Assign :: t LhsL → t AssignOpL → t RhsL → Assign t AssignL
```

Figure 2-13: Generic nodes to model vardecls and assignments

```
do let cSortInjections = [''CExprIsRhs, ''AssignIsCExpr, ...]
   let names = (cSigNames \\ [mkName "Ident", ...])
            ++ cSortInjections ++ [''VarDecl, ''P . Ident, ''Assign, ...]
   runCompTrans (makeSumType "MCSig" names)
```

Figure 2-14: Generating the incremental parametric syntax

First, for each algebraic data type in the original AST, the user must create a language fragment signature similar to the one in Figure 2-11. `comptrans` generates this code automatically using Haskell's compile-time code-generation engine, Template Haskell [155]. For instance, the command to do this for C is

```
runCompTrans (deriveMultiComp ''CTranslationUnit)
```

as `CTranslationUnit` is the root of the C type.

Second, the user sums these language fragments into a signature for the language. For C, the command is `runCompTrans (makeSumType "CSig" cSigNames)`. After this command has been executed, the user may now manually declare types in terms of `CSig`, such as the type of C terms `type CTerm = Term CSig`, and the signature of labeled C terms `type CSigLab = CSig :&: Label`.

Finally, another command is used to generate the translations between the representation of `language—c` and the modularized representation.

Designing a Library of Generic Components Designing a generic language component takes serious thought: it must be possible to instantiate it in a way that models the corresponding construct in every language under consideration.

These come in the form of (completely-standalone) manually-written signatures. Figure 2-13 shows Cubix's definitions for generic variable declarations and assignments, which we designed to model the corresponding language constructs in C, Java, JavaScript, Lua, and Python. The empty data declarations like `data LhsL` denote sorts, while the others are generic nodes. This component comes with many knobs. By providing C-specific nodes of sort `VarDeclAttrsL` and `MultiVarDeclAttrsL`, it can model declarations like `const int x = 1, *y;`. By providing empty nodes of those sorts, it can model Lua and JavaScript variable declarations.

Creating the IPS The user must now decide how to instantiate the generic components in Figure 2-13 to model their language-specific counterparts. For instance, in C, assignments are expressions, and expressions are assignment right-hand sides.

The user specifies this by generating a sort injection from `AssignL` to `CExprL` and from `CExprL` to `RhsL`, and does similar for `LhsL` and `AssignOpL`.

The user is now ready to define the IPS for the language. This is done by expressing the old signature as a compile-time list of symbols, and literallly removing the language-specific components and adding the generic ones. Figure 2-14 gives the code used to generate the IPS C signature, `MCSig`.

Finally, the user must write a translation from the modularized syntax to the IPS. They need only write code for the cases where the syntaxes differ, i.e.: to replace language-specific nodes with generic ones.

This completes the process depicted in Figure 2-3.

Other support Some transformations may require other language-infrastructure, such as a control-flow graph generator. The IPS representation makes it easy to share code across languages; our 5 CFG-generators, described in more detail in Section 2.7.1, average 122 LOC.

In our experience, creating an incremental parametric syntax for a new language takes 1-2 days. We have implemented support for C, Java, JavaScript, Lua, and Python, using the parsers, pretty-printers, and syntax definitions from the Haskell libraries `language-c` [77], `language-java` [21], `language-javascript` [188], `language-lua` [129], and lastly `language-python` [138]. Because of problems with the parser included in `language-java`, we instead use a Java parser written in Java, the `javaparser.org` parser [166], and translate its results into the `language-java` AST. Despite their names, these libraries were all implemented independently by different authors, and share no common infrastructure beyond standard libraries. We fixed bugs in all of their pretty printers but were otherwise not involved with their development. Some of our fixes have yet to be merged upstream.

2.4.2 Transformation Support

A language-parametric transformation makes limited assumptions about its target language. This is done by parameterizing the transformation over operations on the

```
type HasSyntax f =
  (VarDecl ≺ f, MultiVarDecl ≺ f
  , OptVarInit ≺ f, Ident ≺ f, Assign ≺ f, AssignOpEquals ≺ f
  , Block ≺f, ListF ≺ f, ExtractF [] (Term f))

type CanHoist f =
  (HasSyntax f, VarInitToRhs (Term f)
  , VarDeclBinderToLhs (Term f), HTraversable f
  , InjF f MultiVarDeclL BlockItemL, InjF f AssignL BlockItemL)
```

Figure 2-15: Constraints for the elementary hoist transform

nodes and terms of the language, given in the form of Haskell typeclasses.

The constraints for the elementary hoisting transformation, `CanHoist` in Figure 2-15, depicts the full spectrum of such operations. The elementary hoisting transformation can run on any language that satisfies these constraints, and gives a compile error on any that do not. First, there are constraints that the language contain generic nodes. This is given as the ≺ constraint from `compdata`, which provides an injective function from the generic node to terms of the language, `inject`, and its partial inverse, `project`. As a second class, the `InjF` constraints are sort injections as discussed in Section 2.3.3. Finally, `VarDeclBinderToLhs` and `VarInitToRhs` provide the language-specific operations of elementary hoisting, discussed in Section 2.2.1. Overall, these constraints allow a transformation to make a limited set of assumptions about its target languages, allowing it to handle the intricate details of many languages while maintaining a high level of generality.

There are also a couple more technical constraints. The interface `HTraversable` from `compdata` interface offers generic tree-traversal operations. `MaybeF` and `ListF` provide tree nodes representing optional nodes and lists of nodes, so a node representing a list of block items may have sort `[BlockItemL]`. There are then operations `extractF` and `insertF` to convert between values of type `Term f [l]` (term of sort "list of `l`") and values of type `[Term f l]` (list of terms of sort `l`).

We have built a library of *strategy combinators* [101] called `compstrat`. With strategy combinators, the user can turn a set of single-node rewrites into a complicated traversal pattern in a single line of code. `compstrat` provides similar functionality to other strategy combinator libraries such as Scrap Your Boilerplate [99], Strafunski [102], and KURE [65].

We have also built miscellaneous other infrastructure to support our transformations. The most interesting of these is the control-flow based inserter, explained fully in Section 2.7.4. In brief: inserting a statement before a loop condition causes it to be placed before the loop, before the end of the loop, and before every `continuet` statement.

2.4.3 Example: Implementing the Elementary Hoisting Transformation

This section presents and explains the full implementation of the elementary hoisting transformation first described in Section 2.2.1. Figure 2-16 gives the code; Figure 2-15 showed the `CanHoist` constraint. We omit the 30 lines of code giving the three language-specific instances of `VarInitToRhs` and `VarDeclBinderToLhs`.

The code implements the algorithm described in Section 2.2.1. Execution begins at `elementaryHoist`, which runs `hoistBlockItems` over every block. It does so by using `compdata`'s `transform` function to run `inner` over every node, which uses `project` to test if a node is a block. Later, the sort injections are used via `projF` and `injF` to operate on the subset of `BlockItem`'s that the transformation knows about.

In this example, we have tried to avoid many of the vagaries of Haskell syntax as well as more advanced features of Cubix. Nonetheless, some advanced features are present. The sum-of-signatures approach distinguishes between nodes, which may lie in an arbitrary AST, and terms, which are tied to a single language. The vanilla data constructors of Figure 2-13 like `Assign` construct nodes of a signature fragment, while their primed variants like `Assign'` construct and pattern match on terms. We explained the `extractF` and `insertF` functions in Section 2.4.2; these are used to implement the `liftF` and `mapF` functions, which are used to operate on trees of type `Term f [l]`. Finally,

```
declToAssign :: (CanHoist f)
          ⇒ Term f MultiVarDeclAttrsL → Term f VarDeclL → [Term f BlockItemL]
declToAssign mattrs (VarDecl′ lattrs b optInit) = case optInit of
  NoVarInit′        → []
  JustVarInit′ init → [injF (Assign′ (varDeclBinderToLhs b) AssignOpEquals′
                                      (varInitToRhs mattrs b lattrs init))]

removeInit :: (CanHoist f) ⇒ Term f VarDeclL → Term f VarDeclL
removeInit (VarDecl′ a n _) = VarDecl′ a n NoVarInit′

splitDecl :: (CanHoist f)
          ⇒ Term f BlockItemL → ([Term f BlockItemL],[Term f BlockItemL])
splitDecl (projF → (Just (MultiVarDecl′ attrs decls)))
          = ([injF (MultiVarDecl′ attrs (mapF removeInit decls))]
            , concat (map (declToAssign attrs) (extractF decls)))
splitDecl t = ([], [t])

hoistBlockItems :: (CanHoist f) ⇒ [Term f BlockItemL] → [Term f BlockItemL]
hoistBlockItems bs = concat decls ++ concat stmts
  where (decls, stmts) = unzip (map splitDecl bs)

elementaryHoist :: (CanHoist f) ⇒ Term f l → Term f l
elementaryHoist t = transform inner t
  where
    inner :: (CanHoist f) ⇒ Term f l → Term f l
    inner (project → (Just (Block bs e))) = Block′ (liftF hoistBlockItems bs) e
    inner t                                = t
```

Figure 2-16: Implementation of the elementary hoist transformation. This figure is a repeat of Figure 2-2.

Table 2.1: Various term types in CUBIX

Type signature	Description
`Term f AssignL`	Assignments in any language
`Term MJavaSig l`	Java terms of any sort
`(Assign ≺ f) ⇒ Term f IdentL`	An identifier in any language that contains generic assignments
`Term f (StatSort f)`	A statement in any language. The statement sort is language-specific.
`(InjF f IdentL PositionalArgExpL , CallAnalysis f) ⇒ Term f IdentL`	An identifier in any language which supports a call analysis, and where identifiers may be used as ordinary arguments to functions

the syntax `f (view → Just x) = ...` is a Haskell *view pattern* [175] which is syntactic sugar for `f t = case view t of Just x → ...`, with pattern match failure proceeding to the next case.

2.4.4 Choices of Target and Implementation Languages

When we speak about CUBIX, we always find people who want to use it or something like it for their applications. What would it take to implement a system like CUBIX in a different language? And what about supporting other languages, such as ML or Prolog or Haskell itself?

What do we gain from these fancy types? CUBIX's implementation of incremental parametric syntax uses some rather advanced type system features. Case in point: the current implementation uses over 30 GHC extensions. What's the benefit of all this, and can it be replicated in a language other than Haskell?

There are two primary benefits. The first is the precise typing. The second is dispatch: the compiler uses the type information to choose appropriate language-specific and sort-specific operations. Both are indispensible for building multi-language tools. And their synthesis allows generic programming.

Consider the example types in Table 2.1. They show how, using the `Term f l` type, developers can restrict operations to certain sorts of terms, languages, and properties of the language. These restrictions are all enforced by the compiler.

Without these types, it's still quite easy to write a function that can accept many kinds of terms, by giving them all a single "Node" type. This is the dynamically-typed or "generic node" approach, used in many language workbenches. This is not enough to get language-parametric transformation, as removing the types does not remove the network of constraints between AST nodes. There must be the extra step of converting part of the tree to some common form, as done in IPS. And these conversions introduce massive room for errors.

Our own experience attempting this kind of generic programing in JavaScript, as well as fixing type errors during normal Cubix development, makes us pessimistic about trying it without precise types. It's far too easy to e.g.: attempt to use an assignment as an expression, when that is not legal in every language.

The second major benefit is dispatch. Consider writing a transformation that works on any language where functions can be turned into lambdas. If we were to implement this transformation in a language without typeclasses, we would make the transformation take a "turn function into lambda" operation as a parameter. This operation would then need to be transitively supplied to every piece of the transformation that needed it.

Conversely, in Haskell, we'd simply add a condition to the constraints for the transformation, as in Figure 2-15, and it would be propagated to all components. And when attempting to call this transformation on a specific language, the compiler will automatically find and supply the correct instance of the "turn function into lambda" operation.

In other words, it is very easy to write a generic operation that includes language-specific pieces. Doing this at a smaller level is generic programming. For instance, `subterms e :: [Term f IdentL]` gives all identifiers contained in `e`. The equivalent code in a language without typeclasses would be `subterms(e, Filters.checkSort(SORT_IDENT))` or similar. Meanwhile, Haskell automatically supplies `subterms` with the correct sort-classification check by looking at the types. So, this representation is useful even when only working with one language.

IPS in other languages Our implementation of incremental parametric syntax relies heavily on three distinctive features. We discussed the use of type classes above. The `Term f l` type relies on GADTs to work (else `Term f IdentL` could not be fundamentally different from `Term f AssignL`). And we've used Haskell's built-in code generation, Template Haskell, throughout this chapter. Haskell is the only language we know of that supports all three features. But even Haskell is not a perfect language for building this kind of system, and we still have a wishlist of language features that would make Cubix development much easier (e.g.: pattern matching that works better with modular syntax).

We can envision a Cubix-like framework in a language without any of these three features. It would use an extra-linguistic tool to generate boilerplate code. Users would pass in operations manually in lieu of typeclasses, at some inconvenience. But without GADTs, we see only two options, neither of them appealing: either write a custom type-checker/analyzer, or face the pitfalls of dynamically-typed terms.

Supporting more languages To share code between languages, these languages must have nodes which are similar enough to design a generic node whose semantics model all of them. Expressions are similar in C and ML, and we see no barrier writing transformations that can operate on them both. On the other hand, the execution semantics of Prolog nodes differs substantially from imperative and functional languages, and so we do not expect to be able to write C/Prolog transformations.

Integrating Cubix with a third-party parser only requires that the parser output to a Haskell ADT. We hence picked languages that already had good Haskell libraries, but it can integrate with parsers written in other languages by writing a wrapper, as we have done for Java.

Beyond Haskell Though we previously argued that Haskell uniquely provides features important to Cubix, even Haskell has limitations which prevent us from extending Cubix in desired ways. Such limitations include:

- **Negation**: There is no way to write a constraint of the form "for any language that *does not* contain the generic PointerArithmetic" fragment, owing to Haskell's open-world interpretation of typeclass instances.

- **Slowness of type-level programming**: It would be useful to define separate languages for each version of Java, or for core C and the GCC dialect, which adds new features like nested function definitions. Similarly, we would like to support hybrid languages such as Cython (C/Python hybrid) and JSX (JavaScript/XML hybrid). (Indeed, several open-source authors we contacted when attempting to find potential case studies for YOGO asked about Cython support.) All of these could be supported using **language subset constraints**. In this scheme, we first define a signature for the core language, and then extend it with additional fragments, e.g.: `MCSig :+: GCCExts` for GCC C, or `MCJava :+: Java7NewSyntax :+: Java8NewSyntax` for Java 8. Language-specific operations could then be defined for all dialects by e.g.: defining a typeclass instance for all signatures which contain all 80 or so fragments in C (written `MCSig :<f`). However, although Haskell is perfectly capable of expressing this, we have been unable to actually run programs using this approach because the compiler chokes when given such constraints.

- **Pattern matching**: If one writes a function in CUBIX pattern-matching over terms of a fixed sort and language, GHC cannot detect whether this match is exhaustive or missed a case. This was later improved: in 2018, we filed a bug report about GHC's exhaustiveness checking, which was credited (in personal correspondence and in one of their talks) as the inspiration for an entire paper [69]. However, this only benefits functions written without view patterns, as used by the `splitDecl` and `inner` functions in Figure 2-16. We would further like to be able to write these same functions without the clunky view-pattern syntax.

2.5 Evaluation

In previous sections, we argued that the insights of Cubix make language-parametric transformations easy to write. In this section, we demonstrate a realistic language-parametric transformation and its application to real software, and further evaluate the following two claims:

- *Readability*: These transformations produce readable output, similar to what a human would write. They do not needlessly destroy the program's structure, as do IR-based transformations.
- *Correctness*: Despite the low effort needed per language, transformations can maintain correctness even when faced with the intricacies of multiple languages.

Additionally, because we built the tool of Section 2.5.1 after the rest of the work in this section, our experience building also supports our claim that it is easy to extend an IPS as more features are needed to support new transformations.

2.5.1 A Realistic Whole-Program Refactoring

In this section, we present the IPT tool (**i**nterprocedural **p**lumbing **t**ransformation) for threading variables through chains of function calls, inspired by the Dropbox and Facebook stories in Section 2.1. We built the IPT tool as a language-parametric transformation which we developed simultaneously for all 5 languages supported by Cubix.

The IPT tool takes a method and a parameter name, and recursively has all callers pass down said parameter, asking for user approval for each change. Figure 2-17 presents a scenario where the end goal is to replace the call to `strcpy` within `f1` with `strncpy`, and the barrier is that the programmer is missing the `len` parameter needed by `strncpy`. He invokes the IPT tool to add an `int len` parameter to `f1`, pressing "Yes" 4 times to change lines 1, 5, 4, and 9, so that the existing `len` parameter in `f3` is passed through 2 layers of function calls to `f1`. After the IPT tool is done, the programmer can now manually change line 2 to `strncpy(buf, "Hello", len);`. The IPT tool has

```c
void f1(char* buf) {
  strcpy(buf, "Hello");
}

void f2(char* buf) {
  f1(buf);
}

void f3(int len) {
  char *buf = malloc(len);
  f2(buf);
}
```

```c
void f1(char* buf, int len) {
  strcpy(buf, "Hello");
}

void f2(char* buf, int len) {
  f1(buf, len);
}

void f3(int len) {
  char *buf = malloc(len);
  f2(buf, len);
}
```

Figure 2-17: Input/output example of the IPT tool on C

automated plumbing data through the system; all that is left for the programmer is to choose where it comes from, and how it's used.

Building this tool also shows that it is easy to extend an IPS to support new transformations. We had not needed a generic notion of functions for our previous transformations we implemented (see Section 2.5.2), so we made an incremental change to our parametric syntax. In 3 hours, we designed a generic fragment for function definitions and calls that could be instantiated to model the features of all languages under consideration. It took us 21 hours to design and implement the changes to all 5 language representations, proportional to the complexity of each language (e.g.: 5 hours to understand C declarations and their many variations). We thus obtained the incrementality benefits of IPS: we didn't need to build up-front support for functions, but could still build transformations that needed them, and we now have support for functions for all future transformations.

With the generic syntax for functions in place, building the tool itself took only 19 hours. Altogether, the extensions to Cubix averaged 5 hours per language, while the tool itself averaged another 4 hours per language.

The implementation is fairly straightforward: it maintains two queues of function calls and definitions to be modified, and prompts the user about each potential change.

After modifying each function definition, it uses a static analysis to find all callers and add them to the queue. This static analysis is a parameter of each language, but the analyses for each language may use a shared implementation using techniques of multi-language analysis. Making this analysis more precise means the user will be prompted for fewer erroneous changes.

Our IPT tool is still a prototype, with a minimal command-line UI, an imprecise call analysis, and incomplete support for C function prototypes. Nonetheless, we have used it in three real case studies in Java and Python, in addition to toy programs in the other three languages.

We first used it on SimpleDB [113], a teaching database used at several universities. SimpleDB totals 23,000 lines of Java, and 11,500 lines of tests. It frequently accesses a global `BufferPool` object by calling `Database.getBufferPool()`. We used a two-step process to eliminate this global. First, we used the IPT tool to thread a `bufferPool` parameter throughout the program. This changed all `Database.getBufferPool()` calls to instead read `Database.getBufferPool(bufferPool)`. Second, we applied a find-and-replace to the entire program to simplify them to `bufferPool`. We then manually changed entry points to the program to supply this `bufferPool` parameter. Altogether, the IPT tool modified 484 lines across 41 files, while we manually modified 50 lines across 24 files. All tests pass.

We then did two smaller case studies in Python. Flask [71] is a Python web micro-framework which totals 6500 lines of Python and 5700 lines of tests. We used the IPT tool to modify the `_get_exc_class_and_code` function to take a default exception code, and propagated this parameter up several layers. The IPT tool modified 21 lines across 2 files. We only manually changed 2 lines: to use this parameter, and to supply it at the top of the chain. Tornado [52] is a Python web server owned by Facebook. It comprises 22,000 lines of Python and 16,000 lines of tests. We changed the `is_valid_ip` function to take an `accept_ipv6` parameter, and propagated this parameter up several layers. The IPT tool changed 22 lines across 9 files. We used a find-and-replace to provide a default value to many new parameters, and then changed 3 lines manually. All tests pass for both Flask and Tornado.

For all five languages, we also tested the IPT tool on a toy program consisting of three functions across three files that call each other; the tool successfully propagated a parameter through all three functions.

2.5.2 Benchmark Transformations

To more rigorously evaluate our system, we have implemented three smaller source-to-source transformations. These were chosen to explore the space of operations used by program transformations and to require a minimum of user input. Table 2.2 lists them and their line counts.

- The hoisting transformation **Hoist**, which lifts variable declarations to the top of their scope. This is similar to elementary hoisting in Sections 2.2.1 and 2.4.3, except that it also supports Lua, and uses additional machinery to avoid hoisting shadowed variables, and to deal with language special-cases such as C's structure initializers. Figure 2-1 gave a mundane example; Figure 2-18 gives an example handling a JavaScript special case. This transformation supports all languages except Python, which lacks variable declarations.

- The test-coverage instrumentation transformation **Testcov**, which prefixes each basic block in the source code with an assignment which marks that that block has executed. This produces data that could be fed into a test coverage tool. It was inspired a Semantic Designs tool which implements this transform separately for a dozen languages [151]. This transformation supports all languages. Figure 2-19 shows an example special case for Java.

- The three-address code transformation **TAC** hoists all nested computations into temporary variables, e.g.: changing `1+1+1` into `t=1+1; t+1`. This is a deceptively complicated transformation, difficult to write at the source level for even one language. Figures 2-20 and 2-21 show a few of the complexities it supports, all handled cleanly by CUBIX's general infrastructure for operator strictness and CFG-based insertion. This transformation supports JS, Lua,

```
function f() {
  "use strict ";
  if (x) {
        var y =1;
  }
}
```

```
function f() {
  "use strict ";
  var y;
  if (x) {
    y = 1;
  }
}
```

Figure 2-18: Hoisting JavaScript, showing interactions with JS's `"use strict ";` pragmas and lack of inner scopes. No JS-specific hoisting code is needed, only a precise representation of JS blocks.

Table 2.2: Transformations implemented and their size. Line counts are split into the core code of the transformation, plus the per-language code to support language-specific operations and customization. Line counts exclude the file prologue, i.e.: they count from the first line of code which is not an `import` statement.

Transformation	**Languages Supported**	**Core LOC**	**Extra LOC per language**
Hoist	C, Java, JavaScript, Lua	154	65
Testcov	All	77	25
TAC	JavaScript, Lua, Python	360	116

and Python. It does not support Java or C because declaring the temporary variables would require type inference, which in turn requires symbol-table construction, a heavyweight piece of language infrastructure.

All transformations use a mixture of generic and language-specific code. However, the language-specific code is usually much less complex, and fewer lines are needed per-language, as shown in Table 2.2.

2.5.3 Correctness

We claim it's feasible to write semantics-preserving language-parametric transformations with our approach. Hence, we collected language test suites for each of the 5 languages, and improved our transformations until we had a 100% pass rate for all transformations on all languages.

```java
1 public void foo(int x) {
2   if (x > 0) {
3     while(true)
4       x++;
5     // Unreachable code
6   }
7 }
```

```java
1 public void foo(int x) {
2   TestCoverage.coverage[0] = true;
3   if (x > 0) {
4     TestCoverage.coverage[1] = true;
5     while (true)
6       x++;
7   }
8   TestCoverage.coverage[2] = true;
9 }
```

Figure 2-19: Test coverage for Java. A naive transformation would insert a test coverage statement on line 5 after the while loop, causing an "unreachable code" compile error. This case is supported purely through the CFG-generator (by emitting a CFG in which control cannot transfer to code after the loop'), requiring no Java-specific code in the transformation itself.

```javascript
while (f() && g(1+1)) {
    x++;
}
```

```javascript
var t1 = f();
var t2;
if (t1) {
  var t3 = 1 + 1;
  t2 = g(t3);
}
var t4 = t1 && t2;
while (t4) {
  x++;
  t1 = f();
  if (t1) {
    var t3 = 1 + 1;
    t2 = g(t3);
  }
  t4 = t1 && t2;
}
```

Figure 2-20: TAC transformation example for JavaScript, showing handling of loops and non-strict operators.

```
if x is None:
  doThing1()
elif x.foo():
  doThing2()
```

```
t1 = x is None
if t1:
    del t1
    doThing1()
else:
    del t1
    t2 = x.foo()
    if t2:
        del t2
        doThing2()
    else:
        del t2
```

Figure 2-21: TAC transformation example for Python. It avoids computing `x.foo()` when `x` is `None`, and deletes all temporaries immediately after use, as Python is sensitive to the GC behavior. Adding the `del` statements is 5 lines of Python-specific code.

The caveat, though, is that there are some tests which the transformations should not pass. First, we use third-party parsers and pretty-printers, all of which have bugs. We contributed some bug fixes to all of these projects, but issues still remain. Second, all of the dynamic languages have self-referential tests which will never pass (e.g.: "assert this function was declared on line 37"). We rule out these cases by first checking if the test still passes after running the identity transformation **Ident**, consisting of parsing and pretty-printing the program. 93.4% of tests pass this **Ident** transformation. This discussion excludes the Lua test suite, which has other issues explained below.

Table 2.3 lists the language implementations and test suites used in our evaluation. The C, Lua, and Python tests come from their implementations, while the JavaScript ones come from the official specification conformance test suite. The authors of K-Java, [20], report that no Java language tests are publicly available, and hence created their own specification tests, which we use. We restricted ourselves to the core language tests of `test262`, using the same subset as the JavaScript semantics KJS [132], and omitted a small handful of multi-file Java tests among the Java ones, which caused problems with our test harness. We used the entirety of the Lua, Python, and

Table 2.3: Compilers/interpreters and test suites used in evaluation

Language	**Compiler/Interpreter**	**Test Suite**	**Test Files**	**Test LOC**
C	GCC 6.3.0_1	gcc-torture	1394	53,637
Java	JDK 1.8.0_65	K-Java	755	26,568
JS	Node.js v0.10.24	test262	2782	128,698
Lua	Lua.org 5.3.3	Lua Tests	28	12,017
Python	CPython 3.7.0a0	CPython Tests	404	249,499

C test suites.

Table 2.4 shows the number of passing tests for each language and transformation. The **Ident** transformation is a baseline transformation which simply parses and pretty prints a program, in order to filter out “bad tests” as described above. The **Hoist**, **Testcov**, and **TAC** columns show the results of their respective transformations. Comparing the other transformations to **Ident**, all but 12 tests pass. The failing JavaScript and Python tests are all self-referential tests that were not ruled out by **Ident**. The failing JavaScript tests all use `function.toString()`, which retrieves the textual source code of the function. The Python ones inspect the runtime representation of functions, such as the presence of opcodes in the compiled bytecode or the number of bytes used per stack frame. The failing Java hoist test is actually due to a crash of `javac`. Manual inspection shows that this program is indeed correct, and the bug has been confirmed and fixed by the JDK developers [82].

While we were very successful with the other language test suites, we found substantial barriers using the Lua test suite to test our transformations. Its tests are highly self-referential, including a check that the “test” function is defined on line 17, points where it undefines every global variable, and even a test file that reads its own source as input and looks at certain offsets, thereby breaking if the file changes character encoding. We nonetheless tried.

As the Lua tests are distributed as a single program, we modified the Lua test suite to maintain a count of passed assertions, instead of stopping at the first failure, and deleted some of the overly self-referential assertions. We found that the total number of calls to `assert` was nondeterministic, but the number of failing assertions was not. In

Table 2.4: Results of each transformation on the test suites

Lang	**Total**	**Ident**	**Hoist**	**Testcov**	**TAC**
C	1394	1305	1305	1305	N/A
Java	755	745	*744	745	N/A
JS	2782	2573	2573	2568	2572
Python	404	360	N/A	358	357
Lua	Reported separately				
* Not including test which crashed `javac`					

one set of runs, we obtained the following numbers: 70440/70456 passing assertions for the original, 70279/70295 for the identity transformation, and 70463/70479 for hoisting. We gave up attempting to get it working for the test coverage transform, due to crashes related to its metaprogramming around global variables. We similarly gave up for the TAC transformation, because the Lua VM does not allow for more than 200 local variables in any scope, and the TAC transformation overwhelms this easily. We conclude that the Lua test suite is unsuitable for testing program transformations.

2.6 Readability Study

We ran a study to evaluate the readability of our transformations' output. The overall setup of our experiment is like a Turing test. First, we ask a set of human contributors to transform programs by hand. We then give a separate set of human judges these programs, alongside the corresponding automatically transformed programs, and ask them to rate them both on correctness and quality. Because low-level code formatting is outside the scope of our claims, we automatically reformat the human-written code before presenting them for comparison. Outside of formatting, we attempted to bias the experiment in favor of the humans, allowing them to resubmit until their transformed programs were correct according to our extremely thorough test suites. Despite this, in our final results, the judges gave the automatically transformed programs a higher average rating.

Our experiment proceeds in three phases. In the first phase, we construct the RWUS suite, providing suitable programs on which to run the study. In the second phase, we ask human participants to manually apply each of the three studied transformations on a code sample. In the final phase, human judges from Mechanical Turk rate the manually-transformed code against the same code transformed by our system. Note that this study was completed using earlier versions of the transformations which failed some tests.

2.6.1 Phase 1: Constructing the RWUS Suite

As objects in our study, we needed (1) representative samples of real-world code, and (2) an objective measure of whether the code was transformed correctly. The second criterion is the main difficulty, as random samples of code typically do not come with thorough tests, and certainly not tests that are easy to run. Hence, we created our own.

The RWUS (Real World, Unchanged Semantics) suite consists of 50 functions across 5 languages randomly selected from top GitHub projects. For each, it also includes a test suite designed with the intention that only functions semantically equivalent to that function will pass. Each function is distributed as an *entry*. An entry is a file containing the original sample, mocks for all referenced symbols, tests, and a wrapper `main` procedure which invokes the tests. The files can all be compiled and executed without any dependencies. The tests are used by invoking a script that replaces the sample with a transformed version, and then executes the resulting file.

We selected the functions for the RWUS suite as follows: For each of C, Java, JavaScript, Lua, and Python, we downloaded the top 20 projects in that language on GitHub from those with at least 500 lines, sorted by number of users who "starred" that project. We then uniformly at random selected a line of code from the projects. If this line of code lies within a function, we took the innermost such function as a sample; else, we repeated the process. We discarded all samples which were not between 5 and 50 lines of code, excluding function signatures, blank lines, and comments. We repeated this process until we had 10 samples for each language. One shortcoming of this approach is that the top-rated projects on GitHub vary in size by orders of magnitude. As the extreme, 90% of our C corpus and all 10 C samples come from Linux. The other 40 samples come from 24 different projects.

For each sample, we constructed test cases ensuring full path coverage, and added checks to ensure all mocked functions are called in the expected order with the expected arguments. The resulting tests are incredibly thorough. While the actual samples total 1158 lines of code, the RWUS suite totals 8070 lines of code.

The RWUS suite is available from:

`https://github.com/cubix-framework/rwus`

2.6.2 Phase 2: Obtaining Human-Written Transformations

We recruited programmers through department mailing lists, flyers posted around the department, and social media. Due to the relative scarcity of Lua programmers, we also posted on Lua forums, and asked Lua participants to spread the study by word of mouth.

Participants were sent to a website, where they would download a single sample from the RWUS suite along with its tests, and were asked to perform each of our transformations by hand on the file. They were allowed to contribute one sample per language, and were offered a $10 Amazon gift card for each.

We inspected each submission by hand. Participants were asked to resubmit until their transformed samples passed all tests, and had no significant transformation errors, such as unhoisted variables.

2.6.3 Preparing the Samples

After we had collected all 50 human-transformed samples, we ran them through the corresponding parser and pretty printer to normalize formatting. We then ran our transformations on each of the RWUS samples, and evaluated them with the RWUS test suites.

We did not run any transformation on the RWUS samples until all development on the transformations had ceased. We also attempted to avoid allowing knowledge

Table 2.5: Counts of programs where presentation to the human judges was inappropriate

	C	Java	JS	Lua	Python
Identical	6	9	1	5	3
Failed	0	1	4	0	1

of the samples in the RWUS suite to influence development of the transformations, although a single researcher was responsible for both.

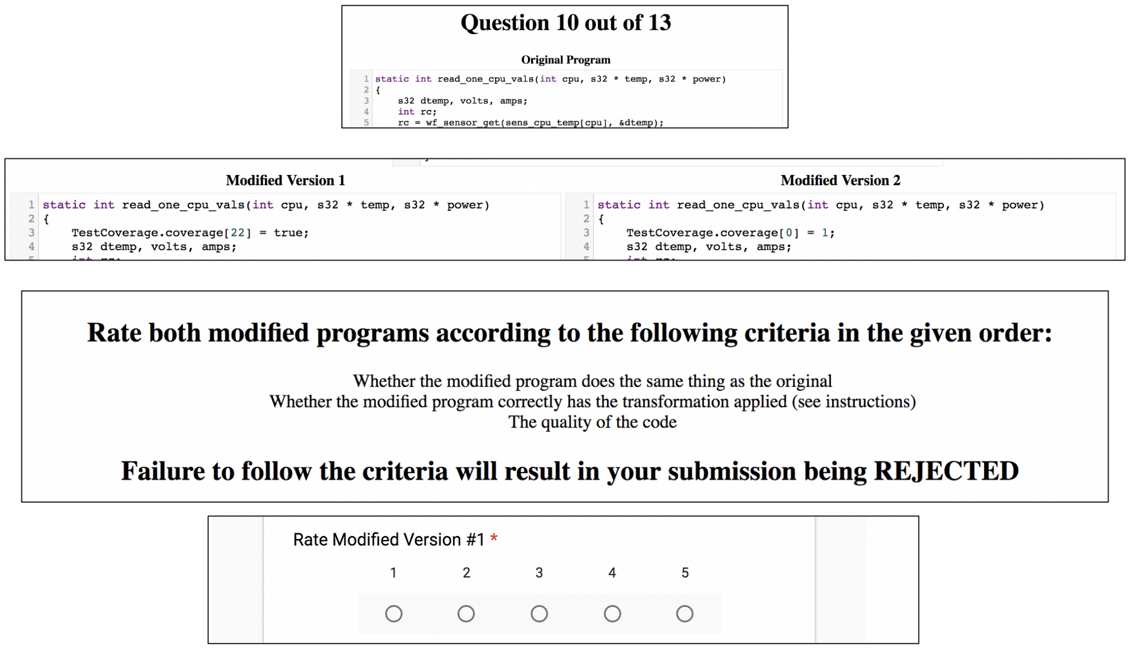
Of the 120 transformed pairs, for 24 of them, the automatically transformed version was identical to the human written one after reformatting. These are broken down per language in Table 2.5. Because this study was done using older versions of the transformations, before their pass rate had been perfected, six[1] automatically transformed samples either failed their test suites or caused an error in the transformed program. Also, for one sample, a pretty-printer bug caused both the human-transformed and automatically transformed versions to fail to compile. The remaining 89 pairs were sent to human judges for evaluation in Phase 3.

2.6.4 Phase 3: Comparing Human and Machine-Written Transformations

In Phase 3, we asked human judges from Mechanical Turk to rate the manually-transformed code from Phase 2 along with their automatically transformed counterparts.

We created one task on Mechanical Turk for each language/transformation combination. For each judge entering our website interface, we began by presenting an explanation and example of the transformation, before presenting the questions. Each question shows a sample program, along with the automatically transformed version produced by our system, and the manually-transformed version collected in Phase 1. They were asked to rate both on a 1–5 scale. We instructed that they should first

[1]By chance, samples that trigger bugs and unsupported cases were overrepresented in the samples, and it would bias the study to modify the transformations in response to the randomly-chosen samples.



Question 10 out of 13

Original Program

```
static int read_one_cpu_vals(int cpu, s32 * temp, s32 * power)
{
    s32 dtemp, volts, amps;
    int rc;
    rc = wf_sensor_get(sens_cpu_temp[cpu], &dtemp);
```

Modified Version 1

```
static int read_one_cpu_vals(int cpu, s32 * temp, s32 * power)
{
    TestCoverage.coverage[22] = true;
    s32 dtemp, volts, amps;
```

Modified Version 2

```
static int read_one_cpu_vals(int cpu, s32 * temp, s32 * power)
{
    TestCoverage.coverage[0] = 1;
    s32 dtemp, volts, amps;
```

Rate both modified programs according to the following criteria in the given order:

Whether the modified program does the same thing as the original
Whether the modified program correctly has the transformation applied (see instructions)
The quality of the code

Failure to follow the criteria will result in your submission being REJECTED

Rate Modified Version #1 *

1	2	3	4	5
○	○	○	○	○

Figure 2-22: Main parts of GUI shown to human judges in the readability study.

rate the transformed programs on correctness vs. the original program, second on faithfulness to the intended transformation, and only third on general prettiness and code quality. Both the order of questions and the order of the transformed pairs were randomized. We assigned each of the 20-30 transformed samples to 10 judges, giving us up to 300 ratings per language. Figure 2-22 shows the key parts of the website interface shown to judges.

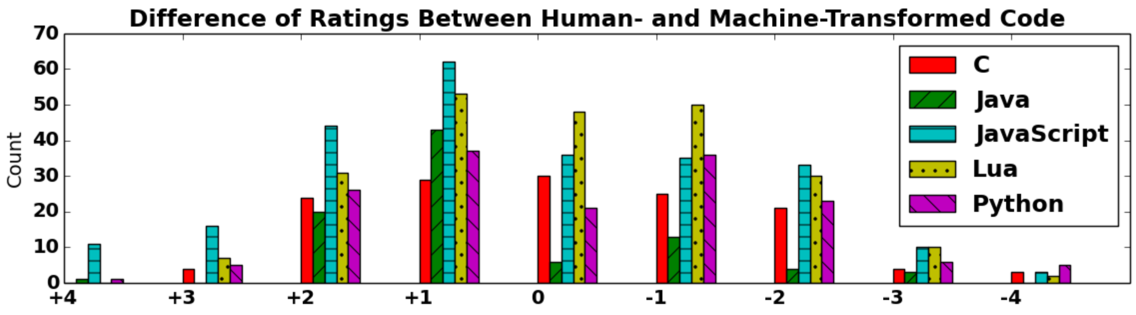


Figure 2-23: Counts of differences between the ratings of the machine transformations and the human transformations. The leftmost bars represent cases where the judge rated the machine-produced output higher than the human-produced.

2.6.5 Quality Control

The setup described above does not preclude someone from rating programs randomly, so we employed two quality-control mechanisms. Our primary form of quality control was the creation of "canary" questions. Canaries appear as normal questions, except that the programs contained therein were contrived. In two of the canaries, one of the programs was clearly not a transformed version of the original. In the third canary, both displayed programs were identical. We rejected any submission in which the worker did not rate the correct program higher for the first two canaries, or did not rate both programs of the third canary the same. Second, if a worker ever submitted two answers within 11 seconds of each other, we marked this worker as untrustworthy, and rejected all submissions by him. We picked this value after observing the times spent on each question in dry runs of the study.

We noticed substantial differences between workers who did and did not pass the quality controls. Workers with one rejected submission typically had rejected submissions for many different languages. Workers with accepted submissions were much more likely to only submit for one language. Workers typically either had all their submissions accepted or all rejected. Furthermore, we noticed that rejected submissions were typically completed in much less time than accepted ones, although many workers who failed the canaries were substantially slower than the fastest correct workers.

The experimenters manually inspected a selection of judgments from accepted submissions, and found them all reasonable. Overall, our observations suggest that our quality control mechanisms did effectively classify workers on skill, and that our data is high-quality.

2.6.6 Results

For each language, we tabulated the difference in ratings between the human-written and automatically transformed programs. Our results are given in Figure 2-23. The average differences in ratings ranged from -0.075 for Python (favoring the humans) to +0.633 for Java (favoring the machine). The differences for C, JavaScript, and Lua were −0.014, +0.396, and −0.052 respectively.

Our goal was to show that the output of our transformations is not less readable than the human-transformed code. This is a problem in statistics known as non-inferiority testing [180]. For each language, we formulated a hypothesis that the average difference in ratings between each the machine- and human-transformed code is at least −1. We then factored in the pairs that were not sent to Phase 3: each identical pair was counted as 10 judgments of equality (difference 0), and each pair where the machine-transformed version was incorrect was counted as 10 judgments that maximally penalize the machine version (difference −4). We tested each of the 5 hypotheses using a paired t-test. For each language, it showed that the machine-transformed code was non-inferior by a non-inferiority margin of at most 1 with $p < 10^{-8}$. In retrospect, this data had the power to prove the hypothesis with a much smaller non-inferiority margin.

Considering both the raw data and the statistical tests, our study provides strong evidence that the output of transformations in Cubix is no less readable than hand-transformed code, showing that implementing source-to-source transformations with incremental parametric syntax avoids the mangling common to IR-based approaches.

2.6.7 Threats to Validity

Our results are potentially biased by using a real-world distribution of programming constructs, as opposed to intentionally constructing a suite filled with corner cases. The humans are hindered by a lack of learning: they only perform each transformation once per language. Finally, we cannot be certain of the quality of the data from Mechanical Turk. In our dry runs, we found that workers on Mechanical Turk tend to rate simple programs more highly, even when the transformation is incorrect. Two of our canaries are specifically designed to prevent this behavior.

2.7 CFG System

2.7.1 The Need for Advanced CFG Manipulation

Languages differ in syntax. Cubix gains much of its multi-language capability from its ability to massage differing syntaxes to share identical components. But another approach is to write a transformation beyond the syntactic level.

In this section, we introduce just a non-syntactic transformation operator offered by Cubix, *control-flow-based insertion*. And we also explain the bedrock under this operations rests: Cubix's CFG-generation machinery, showing how Cubix's generic programming capabilities shine in a non-transformation task, enabling Cubix to generate high quality control-flow graphs with relatively little code, averaging only 122 language-specific lines per CFG generator.

We now illustrate the use of control-flow based insertion with a simple challenge: Some analysis has identified that a string variable `s` is unsanitized at a certain use-site. How would you build a tool that inserts the line `s = sanitize(s);` before that use-site, as near as possible?

This operation seems trivial when one imagines the common case where `s` sits within a one-line statement, and the transformation must merely insert the sanitization on the previous line. Yet complexity appears when one considers that the use of `s` may be awkwardly situated inside some construct with interesting control-flow, and may have multiple predecessors, separated by some distance. A common case is when `s` lies in the condition of a for-loop; then the sanitization must be inserted before the loop, at the end of the loop, and before every `continue` statement. Figure 2-24 gives example input and output for this case in C. Similar problems arise with other control-flow constructs. When use is in the condition of a Python `elif`, for instance, as in Figure 2-25, adding the sanitization will require splitting the `elif` into a nested conditional (Figure 2-25b).

There are many existing techniques for program transformation, such as those surveyed in [172]. To a first approximation, all of them require the programmer to eventually specify tree rewrites, and would hence require giving many cases to work

```
for (int i = 0; i++ && !isStopCode(s); i++) {
  if (!containsCommand(s))) {
    s = nextInput();
    continue;
  }

  process(s);
  s = nextInput(s);
}
```

(a)

```
s = sanitize(s)
for (int i = 0; i++ && !isStopCode(s); i++) {
  if (!containsCommand(s))) {
    s = nextInput();
    s = sanitize(s)
    continue;
  }

  process(s);
  s = nextInput(s);
  s = sanitize(s)
}
```

(b)

Figure 2-24: Sample C input (a) and output (b) for sanitization transformation, targeting `isStopCode(s)`.

```
if notificationReceived():
  handleNotification()
elif isCommand(s):
  process(s)
else:
  log("Skipped")
```

(a)

```
if notificationReceived():
  handleNotification()
else:
  s = sanitize(s)
  if isCommand(s):
    process(s)
  else:
    log("Skipped")
```

(b)

Figure 2-25: Sample Python input (a) and output (b) for sanitization-transformation, targeting `isCommand(s)`.

on these examples. But, CUBIX's multi-language CFG support and its control-flow based insertion operation, the following function performs this transformation in all of CUBIX's 5 languages, correctly handling the special cases above.

```
insertSanitization targetNode var =
  dominatingPrepend targetNode
                    (assign (ident var)
                            (functionCall "sanitize"
                                          [ident var]))
```

Half of the magic of this snippet comes from CUBIX's incremental parametric syntax, allowing the `(assign ...)` expression to expand into one of several language-specific variants depending on the inferred type of each call-site. We now introduce the other half of the magic: the new source-to-source transformation paradigm of *control-flow based insertion*, invoked here through the `dominatingPrepend` function. The idea of CFG-based insertion is to provide a new primitive operation "Insert code A at points ensuring that it always runs before/after code B." Behind this lies a graph search and some language-specific operations for checking where the insertion is possible; these together turn what would be a menagerie of casework into one declarative statement of intention.

Just as CUBIX innovates in using control-flow graph for transformation so as to amortize the cost of building tools across multiple languages, it also innovates in amortizing the cost of CFG infrastructure. In the other half of this section, we present a new monadic decomposition of CFG-generators which allows them to be written in a language-modular fashion. The upshot of this is that we were able to construct complete CFG generators for 5 languages, passing an extensive test suite which includes CFG "challenge problems" such as Duff's device [181], in only 1100 total lines of code — and averaging 122 lines of language-specific code — compared to a combined 2400 lines of code in the best pre-existing comparison CFG-generators we found. We demonstrate this in miniature with a fully-worked tutorial: 19 lines of code for a full CFG-generator on a language with 17 distinct node types.

While incremental parametric syntax attacks the modularity problem caused by syntactic differences, CFG-generation presents its own modularity challenges, caused not only by syntactic differences but also by *control effects*. For example, even when multiple languages have syntactically-identical while-loops, a naive "make CFG for while" function does not work because loops in different languages may interact differently with `break`, `continue`, and `goto` statements within their bodies. Similar factors prevent code reuse between while- and foreach-loops, even though the control-flow structure is the same. We shall also find that control-flow graphs, under their most common designs, require non-local information to construct, demanding recursion patterns which break the separation between language fragments.

We answer with a fully-compositional design for CFG-generators. Our design uses monads to separate handling of control effects, and removes the need for non-local information. Under this design, it becomes quite straightforward to create a single, reusable "make CFG for while" function. Our design further allows several standard cases to be handled declaratively, including nodes with a default left-to-right evaluation order, nodes which do not take part in computation (e.g.: type declarations), and nodes which introduce a new control-flow contour (e.g.: lambdas). We explain these techniques with a tutorial in which we construct full CFG-generator for a small IMP language with 17 distinct node types; including loops, lambdas, and goto; in only 19 lines of code.

In the remainder of this section, we present a tutorial for building language-modular CFG-generators, the design of control-flow-based insertion, and the implementation in CUBIX, shown correct by a test suite of over 4000 lines.

2.7.2 CFG Generation: Not So Easy

Though our ultimate goal is to build language-modular CFG-generators, modularity problems appear in CFG generators even for one language. In this, we illustrate the problems of naive definitions CFG generators. In a tutorial reconstruction, we proceed to refine such a naive generator into a new, compositional design, in preparation for generalizing this design in Section 2.7.3.

A Naive CFG-Generator

We present this datatype for a simple imperative language, along with an interface for graphs:

```
type Var = String
type Label = Int

data Exp = Add Exp Exp | Lt Exp Exp | VarExp Var

data Stmt = Assign Var Exp
          | If Exp LabStmt LabStmt
          | While Exp Stmt
          | Block [LabStmt]

type LabStmt = (Label, Stmt)

type Graph -- definition not shown
addEdge :: Graph → Label → Label → Graph
connect :: Label → Label → State Graph ()

optConnect :: Label → Maybe Label →State Graph ()
optConnect l1 (Just l2) = connect l1 l2
optConnect l1 Nothing   = return ()
```

Disregarding the extra complexity to construct basic-blocks, a traditional statement-level CFG-generator would create one node per statement, as a recursive traversal. We construct an implementation below, and then discuss why its design does not readily extend to a language-generic implementation.

First we design the recursion. Let us think about what information must be passed down and up the stack in such a traversal. There will be an edge between the last statement in each branch of an if-statement, and the first statement after the if. Some part of the code must have access to the nodes of both. This can be accomplished either by passing down the node after the **if**, or by passing up the set of nodes which may be the last thing executed. In our implementation, we choose the former: the recursion passes down the the next node that runs after the current term.

Below is code for the CFG generator. As the language is small, the code is short, yet contains a high density of special cases.

```
genCfg :: Maybe Label →LabStmt → State Graph Label
genCfg next (l, Assign _ _) = do optConnect l next
                                 return next
genCfg next (l, If _ s1 s2) = do l1 ← genCfg next s1
                                 l2 ← genCfg next s2
                                 connect l l1
                                 connect l l2
                                 return l
genCfg next (l, While _ s)  = do l' ← genCfg (Just l) s
                                 connect l l'
                                 optConnect l next
                                 return l
genCfg next (l, Block ss)   = do ml' ← genCfgBlock next ss
                                 case ml' of
                                   Just l' → connect l l'
                                   Nothing → optConnect l next


                                 return l


genCfgBlock :: Maybe Label →[LabStmt]
                           → State Graph (Maybe Label)
genCfgBlock next []      = return Nothing
genCfgBlock next [s]     = do l' ← genCfg next s
                              optConnect l' next
                              return (Just l')
genCfgBlock next (s:ss) = do l ← genCfgBlock next ss
```

```
                    l' ← genCfg (Just l) s
                    return (Just l')
```

For a language so small, any problems that could exist in this code would be trivial inconveniences. Yet, viewed through a magnifying glass, it is already possible to spot two issues that may blossom into barriers as the language grows, or when trying to make this code language-modular.

The first is that this code is not *compositional*. This means that the CFG for one node is not a function of the CFGs for its child nodes. Indeed, some lack of modularity is already present: thinking carefully about the reason for each line, the handling of `next` nodes is derived from the design of `Block` nodes (e.g.: it would be different if the cons-lists of `Block` were replaced with snoc-lists or a binary `Seq` node).

The second is that the implementation of each case assumes this language has no nonlocal control-flow. Upon the introduction of new nodes with nonlocal control-flow, every case of the CFG-generator would need to be rewritten to thread extra state throughout the generator.

In the following subsections, we construct a new architecture that addresses all these problems.

The Case for Enter/Exit

In the previous subsection, we determined that the `genCfg` function needed to pass around non-local information because it may sometimes need to draw an edge connecting an AST node and its (great-)grandparent. This is also the reason `genCfg` is not compositional. To make it compositional, we must change the CFG design. Our proposal: create two CFG nodes per AST node, representing its entry and exit.

With this change, each AST node's CFG fragment depends only on its children. No extra information is passed down in recursion. And, as an added bonus, the special casing has also been eliminated, for it stemmed from decisions over where to handle the non-local information. Here are the first three cases:

```
makeEnterExit :: State Graph (Label, Label)
```

```
-- Implementation not shown

genCfg :: LabStmt → State Graph (Label, Label)
genCfg (_, Assign _ _) = do (enter, exit) ← makeEnterExit
                            connect enter exit
                            return (enter, exit)
genCfg (_, If _ s1 s2) = do (enter, exit) ← makeEnterExit
                            (enter1, exit1) ← genCfg s1
                            (enter2, exit2) ← genCfg s2
                            connect enter enter1
                            connect enter enter2
                            connect exit1 exit
                            connect exit2 exit
                            return (enter, exit)
genCfg (_, While _ s)  = do (enter, exit) ← makeEnterExit
                            (enterBody, exitBody) ← genCfg s
                            connect enter enterBody
                            connect exitBody enter
                            connect enter exit
                            return (enter, exit)
```

While the increase in the number of CFG edges has correspondingly increased the size of the code, the amount of information has decreased thanks to the higher symmetry. Gone are the branches; instead, each case reads as a list of the edges corresponding to the current node.

However, the casework needed to deal with empty blocks has not gone away — it's gotten worse! Continuing, the case for `Block`'s looks like this:

```
genCfg (_, Block ss) = do
  (enter, exit) ← makeEnterExit
  mEnterExitSS ← genCfgBlock ss
  case mEnterExitSS of
    Nothing                → return ()
    Just (enterSS, exitSS) → do connect enter enterSS
                                connect exitSS exit

  return (enter, exit)

genCfgBlock :: [LabStmt] → State Graph (Maybe (Label, Label))
-- Implementation not shown
```

We shall find that dealing with possibly-empty returned nodes is such a common situation that it merits its own primitive, and all the casework can be encapsulated in a new operation for combining a possibly-empty pair of nodes.

```
type EnterExitPair = Maybe (Label, Label)

combineEnterExit :: EnterExitPair → EnterExitPair
                                  → State Graph EnterExitPair
combineEnterExit Nothing          p2               = return p2
combineEnterExit p1               Nothing          = return p1
combineEnterExit (Just (l1, l2)) (Just (l3, l4)) = do
    connect l2 l3
    return (Just (l1, l4))
```

With this new primitive, branching is eliminated. Here is the new code for `Block`:

```
genCfg (_, Block ss) = do
  (enter, exit) ← makeEnterExit
  eepSS ← genCfgBlock
  x ← combineEnterExit (Just (enter, enter)) eepSS
  combineEnterExitPair x (Just (exit, exit))

genCfgBlock :: [LabStmt] → State Graph EnterExitPair
genCfgBlock ss = fold (\s mEepRest → do
                           eepS ← genCfg s
                           eepRest ← mEepRest
                           combineEnterExit eepS eepRest)
```

```
                  Nothing
                  ss
```

Monadic Deferral

Having made CFG-generation compositional, we chase the next milestone of modularity: how to make each case of `genCfg` into an independent function? Doing so would make it possible, for instance, to share CFG-generation code across all languages with while-loops.

```
genCfgWhile :: (Label, Label) → State Graph EnterExitPair


genCfgPython :: PyTerm → State Graph EnterExitPair
genCfgPython (PyWhile _ s) = do sEnterExit ← genCfgPython s
                                genCfgWhile sEnterExit
-- ...other cases


genCfgC :: CTerm → State Graph (Label, Label)
genCfgC (CWhile _ s) = do sEnterExit ← genCfgC s
                          genCfgWhile sEnterExit
-- ...other cases
```

This refactoring involves hoisting the recursive `genCfg` call out of the case for `While`. This works for the simple language we have used thus far. It does not work when `genCfg` may have non-commutative effects. Such effects in the CFG-generator occur when there are control-effects in the code. Let us add `break` and `continue` to our language:

```
data Stmt = ...
          | Break
          | Continue
```

Generating a CFG for these new nodes requires tracking the break and continue targets. This requires adding a new parameter to CFG generation, either explicitly, or by rolling it into the state. The latter is clearly the more modular option. We thus extend the state of the CFG-generator to also include a stack of break/continue targets, and update the signatures of other functions accordingly.

```
data CfgGenState = CfgGenState { graph :: Graph
                               , loopStack :: [(Label, Label)] }

type CfgGen a = State CfgGgenState a

combineEnterExit :: EnterExitPair → EnterExitPair
                                   → CfgGen EnterExitPair
connect :: Label → Label → CfgGen ()
pushLoop :: Label → Label → CfgGen ()
popLoop :: CfgGen ()
```

With these additions, it is no longer possible to write `genCfgWhile` with the given signature. `genCfgWhile` must use `pushLoop` and `popLoop` to communicate the break/continue targets to invocations of `genCfgC`/`genCfgPython` on nodes in the loop body. However, the CFGs for these nodes are generated before `genCfgWhile` is even called. In this new language, the CFG fragment for a node is no longer a function of the CFG fragments for its subnodes. `genCfg` is no longer compositional. Instead, the CFG fragment for a node also depends on the control effects of its children. And so, by accepting a monadic value for the CFGs of its children, `genCfgWhile` can control the state passed to the recursive calls to `genCfg`, and again becomes compositional:

```
genCfgWhile :: CfgGen (Label, Label) → CfgGen (Label, Label)
genCfgWhile mBodyEnterExit = do
  (enter, exit) ← makeEnterExit
  pushLoop enter exit
  (bodyEnter, bodyExit) ← mBodyEnterExit
  popLoop
  connect enter bodyEnter
  connect bodyExit exit
  connect enter exit
  return (enter, exit)
```

```
genCfgBreak :: CfgGen (Label, Label)
genCfgBreak = do
  (enter, exit) ← makeEnterExit
  (breakTarget, continueTarget):rest ← gets loopStack
  connect enter breakTarget
  return (enter, exit)

genCfgContinue :: CfgGen (Label, Label)
genCfgContinue = do
  (enter, exit) ← makeEnterExit
  (breakTarget, continueTarget):rest ← gets loopStack
  connect enter continueTarget
  return (enter, exit)
```

It is now straightforward to define a language-specific `genCfg` function which defers to these cases.

```
genCfg :: Stmt → CfgGen (Label, Label)
genCfg (While _ s) = genCfgWhile (genCfg s)
genCfg Break       = genCfgBreak
genCfg Continue    = genCfgContinue
-- cases for Assign, If, Block not shown
```

Finer-Grained CFGs

Control-flow graphs are best known from their use in compilers, where the definition "a CFG is a directed graph of basic blocks" has become sacrosanct. We have already departed from this somewhat by not collapsing consecutive statements into basic blocks, and by using two nodes per statement.

Yet smaller units also have control-flow (e.g.: `f() + g()` evaluates `f()` before `g()`), and we argue that, for static analysis and transformation tools, finer-grained control-flow graphs are a better choice. Indeed, many static analysis frameworks, such as

IncA [161] and Polyglot [126], already use expression-level CFGs. We will revisit this question in Chapter 4.1, but, for the problem at hand, the following observation resolves the issue decisively: in the examples of Section 2.7.1, the CFG-based inserter must find the predecessors of the *condition* of the loop and conditional. It is thus **not possible** to build the CFG-based inserter without a finer-grained CFG.

In the remainder of this subsection, all control-flow graphs will be expression-level.

2.7.3 Language-Modular CFG Generation

Having discovered and eliminated the bottlenecks to modularity, we can now aggressively factor out and automate common parts. In this subsection, we present the remainder of our approach to CFG-generation, and use it to create a complete CFG-generator for a language with 17 node types in only 19 lines of language-specific code.

Additional background: Operations on parametric syntax terms

An advantage of using explicit type-level recursion not previously discussed is that it enables *structured recursion schemes*. We stated in Section 2.7.2 that the new CFG-generator design is compositional, meaning that the CFG of a node is a function of those of its subterms. The *catamorphism* recursion scheme formalizes this. A recursive function like `genCfg`, implemented as a catamorphism, is built out of a function of type `TermSig (m (Node, Node))` $\to$ `m (Node, Node)`, for some appropriate monad `m`. This is called an *algebra* of the `TermSig` functor with *carrier* `m (Node, Node)`. The input to this function is a term where each child node has been replaced by a value of type `CfgGen (CfgNode, CfgNode)` (i.e.: a command that, when run, produces a CFG and its enter/exit nodes). The catamorphism construction combinator `cata` then lifts this algebra into a recursive function over an entire term.

We define some operations for multi-sorted terms, followed by an implementation of catamorphisms. `HFunctor` and `HFoldable` are higher-kinded analogues of the standard `Functor` and `Foldable` typeclasses. `K` is a type-level K-combinator, needed for embedding types that do not have a sort parameter inside a multi-sorted tree.

```
class HFunctor f where
  hfmap :: (forall l. f l → g  l) → (forall l. h f l → h g l)


newtype K a l = K { unK :: a}


class (HFunctor h) ⇒ HFoldable h where
  hfold : (Monoid m) ⇒ h (K m) l → m


−− Instances are auto−generated


hcata :: (HFunctor f) ⇒ (f a i → a i) → Term f i → a i
hcata f (Term t) = f (hfmap (hcata f) t)
```

As a simple example of a catamorphism, consider this function to compute the number of statements in a term. Notice how there are no explicit recursive calls; instead, `sizeF` inputs a "pre-digested" term, in which each child has been replaced by its size.

```
sizeF :: ImpTerm (K Int) l → K Int l
sizeF (Assign _ _)           = K 1
sizeF (If _ (K n1) (K n2)) = K (1 + n1 + n2)
sizeF (While _ (K n))        = K (1 + n)
 ...


size :: ImpTerm l → Int
size = hcata sizeF
```

We shall demonstrate a modular `genCfg` built in this fashion.[2]

A Generic Setup

We now proceed to develop a language-modular infrastructure for CFG-generators. We begin by defining the language IMP, which we defined as a showcase for our

[2]In the actual implementation, we use a recursion scheme which also permits cases to inspect the children themselves. This is rarely used.

techniques. Imp features two kinds of control effects, `break` and `goto`, and contains both lambdas and function definitions, constructs which define new control-flow contours (i.e.: separate intraprocedural CFGs). It also has type declarations, for the sole purpose of demonstrating how our approach handles nodes which do not take part in computation. Its signature is below. Imp has 5 base sorts: expressions, lambdas, statements, function definitions, and types; we repurpose Haskell's list constructor to create new sorts for lists of statements and function definitions. A top-level Imp program is then given by a value of type `Imp [FunDefL]`.

```
type Var = String

data ExpL; data TypeL; data LambdaL; data StmtL; data FunDefL

data ImpSig e l where
  Add       :: e ExpL → e ExpL → ImpSig e ExpL
  Lt        :: e ExpL → e ExpL → ImpSig e ExpL
  VarExp    :: Var             → ImpSig e ExpL
  LambdaExp :: e LambdaL       → ImpSig e ExpL
  CallExp   :: e ExpL → e ExpL → ImpSig e ExpL

  IntType  :: ImpSig e TypeL
  BoolType :: ImpSig e TypeL

  Lambda :: Var → e TypeL → e ExpL → ImpSig e LambdaL

  Assign    :: Var    → e ExpL                → ImpSig e StmtL
  If        :: e ExpL → e StmtL → e StmtL → ImpSig e StmtL
  While     :: e ExpL → e StmtL               → ImpSig e StmtL
  Break     ::                                  ImpSig e StmtL
  GotoLabel :: String                         → ImpSig e StmtL
  Goto      :: String                         → ImpSig e StmtL
  Block     :: e [StmtL] →                    → ImpSig e StmtL
  FunDef :: Var → e StmtL → ImpSig e FunDefL
```

```
data Fix f l = In (f (Fix f) l)
```

```
type Imp = Fix ImpSig
```

We have previously mentioned CUBIX's generic support for lists embedded in a tree. Its constructors are called `ConsF` and `NilF` (smart constructors: `ConsF′` and `NilF′`), and they can be used to provide terms of sort `[StmtL]` and `[FunDef]`.

We now state the core monad operations needed in CFG construction.

```
type CfgNode = Int
```

```
class MonadCfgGen m where
  makeEnterExit :: m (CfgNode, CfgNode)
  connect       :: CfgNode → CfgNode → m ()
```

These operations can be implemented on any state monad whose state contains certain fields, discussed in the next subsection.

```
class HasCfgGenState s where
  -- defined in next subsection
```

```
instance (HasCfgGenState s) ⇒ MonadCfgGen (State s)
```

We define an `EnterExitPair` type as in Section 2.7.2. We abbreviate it to `EEP`, as it is used quite frequently.

```
type EEP = Maybe (CfgNode, CfgNode)
```

```
combineEnterExit :: (MonadCfgGen m) ⇒  EEP → EEP → m EEP
```

A useful fact is that enter/exit pairs, in the presence of a monad capable of adding edges, actually form a monoid under `combineEnterExit`. If `a`, `b`, and `c` are monadic values which evaluate to enter and exit nodes of three separate CFG fragments, then `a <>b <>c` is a monadic value which, when evaluated, connects `a, b, c` in sequence, returning the enter and exit of the combined graph.

```
instance (MonadCfg m) ⇒ Monoid (m EEP)
  mempty     = return Nothing
  ma <> mb = do a ← ma
                b ← mb
                combineEnterExit a b
```

We shall soon give the general interface for CFG generators, as a catamorphism whose result is a stateful computation. One wrinkle is that values in the `State` monad do not have the right type for use in a catamorphism. They must be modified to take an extra sort parameter, which can be done by wrapping them with the `K` combinator.

```
type HState s a = K (State s a)
```

A downside is that using `HState` requires frequent unwrapping and rewrapping in order to use the monad operations. We are now ready to give the final interface for language-specific CFG-generators:

```
class (HFoldable f) ⇒ ConstructCfg f s where
  constructCfg' :: f (HState s EEP) l  → HState s EEP l

constructCfg :: (ConstructCfg f s) ⇒ Fix f l → State s ()
constructCfg t = void (unK (hcata constructCfg' t))
```

Open Products for Control Effects

Different control effects need different state. As we have shown in Section 2.7.2, break and continue statements demand maintenance of a stack of break and continue targets. Goto statements, meanwhile, demand a map of label names to targets. As languages may have their own exotic control effects, there can be no common CFG-generation state shared by all languages.

Our solution is to express the current CFG-generation state as an *open product*.

Open products permit the definition of functions which run on **any state which has a certain field**. The generic `constructCfgGoto` case, for instance, runs on any state that contains a map of goto labels to targets.

A common way to implement open products is with lenses [60]. A lens from `a` to `b` is a pair of a getter `a → b`, which looks up a field of type `b` from a value of type `a`, and a *setter* `a → b → a`, which replaces said field with a modified value.

```
type Lens a b = (a → b, a → b → a)
```

There are already many Haskell libraries for automatically generating lenses of this type (or one isomorphic to it) [90, 92, 173]. We sweep under the rug which one is used, and instead use the pseudocode `magicMakeLens` to denote the proper invocation of whichever library. We can now define the state needed for each kind of control effect: the current graph and an node-ID generator for general CFG-construction, a stack of break targets for loops, and a goto-label map for goto statements.

```
type NodeGen = Int


data CfgGenState = CfgGenState { curGraph :: Graph
                               , nodeGen  :: LabelGen }


class HasCfgGenState s where
  cfgGenState :: Lens s CfgGenState


class (HasCfgGenState s) ⇒ HasBreakStack s where
  breakStack :: Lens s [CfgNode]


class (HasCfgGenState s) ⇒ HasGotoMap s where
  gotoMap :: Lens s (Map String CfgNode)
```

We are now able to define language-agnostic CFG construction functions for all relevant statements, similar to the examples in Section 2.7.2.

```
constructCfgIf :: (HasCfgGenState s)
               ⇒ HState s EEP i → HState s EEP j
```

```
                → HState s EEP k → HState s EEP l


contsructCfgBreak :: (HasBreakStack s) ⇒ HState s EEP l
constructCfgWhile :: (HasBreakStack s)
          ⇒ HState s EEP i → HState s EEP j → HState s EEP k


constructCfgGoto      :: (HasGotoMap s) ⇒ String → HState s EEP l
constructCfgGotoLabel :: (HasGotoMap s) ⇒ String → HState s EEP l
```

Of note is that, for forward-gotos, `constructCfgGoto` must allocate a CFG node for the `GotoLabel` statement before it is seen.

Dispatching with Easy Cases

Most AST nodes have no interesting control-behavior. In this subsection, we define a way to deal with the common cases without writing any custom code.

We group these non-interesting nodes into three categories, discriminating by sort. Computation sorts describe those nodes which have control flow. These AST nodes will be given their own CFG nodes. Suspended-computation sorts, abbreviated "suspend sorts," are those nodes whose bodies may have control flow, but which should be in a separate contour, not connected to the CFG nodes of the surrounding context. Lambda expressions are a typical example. Finally, all other sorts are considered to not participate in the computation. For a node `n` of such a sort, its children, if any exist, will be sequenced and connected to the surrounding CFG nodes, as if `n` did not exist.

We define type families to specify these categories. For each language signature `f`, `ComputationSort f` returns the computation sorts for that language as a type-level list, and similar for suspend sorts. It is then possible to implement dynamic checks to see if a term is of a computation sort, using standard type-level programming techniques.

```
type family ComputationSorts (f :: (* → *) → * → *) :: [*]
type family SuspendSorts     (f :: (* → *) → * → *) :: [*]
```

```
isComputationSort :: Fix f l → Bool
isSuspendSort     :: Fix f l → Bool
```

As a preview, here are the definitions for `ImpSig`.

```
type instance ComputationSorts ImpSig = '[ExpL, StmtL, [StmtL]]
type instance SuspendSorts ImpSig = '[LambdaL, FunDefL]
```

Note that. as IMP has no expressions with interesting control-flow (e.g.: short-circuiting operators), simply removing `ExpL` from the list of computation sorts would change the CFG-generator definition to instead produce statement-level CFGs. We intentionally include `[StmtL]` in the list of computation sorts, giving `ConsF` nodes their own CFG nodes, as this will be useful when building the CFG-based inserter.

We now turn to defining the generic cases, beginning with the case for computation sorts. Thanks to our careful definitions of `HState` and the monoid instance for enter/exit pairs, the `hfold` function will run and sequence all children. The remainder of this function allocates CFG nodes for the current term, and connects them to the children.

```
constructCfgCompSort :: (HFoldable f)
                     ⇒ f (HState s EEP) l → HState s EEP l
constructCfgCompSort t = K do
  (enter, exit) ← makeEnterExit
  let left = return (Just (enter, enter))
  let body = hfold t
  let right = return (Just (enter, enter))
  left <> body <> right
```

Notice also what happens when there are no children: `body` evaluates to **Nothing**, and the final line simply connects `enter` to `exit`.

The final definition of `constructCfgDefault` dispatches on the sort of a term. For suspend sorts, it sequences the CFGs of all children, but returns an empty enter/exit pair, so that nothing will connect to the children. For non-computation nodes, it sequences the CFGs of the children and returns them

```
constructCfgDefault :: (HFoldable f)
                    ⇒ f (HState s EEP) l → HState s EEP l
constructCfgDefault t =
  if isComputationSort t then
    constructCfgCompSort t
  else if isSuspendSort t then
    K (hfold t >> return Nothing)
  else
    K (hfold t)
```

Notice that, when `t` is `IntType` or `BoolType`, `constructCfgDefault t` has no effects and evaluates to `K` **Nothing**. Notice also how it handles terms of sort `[StmtL]` (it sequences them, also creating CFG nodes for the `ConsF` and `NilF` nodes) and terms of sort `[FunDefL]` (it runs them independently).

Victory

We now give full source code for the CFG generator for Imp, validating our claim to construct a CFG-generator for it in only 19 lines of language-specific code. In fact, there are only 18 non-empty lines in the example below. Not shown is the code to generate lenses in whichever library is chosen; in Edward Kmett's `lens` library [92], this would be a 1-line Template Haskell invocation, `makeLenses ''ImpSigCfgState`, yielding our final count of 19 lines.

```
type instance ComputationSorts ImpSig = '[ExpL, StmtL, [StmtL]]
type instance SuspendSorts ImpSig = '[LambdaL, FunDefL]


data ImpSigCfgState = { breakStack  :: [CfgNode]
                      , gotoMap     :: Map String CfgNode
                      , cfgGenState :: CfgGenState }


instance HasBreakStack ImpSigCfgState where
  breakStack = magicMakeLens
```

```
instance HasGotoMap ImpSigCfgState where
  gotoMap = magicMakeLens
instance HasCfgGenState ImpSigCfgState where
  cfgGenState = magicMakeLens

instance ConstructCfg ImpSig ImpSigCfgState where
  constructCfg' (While e s)     = constructCfgWhile e s
  constructCfg' (If e s1 s2)    = constructCfgIf e s1 s2
  constructCfg' Break           = constructCfgBreak
  constructCfg' (GotoLabel s)   = constructCfgGotoLabel s
  constructCfg' (Goto s)        = constructCfgGoto s
  constructCfg' t               = constructCfgDefault t
```

With this code, one can now run `constructCfg t` for any Imp program `t` and obtain a control-flow graph.

The brevity of this example came in part because this language has no "unusual" nodes: each node either fits one of the defaults, or is a common control-flow construct for which it is reasonable to place the behavior in a language-agnostic function. In Cubix, we implemented CFG-generators for 5 real languages, all of which did have "unusual" nodes requiring custom code. Yet they also all benefited greatly from the predefined common cases, For JavaScript, for instance, we were still able to define a complete CFG-generator passing an exhaustive test suite in only 79 lines of language-specific code, compared to 184 distinct node types.

2.7.4 CFG-Based Program Transformation

In this subsection, we explain our new transformation primitive, CFG-based insertion. The code in this subsection involves more algorithmic details and bookkeeping compared to that of the subsections on modular CFG-generation. Consequently, whereas the code in those previous subsections would run verbatim except for their use of lenses, the code in this subsection tends slightly more towards pseudocode.

The goal of this subsection is to define the `dominatingPrepend` operation. The invocation `dominatingPrepend targ s` modifies the program to ensure that `s` always runs before `targ` by inserting `s` at the first possible predecessor along every control path.

```
dominatingPrepend :: (InsertAt f l) ⇒ Fix f i
                                     → Fix f l
                                     → CfgInsertion ()
```

There are many variants of this operation which we do not present here. Aside from the obvious `dominatingAppend`, which would instead ensure `s` runs *after* `targ`, there are variants in how to behave if multiple insertions are performed at the same point, and variants that can choose to treat each insertion point differently (to e.g.: only declare a temporary variable once).

The `dominatingPrepend` operation rests on both the presence of a CFG and the `InsertAt` interface. The `InsertAt` interface provides the twin operations of "is it possible to insert this node at this point" and of performing the actual insertion. The utility of this interface is that it describes both the mundane operation of inserting a statement into a list of statements **as well as** more peculiar ones which cleave open a statement to insert another.

A program point is identified by an AST node and an *evaluation point*; these shall also correspond to CFG nodes. Most AST nodes only have two evaluation points: before (`Enter`) and after (`Exit`) it executes. But a few, such as Python if-elif-elif-...-else chains, also have intermediate evaluation points (they correspondingly have more than two CFG nodes!). We will revisit this in Chapter 4, where we provide a theoretical account of "program points" from first principles.

```
data EvaluationPoint = Enter | Exit | Intermediate Int
```

We now define the `InsertAt` interface. `canInsertAt @l p targ` returns whether a term of sort `l`[3] can be inserted to run at the program point defined by `p` and `targ`. Then the code `insertAt p s targ` modifies `targ` to perform the insertion.

```
class InsertAt f l where
  canInsertAt :: EvaluationPoint → Fix f i → Bool
```

[3]The `@l` argument is a Haskell explicit type parameter.

```
  insertAt     :: EvaluationPoint → Fix f l → Fix f i → Fix f i
```

In the previous subsection, it sufficed for demonstration to use raw integer labels as CFG nodes, with no way to map between corresponding CFG and AST nodes. This subection requires CFG nodes with a little bit more structure (as is the case in the actual implementation), with the ability to map between AST and CFG nodes, where each CFG node corresponds to an AST node / evaluation point pair. We shall also need to refer to terms of unknown sort, which we do with the existential combinator `E`: `E (Fix f)` refers to a term `(Fix f l)` for some unknown sort `l`.

```
data E f = forall i. E {unE :: f i}


type Graph f
type CfgNode f
evalPoint :: CfgNode f → EvaluationPoint
termFor   :: CfgNode f → E (Fix f l)
predecessors :: CfgNode f → [CfgNode f]


nodeFor :: Graph f → EvaluationPoint → Fix f l → Maybe CfgNode
```

Note that `nodeFor` is not actually implementable as written, for it cannot distinguish between identical terms at different program points. This does not substantively alter our work, as it is possible to (as is actually done in CUBIX) modularly add label annotations via an alternate fixpoint operation

```
data FixLab f l = In (f (FixLab f l), Label)
```

but we choose to omit such bookkeeping from this presentation. An alternative version uses paths from the root to identify unique subtrees.

We also define this helper function:

```
canInsertAtNode :: (InsertAt f l) ⇒ CfgNode f → Bool
canInsertAtNode n = canInsertAt @l (evalPoint n) (unE (termFor n))
```

We now begin to implement CFG-insertion. We shall demonstrate on a tiny model of Python called WORM, capable of expressing the two special cases described in Section 2.7.1.

```
data WormSig e l where
  While :: e ExpL → e StmtL → WormSig e StmtL
  Continue :: WormSig e Stmtl
  IfElse :: e ExpL → e StmtL → e StmtL → WormSig e StmtL
  IfElifElse :: e ExpL → e StmtL -- if
             → e ExpL → e StmtL -- elif
                      → e StmtL -- else
                      → WormSig e StmtL
  -- other cases not shown
```

As we will need to construct new terms, as is customary with unfixed data types, we shall define *smart constructors* as syntactic sugar.

```
iWhile :: Fix WormSig ExpL → Fix WormSig StmtL
                          → Fix WormSig Stmtl
iWhile e s = In (While e s)

-- iContinue, iIfElse , etc not shown
```

We begin by defining `InsertAt`. This benefits from an operation to check the sort of a term: `isSort @StmtL t` tests if `t` is a statement. If it passes, an actual implementation would need to cast `t` to `Fix f StmtL` in order to use it; we gloss over this detail.

```
isSort :: Fix f i → Bool

instance InsertAt WormSig StmtL where
  canInsertAt (Intermediate 0) _ (In (IfElifElse _ _ _ _ _)) = True
  canInsertAt Enter _ t = isSort @StmtL t
  canInsertAt _     _ _ = False

  insertAt (Intermediate 0) stmt (In (IfElifElse c1 S1 c2 s2 c3)) =
    iIfElse c1 s1
        (ConsF' stmt
            (ConsF' (iIfElse c2 s2 s3)
                    NothingF'))

  insertAt Enter stmt t | isSort @StmtL t = ConsF' stmt t
  insertAt _     _    _                   = error "Unreachable"
```

In the real implementation, most `InsertAt` cases are variants of inserting a statement into a list of statements. In addition to this case for if-elif-else, other special cases include replacing a singleton statement with a block, and splitting a Python **with** (e.g.: inserting code between the calls to `A()` and `B` in **with** `a as A(), b as B():`).

We now turn to the components of the implementation of `dominatingPrepend`. The `satisfyingBoundary` function implements a standard graph algorithm which finds all predecessors of a node which satisfy some predicate. When used with `canInsertAt`, it locates the places at which to perform the insertion.

```
satisfyingBoundary :: Graph f → CfgNode f → (CfgNode f → Bool)
                                          → Set (CfgNode f)
satisfyingBoundary cfg start pred = go Set.empty start
  where
    go :: Set (CfgNode f) → Cfgnode f → Set (CfgNode f)
    go seen x = if Set.member seen x then
                  Set.empty
                else if pred x then
                  Set.singleton x
                else
                  fold (map (go (Set.insert x seen))
                            (predecessors x))
```

Because this implementation runs on immutable trees, the function `dominatingPrepend` must run in two phases: first marking the set of insertions to perform, and then performing them. We define the monad to hold the necessary state:

```
data CfgInsertionState f l =
  CfgInsertionState
    { pendingInsertions :: Map (CfgNode f) (Fix f l)
    , cfg :: Graph }

type CfgInsertion f l = State (CfgInsertionState f l)
```

`dominatingPrepend` simply finds the insertion points and marks the intended insertion.

```
dominatingPrepend :: (InsertAt f l) ⇒ Fix f i
                                    → Fix f l
                                    → CfgInsertion f l ()
dominatingPrepend target item = do
  state ← get
  let curCfg = cfg state
  let startNode = fromJust (nodeFor curCfg Enter)
  let insertionPoints =
         satisfyingBoundary curCfg startNode canInsertAtNode
  let insertions =
         Set.fold (Set.map (\node →Map.singleton node item)
                           insertionPoints)
  puts (\s → s {pendingInsertions = insertions })
```

For performing the insertions, we assume a primitive `rewriteAt`. `rewriteAt x t f` finds the subtree in `t` equal to `x`, and then rewrites it with `f`, returning an updated `t`. Similar to the `nodeFor` function, a real implementation would require either labeled terms or paths from the root.

```
rewriteAt :: Term f i → Term f j
          → (Term f i → Term f i) → Term f j
```

The final step loops over the intended insertions and performs them.

```
performInsertions :: (InsertAt f l) ⇒ Fix f i
                                    → Graph f
                                    → Cfginsertion f l ()
                                    → Fix f i
performInsertions t g m =
    let initialState = CfgInsertionState Map.empty g
```

Table 2.6: Line and token counts of CFG-generators

Language	Cubix SLOC	Cubix tokens	Comparison	Comp. SLOC	Comp. Tokens
C	168	1837	Clang* [142]	1158	7962
Java	107	1374	Polyglot [143]	573	3888
JavaScript	79	1181	ast-flow-graph [83]	400	2924
Lua	148	1522	None found	N/A	N/A
Python	110	1261	StatiCFG [34]	232	2525
Infrastructure	475	4814			
Total	1087	11989		2463	17299

*C-relevant portions only

```
let insertions = pendingInsertions (execState m initialState)

Map.foldrWithKey
  (\t' node item →
      let p = evalPointNode in
      rewriteAt p (termFor node) t' (insertAt p item))
  t
  insertions
```

2.7.5 Implementation

Within this subsection, the term "significant lines" refers to the number of lines in a file after removing the file prologue (i.e.: import statements), comments/docstrings, and blank lines. All line counts (SLOC) refer to significant lines.

Language-Modular CFGs We implemented CFG generation in the language-modular style in Cubix, and built CFG-generators for C, Java, JavaScript, Lua, and Python, along with a thorough test suite with over 3000 SLOC of unit tests, together with 66 end-to-end tests, consisting of an input program and its full expected graph. Even though most CFG-generation is performed by language-agnostic code, the unit tests are language-specific, giving us high-confidence that our approach is flexible enough to adapt to the peculiarities of each language while still saving substantial labor.

Table 2.6 gives the size of the CFG generators built in Cubix. For each language, we searched for a comparison CFG generator, restricted to those which build a CFG from the original AST and not for an IR. We found comparison generators for all languages except for Lua, after having searched for one in Lua static analysis tools and in the source code of the Lua.org and LuaJIT implementations. We include CFG construction code, but not code for the graph data type. We also report token counts, computed by a lexer for the relevant implementation language, which tend to be more robust than line counts.

This is an imperfect comparison in every way. The comparison CFG-generators range in quality from "part of a major compiler" (Clang) to "visualization tool with 20 stars on Github" (ast-flow-graph). All of the Cubix generators are expression-level, whereas all comparison generators except Polyglot are statement-level. On the other hand, some of the comparison generators also do basic-block compression. We also biased these counts against Cubix by including infrastructure for Cubix but not the comparison generators; the relevant equivalents of Section 2.7.3 total 1659 SLOC and $12,536$ tokens across the 4 comparison generators. Finally, while they are generally better than line counts, token counts are not known to be comparable across languages.

Nonetheless, this table gives strong evidence that our approach provides substantial labor savings compared to a single-language baseline lacking generic programming techniques. Our combined generators take approximately 1100 lines compared to 2400 for the comparison generators— and this is excluding a comparison Lua generator. And the per-language marginal cost is over 3x fewer lines — and would be even better if we included each other generator's infrastructure.

CFG-Based Insertion We implemented a CFG-based inserter in Cubix in 142 SLOC. We used it to implement the test-coverage and three-address code transformations described in Section 2.5.2 (with the test-coverage transformation being strictly more complicated than the sanitization example in Section 2.7.1). These transformations attained full semantics-preservation as measured by compiler test suites totalling over 5000 test files, as shown in Section 2.5.3.

2.8 Application: Semantic Code Search

In this section, we present the Yogo semantic code search tool, currently the largest application of Cubix. Yogo addresses the common and commonly-intimidating task of discovering all places in a codebase which perform some similar operation. For instance, if a common idiom is discovered to have a bug, then all instances of that idiom must be changed. Similarly, if there are many places in the codebase which manually access a data representation, then changing the representation demands not only locating all code which accesses the data structure, but also recognizing which high-level operation they perform. While conventional code search is helpful in identifying these locations, it is not enough; it can still miss a long tail of unexpected variations of the common pattern.

For example, imagine an E-commerce app that represents the items in a shopping cart as an unordered array with duplicates. The programmer wishes to find all code that counts the frequency of a given item in the list, as part of some larger refactoring, whether to replace all of them with a shorter or more efficient implementation of frequency counting, or to switch to an alternate representation of shopping carts altogether. Figures 2-26a and 2-26e show example code and a refactored version. Although this is one of the simpler instances of this problem, finding all equivalent code in a codebase is already hard, as the code sought may be *interleaved* with other code (as in Figure 2-26c), and may be *paraphrased* via syntactic variation or different approaches altogether (as in Figures 2-26b and 2-26d).

Abstracting Away Syntax How can a program recognize that the examples in Figure 2-26 are all instances of the same pattern? This has long been the goal of *semantic code search* (for recognizing a concept in a codebase), and the closely-related

```
count = 0
for a in cart:
  if a == item:
    count += 1
use(count)
```

(a)

```
count = 0
for i in range(len(arr)):
  if itm != arr[i]:
    continue
  count += 1
use(count)
```

(b)

```
count = 0
for i in cart:
  if debug:
    print(cart[i])
  if cart[i] == item:
    count += 1
use(count)
```

(c)

```
count = 0
i = 0
while i < len(cart):
  if cart[i] == k:
    count += 1
  i += 1
use(count)
```

(d)

```
use(cart.count(item))
```

(e)

Figure 2-26: Variations over the array frequency count pattern in Python

problem *semantic clone detection* (identifying semantically-similar code snippets in a codebase). There are a wide range of code search tools intended for different purposes (see [141] for a survey). In this work, we are concerned specifically with code searches intended to help a programmer change a codebase, measured to be 16% of code searches [149]. We specifically attack an extreme version of the problem, motivated by large refactoring tasks, where every result returned is programmer time saved — or a bug averted. In this extreme, the goal is to exactly match all instances of a semantic concept within a single codebase. To do so, we are willing to spend computation on the most powerful reasoning techniques available. Our results offer a search procedure which is complete in semantic equivalence up to a set of graph-rewrite rules, and which is embarrassingly parallel, able to query a 1.2 million line codebase in 2.5 hours using 30 machines.

Yogo: You Only Grep Once In this section, we present a new approach to semantic code search based on dataflow equivalences, and our Yogo tool built on it. Yogo takes as input the code to search, a library of pre-written rewrite rules, and a high-level concept expressed as a dataflow pattern. Using a fusion of techniques from Tate et al's *equality saturation* [162] and the Programmer's Apprentice [146], it is able to recognize when a code fragment is equivalent to one of many implementations of said concept.

From equality saturation, we borrow the Program Expression Graph (PEG) representation. Like program dependence graphs, Program Expression Graphs ignore statement ordering and can match patterns interleaved with other code. However, they go further by representing all of a program's semantics, including mutation and loops, as pure data-flow. In doing so, it becomes possible to discover equivalent fragments by applying low-level equations and rewrite rules, and then compactly represent all such equivalent fragments as a structure called an *e-graph* [123, 122, 49]. The result is an efficient procedure for discovering if a program contains a subprogram equivalent to the search pattern. And, if the rules given to the system are sound, and they only entail a finite number of equivalent programs, then the search procedure is sound, and complete with respect to the rules. In our experiments, given the default rewrite rules, Yogo's search terminates in under 3 minutes for over 99% of methods.

From the Programmer's Apprentice, we borrow the idea that high-level concepts can be identified as dataflow patterns. Based on this idea, we can e.g.: create rewrite rules that recognize many implementations of the concept of "iterating through a sequence." A dataflow pattern for array frequency-counting can then use this concept as a subnode, so that it may match any of the varieties of iteration in Figures 2-26 and 2-27. The same idea allows our approach to recognize when a program accomplishes the same goal through an alternate API, or even a different algorithm. With this switch to high-level concepts, our technique can even abstract away language-dependent features. The upshot is that, from a single query for array-count frequency, Yogo can recognize not only all five Python variations in Figure 2-26, but also the three Java variations in Figure 2-27.

And Yogo's equational reasoning capabilities extend beyond code search. We later extended Yogo with a mode for proving the equivalence of two functions, added rudimentary support for C, and used it to prune equivalent programs in a program induction benchmark set (Section 2.8.3).

This chapter presents an abridged explanation of Yogo, with an emphasis on its raw capabilities and on its use of Cubix. Full details, including additional motivation and evaluation, are available in the paper [141] and accompanying tech report [140]. Details on the equivalence checking mode are available in the aforementioned benchmark paper [5].

2.8.1 A Run Through Yogo

In this section, we demonstrate using Yogo to search for a 1-dimensional bounds-check equivalent to `i` $<$ `lo` $||$ `i` $>=$ `hi`. Although small, this is already challenging. Figure 2-28 shows 4 examples of bounds-checking code, including examples in both positive and negated form, in both Java and Python, and including variants which have no tokens in common with the given pattern.

To search for a bounds check, the user writes a query in Figure 2-30, using Yogo's

```
int count = 0;
for (Item x : list)
  if (x == k)
    count += 1;
use(count);
```

(a)

```
int count = 0;
for (i = 0; i < list.size(); i++)
  if (list.get(i) == k)
    count += 1;
use(count);
```

(b)

```
int count = Collections.frequency(list, k);
use(count);
```

(c)

Figure 2-27: Java variations of array frequency count

```
if (x >= end || x < pos) {
  doSomething();
}
```

(a)

```
if 10 <= x < 20:
  doSomething()
```

(b)

```
boolean b = i >= left;
int j = i;
boolean c = j < right;
if (!(b && c)) {
  doSomething();
}
```

(c)

```
if (x >= rect.left()) {
  if (x < rect.right()) {
    doSomething();
  }
}
```

(d)

Figure 2-28: Example 1D bounds checks

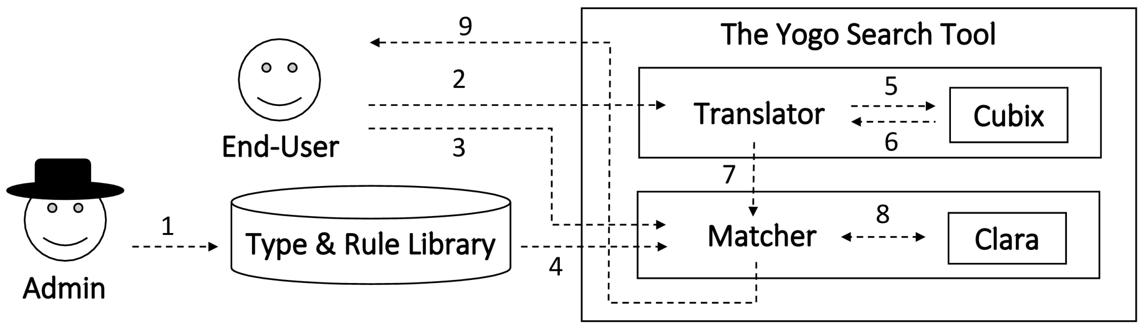


Figure 2-29: The organization of the Yogo Search Tool and its deployment. The system admin maintains a long-term library of rules and custom types (1), which are reused in every search session (4). Then for each search session, the end-user provides source files (2) and search patterns (3). The tool outputs match results to the end-user (9).

DSL[4]. This query is a textual form of the dataflow graph pattern in Figure 2-31a.

The user then invokes Yogo, indicating the target language and files, the query, and **a standard library of rules**.

```
./yogo java "general_rules.yogo,java_rules.yogo"
        query.yogo *.java
```

The general rules library includes rules for reasoning about boolean and comparison operators, such as the one in Figure 2-32, which implements De Morgan's law $a \vee b = \neg(\neg a \wedge \neg b)$. The Java- and Python-specific rule libraries contain rules for reasoning about language-specific constructs and APIs, and for mapping them into language-generic concepts. For example, there is the Python rule giving the isomorphism between (`a` $<=$ `b` $<$ `c`) and (`a` $<=$ `b` and `b` $<$ `c`)[5]. Over time, a small set of power-users can add rules for reasoning about new libraries and domains, enabling a large set of end-users to rapidly construct deep semantic queries. Figure 2-29 gives an overview of how the two kinds of users interact with Yogo.

[4]Yogo's DSL is more verbose than the concrete syntax to simplify parsing in the implementation. This is not fundamental to the approach.

[5]The purely-functional PEG representation renders short-circuiting and duplication a non-issue, at the cost that PEGs cannot always be mapped back into code.

During execution, YOGO constructs a Program Expression Graph for each method under search. (YOGO only accepts intraprocedural queries.) It then runs its **equality saturation** engine to turn that PEG into an *equivalence graph* on this PEG, or E-PEG, which represents both the original method as well as all methods which can be shown equivalent using the provided rules library.

For example, we illustrate how YOGO matches the query against the code snippet `x` $>=$ `lo` && `x` $<$ `hi`. YOGO first translates this code into the PEG in Box 1 of Figure 2-31b. After matching the rule for De Morgan's law, YOGO's equality saturation engine extends the PEG with the new nodes in Box 2, and adds the dashed equivalence edge between the `and` and `not`. Finally, it matches and runs two rules witnessing both directions of the equivalences $\neg(a \geq b) = (b < a)$, adding the nodes of Box 3. The entire e-graph in Figure 2-31b now represents 5 variations of the original code.[6] The `or` node is equivalent to the search query, and hence YOGO returns a match, with the query variable `root` bound to the `or` node.

Other rules in YOGO's standard libraries give it the ability to reason about nested conditionals, memory, and assignments. Using these rules, it can expand the four programs of Figure 2-28 into E-PEGs that compactly represent the exponentially large spaces of equivalent programs, and discover that all four of them contain a 1-dimensional bounds-check.

Of course, our worked example only shows YOGO reasoning about pure code, using the classic techniques of e-graphs and congruence-closure [123, 122, 49]. In order to scale to all the examples of Figure 2-28, YOGO must translate stateful code to a form amenable to equational reasoning. The full paper [141] explains how YOGO leverages E-PEGs [162] to handle stateful and loopy code, and the insights of the Programmer's Apprentice [146] to match high-level concepts rather than specific code.

[6]We'll have more discussion in Chapter 3 about the ability of e-graphs to compactly represent large sets of programs and its limitations in doing so, and how e-graphs are equivalent in expressiveness to tree automata.

```
(defsearch bound—checking
  (root <— (generic/binop :or
                          (generic/binop :< x lo)
                          (generic/binop :>= x hi))))
```

Figure 2-30: A search query for bounds-checking

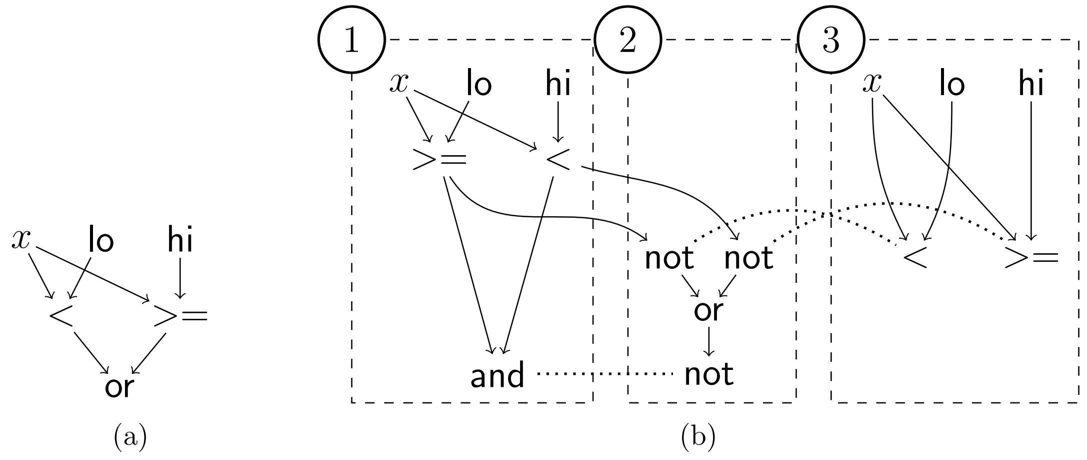


Figure 2-31: (a): PEG for the query in Figure 2-30. (b): E-graph representing 5 programs equivalent to `x` >= `lo` && `x` < `hi`, with nodes grouped by order of discovery. Dashed lines are between equivalent nodes. Some nodes duplicated for clarity.

```
(defeqrule demorgan1
  (generic/binop :and a b)
  =>
  (generic/unop :not
                (generic/binop :or
                               (generic/unop :not a)
                               (generic/unop :not b))))
```

Figure 2-32: Yogo rule for one direction of De Morgan's law

2.8.2 Implementation

Yogo is implemented in 900 lines of Clojure and 2700 lines of Haskell. The Haskell portion, based on Cubix, defines translators from Python and Java to PEGs. The Clojure portion defines a DSL for rewrite rules and queries, as well as an equality saturation engine based on the Clara implementation of the Rete algorithm [59]. Yogo comes with 500 lines of generic rewrite rules, 200 lines of Java-specific rules, 200 lines of Python-specific rules, and 30 lines of C-specific rules. Yogo uses a heuristic analysis based on method names for inferring method purity, e.g.: it assumes that Java methods starting with "get" or "to" are pure.

Yogo also features an *equivalence-checking* mode. To check for equivalences between functions, Yogo generates a single joint E-PEG containing both, assigning each the same start state. It then reports two subprograms as equivalent if equality saturation groups their return values in the same equivalence class.

2.8.3 Evaluation

In this subsection, we set out to prove Yogo's ability to search real codebases and to find paraphrases and discontiguous matches in multiple languages. We first present a brief summary of our systematic study of 9 search patterns Section 2.8.3 on a corpus of 3 codebases. As this evaluation involves artificial searches on small (under $60K$ LOC) codebases, we follow this with our case study on Oracle's Graal project (Section 2.8.3), where a Yogo query discovered a bug in a 1.2M LOC codebase that had been missed by a custom static analyzer designed for that exact class of bugs. We conclude with a discussion of results from Yogo's equivalence checking mode.

General Patterns, Multiple Codebases

As a test of Yogo's generality and cross-language abilities, we developed 9 language- and codebase-independent queries and evaluated them on 3 codebases. We present our results briefly here. Full discussion is available in the dedicated Yogo paper [141] and its accompanying technical report [140]. These experiments use only Java and

Table 2.7: Codebases searched, and number of time-outs

				Methods		Non-methods	
	Lang.	Commit	LOC	Total	TO	Total	TO
PyGame	Python	ad681aee	49k	2345	21	481	3
Cocos2D	Python	9bb2808	53k	2876	2	1082	0
LitiEngine	Java	c188504	59k	3625	28	997	2

Table 2.8: The number of matches found for each search pattern and codebase.

Pattern	Cocos2D	PyGame	LitiEngine
SP1. Bound checking	4	3	13
SP2. Squared 2D distance	8	8	10
SP3. Put-if-not-present	3	4	11
SP4. Frequency count	0	0	0
SP5. Time elapsed	1	20	1
SP6. Loop index	120	126	42
SP7. Dictionary iteration	17	6	9
SP8. MD5 hashing	1	1	0
SP9. File writing	8	7	0

Python, as YOGO's C frontend was added later.

We first chose the 9 patterns, using a combination of author insight and a systematic methodology based on StackOverflow questions. The 9 patterns are listed in Table 2.8. Because several of our patterns were geometric, and because game engines tend to contain many interesting intraprocedural code snippets, we chose 3 game engines as our codebases to analyze on: PyGame, Cocos2D, and LitiEngine. Information about them is listed in Table 2.7. Using the Docker container for YOGO shipped in the accompanying artifact, on one author's computer, YOGO searched PyGame, Cocos2D, and LitiEngine in 3.5, 1, and 4.5 hours respectively. We set YOGO to a 3-minute timeout. As shown in Table 2.7, YOGO timed out on fewer than 1% of methods; note that YOGO can still find some matches even if the generated E-PEG does not reach saturation. Table 2.8 gives the full results.

Here are a few example YOGO matching, demonstrating how it is able to find matches in the presence of paraphrases, match discontiguity, and multiple languages.

- Figure 2-33 gives three examples of SP2, squared distance. Figures 2-33a and 2-33c use explicit power-of-2, while 2-33b uses self-multiplication. These para-

```
double distSq =
  Math.pow((a.getCenterX() - b.getCenterX()), 2)
   + Math.pow((a.getCenterY() - b.getCenterY()), 2);
```

(a) LitiEngine (Java)

```
return Math.sqrt(
   (p1.getX() - p2.getX()) * (p1.getX() - p2.getX())
     + (p1.getY()-p2.getY()) * (p1.getY()-p2.getY()));
```

(b) LitiEngine (Java)

```
xdistance = left.rect.centerx - right.rect.centerx
ydistance = left.rect.centery - right.rect.centery
distancesquared = xdistance ** 2 + ydistance ** 2
```

(c) PyGame (Python)

Figure 2-33: Excerpts that match SP2 (squared 2D distance).

```
for n, k in enumerate(self.tileset):
    s = pyglet.sprite.Sprite(self.tileset[k].image, y=y,
                             x=x, batch=self.tiles_batch)
    # ...
```

(a) Cocos2D (Python)

```
for (final Map.Entry<String, List<Consumer<Float>>> entry
        : this.componentReleasedConsumer.entrySet()) {
    for (final Consumer<Float> cons : entry.getValue()) {
        pad.onReleased(entry.getKey(), cons);
    }
}
```

(b) LitiEngine (Java)

Figure 2-34: Excerpts that match SP7 (iterating over a map).

phrases are bridged by a Java-specific equality rule that rewrites the value of `Math.pow` function call to a generic power operation and a language-generic equality rule that rewrites `x * x` to `x ** 2`, as well as the generic rules for local variables.

- Figure 2-34 shows different way one can iterate over key-value entries of a map (SP7). Although this required language-specific rules to understand iterating over the key set vs the entry set of a map, we reused a lot of rules and abstractions from SP3 and SP4, which also involved maps and iteration, respectively. As a result, SP7 is fairly concise, with only 5 node patterns.

More discussion is available in the full paper [141] and its accompanying technical report [140], and the set of all matches is available in the accompanying artifact.

Graal Case Study

The GraalVM [183] is a Java VM and JDK built by Oracle designed to support language-interoperability and ahead-of-time compilation. As its associated Java compiler, Graal, is itsef written in Java, the project contains 13 hand-written static analyzers, built atop Graal's own bytecode analysis infrastructure, to enforce coding guidelines and catch bugs in the project. We determined that the buggy patterns sought by 4 of these analyzers could be expressed as Yogo queries. We implemented 3 of these as Yogo queries, and verified they all caught the defects in past revisions of Graal that inspired the corresponding analyzer. But, for one of these, we realized that the Yogo query would naturally be more general than the existing analyzer, opening the possibility that Yogo could detect new bugs in the codebase. And, excitingly, this happened to be the one analyzer relevant to other codebases.

`VerifyDebugUsage` searches for several patterns akin to `Debug.log("A: " + str)` or `Debug.log("A: \%s", node.toString())`. Code of these patterns all perform string computations whose results are discarded when debug-logging is not enabled. These are all minor performance bugs, but ones which the Graal team has aimed to eradicate from their compiler. The preferred alternative is `Debug.log("A: \%n", node)`, which

```
(deftrigger plus-string-string
  (e <- (generic/binop :+ s1 s2))
  (rules/is-string s1)
  (rules/is-string s2)
   =>
  (rules/is-string e))

(deftrigger string-alloced-argument
  (rules/obj-to-string arg)
   =>
  (rules/bad-debug-argument arg))
```

Figure 2-35: Extracts of query for incorrect debug usage

does no string computations unless debug-logging is enabled.

The `VerifyDebugUsage` analyzer is 330 lines,[7] and works by manual tree-pattern matching. We immediately noticed a limitation: it would fail to detect the target defect if there was indirection through temporary variables, as in the example `str = n.toString(); Debug.log("\%s", str);` . However, YOGO by default treats such a snippet as indistinguishable from its inlined version, `Debug.log("\%s", n.toString())`.

After identifying the opportunity, to help with our deadline, we outsourced the remaining work to a programmer in India, Sreenidhi Nair. As evidence for YOGO's usablity, **in 3 days, he learned the tool well enough to implement this query**, along with queries for the other Graal checkers. The debug-usage query is 69 SLOC of YOGO DSL. Of these, 11 lines merely tag expressions as having type `String`, owing to the lack of type information in YOGO's current information, leaving 58 lines of actual query code. Figure 2-35 gives extracts of this code, which uses YOGO's support for auxiliary facts to identify string expressions and to mark expressions as unsuitable for use in debug-logging.

For testing, we inspected past commits which modified their checker to find instances of the bug it was designed to catch. To avoid overfitting the query, the programmer developing the query was told which commits and directories contained instances of the buggy pattern, but was not shown the instances until after success-

[7] As of SHA 4ce223a1dc

```
if (object instanceof Node) {
  Node node = (Node) object;
  String loc = GraphUtil.approxSourceLocation(node);
  String name = node.toString(Verbosity.Debugger);
  if (loc != null) {
    debug.log("Context obj %s (approx. location: %s)",
              name,
              loc);
  } else {
    debug.log("Context obj %s", name);
  }
}
```

Figure 2-36: Defect caught by YOGO but not custom checker.

fully detecting it with YOGO.

Although Graal has a 1.2 million line codebase, as YOGO runs on each method independently, parallelizing this search was straightforward. The final run took 2.5 hours using 30 AWS instances (type `c5n.xlarge`). The search turned up many uninteresting true positives, such as defects in test code, which `VerifyDebugUsage` is not configured to check. It also turned up one example which, while an instance of the buggy pattern which should have been caught by their checker, was not an actual defect, as it was wrapped by the condition `if (log.isLoggable(Level.FINE))`. Along with these uninteresting-yet-correct matches, it also turned up one defect that could not have been found by `VerifyDebugUsage`.

Figure 2-36 gives the buggy code, which `VerifyDebugUsage` missed because of the indirection through `nodeName`. Our fix wrapped this code in a condition checking that logging was enabled, and was accepted into the Graal codebase.[8] And thus, **a 60-line YOGO query found a bug missed by a 330-line checker designed for that exact purpose**.

[8]https://github.com/oracle/graal/pull/1965/

Equivalence-Checking

6 months after the initial publication of Yogo, we were contacted by a group developing a new benchmark set of small programs for use in program induction and meta-learning. They had been chopping mined functions into small "subprogram" snippets, and were interested in using Yogo to prune semantically-duplicate programs. We agreed to build an equivalence-checking mode for Cubix and a limited C frontend.

They first presented us with 13 challenge problems inspired by their corpus, intended as hard examples for showing equivalence. Each such problem consists of two short semantically-identical functions, and a third function which is syntactically similar yet semantically distinct. Figure 2-37 gives two of the harder such problems, one involving standard library functions, the other involving arithmetic.

We added 24 general and 6 Java-specific rules in order to handle these equivalences, addressing both standard functions and arithmetic. For example, for Figure 2-37a, we added rules to track whether a number is non-negative, a general concept of the `abs` function, a a rule mapping the `java.lang.Math.abs` function to the generic `abs` function, and a rule stating that the `abs` function is the identity on non-negative inputs. For Figure 2-37b, we added more arithmetic rules, including ones for constant folding.

After adding these rules, we found that Yogo was successfully able to identify the 13 semantically-equivalent pairs when simultaneously fed the 39 functions from the problems. As a caveat, this required turning on rules for associativity, which causes equality saturation to not terminate due to an unsolved problem in e-graph saturation. We wrote up a description of this problem at `https://github.com/egraphs-good/egg/discussions/60`.

We next turned to finding equivalent programs in their real corpus, feeding Yogo a set of programs of the same type signature that behaved the same on a small set of random inputs. In our initial run, Yogo found 731 equivalence classes containing more than one subprogram; these 731 non-trivial equivalence classes total 2307 subprograms.

A full description of the benchmark set and YOGO's role in its creation is available in the corresponding paper [5].

2.9 Conclusion

Incremental parametric syntax fulfills a simple promise: when writing similar transformations for multiple languages, they should be able to share code to the extent the languages are similar. And we have delivered. In an age when even commercial analysis and transformation tools rarely support multiple languages, a student researcher was able to develop a prototype to solve the Facebook/Dropbox problem for 5 languages, write battle-tested CFG-generators for 5 languages in under 125 lines of language-specific code on average, and develop the world's most advanced semantic code search tool for 3 languages.

We are the first to allow a single program to perform source-to-source transformations on multiple real languages while preserving the information of each. We believe incremental parametric syntax solves a key problem in writing multi-language tools, and greatly increases the cost/benefit ratio of doing so.

Work on CUBIX is ongoing, with the framework now featuring a dedicated website, high-quality tutorial materials, and a mailing list for users. We are working to bring about the future of multi-language tools, and the world is listening.

For the tutorial, documentation, and full source code for CUBIX, including the RWUS suite, go to:

`http://www.cubix-framework.com`

```
public int prob1a(int a, int b) {
  a = Math.abs(a) + Math.abs(b);
  return a;
}

public int prob1b(int a, int b) {
  a = Math.abs(a) + Math.abs(b);
  a = Math.abs(a);
  return a;
}

public int prob1_decoy(int a, int b) {
  a = Math.abs(a + b);
  return a;
}
```

(a)

```
public int prob2a(int a, int b) {
  b = b - a;
  b = b * 2;
  b = b/2;
  return b;
}

public int prob2b(int a, int b) {
  b = b - a;
  b = b * 4;
  b = b / 2;
  b = b/2;
  return b;
}

public int prob2_decoy(int a, int b) {
  b = b - a;
  b = b/2;
  return b;
}
```

(b)

Figure 2-37: Example triplets of functions fed to the equivalence-checker.

Chapter 3

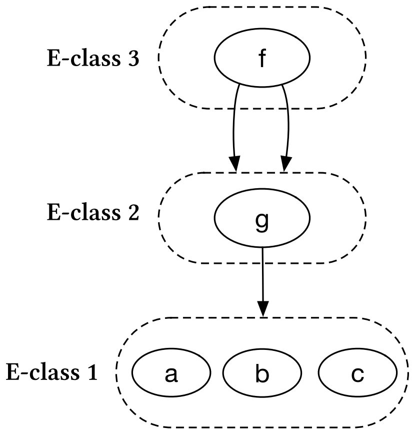
ECTAs: Compact Spaces of Coupled Terms

A general recipe for solving programming problems is to generate a large set of programs, and then pick the highest-scoring one. Since even small bounds admit astronomically many possible programs, doing so requires simultaneously considering many programs with shared commonalities. In that vein, enumerative program synthesis has thrived in the past decade by using version space algebras [107, 135] to compactly represent all programs in an exponentially-large space, while e-graphs have done the same for rewriting-based optimizers [182, 163], deductive synthesis [121], and a certain kind of semantic code search [141]. We ourselves used this idea for code search in our Yogo tool (Section 2.8), which efficiently searches an exponentially-large space of programs equivalent to the input, searching for any that "score highly" by matching the query.

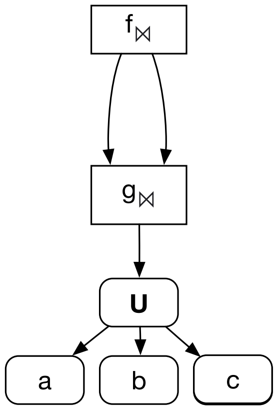
Both VSAs and e-graphs are now known [134] to be equivalent to special cases of tree automata, which have independently experienced a surge of interest in recent years [2, 177, 178]. All three techniques embody a divide-and-conquer approach that thrives when separate portions of a program may be chosen independently. They falter when there are constraints in which portions may appear together.

E-graphs, VSAs, DFTAs: On subterms divided they stand, on subterms united they fall. Consider representing the set of 9 terms $\mathcal{T} = \{f(g(X), g(Y))\}$ where $X, Y \in \{a, b, c\}$. In a rewriting-based optimizer or deductive synthesizer, this might emerge by starting with $f(g(a), g(a))$ and discovering the equalities $a = b = c$. The corresponding e-graph is built by beginning with an AST for $f(g(a), g(a))$ and replacing a with the equivalence class $\{a, b, c\}$, resulting in the e-graph shown in Figure 3-1a. In an inductive synthesis domain like FlashFill [73] or other members of the FlashMeta/PROSE suite [135], this might emerge by discovering that $g(a)$, $g(b)$, and $g(c)$ all give the same value, needed as inputs to f to produce some final goal output. These tools construct a version space algebra (VSA); either bottom-up, by applying g to all constants and grouping by observational equivalence; or top-down, by inverting f, seeking a value for its arguments, and finding that $g(a)$, $g(b)$, and $g(c)$ all suffice. Figure 3-1b gives the resulting VSA. A third, lesser-known, representation is the tree automaton. Figure 3-1c gives the deterministic finite tree automaton (DFTA) for the same set. The DFTA associates each subterm with a state q, determining the state of each term by matching its symbol and states of its children with a set of transitions. Although the three representations are typically constructed differently, the similarity in their final structure is striking; it is immediately apparent, for instance, how nodes of the DFTA correspond to e-classes of the e-graph, and transitions (hyperedges) of the DFTA correspond to the e-graph's nodes. All three function similarly, and it is now known that both VSAs and e-graphs are isomorphic to special cases of tree automata. [134, 93]

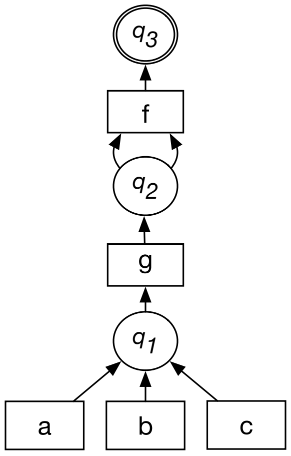
These three approaches all compactly represent this space of 9 terms by exploiting how the choices for $g(X)$ and $g(Y)$ may be made independently. The picture changes, however, when one instead considers the set of 3 terms of the form $\mathcal{U} = \{f(g(X), g(X))\}$, where $X \in \{a, b, c\}$. Now, the e-graph, shown in Figure 3-2, cannot exploit the sharing at all, and can only represent the set by giving all three terms. Though this set is a third the size of $\mathcal{T}$, the representation is nearly thrice as large. The VSA and DFTA behave similarly. This problem occurs in Yogo as well in a different way: while it derives much power from treating equivalent terms



(a) E-graph



(b) Version-space algebra



(c) DFTA. Rectangles represent transitions.

Figure 3-1: Compact representations of $\mathcal{T}$. The numbered DFTA nodes correspond to the numbered e-classes.

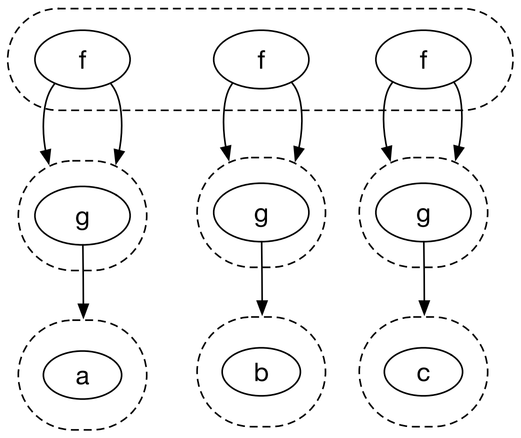


Figure 3-2: E-graph for $\mathcal{U}$.

identically, it is also incapable of expressing a rule like $f(x, x) \Rightarrow g(x)$ where the copies of x must be syntactically identical, not merely equivalent.

ECTAs to the rescue We propose using a new data structure, the *equality-constrained tree automaton* (ECTA), for compactly representing and searching such sets. ECTAs are tree automata extended with *equality constraints.* To construct an ECTA representing the term set $\{f(g(X), g(X))\}$, one first constructs a tree automata representing the set $\{f(g(X), g(Y))\}$, and then adds a constraint implying $X = Y$. Rather than deal with named variables and their well-known complications, ECTAs use a *nameless representation* referring to subterms by paths. In this case, the $X = Y$ constraint is encoded as the path constraint $0.0 = 1.0$ on the root, i.e.: that the term at path 0.0 (first child of the first child, named X in the previous sentence) is equal to the term at path 1.0 (first child of the second child, named Y). The resulting ECTA for $\mathcal{U}$ is given in Figure 3-3, is identical to the DFTA for $\mathcal{T}$, Figure 3-1c, save for the $0.0 = 1.0$ path constraint on the f transition. Even on this tiny example, we see that the ECTA greatly outperforms the e-graph in the amount of sharing obtained. And yet it is still clear how to efficiently (and, for simply-constrained cases like this, in time linear in the number of outputs) enumerate the represented terms, by selecting a value of a, b, or c on the left path and reusing it on the right.

Adding equality constraints expands the set of problems e-graphs/VSAs/tree automata can solve. While these constraints emerge directly when considering the set of terms generated by a nonlinear rewrite system, they can also be used to encode many other kinds of dependencies between terms, such as type unification in polymorphic function applications , and staging constraints in multi-stage programming. Effectively, whereas e-graphs/VSAs/tree automata construct spaces of terms as unions and cross products of smaller spaces, ECTAs can also represent reduced products. Thanks to this expanded vocabulary, ECTAs can express many problems outside the reach of these other techniques, letting more problems be solved by a single optimized ECTA library.

Tree automata have long been used to represent sets of terms in term rewriting [45, 57, 64], and we are not the first to consider adding equality constraints to handle nonlinear rewrites. In fact, in 1995, Dauchet introduced a data structure very similar to our ECTAs called "reduction automata" [46] — and used them to prove the decidability of a certain variant of first-order logic. Other prior work on constrained tree automata (e.g.: [18, 19, 10, 11, 145]) similarly focus on worst-case complexity and decidability results, and we have found no reference to these data structures being used in a practical system in the 30 years since their introduction. Our work, in contrast, approaches them from the view of applications, in the process introducing both low-level optimizations, high-level restrictions, and pragmatic algorithms for reducing the search space and enumerating terms (Section 3.3).

3.1 Overview

We demonstrate performing type-directed synthesis based on polymorphic components, which we dub the **Hoogle+ domain** after the main existing work in this area [81], based on ECTAs and specifically the ECTA library. Prior work encodes a problem into Petri net reachability, searching only a single term at a time, bounds polymorphism, and cannot directly handle partial application. In contrast, the encoding into ECTAs simultaneously searches for all terms, and seamlessly handles arbitrary polymorphism and partial application. Thanks to this ability to directly encode the

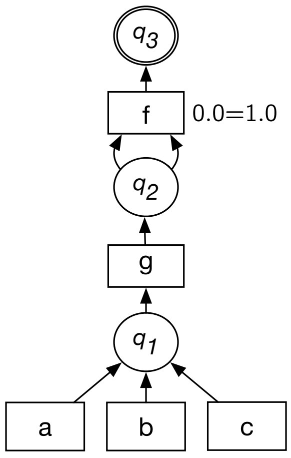


Figure 3-3: ECTA for $\mathcal{U}$.

problem into the ECTA constraint system, our implementation is under 400 lines, and is competitive with the much-larger original HOOGLE+ implementation, as shown in Section 3.4.1.

Using either VSAs, e-graphs, or tree automata, it is easy to express the space of all not-necessarily-well-typed terms. With ECTAs, we can further encode the well-typedness constraint so that a generic fast enumeration procedure will find all well-typed terms in the space. Figure 3-4b depicts on ECTA containing all such well-typed terms; we incrementally describe the ideas used to construct it below. Notice first the application constructor, called `app`, in the topmost hyperedge, which contains two copies of the list of all available components `($)`, `foldro`, `map`, etc as children, indicating that, before applying constraints, the space contains all components applied to any other component. The root node hence will describe all well-typed terms of size 2. Nodes describing all well-typed terms of sizes 3, 4, etc are then built using this node as a potential input, and a node containing all terms upto size k can be built as the union of such nodes.

Our examples below deal extensively with the Haskell application function , which serves as an example of multiple aspects of the ECTA encoding. The (\$) operator (\$) : $(a \to b) \to (a \to b)$ is mostly used in Haskell as syntactic sugar to avoid nested parentheses, so that one can write `f $ g $ h $ x` instead of `f (g (h x))`, but is also useful in its own right, as in a HOOGLE+ benchmark for applying a function n times to a given argument, `foldr ($) x (replicate n g)`.

The basic idea of the encoding is to add an annotation to each term node giving its type, which by convention we place at index 0 of all term (non-type) nodes. Equality constraints on the application constructor `app` then enforce that types match. **The first set of constraints** on the `app` constructor use equalities to enforce that the type of the argument matches the argument type of the function, and that the overall type of the `app` matches the return type of the function. We will explain momentarily what positions 1 of the `app` constructor and 0 of the $(\rightarrow)$ type constructor are used for. These two constraints are then $2.0.1 = 3.0$ (argument type (child 1) of type (child 0) of function (child 2 of the `app`) equals type (child 0) of the argument child (child 3 of the app)), and $0 = 2.0.2$ (overall type of the `app` (child 0) equals return type (child 2) of the type (child 0) of the function argument (child 2 of the `app`)).

There is a problem with the above encoding: what if the component under consideration does not have a function type $(\rightarrow)$, but a different binary type such as `Either`? The constraints mentioned above would be satisfied by the application $(Either 12)3$, which is ill-typed. To express the constraint that the term in function position must have a function type, there must be some extra child on the function type constructor indicating that is a function type, as only these children are visible to the constraint system. Our encoding adds extra hyperedges containing the symbol $(\rightarrow)$ to all function type constructors at position 0, depicted as the small side nodes in Figure 3-4b. There is then an extra (final) constraint on the `app` transitions enforcing that the child in function position indeed has a function type. This is done by adding that same $(\rightarrow)$ symbol at position 1 of the `app` hyperedge, so that the constraint $1 = 2.0.0$ requires that the child in function position indeed have a function type.

We now turn to the handling of polymorphism. Our encoding expresses a polymorphic type such as $(a \rightarrow b) \rightarrow (a \rightarrow b)$ as the type $(\mathsf{Any} \rightarrow \mathsf{Any}) \rightarrow (\mathsf{Any} \rightarrow \mathsf{Any})$ with constraints on the top-level $(\rightarrow)$ that the corresponding Any's must match: $1.1 = 2.1$ (for the "a" variable) and $1.2 = 2.2$ (for the "b" variable). This Any type can then be written as a recursive ECTA node which contains the space of all valid types, depicted in Figure 3-4a. When constrained to be equal to another polymorphic type, such as

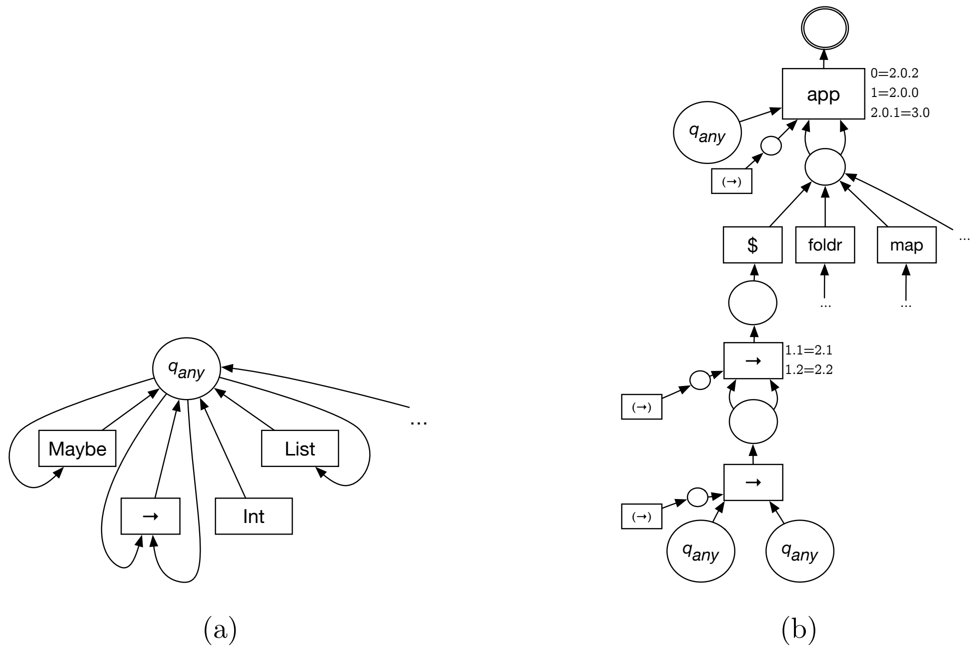


Figure 3-4: (a) Encoding the "Any" type in Haskell as a recursive ECTA node (b) An ECTA for all well-typed Haskell programs of the form $f(g)$, where f and g are drawn from a fixed set of components.

in the application `foldr ($)`, where `foldr` : $(c \to d \to d) \to d \to \text{List } c \to d$, the `app` constraints imply that both the c and d in `foldr` must be equal to $a \to b$, giving rise to the result that **unification is automata intersection**.

This concludes the encoding of the Hoogle+ domain into ECTAs. We have now explained everything in Figure 3-4. To synthesize terms of type $X \to Y \to Z$, one simply constructs a similar ECTA adding terms of types X and Y to the space of available components, adds an extra constraint to the top node restricting the overall return type to Z, and then runs a generic enumeration procedure to synthesize terms of the desired type $X \to Y \to Z$.

The encoding can be refine further with domain knowledge. For instance, we gave an example where (\$) was useful in argument position, `foldr ($) x (replicate n g)`. However, it is never useful in function position, as it is simply the identity function; `($ foldr)` is equivalent to just `foldr`. We can modify the encoding so that (\$) is elided from the list of available components in the child of `app` nodes representing terms in the function-position; doing so greatly reduces the size of the search space.

3.2 Basic Formalism

In this section, we introduce our basic mathematical formalism for ECTAs, along with brief descriptions of their operations.

As explained in Section 3 and elaborated in , there are several variants of tree automata with equality and other constraints, though they have seen scarce application. ECTAs are in fact a restriction of Dauchet's *reduction automata* [46], introduced in 1995 to prove the decidability of first-order logic with a primitive for subsumption of terms. ECTAs are identical to reduction automata except that reduction automata also feature disequality constraints. This restriction is interesting because it admits a *static reduction operation* (Section 3.2.3), which modifies the graph so as to reduce the search space of terms which may satisfy the constraints, and overall gives rise to efficient enumeration.

The formalism of this section is more general than the representation used by the ECTA library. This formalism presents ECTAs as a graph, with arbitrarily-many final states and cycles (so long as the cycles contain no equality constraints). In Section 3.3, we introduce a second formalism which adds additional restrictions, but enables more optimization.

3.2.1 Preliminaries

We first present standard definitions of signatures, terms, paths, the subpath relation, and the prefix-free property. The only departure from other presentations is that symbols do not come with an explicit arity.

We use Σ to denote a *signature*, defined as a set of symbols. If $\mathcal{X}$ is a set of variables, then the *terms* of Σ with variables $\mathcal{X}$ is denoted $\mathcal{T}(\Sigma, \mathcal{X})$, and consists of objects either of the form $x \in \mathcal{X}$ or $s(\overline{t_i})$, where $\overline{t_i}$ is a (possibly-empty) list with each $t_i \in \mathcal{T}(\Sigma, \mathcal{X})$ and s is some "constructor" symbol. We abbreviate the set of closed terms of Σ, $\mathcal{T}(\Sigma, \varnothing)$, as $\mathcal{T}(\Sigma)$. We sometimes abbreviate nullary terms like $a()$ as a.

A *path* $p \in P$ is a list of integers $i_1.i_2.\dots.i_k \in \mathbb{N}^*$. The empty path is denoted ϵ.

The subterm of t at path p, written $t\big|_p$, is inductively defined:

- $t\big|_{\epsilon} = t$

- $s(t_0, \ldots, t_k)\big|_{i.p} = t_i\big|_p$ if $0 \leq i \leq k$, undefined otherwise.

For example, for $t = f(g(a), g(b))$, $t\big|_{0.0} = a$ and $t\big|_{1.0} = b$.

For paths $p_1, p_2 \in P$, if $\exists i_1, \ldots, i_k \in \mathbb{N}^*$ such that $p_1 = p_2.i_1 \ldots i_k$, then p_1 is a **subpath** of p_2. If $k > 0$, p_1 is a **strict subpath** of $p_)$.

A set P of paths is *prefix-free* if there are no $p_1, p_2 \in P$ such that p_1 is a strict subpath of p_2.

3.2.2 Equality-Constrained Tree Automata

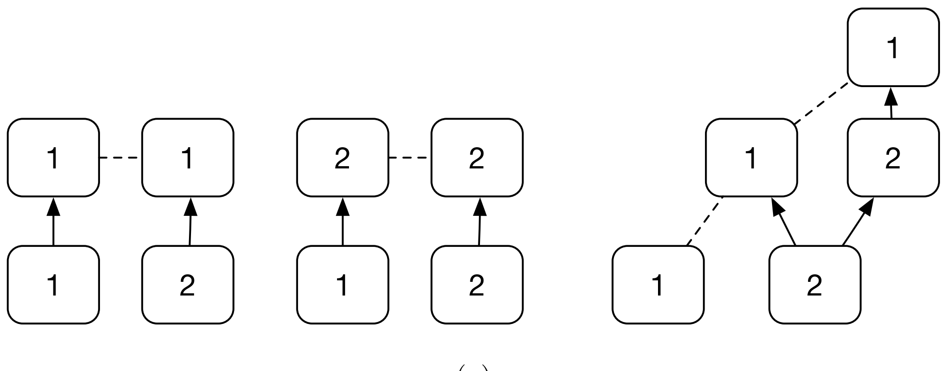
We now build the definition of the main data structure of this chapter, ECTAs. ECTAs are identical to ordinary tree automata as defined in e.g.: [177, 35], except that certain transitions are equipped with a set of constraints called "path equivalence classes" (PECs), each restricting a set of subterms to be mutually equal. We first present PECs.

Definition 3.2.1 (Path Equivalence Classes (PECs))**.** *A path equivalence class $c \in$ $\mathsf{PEC} \subseteq \mathbb{P}(P)$, is a prefix-free set of paths. We write a PEC $\{p_1, p_2, \ldots, p_k\}$ as $\{p_1 = p_2 = \cdots = p_k\}$.*

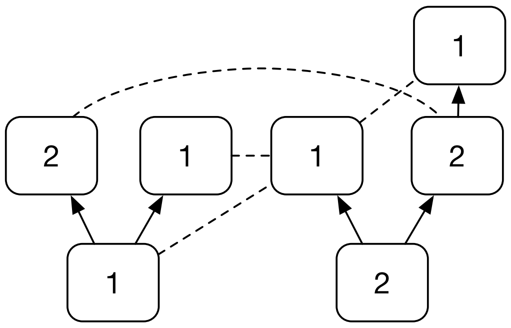
Definition 3.2.2 (Satisfaction of a PEC, Value at a PEC)**.** *A path equivalence class $c = \{p_1 = \cdots = p_k\}$ is satisfied on term t if there is some t' such that, $\forall p_i \in c, t\big|_{p_i} = t'$. We write $c(t)$ if this condition holds, and $t\big|_c$ to denote this unique t'.*

Definition 3.2.3 (Path Constraint Sets, Satisfaction, Consistency)**.** *A path constraint set $C = \{c_1, \ldots, c_k\}$ is a set of path equivalence classes. Term t satisfies path constraint set C, written $C(t)$, if $\forall c \in C, c(t)$. If there exists a t such that $C(t)$, then C is consistent; otherwise, it is inconsistent.*

An example of an inconsistent path constraint set is $\{\{0 = 1.0\}, \{0.0 = 1\}\}$, which corresponds to unifying e.g.: the expression $f(A, g(A))$ with $f(g(B), B)$, which generates the contradictory constraints $A = g(B)$ and $B = g(A)$.



(a)



(b)

Figure 3-5: (a) E-graphs for the PECs $\{1.1 = 2.1\}$, $\{1.2 = 2.2\}$, and $\{1 = 2.1 = 2.2.1\}$, corresponding to the types $(\$) : (a \to b) \to (a \to b)$ and pairToList : $a \to a \to [a]$ (b) The congruence-closure of the e-graphs in (a), showing that it contains the contradictory PEC $\{1.1 = 1 = 2.1 = 2.2.1\}$

Although ECTAs generalize e-graphs, they actually contain e-graphs as a subcomponent: a path constraint set can be *completed* using the congruence closure algorithm used in e-graphs [123], meaning that the constituent PECs are merged and extended to explicitly contain all equalities implied by the path constraint set. Figure 3-5 gives an example set of PECs as e-graphs, and illustrates using congruence closure to compute the combined PEC set, finding an inconsistency.

It turns out that a path constraint set is consistent if and only if each PEC is prefix-free after completion.

Theorem 3.2.4. *Let $C = \overline{c_i}$ be a completed set of PECs. Then C is inconsistent if and only if one of the c_i is not prefix-free.*

Proof. It is easy to see that a non-prefix-free PEC is inconsistent.

For the reverse direction, we shall show that if each of the c_i are prefix-free, then C is consistent. Consider $c_1, c_2 \in C$, and define $c_1 \sqsubseteq c_2$ if $\exists p_1 \in c_1, p_2 \in c_2$ such that p_1 is a subpath of p_2. We show that $\sqsubseteq$ is a partial order. Reflexivity and transitivity are trivial. For anti-symmetry: suppose $p_1, p_3 \in c_1$ and $p_2, p_4 \in c_2$ with p_1 a subpath of p_2, p_4 a subpath of p_3. If $c_1 = c_2$, we are done; otherwise, we must have $p_1 \neq p_2$ and $p_3 \neq p_4$ by completeness of C. We can thus write $p_2 = p_1.p_2'$, $p_3 = p_4.p_3'$ for some $p_2', p_3' \neq \epsilon$. By completeness of C, it follows that c_1 contains $p_1 = p_3 = p_4.p_3' = p_2.p_3' = p_1.p_2'.p_3'$, which would mean that c_1 is not prefix-free. Thus, $\sqsubseteq$ is a partial order.

Because C can be topologically sorted by $\sqsubseteq$, it is trivially possible to construct a term t such that $C(t)$ by inductively choosing a term for each successive minimal c_i. Thus, if each of the c_i are prefix-free, C is consistent. □

Having developed path constraint sets, we now define ECTAs. We also introduce the restriction against constraints on cycles, which is necessary for the fast enumeration algorithm of Section 3.3.3 to terminate, and, indeed, for decidability.

Definition 3.2.5 (Equality-Constrained Tree Automata (ECTA)). *We define an equality-constrained tree automaton $\mathcal{G} \in \mathsf{ECTA}$ to be a tuple $(\mathcal{Q}, \Sigma, \mathcal{Q}_f, \Delta)$ where $\mathcal{Q}$ is a set of states, Σ a signature, $\mathcal{Q}_f \subseteq \mathcal{Q}$ a set of final states, and $\Delta \in \Sigma \times \mathcal{Q}^* \times \mathbb{P}(PEC) \times \mathcal{Q}$ is a set of constrained transitions (a.k.a.: hyperedges). Further, there may be no cycle containing an equality constraint, i.e.: if there is a sequence of transitions $s_1(\dots, q_0, \dots) \xrightarrow{C_1} q_1, s_2(\dots, q_1, \dots) \xrightarrow{C_2} q_2, \dots, s_k(\dots, q_{k-1}, \dots) \xrightarrow{C_k} q_0$, then all of the C_i must be equal to $\varnothing$.*

Remark 3.2.6. *The restriction against equality-constraints on a cycle is also present in Dauchet's reduction automata [46] as necessary to make finding a represented term decidable, although it is phrased there in terms of an ordering on states.*

Example 3.2.7. *The ECTA of Figure 3-3 is given by $(\mathcal{Q}, \Sigma, \mathcal{Q}_f, \Delta)$ where:*

$$
\begin{aligned}
&\mathcal{Q} = \{q_1, q_2, q_3\} \\
&\Sigma = \{a, b, c, f, g\} \\
&\mathcal{Q}_f = \{q_3\} \\
&\Delta = \left\{ a() \to q_1, b() \to q_1, c() \to q_1, g(q_1) \to q_2, f(q_2, q_2) \xrightarrow{\{\{0.0=1.0\}\}} q_3 \right\}
\end{aligned}
$$

A term $t = s(\overline{t_i})$ transitions to a state q if there is a transition $s(\overline{q_i}) \xrightarrow{C} q$, each t_i transitions to $\overline{q_i}$, and $C(t)$. The denotation of an ECTA is then the set of terms which transition to a final state.

A **run** is a mapping from the positions of a term to the states of an ECTA which is compatible with the transition relation. The **skeleton** of an ECTA is the tree automata obtained from an ECTA G by removing all path constraints. A **spurious run** of G is a run of the skeleton of G which is not a run of G.

Analogously to nondeterministic string automata, union of ECTAs may be defined as a union of states and transitions, while intersection may be defined using a product construction, in which the constraint sets of intersected edges are conjoined. We defer detailed discussion of these topics to the development of our optimized/restricted formalism in Section 3.3.

3.2.3 Static Reduction

ECTAs are a type of constrained automata where the only available constraints are path equalities. All other kinds of constraints must be encoded into equalities. This restriction comes with several advantages for fast enumeration. We now present the first: static reduction, or using automata intersection to reduce the number of spurious runs.

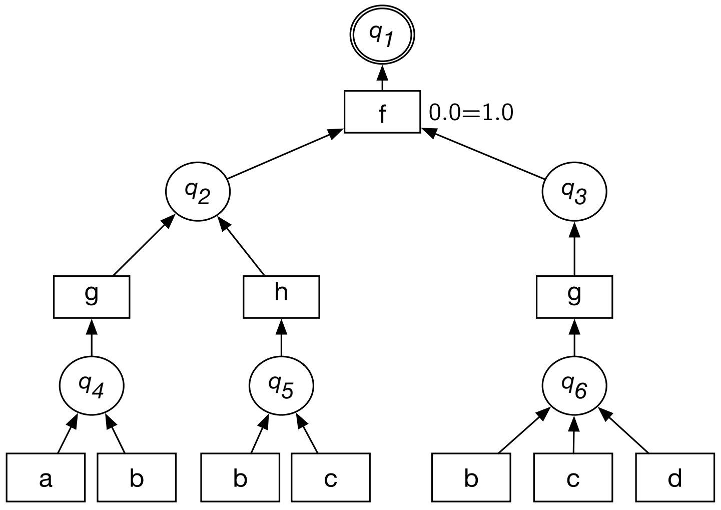
Consider Figure 3-6. Though it represents only 3 terms, the skeleton of the ECTA in Figure 3-6a admits 12 runs. The ECTA in Figure 3-6b, in contrast, represents the same 3 terms, but its skeleton admits only 6 runs. This suggests that enumeration will be faster for the ECTA in Figure 3-6b. This ECTA is the output of *static reduction* of the $0.0 = 1.0$ constraint. Static reduction of a PEC removes all transitions at the end of a path in the constraint except those which match a transition along the other paths in the constraint.

For an edge e in an ECTA, the the set of nodes from e at path $p = i_1 \ldots i_k$, $e\big|_p$, is defined as the set of nodes obtained from first finding the i_1th child of e n_1, then taking the i_2th children of edges into n_1, etc. A PEC $c = \{p_1 = \cdots = p_k\}$, is then reduced by intersecting each node at p_1 with the intersection of the nodes at $p_2, \ldots, p_k$, and repeating likewise for the nodes at p_2, etc. We give a formal definition in Section 3.3.2, as the algorithm requires stating a definition of intersecting ECTA states (as opposed to entire ECTAs), which we have not done here.

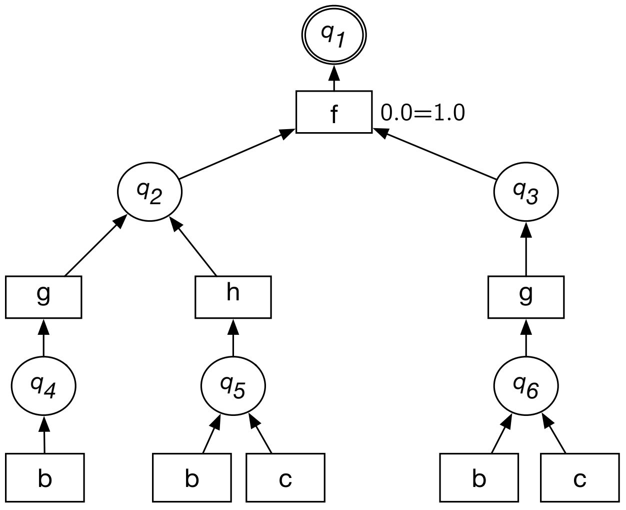
3.3 Optimized Formalism and Implementation

The formalism of Section 3.2 is simple and slow. It gives a representation as an arbitrary graph, which hinders the most important optimization (memoization), and it suggests no enumeration algorithm other than brute-force. Turning this into an ECTA library capable of strong performance on a wide range of applications will require more than low-level optimizations. In this section, we explain the algorithmic, representational, and optimization insights behind the ECTA library, culminating in the fast, flexible enumeration algorithm of Section 3.3.3,

We have implemented this optimized formalism along with lower-level optimizations in our ECTA library. ECTA comes with implementations of the applications in Section 3.4, as well as a program which reduces boolean satisfiability to ECTA enumeration, which functions both as a proof of NP-hardness and a testbed for observing its behavior when faced with a high ratio of constraints to nodes. It contains approximately 3000 lines of implementation and 600 lines of tests. The core of ECTA is approximately 1900 lines, including its processing of ECTAs, path constraints, and terms, and the GraphViz-based visualizer for ECTAs. Slightly over 1300 lines are spent on applications, and the rest on its miscellaneous libraries.



(a)



(b)

Figure 3-6: (a) A non-minimal ECTA representing $\{f(g(b), g(b)), f(h(b), g(b)), f(h(c), g(c))\}$ (b) This same ECTA after reducing the constraint $0.0 = 1.0$. In particular, every leaf is used in at least one term.

3.3.1 Pseudo-Tree ECTAs and the Globally-Unique Recursion Restriction

We first address the representation of ECTAs used by the ECTA library. Our goal is to produce a representation amenable to memoization and sharing, which requires that operations be defined on portions of an ECTA rather than an entire ECTA. Though ECTAs are graphs, this is much more easily done with a DAG or tree structure. We thus introduce pseudo-tree ECTAs, which presents an ECTA as an AST, along with an additional restriction that enables high-performance treatment of recursive references (i.e.: back-edges).

How does one treat a graph as a tree? The first step is to choose a root; since every ECTA is equivalent to an ECTA with only one final state, this step is trivial. After that, following edges away from the root will immediately give a tree-like structure if the graph is acyclic; this is well-known, and was exploited by e.g.: previous implementations of version space algebras [135]. Oliveira and Cook's "structured graph" representation [127] goes further in allowing arbitrary graphs to be manipulated through a tree-like interface. It does this by using (parametric) higher-order abstract syntax [27] to represent explicit recursive μ nodes in graphs, so that e.g.: the graph $A \to B \to C \to A$ would be represented as the pseudo-tree $\mathrm{Mu}(\lambda x.A(B(C(x))))$.

We considered using the structured graph representation, but decided against it for performance reasons: without the ability to inline functions at runtime, every modification to the body of a Mu node would be represented in memory as a chain of function compositions which must be evaluated anew each time a concrete node must be extracted. In particular, we predicted that repeatedly intersecting ECTAs would create such composition chains of arbitrary length. A related alternative solution is to have explicit Mu nodes whose bodies are ECTAs containing bound variables, replacing the problem of performance with that of dealing with binders, which is especially difficult when subterms are shared.[1] These and other approaches share the common theme of treating a graph as a DAG where certain nodes are marked as

[1]An algorithm for hashing modulo alpha-equivalence [115] was published during the development of ECTA, which suggests a new approach for sharing subterms with named variables.

recursive Mu nodes that may be the target of backedges from their child subgraphs.

However, we realized during development that most applications under consideration only deal with finite spaces of terms and hence do not require recursive nodes at all. The main application we conceived where recursive nodes would be useful was the Hoogle+ domain, where a polymorphic variable may be instantiated to an arbitrary type, and hence a recursive node representing an unrestricted arbitrary type would be useful. This inspired us to introduce the **globally-unique recursion restriction**: in any execution containing a sequence of ECTA operations, the combined ECTAs must have at most one globally unique recursive node. While this may sound unduly restrictive at first glance, it actually means that an application may only enumerate from one distinct infinite space of terms along with many finite spaces, which is effectively a very weak restriction.

One caveat is that, in the case where the body of the globally-unique Mu node contains multiple nodes, the intersection algorithm of Section 3.3.2 must use least-fixed-point semantics to avoid infinite unrolling. However, the encoding of an arbitrary term in a DSL requires at most one distinct node per sort. (There would be, in contrast, many hyperedges, one per constructor.) As none of the potential applications we have encountered involve a multi-sorted DSL, we do not presently foresee this case actually arising in applications, and thus we have not implemented this case at time of writing.

Definition 3.3.1 (Pseudo-Tree ECTA)**.** *A pseudo-tree ECTA is a DAG N given by the grammar*

$$N ::= \mathit{Node}(\overline{E}) \mid \mathit{Mu}(N) \mid \mathit{Rec} \qquad \text{(Nodes)}$$

$$E ::= \mathit{Edge}(\mathsf{symbol}, \overline{N}, \overline{\mathsf{PEC}}) \qquad \text{(Edges)}$$

where subterms have maximal sharing, and subject to the constraints that

- *There is at most one globally-unique Mu node: If $Mu(N_1)$ and $Mu(N_2)$ appear* *in any two ECTAs under consideration, then $N_1 = N_2$.*

- *All Rec nodes are descendants of Mu nodes.*
- *No constraints appear under a Mu node: if $Edge(s, \overline{n}, \overline{c})$ is a descendant of a Mu node, then $\overline{c} = \epsilon$.*
- *No constraints on nodes under a Mu node: For any edge $e = Edge(s, \overline{n}, \overline{c})$, if $\overline{c}$ contains the path p, then no node at $e\big|_p$ may be a descendant of a Mu node.*

Their denotation is:

$$\begin{aligned}
[\![\cdot]\!]^N &: N \to \mathbb{P}(\mathsf{Term}) \\
[\![Node(\overline{e_i})]\!]^N &= \bigcup [\![e_i]\!] \\
[\![Mu(n)]\!]^N &= [\![[Mu(n)/Rec]n]\!]
\end{aligned}$$

$$\begin{aligned}
[\![\cdot]\!]^E &: E \to \mathbb{P}(\mathsf{Term}) \\
[\![Edge(s, \overline{n_i}, \overline{c_j})]\!]^E &= \left\{ s(\overline{t_i}) \middle| t_i \in [\![n_i]\!], \forall c_j . c_j(s(\overline{t_i})) \right\}
\end{aligned}$$

where $[Mu(n)/Rec]n$ denotes substituting Mun for Rec in all descendants of n.

Optimizations

In the ECTA library, all nodes, hyperedges, and symbols are shared by hash-consing. After finding existing Haskell hash-consing libraries insufficiently performant, we built our own using mutable hashtables. To improve sharing, each node is normalized by sorting its children and removing duplicates. Path equivalence classes are represented using a trie. The restriction that no path constraint may refer to a term under a Mu node is not strictly enforced, but rather implemented by lazy unfolding.

3.3.2 ECTA Operations: Union, Intersection, and Reduction

On top of the ease of memoization, tree-like data structures tend to admit simple recursive algorithms. We present definitions of union, intersection, and static reduction of pseudo-tree ECTAs.

Union

The union of two pseudo-tree ECTA nodes is trivial: a node containing the edges of both.

Definition 3.3.2 (Union). *Let $n_1 = \mathit{Node}(E_1), n_2 = \mathit{Node}(E_2)$. Then define*

$$n_1 \cup n_2 = \mathit{Node}(E_1 \cup E_2)$$

Intersection

Let Closure(C) denote the congruence closure of path constraint sets as described in Section 3.2, with Closure(C) $= \perp$ if C is found to be inconsistent using the criterion of Theorem 3.2.4. We now present a mutually-recursive definition of the intersection of pseudo-tree ECTA nodes and edges.

Definition 3.3.3 (Intersection). *Let $e_1 = \mathit{Edge}(s_1, \overline{n_{1i}}, C_1)$, $e_2 = \mathit{Edge}(s_2, \overline{n_{2i}}, C_2)$. Then we define $e_1 \cap e_2$ of type $E \sqcup \mathit{UndefinedEdge}$ as*

$$e_1 \cap e_2 = \begin{cases} \mathit{Edge}(s_1, \overline{n_{1i} \cap n_{2i}}, \mathit{Closure}(C_1 \cup C_2)) & s_1 = s_2 \land \mathit{Closure}(C_1 \cup C_2) \neq \perp \\ \mathit{UndefinedEdge} & \mathit{otherwise} \end{cases}$$

Let $n_1 = \mathit{Node}(E_1)$, $n_2 = \mathit{Node}(E_2)$. Then

$$n_1 \cap n_2 = \mathit{Node}\left(\left\{e_1 \cap e_2 \middle| e_1 \in E_1, e_2 \in e_2, e_1 \cap e_2 \neq \mathit{UndefinedEdge}\right\}\right)$$

Optimizations The congruence closure operation is memoized; congruence closure will typically be performed on very few distinct inputs during an execution. The definition of $n_1 \cap n_2$ effectively performs a nested-loop join, performing $|E_1| * |E_2|$ edge intersections; the ECTA implementation performs a simple hash join instead. All intersections are memoized; nodes/edges are ordered to improve the cache hit rate.

Reduction

An algorithm for static reduction is particularly easy to give on pseudo-tree ECTAs.

Definition 3.3.4 ((Intersecting) States at a path in an ECTA)**.** *Let $n = Node(\overline{e_i})$, where $e_i = Edge(s_i, \overline{n_{ij}}, C_i)$ be a pseudo-tree ECTA node. Define the states at path p starting at n as*

$$\begin{aligned} n\big|_{\epsilon} &= \{q\} \\ n\big|_{j.p} &= \bigcup n_{ij}\big|_{p} \end{aligned}$$

The states at path p of n intersected with state n', $n\big|_{p}^{\cap n'}$, is then defined:

$$\begin{aligned} n\big|_{\epsilon}^{\cap n'} &= n \cap n' \\ n\big|_{j.p}^{\cap n'} &= Node\left(\left\{ Edge(s_i, \{n_{i1}, \ldots n_{i(j-1)}, n_{ij}\big|_{p}^{\cap n'}, n_{i(j+1)}, \ldots, n_{ik}\}, C_i) \middle| e_{ij} \text{ exists}\right\}\right) \end{aligned}$$

For non-empty paths p, we can define similar values for an edge $e = Edge(s, \overline{n_i}, C)$.

$$\begin{aligned} e\big|_{j.p} &= n_j\big|_{p} \\ e\big|_{j.p}^{\cap n'} &= Edge(s, \{n_1, \ldots, n_{j-1}, n_j\big|_{p}^{\cap n'}, n_{j+1}, \ldots, n_k\}, C) \end{aligned}$$

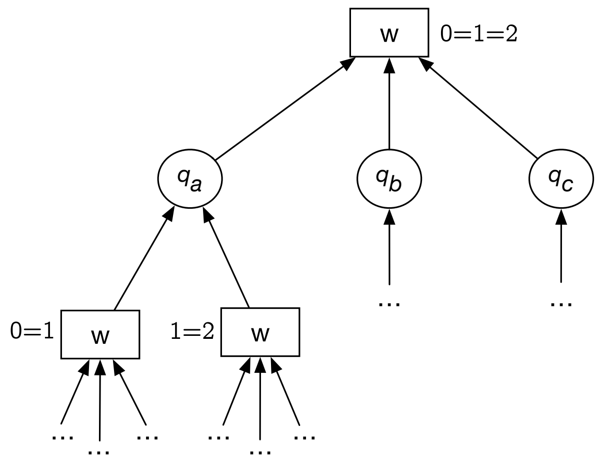


Figure 3-7

With this new vocabulary, we give the algorithm for static reduction:

Definition 3.3.5 (Reduction)**.** *First, let PEC* $c = \{p_1 = \cdots = p_k\}$*. Then let* $c_{\bar{i}} = \{p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_k\}$*. Then let* e *be an edge, and define*

$$m_i = \bigcap_{p \in c_{\bar{i}}} e\big|_p$$

Then define

$$\mathit{reduce}(e, c) = e\big|_{p_1}^{\cap m_1} \ldots \big|_{p_k}^{\cap m_k}$$

For example, running this definition on the f edge of Figure 3-6a produces Figure 3-6b.

An Optimization Not Taken: Avoiding "Junk Edges" Consider statically reducing the root w transition of Figure 3-7. This involves intersecting each of the two edges under q_a by $q_b \cap q_c$, each edge under q_b by $q_a \cap q_c$, and each edge under q_c by $q_a \cap q_b$. It would be semantically equivalent to intersect all three of them instead by $q_a \cap q_b \cap q_c$, which reduces the number of distinct intersections that must be performed. Counterintuitively, doing so actually degrades performance.

The reason is that intersecting $q_a \cap q_a$ produces three child edges: one with constraint $\{0 = 1\}$, one with constraint $\{1 = 2\}$, and a "junk edge" with constraint $\{0 = 1 = 2\}$. Although this last edge is redundant with the first two, detecting and eliminating these redundant edges is actually rather expensive. Particularly for the Hoogle+ domain (Section 3.4.1), where a large proportion of hyperedges have the same symbol (the $\rightarrow$ function type symbol) with different constraints, the creation of a redundant edge can proliferate into many more redundant edges upon reducing other constraints. Thus, ECTA takes the "slow" route of computing $q_a \cap q_b$, $q_a \cap q_c$, and $q_b \cap q_c$ separately.

3.3.3 Flexible, Fast Enumeration

ECTAs were designed as way to compactly represent large sets of programs and efficiently search them, and we now turn to efficient search. In spite of the expressiveness of the constraint language, as shown in Section 3.4, the use of only equality constraints offers three advantages for finding satisfying terms quickly: (1) the use of static reduction to reduce the number of spurious runs (2) the ability to use intersection during enumeration to discover early that a choice leads to no satisfying terms, and (3) the ability to choose the same term along multiple paths as to constructively satisfy an equality constraint. In contrast, for reduction automata [46], which are essentially ECTAs with disequality constraints, none of these are true.

Nonetheless, different constraints may affect the same subterms, and thus which portions of subterms are chosen first during enumeration can have a substantial affect on the amount of time needed to find a satisfying term or discover that a choice is contradictory, Given that we managed to encode SAT into ECTAs, there is little prospect for anything better than a heuristic approach to this ordering. Thus, inspired by presentations of DPLL(T) and Knuth-Bendix completion [8, 125], instead of providing a fixed algorithm, we provide a set of branching rules such that the successive application of rules in any order is guaranteed to terminate and yield every satisfying term — or, in cases with infinitely many satisfying terms, to yield representations containing unconstrained tree automata from which all such terms may be trivially generated. This approach permits applications to use a custom enumeration ordering, though ECTA also offers a default ordering.

Enumeration State

Each node in an ECTA is a set of alternatives to be chosen from; as each choice is made, the graph gradually crystallizes into a term. We begin by defining partially-enumerated terms, consisting of a tree fragment at the top, with as-yet-unenumerated ECTAs among the branches. These unenumerated ECTAs may not always be enumerated independently, but may be constrained by path constraints on their ancestors to be equal to cousins. We thus define the enumeration state as a map from named variables to partially-enumerated terms, so that a subnode may be enumerated once and included by named-variable reference into multiple positions in a tree. Unenumerated ECTA nodes may also contain fragments of constraints inherited from their ancestors, each paired with a named variable so that choices of constrained subterms may be shared with other branches.

Definition 3.3.6 (Partially-Enumerated Term, Enumeration State, Fully Enumerated, Denotation)**.** *Let Var be a countably infinite set of variables, and let $v_\top \in \mathsf{Var}$ a special initial variable. A partially enumerated term is given by the grammar:*

$$\mathsf{PTerm} ::= T(symbol, \overline{\mathsf{PTerm}}) \mid Unenumerated(N, \overline{(\mathsf{PEC}, \mathsf{Var})}) \mid Ref(\mathsf{Var})$$

An enumeration state is then defined as any σ, where

$$\sigma : \mathsf{Var} \rightharpoonup \mathsf{PTerm}$$

and all $v \in dom(\sigma)$ are reachable, defined:

- *$v_\top$ is reachable.*
- *If v is reachable, then all variables referenced in $\sigma(v)$ are reachable.*

$$
\begin{aligned}
[\![\cdot]\!]^{\mathrm{PTerm}}_{\sigma} &: \mathsf{PTerm} \times (\mathsf{Var} \rightharpoonup \mathsf{PTerm}) \to \mathbb{P}(\mathsf{Term} \times (\mathsf{Var} \to \mathsf{Term}))\\
[\![\mathrm{T}(s, \overline{pt_i})]\!]^{\mathrm{PTerm}}_{\sigma} &= \Big\{(s(\overline{t_i}), \rho)\Big|(t_i, \rho_i) \in [\![pt_i]\!]^{\mathrm{PTerm}}_{\sigma},\\
&\qquad \rho \text{ arbitrary}, \forall j \forall v, \rho(v) = \rho_j(v)\Big\}\\
[\![\mathrm{Unenumerated}(n, \overline{(c_i, v_i)})]\!]^{\mathrm{PTerm}}_{\sigma} &= \Big\{(t, \rho)\Big|\rho \text{ arbitrary}, t \in [\![n]\!]^{\mathrm{N}}, c_i(t), t\big|_{c_i} = \rho(v_i)\Big\}\\
[\![\mathrm{Ref}(v)]\!]^{\mathrm{PTerm}}_{\sigma} &= \Big\{(t, \rho)\Big|\rho \text{ arbitrary}, \rho(v) = t\Big\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\sigma]\!]^{\mathrm{ES}} &: (\mathsf{Var} \rightharpoonup \mathsf{PTerm}) \to \mathbb{P}(\mathsf{Var} \to \mathsf{Term})\\
[\![\sigma]\!]^{\mathrm{ES}} &= \Big\{\rho\big|_{\mathrm{dom}(\sigma)}\Big|(t_v, \rho) \in [\![\sigma(v)]\!]^{\mathrm{PTerm}}_{\sigma}, \rho(v) = t_v\Big\}
\end{aligned}
$$

Figure 3-8

*An enumeration state σ is **fully enumerated** if, for all $v \in dom(\sigma)$, no path constraints appear in $\sigma(v)$.*

We give an impredicative definition of the denotation of partially-enumerated terms and enumeration states. The denotation of a partially-enumerated term within an enumeration state is defined as the pair of a term compatible with the PTerm, and any from a large universe of states which matches the constraints imposed by the choices made in enumerating this term. The denotation of an enumeration state σ will then be given by intersecting the states compatible with its components and restricting to the domain of σ (written $\rho\big|_{dom(\sigma)}$).

Figure 3-8 gives the denotation of enumeration states.

Example 3.3.7. *Let $v_\top = v_0$ and consider the enumeration state*

$$
\begin{aligned}
\sigma = [&v_0 \mapsto T(\text{“}f\text{''}, Unenumerated(Node(Edge(\text{“}a\text{''}, Edge(\text{“}b\text{''})), (\{\epsilon\}, v_1)), Ref(v_1)),\\
&v_1 \mapsto Unenumerated(Node(Edge(\text{“}a\text{''}), Edge(\text{“}b\text{''})))]
\end{aligned}
$$

Then

$$[\![\sigma(v_1)]\!]^{PTerm}_{\sigma} = \bigcup_{\rho} \{(a(), \rho), (b, \rho)\}$$

, where the union is over all $\rho : \mathsf{Var} \to \mathsf{Term}$*, and*

$$[\![\sigma(v_0)]\!]^{PTerm}_{\sigma} = \Big(\bigcup_{\rho \big| \rho(v_1)=a()} \{(f(a(), \rho(v_1)), \rho)\} \Big) \cup \Big(\bigcup_{\rho \big| \rho(v_1)=b()} (f(b(), \rho(v_1)), \rho)\} \Big)$$

, which reduces to the terms $f(a(), a())$ *and* $f(b(), b())$ *with* $\rho(v_1)$ *being* $a()$ *or* $b()$ *accordingly. This yields the final denotation*

$$[\![\sigma]\!]^{ES} = \{[v_0 \mapsto f(a(), a()), v_1 \mapsto a()], [v_0 \mapsto f(b(), b()), v_1 \mapsto b()]\}$$

Enumeration Algorithm and Rules

We now present the enumeration algorithm. To enumerate the ECTA with root n, the algorithm first creates the initial partial enumeration state

$$\sigma = [v_\top \mapsto \text{Unenumerated(n)}]$$

, and then repeatedly applies one of the following two rules:

Choose Take an unenumerated node, and create one branch for each potential choice.

Suspend Replace an unenumerated node with a reference to a variable.

Each rule breaks down an unenumerated node in some manner, bringing the state closer to being fully-enumerated. These rules are designed and constrained so that any schedule for applying them to nodes yields a complete and correct enumeration procedure. Our implementation in the ecta library provides a default ordering — depth-first, left to right. But different orderings can result in drastically different performance (explained in Remark 3.3.13 below), and thus it is valuable to provide a general framework in place of a fixed algorithm.

We shall first give an example of these rules in action in the Hoogle+ domain, and then present the formal definitions of CHOOSE and SUSPEND, followed by proofs of correctness.

Example 3.3.8. *Figure 3-9 shows the state of enumerating the ECTA in Figure 3-4b after three rule applications, selecting that the root of the generated term to be* app *(the only choice) and suspending child* 0 *(the type annotation) and child* 1 *(the* $(\to)$ *constant used to match against function-type annotations). Figure 3-10 gives this same example after completely enumerating* $V0$*, suspending all subterms touched by equaality constraints. In doing so, it has selected the program* (\$ \$)*; the remaining portions to be enumerated correspond to the instantiated types of each subprogram. Showing these are nonempty is equivalent to showing that their relevant types are unifiable.*

Note how, after completing this "unification check" (specifically, showing the non-emptiness of $V1, V2, V3$ *by enumerating a single value), one will be able to obtain the final term* (\$ \$) *by inspecting V0, without constructing the type annotations. The algorithm will also terminate without enumerating any of the* q_{any} *nodes, which may take on infinitely many values. This is equivalent to inferring only the most general type for a term.*

Definition of CHOOSE CHOOSE consists of a local nondeterministic rewrite rule that runs on unenumerated nodes, which we then lift to run on an entire partial enumeration state. The core rule is defined

$$\frac{\begin{array}{cc} \forall i, \epsilon \notin c_i & \exists i, e_i = \mathrm{Edge}(s, \overline{n_j}, \overline{d_k}) \\ \overline{w_k = \mathrm{freshVar}()} & C = \overline{(c_i, v_i)} \cup \overline{(d_k, w_k)} \end{array}}{\mathrm{Unenumerated}(\mathrm{Node}(\overline{e_i}), \overline{(c_i, v_i)}) \to \mathrm{T}(s, \overline{\mathrm{Unenumerated}(n_j, \mathrm{desc}(j, C))})}\ \mathit{ChooseNode}$$

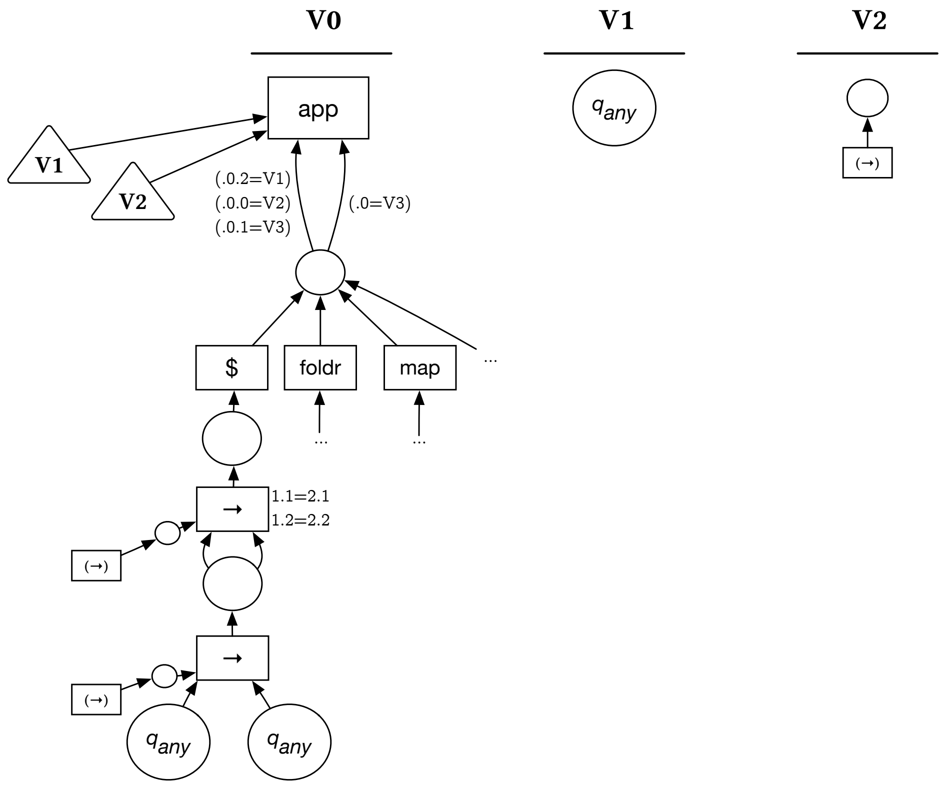


Figure 3-9: The ECTA of Figure 3-4b after applying the CHOOSE rule to the top node and the SUSPEND rules to its first two children. Triangles represent Ref nodes. The arrows marked with path/variable pairs on the app term fragment represent Unenumerated nodes.

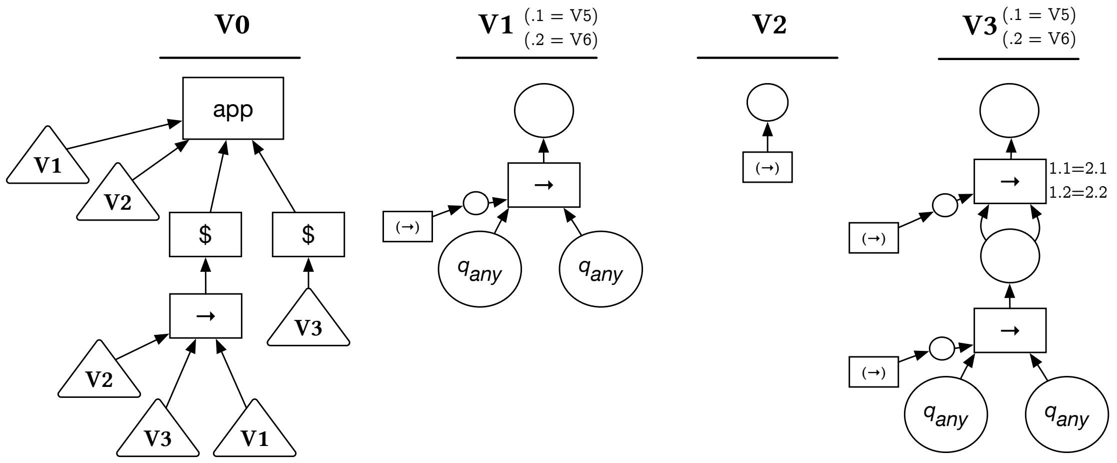


Figure 3-10: The example of Figure 3-9 after fully enumerating V0. The path/variable pairs on V1 and V3 indicate that the value at this variable is an Unenumerated node.

The first premise on Choose states that this rule cannot run if one of the constraints inherited from parents covers the current node, as this constraint would otherwise be lost. The next two premises nondeterministically pick an edge within the enumerated node, The last premise defines the metavariable C as a temporary container of all constraint/variable pairs, both those inherited from parents and those generated by the enumerated edge. Finally, the Choose rule rewrites the unenumerated node to a partially-enumerated node with the symbol of the chosen edge, and whose children are unenumerated nodes corresponding to each child of said edge, with constraints updated appropriately to descend into a child at that index. The latter is accomplished using the desc function, defined

$$\mathrm{desc}(i, \overline{c_j, v_j}) = \left\{ (c'_k, v_k) \middle| c'_k = \left\{ p \middle| i.p \in c_k \right\}, c'_k \neq \varnothing \right\}$$

We now give the final Choose rule, which runs the ChooseNode rule within a partial enumeration state, subject to restrictions

$$\frac{\sigma(v) = E[pt] \quad (\sigma(v) \neq pt) \vee (\forall w, \sigma(w) \text{ does not mention } v) \quad pt \rightarrow pt'}{\sigma \rightarrow \sigma[v \mapsto E[pt']]} \; \mathit{Choose}$$

Example 3.3.9. *Consider the partially-enumerated term*

$$pt = \mathit{Unenumerated}(\mathit{Node}(\mathit{Edge}(\text{“}f'', n_1, n_2, \{0 = 1.0\})), (\{1.1\}, v_2))$$

for some nodes n_1, n_2. Then ChooseNode*(pt) has one possible result, namely*

$$T(\text{“}f'', \mathit{Unenumerated}(n_1, (\{\epsilon\}, v_3)), \mathit{Unenumerated}(n_2, (\{0\}, v_3), (\{1\}, v_2)))$$

Definition of Suspend The Suspend rule consists of two phases: First, an Unenemurated node pt containing ECTA node n is replaced with a reference to another component of σ, and its constraints are collected. Second, σ is updated to set this new component to a copy of *pt* if not already present, or, if it is, to intersect that component with n. In doing so, if the root of *pt* is simultaneously constrained to be equal to multiple distinct variables, these variables are merged, and all components of σ are updated to refer to only the merged variable.

The SUSPEND rule itself chooses an unenumerated node *pt* in some $\sigma(v)$, and then delegates to two helper functions: SUSPENDVAR and MERGEVAR. SUSPENDVAR returns a 5-tuple of the partially-enumerated node with which to replace *pt*, the ECTA node in its body, the variable at which to place the new component, the set of variables that must be equal to this component, and the set of path constraints on children of n. SUSPEND updates σ to replace *pt* with the Ref node before invoking MERGEVARS, which updates it into the final σ'.

$$\frac{\sigma(v) = E[pt] \qquad \textsc{SuspendVar}(pt) = (pt', n, v', V, C)}{\sigma \to \textsc{MergeVars}(n, v', V \cap \mathrm{dom}(\sigma), V, C, \sigma[v \mapsto E[pt']])}\ \textit{Suspend}$$

$$\frac{V = \left\{v_i \middle| c_i = \{\epsilon\}\right\} \quad C = \left\{(c_i, v_i) \middle| c_i \neq \{\epsilon\}\right\} \quad v \in V}{\textsc{SuspendVar}(\mathrm{Unenumerated}(n, \overline{(c_i, v_i)})) = (\mathrm{Ref}(v), n, v, V, C)}\ \textit{SuspendVar}$$

Note that a consequence of the $v \in V$ premise of SUSPENDVAR is that there must be at least one constraint on the empty path ϵ among the c_i. This also prevents any $\sigma(v)$ from becoming a Ref node, as no such constraints will appear at a root of σ.

MERGEVARS performs the heavy-lifting of the SUSPEND rule. It takes as arguments, respectively: the ECTA node n to form the contents of the new component, the variable v at which to place the new component, the possibly-empty list $V = \overline{v_i}$ of variables v_i already in $\mathrm{dom}(\sigma)$ such that v_i is constrained to be equal to the new component, the list $W \supseteq V$ of all variables constrained to be equal to the component, the inherited constraints C on n, and, finally, the partial enumeration state σ to be updated. The premises of MERGEVARS intersect $\mathrm{Unenumerated}(n, C)$ with all

Unenumerated nodes pointed to by the v_i, storing the result in the target variable v. The conclusion replaces all references to the w_i in all of σ with v.

$$\frac{\begin{array}{c}\sigma(v_i) = \text{Unenumerated}(n_i, C_i) \\ \sigma' = \sigma[v \mapsto \text{Unenumerated}(n \cap \bigcap_i n_i, C \cup \bigcup_i C_i)]\end{array}}{\textsc{MergeVars}(n, v, \overline{v_i}, \overline{w_i}, C, \sigma) = (t \mapsto [v/w_1](\ldots [v/w_k]t)) \circ \sigma'}\ \mathit{MergeVars}$$

Example 3.3.10. *Consider the partial enumeration state*

$$\begin{aligned}\sigma =& [v_\top \mapsto \mathit{T}(\text{“}f\text{''}, \mathit{Unenumerated}(n_1, (\{\epsilon\}, v_1), (\{\epsilon\}, v_2), (\{0\}, v_3)),\\ &\qquad\qquad\qquad \mathit{Unenumerated}(n_2, (\{1\}, v_2)))\\ &, v_1 \mapsto \mathit{Unenumerated}(n_3)\\ &, v_2 \mapsto \mathit{Unenumerated}(n_4, (\{0\}, v_4))]\end{aligned}$$

Only the Unenumerated node for n_1 is a possible target for the SUSPEND *rule.* SUSPENDVAR *returns either the tuple $(\mathit{Ref}(v_1), n_1, v_1, \{v_1, v_2\}, \{(\{0\}, v_3)\})$ or the tuple $(\mathit{Ref}(v_2), n_1, v_2, \{v_1, v_2\}, \{(\{0\}, v_3)\})$. The results for either choice will be comparable; we assume the first. The final output is then*

$$\begin{aligned}\sigma' =& [v_\top \mapsto \mathit{T}(\text{“}f\text{''}, \mathit{Ref}(v_1), \mathit{Unenumerated}(n_2, (\{1\}, v_1)))\\ &, v_1 \mapsto \mathit{Unenumerated}(n_1 \cap n_3 \cap n_4, (\{0\}, v_3), (\{0\}, v_4))\\ &, v_2 \mapsto \mathit{Unenumerated}(n_4, (\{0\}, v_4))]\end{aligned}$$

Notice that the current definition of SUSPEND *leaves the binding for v_2, though it is no longer referenced and can have no effect on the rest of enumeration.*

Remark 3.3.11. *This presentation, specifically the premise in* MERGEVARS *that each $\sigma(v_i)$ is an Unenumerated node and the second premise of* CHOOSE, *requires* *that no variable be enumerated with* Choose *prior to being merged with another variable. This strict ordering can be relaxed by replacing the intersection of ECTA nodes with a compatibility check between partially-enumerated terms..*

Remark 3.3.12. *Implementation note: To implement the* MergeVars *step, the* ecta *library adds an extra layer of indirection to σ using a union-find data structure to track which variables have been merged. In doing so, it avoids the need to modify the body of σ.*

Remark 3.3.13. *The intersections in* MergeVars *may result in having $\sigma(v) =$ Unenumerated(Node(), ...) for some v. Because σ then contains the empty node Node(), its denotation is hence $\varnothing$, and this entire branch of computation may be pruned. This is the reason that node ordering affects performance.*

Correctness

Lemma 3.3.14 (Soundness of Choose)**.** *Consider a partial enumeration state σ, and fix a contained partially-enumerated term $\sigma(v) = E[pt]$. Consider the set P of all pt' such that $pt \to pt'$ by the* ChooseNode *rule. Let S be the set of σ' reachable from σ by one application of the* Choose *rule, namely $\left\{\sigma[v \mapsto E[pt']] \middle| pt' \in P\right\}$. Then $[\![\sigma]\!]^{ES} = \bigcup_{\sigma' \in S} \left\{\rho\big|_{dom(\sigma)} \middle| \rho \in [\![\sigma']\!]^{ES}\right\}$.*

Lemma 3.3.15 (Soundness of Suspend)**.** *Consider a partial enumeration state σ, and let $\sigma \to \sigma'$ by the* Suspend *rule. Then $\left\{\rho(v_\top) \middle| \rho \in [\![\sigma]\!]^{ES}\right\} = \left\{\rho(v_\top) \middle| \rho \in [\![\sigma']\!]^{ES}\right\}$.*

Lemma 3.3.16 (Completeness of Choose and Suspend)**.** *Let n be an ECTA node and σ_0 its corresponding initial partial enumeration state, and let $\sigma_0 \to^* \sigma$ by the* Choose *and* Suspend *rules. Suppose σ is irreducible by the* Choose *and* Suspend *rules. Then either σ is fully enumerated, or $[\![\sigma]\!]^{ES} = \varnothing$.*

Proof. Suppose σ is not fully enumerated. Then, by definition, $\sigma(v)$ contains a path constraint for some v. As no path constraints may appear within a Mu node, $\sigma(v)$ must contain a subterm of the form $pt = textUnenumerated(Node(\overline{e_i}), \overline{(c_i, v_i)})$. We proceed by cases:

1. There is no i with $\epsilon \in c_i$: Then one of the following subcases applies:

   (a) **(Choose case)** There is no v such that $pt = \sigma(v)$, or there is such a v and no variable v' such that $\sigma(v')$ references v. Then the conditions of Choose apply, and σ is not irreducible.

   (b) **(Occurs-check failure case)** $pt = \sigma(v)$ for some v, and $\sigma(v')$ references v for some v'. This case analysis is then repeated for v'. Either this process terminates in one of the other cases, or there is a sequence of variables $v_1, \ldots, v_k$ and $v_{k+1} = v_1$ such that $\sigma(v_i)$ references $\sigma(v_{i+1})$ for $1 \leq i \leq k$. Each v_i has a corresponding path $p_i \neq \epsilon$ such that $\rho \in [\![\sigma]\!]^{\text{ES}}$ must satisfy $\rho(v_i)\big|_{p_i} = \rho(v_{i+1})$ for $1 \leq i \leq k$. Since all terms are finite, this is unsatisfiable, and so $[\![\sigma]\!]^{\text{ES}} = \varnothing$.

2. There is at least one i with $\epsilon \in c_i$: Consider all such i and their corresponding v_i. Then one of the following subcases applies:

   (a) **(Suspend case)** For all such v_i, $\sigma(v_i)$ is an Unenumerated node. Then the Suspend rule applies, and σ is not irreducible.

   (b) **(Referenced-yet-enumerated case)** For some v_i, $\sigma(v_i)$ is either a Ref node or a T node. We refer to these as the "Ref-node-at-root" case and the "T-node-at-root" case. The Ref-node-at-root case cannot occur: it would have to stem from an application of the SuspendVar rule at a root, which would require an Unenumerated node containing the path ϵ at a root, which cannot be created by either rule. The T-node-at-root case also cannot occur: this T node could only be created by running the Choose rule on $\sigma(v_i)$, but this rule could not have been run because of the external reference to v_i.

□

Theorem 3.3.17 (Termination of Enumeration)**.** *There is no infinite sequence $\sigma_0 \to \sigma_1 \to \ldots$ where each transition follows from the* Choose *and* Suspend *rules.*

Proof. For paths p_1, p_2 let $p_1 \sqsubseteq p_2$ if $p_1 = p_2$ or $\text{length}(p_1) < \text{length}(p_2)$, where $\text{length}(p)$ denotes normal list length. For ECTA nodes n_1, n_2, let $n_1 \sqsubseteq n_2$ if $n_1 = n_2$ or $\text{depth}(n_1) < \text{depth}(d_2)$, where $\text{depth}(n)$ denotes normal DAG depth. Order integers normally. Order multisets of paths, ECTA nodes, and integers by their respective multiset orderings [48].

Define $F(\sigma) = (A, B, C)$, where A is the set of paths directly contained in an Unenumerated node, B the set of ECTA nodes directly contained in an Unenumerated node, and C the set of depths of Unenumerated nodes (where each $\sigma(v)$ has depth 0, their children have depth 1, etc). We shall show that both the Choose and Suspend rules decrease $F(\sigma)$ under the lexicographic ordering. Since this ordering is well-founded by construction, that completes the proof of termination.

1. **Choose case**: Decreases A if the node being enumerated contains any constraints. In any case, decreases B.

2. **Suspend case**: If V in the SuspendVar rule is nonempty, decreases A. Otherwise, if V is empty, A is unchanged, B is unchanged because the intersection in the MergeVars rules is not performed, and C is decreased because an Unenumerated node is moved to root position.

□

Theorem 3.3.18 (Correctness of Enumeration)**.** *Let n be an ECTA node, σ_0 the initial partial enumeration state constructed by the enumeration algorithm, and consider all terminating sequences $\sigma_0 \to \cdots \to \sigma_t$, where each transition follows from the* Choose *or* Suspend *rules, and σ_t is irreducible by these same rules. Let S by the set of all such σ_t. Then $[\![n]\!]^N = \left\{\rho(v_\top)\middle|\sigma \in S, \rho \in [\![\sigma]\!]^{ES}\right\}$*

Proof. Given an ECTA node n, the enumeration algorithm first constructs the initial state $\sigma_0 = [v_\top \mapsto \text{Unenumerated}(n)]$. It is trivial from the definitions that $[\![\sigma_0]\!]^{\text{ES}} = \left\{[v_\top \mapsto t]\middle|t \in [\![n]\!]^{\text{N}}\right\}$, i.e.: that $[\![\sigma_0]\!]^{\text{ES}}$ is equivalent to $[\![n]\!]^{\text{N}}$. The theorem then follows immediately from Lemmas 3.3.14, 3.3.15, and 3.3.16 and Theorem 3.3.17 □

3.4 Applications

3.4.1 Hoogle+

We described this domain in Section 3.1. We also implemented an additional optimization: by adding *relevant typing* to the encoding: instead of having a single node for each size of term, there is a node for each combination of size and subset of the input arguments. The final search can thus only include terms that use all input variables.

We ran ECTA and HOOGLE+ on 45 of the HOOGLE+ benchmarks, both with a timeout of 120 seconds. Figure 3-11 shows the times of both. This chart is biased against ECTA: the times shown are for HOOGLE+ to generate a single solution, but for ECTA to enumerate all solutions. ECTA is slower on 13 of them, the same (both time out) on 5, and faster on the other 27. There are 2 benchmarks which time out with ECTA, but not HOOGLE+ (and in fact take under 10 seconds), and 2 which time out with HOOGLE+ but not ECTA.

There are a number of benchmarks which take hundreds of milliseconds with HOOGLE+, but tens of milliseconds with ECTA. These can be explained by HOOGLE+'s overhead in using an SMT solver. However, a number of benchmarks take significant time in HOOGLE+, but not ECTA. The greatest speedup is the `mapEither` benchmark, which takes 17.43 seconds for HOOGLE+, but 0.15 for ECTA, a speedup factor of 114.3. Overall, this shows that the general ECTA library is radically faster than a specialized synthesizer on some benchmarks — and we are still adding more optimizations both to ECTA and to the encoding.

3.4.2 Database Optimization

E-graphs have been productively applied to rewrite-based optimization. In this subsection we describe an optimization task where ECTAs perform significantly better than E-graphs.

Database query optimization is often performed by applying semantics-preserving

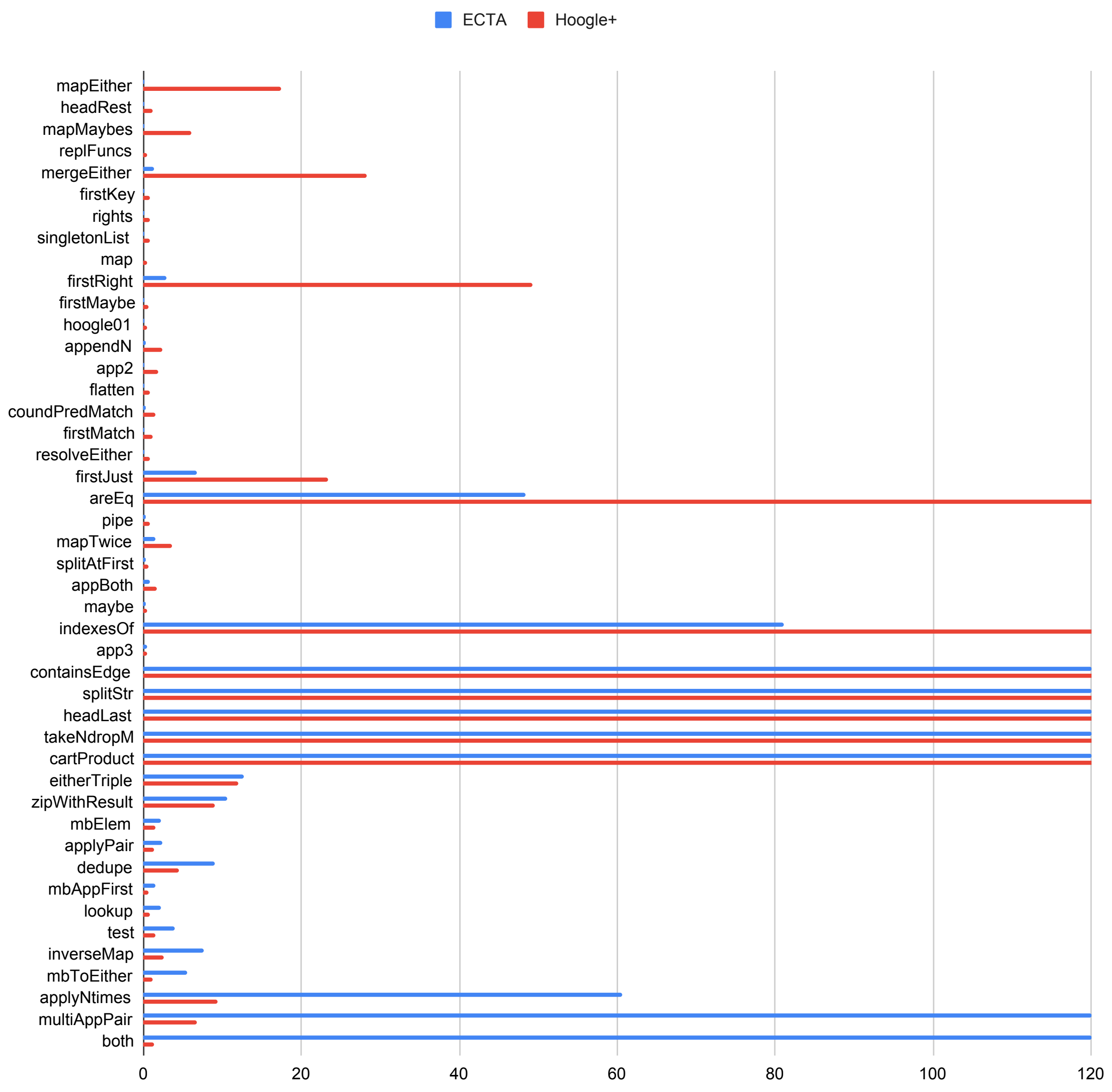

ECTA
Hoogle+
mapEither
headRest
mapMaybes
replFuncs
mergeEither
firstKey
rights
singletonList
map
firstRight
firstMaybe
hoogle01
appendN
app2
flatten
coundPredMatch
firstMatch
resolveEither
firstJust
areEq
pipe
mapTwice
splitAtFirst
appBoth
maybe
indexesOf
app3
containsEdge
splitStr
headLast
takeNdropM
cartProduct
eitherTriple
zipWithResult
mbElem
applyPair
dedupe
mbAppFirst
lookup
test
inverseMap
mbToEither
applyNtimes
multiAppPair
both
0
20
40
60
80
100
120

Figure 3-11

rewrites to the query, with the goal of executing the query more quickly by leveraging views or indices. Previous work [56] has extended the conventional query optimization approach to consider the layout of the data in the database in addition to the query plan. In this work, queries are optimized by applying local rewrites according to a heuristic schedule. In this subsection, we show how ECTAs can be applied to this problem and show that ECTAs offer significant performance advantages over E-graphs in this domain.

The goal of our optimizer is to select an optimal set of data structures and access methods to implement a relational query. The optimizer works on a relational algebra DSL that is extended with operators that describe data structures. A key feature of this DSL is that it uses staging: data structures are constructed in the first (compile-time) stage and access methods execute in the second (run-time) stage. During optimization, the system will consider programs that are not *well staged* (i.e. the staging constraints are not yet satisfied). After optimization, however, it is only useful to consider programs that have the desired staging structure.

We implement a simple version of the optimizer in [56] using both ECTAs and E-graphs. We show that ECTAs allow us to ignore concerns about staging during the rewriting phase and then when rewriting is complete, ECTAs make it easy to only enumerate well-staged terms.

Implementation

We do not describe the DSL in detail in this subsection, but we describe enough to explain the use of equality constraints to describe the staging status of a term. Each term is available in the *run-time* stage, the *compile-time* stage, both, or neither. A term is available in neither stage if it violates some staging constraint. We write the staging status of a term t as $\mathit{stage}(t) = (\mathit{compile}, \mathit{run})$. The staging status of a term is determined by its constructor and the status of its subterms. For example, $\mathit{stage}(\texttt{filter}(p, q)) = (p_c \wedge q_c, p_r \wedge q_r)$ where $\mathit{stage}(p) = (p_c, p_r), \mathit{stage}(q) = (q_c, q_r)$. In general, the staging status of a term is a conjunction of a subset of the statuses of its subterms. After rewriting, the system will enumerate terms that are available at compile time (have the stage (t, α)). We now show how to encode these constraints into an ECTA.

Following the convention in Section 3.1, the first child of every edge representing a term constructor is a metadata node (labeled `meta`). These metadata nodes have children for the two tracked stages: `compile` and `run`. The staging status is a Boolean function of the staging statuses of the subterms. These functions are encoded using truth tables. For example, the `meta` node for `filter` chooses between $\{\texttt{compile}(t, t, t), \texttt{compile}(f, t, f), \ldots\}$. The constraint on `filter` connects the arguments of the truth table to the staging status of the subterms.

Truth tables can grow very large, particularly for variadic operators. To avoid this blowup, our implementation approximates the staging statuses. The DSL only constrains a term to be available in a stage; it never constrains a term to *not* be available. Therefore, a term may have any status unless it is constrained to be available. This approximation allows the truth table for `filter` to be written as $\{\texttt{compile}(t, t, t), \texttt{compile}(f, \top, \top)\}$ where $\top = \{t, f\}$.

The E-graph version of the optimizer also tracks the staging status of its terms. As in the ECTA version, each term constructor has a metadata node as its first argument. Instead of using constraints to track the staging status, the E-graph version proceeds in two phases. While applying optimization rewrites, it assumes that every term has the stage (f, f). Next, it applies propagation rewrites that add the correct stages.

Results

Our ECTA version of the database optimizer has two key advantages over the E-graph version. First, constraint propagation in the E-graph version is greedy; the ECTA version is directed by the well-staged constraint at the top level. This means that the E-graph contains many subterms that do not appear in any well-staged term, which makes it larger. Second, in the E-graph version we use rewrite rules to propagate constraints; in the ECTA version, constraint propagation is handled by the library.

When we run our ECTA optimizer, we find that it produces a graph with 224 nodes and 386 edges before reduction. This graph represents all of the terms produced by applying optimization rewrites, including terms that are not well staged. After applying the well-staged constraint at the top level and performing one step of reduction, the graph has 282 nodes and 428 edges. This reduction step does not ensure that every term that is extractable from the graph satisfies the staging constraints, but it does ensure that there are no nodes in the graph that don't produce any well-staged terms.

The E-graph version produces a graph with 243 nodes and 881 edges. This graph contains all terms—well staged and not—but each term is annotated with its staging status.

These results show that despite the overhead of the constraint encoding in the ECTA, the addition of equality constraints allows us to encode and track complex properties of terms without performing manual constraint propagation or representing terms without the properties we care about.

Chapter 4

Mandate: Deriving Tools from Semantics

4.1 Why Generate CFGs?

Many programming tools use control-flow graphs, from compilers to model-checkers. They provide a simple way to order the subterms of a program, and are usually taken for granted. According to folklore, their definition is simple: “control-flow graphs are an abstraction of control-flow."

In fact, as we shall argue, CFGs are not well understood, and their definition is not so simple. Consider: Even for a single language, no two tools generate the same CFG for the same program, and we have found no prior attempt to define what it means for a given CFG to correctly abstract a program. Before diving deeper into the need for a theory of CFGs, let us illustrate the nuances of CFG-generation: Figure 4-1 is a fragment of a pretty-printer. How would a CFG for it look? Here are three possible answers for different kinds of tools:

1. Compilers want small graphs that use little memory. A compiler may only give one node per basic block, giving the graph in Figure 4-2a.

2. Many static analyzers give abstract values to the inputs and result of every expression. To that end, frameworks such as Polyglot [126] and IncA [161] give

two nodes for every expression: one each for entry and exit. Figure 4-2b shows part of this graph. We previously argued for the utility of this style of CFG in Section 2.7.2.

3. Consider an analyzer that proves this program's output has balanced parentheses. It must show that there are no paths in which one if-branch is taken but not the other. This can be easily written using a path-sensitive CFG that partitions on the value of `b` (Figure 4-2c). Indeed, in Section 4.7, we build such an analyzer atop path-sensitive CFGs in under 50 lines of code.

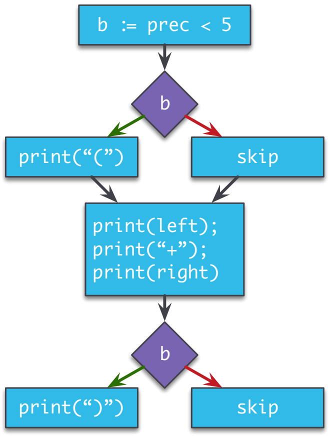
All three tools require separate CFG-generators.

```
b := prec < 5;
if (b) then
  print("(")
else
  skip;
print(left);
print("+");
print(right);
if (b) then
  print(")")
else
  skip
```

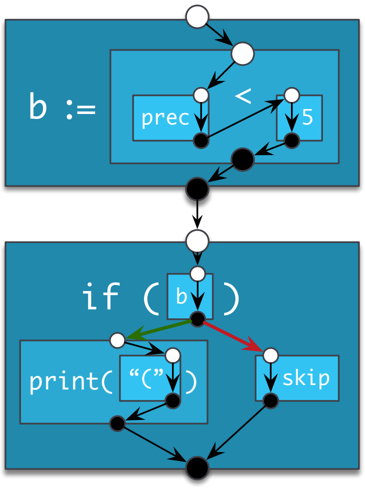
Figure 4-1

Whence CFGs? What if we had a formal semantics (and grammar) for a programming language? In principle, we should be able to automatically derive all tools for the language. In this dream, only one group needs to build a semantics for each, and then all tools will automatically become available — and semantics have already been built for several major languages [20, 132, 75]. In this chapter, we take a step towards that vision by developing the formal correspondence between semantics and control-flow graphs, and use it to automatically derive CFG generators from a large class of operational semantics.

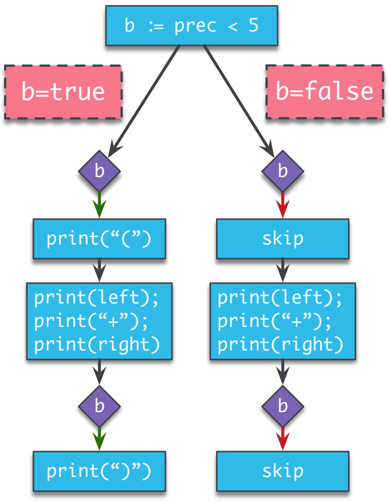
While operational semantics define each step of computation of a program, the correspondence with control-flow graphs is not obvious. The "small-step" variant of operational semantics defines individual steps of program execution. Intuitively, these steps should correspond to the edges of a control-flow graph. In fact, we shall see that many control-flow edges correspond to the second half of one rule, and the first half of another. We shall similarly find the nodes of a control-flow graph correspond to neither the subterms of a program nor its intermediate values. Existing CFG generators skip these questions, taking some notion of labels or "program points" as a given



(a)



(b)



(c)

Figure 4-2: Variants of control-flow graphs. Colors are for readability.

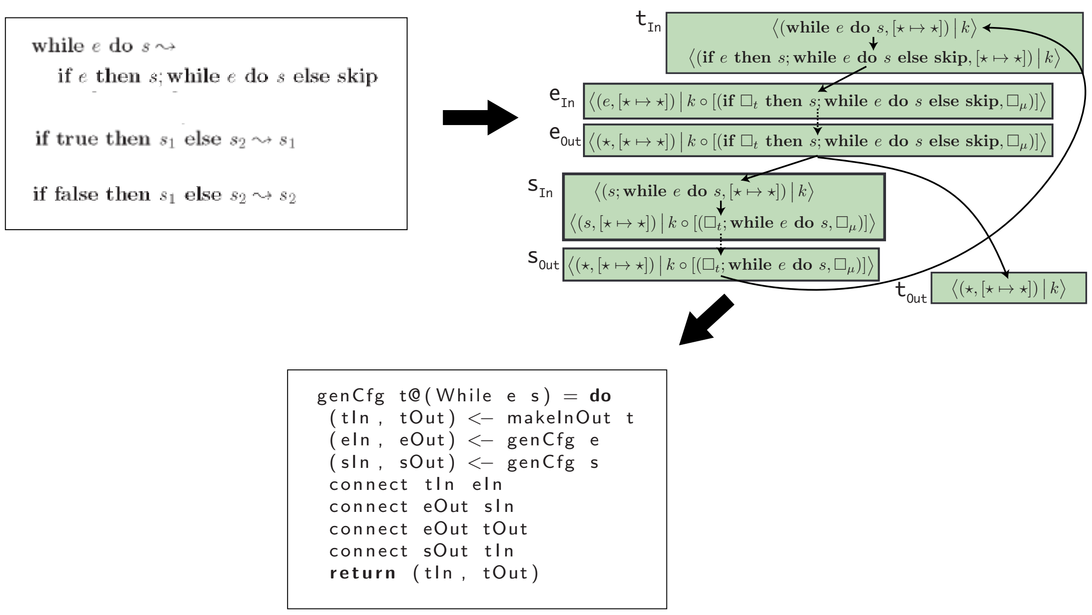
(e.g.: [156]). We instead develop CFGs from first principles, and, after much theory, discover that **"a CFG is a projection of the transition graph of abstracted abstract machine states."**

Abstraction and Projection and Machines The first insight is to transform the operational semantics into another style of semantics, the abstract machine [54], via a new algorithm. Evaluating a program under these semantics generates an infinite-state transition system with recognizable control-flow. Typically at this point, an analysis-designer would manually specify some kind of abstract semantics which collapses this system into a finite-state one. Our approach does this automatically by interpreting the concrete semantics differently, using an obscure technique called *abstract rewriting*. From this reduced transition system, a familiar structure emerges: we have obtained our control-flow graphs!

Now all three variants of control-flow graph given in this section follow from different abstractions of the abstract machine, followed by an optional *projection*, or merging of nodes. With this approach, we can finally give a formal, proven correspondence between the operational semantics and all such variants of a control-flow graph.

Mandate: A CFG-Generator Generator The primary goal of this chapter is to develop the first theory of CFGs from first principles. Yet our theory immediately lends itself to automation. We have implemented our approach in Mandate, the first control-flow-graph generator generator. Mandate takes as input the operational semantics for a language, expressed in an embedded Haskell DSL that can express a large class of systems, along with a choice of abstraction and projection functions. It then outputs a control-flow-graph generator for that language. By varying the abstraction and projection functions, the user can generate any of a large number of CFG-generators.

Mandate has two modes. In the *interpreted mode*, Mandate abstractly executes a program with its language's semantics to produce a CFG. For cases where the


t_{In}
$\langle(\textbf{while } e \textbf{ do } s, [\star \mapsto \star]) \mid k\rangle$
$\langle(\textbf{if } e \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}, [\star \mapsto \star]) \mid k\rangle$
e_{In}
$\langle(e, [\star \mapsto \star]) \mid k \circ [(\textbf{if } \square_t \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}, \square_\mu)]\rangle$
e_{Out}
$\langle(\star, [\star \mapsto \star]) \mid k \circ [(\textbf{if } \square_t \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}, \square_\mu)]\rangle$
s_{In}
$\langle(s; \textbf{while } e \textbf{ do } s, [\star \mapsto \star]) \mid k\rangle$
$\langle(s, [\star \mapsto \star]) \mid k \circ [(\square_t; \textbf{while } e \textbf{ do } s, \square_\mu)]\rangle$
s_{Out}
$\langle(\star, [\star \mapsto \star]) \mid k \circ [(\square_t; \textbf{while } e \textbf{ do } s, \square_\mu)]\rangle$
t_{Out}
$\langle(\star, [\star \mapsto \star]) \mid k\rangle$

$$\textbf{while } e \textbf{ do } s \rightsquigarrow \textbf{if } e \textbf{ then } s; \textbf{while } e \textbf{ do } s \textbf{ else skip}$$

$$\textbf{if true then } s_1 \textbf{ else } s_2 \rightsquigarrow s_1$$

$$\textbf{if false then } s_1 \textbf{ else } s_2 \rightsquigarrow s_2$$

```
genCfg t@(While e s) = do
 (tIn, tOut) <- makeInOut t
 (eIn, eOut) <- genCfg e
 (sIn, sOut) <- genCfg s
 connect tIn eIn
 connect eOut sIn
 connect eOut tOut
 connect sOut tIn
 return (tIn, tOut)
```

Figure 4-3: (Top left) SOS rules for loops and conditionals (Top right) A graph-pattern generated from these rules, describing the control-flow of all while-loops (Bottom) Generated CFG-generation code

control-flow of a node is independent from its context (e.g.: including variants (1) and (2) but not (3)), Mandate's *compiled mode* can output a CFG-generator as a short program, similar to what a human would write (Figure 4-3).

We have evaluated Mandate on several small languages, as well as two larger ones. The first is Tiger, an Algol-family language used in many compilers courses, and made famous by a textbook of Appel [6]. The second is MITScript [26], a JavaScript-like language with objects and higher-order functions used in an undergraduate JIT-compilers course. While these are pedagogical languages without the edge-cases of C or SML, they nonetheless contain all common control-flow features except exceptions. And, since previous work on conversion of semantics features small lambda calculi, ours are the largest examples of automatically converting a semantics into a different form, by a large margin.

Overall, this chapter makes the following contributions:

1. A formal and proven correspondence between the operational semantics and many common variations of CFGs, giving the first from-first-principles theoret-

ical explanation of CFGs.

2. An elegant new algorithm for converting small-step structural operational semantics into abstract machines.

3. An algorithm which derives many types of control-flow graph generators from an abstract machine, determined by choice of abstraction and projection functions, including standalone generators which execute without reference to the semantics ("compiled mode").

4. An "automated termination proof," showing that, if the compiled-mode CFG-generator terminates (run once per language/abstraction pair), then so does the corresponding interpreted-mode CFG generator (run once per program).

5. The first CFG-generator generator, MANDATE, able to automatically derive many types of CFG generators for a language from an operational semantics for that language, and successfully used to generate CFG-generators for two rich languages. The generated CFG-generators were then used to build two static-analyzers.

Further, our approach using *abstract rewriting* offers great promise in deriving other artifacts from language semantics.

4.2 Control-Flow Graphs for IMP

We shall walk through a simple example of generating a control-flow graph generator, using a simple imperative language called IMP. IMP features conditionals, loops, mutable state, and two binary operators. Figure 4-5 gives the syntax. In this syntax, we explicitly split terms into values and non-values to make the rules easier to write. We will do this more systematically in Section 4.3.2.

The approach proceeds in three phases, corresponding roughly to the large boxes in Figure 4-4. In the first phase (top-left box / Section 4.2.1), we transform the semantics of IMP into a form that reveals the control flow. In the second phase

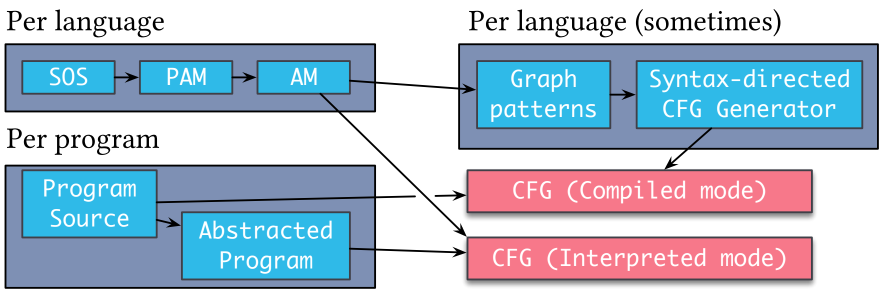


Figure 4-4: Dataflow of our approach

$$
\begin{array}{rlll}
\text{Variables} & x, y, \ldots & \in \mathsf{Var} & \\
\text{Expr. Values} & v & ::= & n \in \mathsf{Int} \mid \textbf{true} \mid \textbf{false} \\
\text{Expressions} & e & ::= & v \mid x \mid e + e \mid e < e \\
\text{Stmt. Values} & w & ::= & \textbf{skip} \\
\text{Statements} & s & ::= & w \mid s; s \mid \textbf{while } e \textbf{ do } s \\
 & & & \mid x := e \mid \textbf{if } e \textbf{ then } s \textbf{ else } s
\end{array}
$$

Figure 4-5: Syntax of IMP

(bottom-left box / Section 4.2.2), we show how to interpret these semantics with abstract rewriting [14] to obtain CFGs. In the finale (top-right box / Section 4.2.3), we show how to use abstract rewriting to discover facts about all IMP programs, resulting in human-readable code for a CFG-generator.

4.2.1 Getting Control of the Semantics

Semantics The language has standard semantics, so we only show a few of the rules, given as structural operational semantics (SOS). Each rule relates an old term and environment (t, μ) to a new one (t', μ') following one step of execution. As our first **running example**, we use the rules for assignments. We will later introduce our two other running examples: addition and while-loops. Together, these cover all features of semantic rules: environments, external semantic functions, branching, and back-edges.

$$\frac{(e, \mu) \rightsquigarrow (e', \mu')}{(x := e, \mu) \rightsquigarrow (x := e', \mu')}\ \mathit{AssnCong}$$

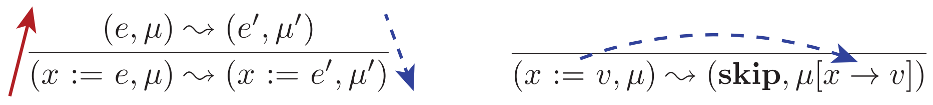
$$\frac{}{(x := v, \mu) \rightsquigarrow (\mathbf{skip}, \mu[x \to v])}\ \mathit{AssnEval}$$

These rules give the basic mechanism to evaluate a program, but don't have the form of stepping from one subterm to another, as a control-flow graph would. So, from these rules, our algorithm will automatically generate an *abstract machine* (AM). This machine acts on states $\langle (t, \mu) \,|\, K \rangle$. K is the *context* or *continuation*, and represents what the rest of the program will do with the result of evaluating t. K is composed of a stack of *frames.* While the general notion of frames is slightly more complicated (Section 4.3.3), in most cases, a frame can be written as e.g.: $[(x := \Box_t, \Box_\mu)]$, which is a frame indicating that, once the preceding computation has produced a term and environment (t', μ'), the next step will be to evaluate $(x := t', \mu')$. Our algorithm generates the following rules. These match the textbook treatment of AMs [54].

$$\begin{aligned}
\langle (x := e, \mu) \,|\, k \rangle &\to \langle (e, \mu) \,|\, k \circ [(x := \Box_t, \Box_\mu)] \rangle \\
\langle (v, \mu) \,|\, k \circ [(x := \Box_t, \Box_\mu)] \rangle &\to \langle (\mathbf{skip}, \mu[x \to v]) \,|\, k \rangle
\end{aligned}$$

The first AM rule, on seeing an assignment $x := e$, will focus on e. Other rules, not shown, then reduce e to a value. The second then takes this value and evaluates the assignment.

While these rules have been previously hand-created, this work is the first to derive them automatically from SOS. These two SOS rules become two AM rules. A naive interpretation is that the first SOS rule corresponds to the first AM rule, and likewise for the second. After all, the left-hand sides of the first rules match each other, as do the right-hand sides of the second. But notice how the RHSs of the firsts do not match, nor do the LHSs of the seconds. The actual story is more complicated. This diagram gives the real correspondence:



$$\frac{(e, \mu) \leadsto (e', \mu')}{(x := e, \mu) \leadsto (x := e', \mu')} \qquad \frac{}{(x := v, \mu) \leadsto (\mathbf{skip}, \mu[x \to v])}$$

The first AM rule is simple enough: it corresponds to the solid arrow, the first half of the first SOS rule. But the second AM rule actually corresponds to the two dashed arrows, AssnEval and the second half of AssnCong. We shall find that jumping straight from SOS to AM skips an intermediate step, which treats each of the three arrows separately.

This fusing of AssnEval and AssnCong happens because only two actions can follow the second half of AssnCong: AssnEval, and the first half of AssnCong — which is the inverse of the second half. Hence, only the pairing of AssnCong with AssnEval is relevant, and two AM rules are enough to describe all computations. And the second AM rule actually does two steps in one: it plugs (v, μ) into $[(x := \square_t, \square_\mu)]$ to obtain $(x := v, \mu)$, and then evaluates this result. So, by fusing these rules, the standard presentation of AM obscures the multiple underlying steps, hiding the correspondence with the SOS.

This insight — that SOS rules can be split into several parts — powers our algorithm to construct abstract machines. The algorithm first creates a representation called the *phased abstract machine*, or PAM, which partitions the two SOS rules into three parts, corresponding to the diagram's three arrows, and gives each part its

```
valueIrrelevance t =
  mapTerm valToStar t
 where
  valToStar (Val _ _) = ValStar
  valToStar t         = t
```

Figure 4-6

own rule. The algorithm then fuses some of these rules, creating the final AM, and completing the first stage of our CFG-creation pipeline.

Section 4.3.3 explains PAM in full, while Section 4.3.5 and Section 4.3.6 give the algorithm for creating abstract machines. Appendix B.1 provides correctness results for the SOS-to-AM procedure.

4.2.2 Run Abstract Program, Get CFG

The AM rules show how focus jumps into and out of an assignment during evaluation — the seeds of control-flow. But these transitions are not control-flow edges; there are still a few important differences. The AM allows for infinitely-many possible states, while control-flow graphs have finite numbers of states, potentially with loops. The AM executes deterministically, with every state stepping into one other state. Even though we assume determistic languages, even for those, the control-flow graph may branch. We will see how abstraction solves both of these issues, turning the AM into the interpreted-mode control-flow graph generator.

To give a complete example, we'll also need to evaluate an expression. Here is the rule for variable lookups, which looks up y in the present environment:

$$\left\langle (y, \mu) \,\middle|\, k \right\rangle \;\rightarrow\; \left\langle (\mu(y), \mu) \,\middle|\, k \right\rangle$$

Now consider the statement $x := y$. It can be executed with an infinite number of environments: the starting configuration $(x := y, [y \mapsto 1])$ results in $(\mathbf{skip}, [y \mapsto 1, x \mapsto 1])$; $(x := y, [y \mapsto 2])$ yields $(\mathbf{skip}, [y \mapsto 2, x \mapsto 2])$; etc.

To yield a control-flow graph, we must find a way to compress this infinitude of possible states into a finite number. The value-irrelevance abstraction, given by the code in Figure 4-6, replaces all values with a single *abstract value* representing any of them: $\star$. Under this abstraction, all starting environments for this program will be abstracted into the single environment $[y \mapsto \star]$.

In a typical use of abstract interpretation, at this point a new abstract semantics must be manually defined in order to define executions on the abstract state. However, with this kind of syntactic abstraction, our system can interpret the exact same AM rules on this abstract state, a process called *abstract rewriting.* Now, running in fixed context K, there is only one execution of this statement.

$$\begin{aligned}
\big\langle (x := y, [y \mapsto \star]) \,\big|\, K \big\rangle &\to \big\langle (y, [y \mapsto \star]) \,\big|\, K \circ [(x := \square_t, \square_\mu)] \big\rangle \\
&\to \big\langle (\star, [y \mapsto \star]) \,\big|\, K \circ [(x := \square_t, \square_\mu)] \big\rangle \\
&\to \big\langle (\mathbf{skip}, [y \mapsto \star, x \mapsto \star]) \,\big|\, K \big\rangle
\end{aligned}$$

This abstract execution is divorced from any runtime values, yet it still shows the flow of control entering and exiting the assignment and expression— exactly as in the expression-level control-flow graph from Figure 4-2b. And thus we can take these abstract states to be our control-flow nodes. The CFG is an abstraction of the transitions of the abstract machine.

Note that, because there are only finitely-many abstract states, this also explains loops in control-flow graphs: looping constructs lead to repeated states, which leads to back-edges in the transition graph. And abstractions also account for branching. Consider the rules for conditionals:

$$\big\langle (\mathbf{true}, \mu) \,\big|\, k \circ [(\mathbf{if}\ \square_t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \square_\mu)] \big\rangle \to \big\langle (s_1, \mu) \,\big|\, k \big\rangle$$
$$\big\langle (\mathbf{false}, \mu) \,\big|\, k \circ [(\mathbf{if}\ \square_t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \square_\mu)] \big\rangle \to \big\langle (s_2, \mu) \,\big|\, k \big\rangle$$

Under the value-irrelevance abstraction, the condition of an if-statement will evaluate to a configuration of the form $(\star, \mu)$. Because $\star$ can represent both **true** and **false**,

both of the above rules will match. This gives control-flow edges from the if-condition into both branches, exactly as desired.

The value-irrelevance abstraction gives an expression-level CFG. But it is not the only choice of abstraction. Section 4.5 presents the CFG-derivation algorithm, and shows how other choices of abstraction lead to other familiar control-flow graphs, and also introduces projections, which give the CFG-designer the ability to specify which transitions are "internal" and should not appear in the CFG.

4.2.3 A Syntax-Directed CFG-Generator

The previous section showed how to abstract away the inputs and concrete values of an execution, turning a program into a CFG. The per-program state-exploration of this algorithm is necessary for a path-sensitive CFG-generator, which may create an arbitrary number of abstract states from a single AST node. But for abstractions which discard all contextual information, only a few additional small ingredients are needed to generate a single artifact that describes the control-flow of all instances of a given node-type. This is done once per language, yielding the compiled-mode CFG-generator. We demonstrate how this works for while-loops, showing how our algorithm can combine many rules to infer control-flow, abstracts away extra steps caused by the internal details of the semantics, and can discover loops in the control-flow even though they are not explicit in the rules.

The semantics of while-loops in IMP are given in terms of other language constructs, by the rule $\textbf{while}\ e\ \textbf{do}\ s \rightsquigarrow \textbf{if}\ e\ \textbf{then}\ (s; \textbf{while}\ e\ \textbf{do}\ s)\ \textbf{else skip}$. Consider an AM state evaluating an arbitrary while-loop $\textbf{while}\ e\ \textbf{do}\ s$ in an arbitrary context k, with an arbitrary abstract environment μ. Such a state can be written $\big\langle(\textbf{while}\ e\ \textbf{do}\ s, \mu)\ \big|\ k\big\rangle$. Any such μ can be represented by the "top" environment $[\star \mapsto \star]$. This means that all possible transitions from any while-loop can be found by finding all rules that could match anything of the form $\big\langle(\textbf{while}\ e\ \textbf{do}\ s, [\star \mapsto \star])\ \big|\ k\big\rangle$. Repeatedly expanding these transitions results in a *graph pattern* describing the control-flow for every possible while-loop.

However, merely searching for matching rules will not result in a finite graph, because of states like $\langle (e, [\star \mapsto \star]) \mid k \rangle$. These states, which represents the intent to evaluate the unknown subterm e, can match rules for any expression. Instead, we note that, for any given e, other rules (corresponding to other graph patterns) would evaluate its control-flow, and their results can all be soundly overapproximated by assuming e is eventually reduced to a value. Hence, when the search procedure encounters such a state, it instead adds a "transitive edge" to a state $\langle (\star, [\star \mapsto \star]) \mid k \rangle$. With this modification, the search procedure finds only 8 unique states for while-loops. It hence terminates in the graph pattern of Figure 4-3 (with dotted lines for transitive edges), which describes the control-flow of all possible while-loops. From this pattern, one could directly generate a CFG fragment for any given while-loop by instantiating e, s, and k. In combination with the graph patterns for all other nodes, this yields a control-flow graph with a proven correspondence to the original program.

But, from these graph patterns, it is also straightforward to output code for a syntax-directed CFG-generator similar to what a human would write. Our code-generator traverses this graph pattern, identifying some states as the entrance and exit nodes of the entire while-loop and its subterms. All other states are considered internal steps which get merged with the enter and exit states (via a *projection*), resulting in a few "composite" states. Figure 4-3 shows how the code-generator groups and labels the states of this graph pattern as well as the generated code[1].

After many steps transforming and analyzing the semantics of IMP, the algorithm has finally boiled down all aspects of the control flow into concise, human-readable code — for an expression-level CFG-generator. To generate a statement-level CFG-generator, the user must merely re-run the algorithm again with a different abstraction. For while-loops, the resulting pattern and code are similar to those of Figure 4-3, except that they skip the evaluation of e.

The last few paragraphs already gave most of the details of graph-pattern construction. Section 4.6 gives the remaining details, while Appendix B.3 proves the algorithm's correctness.

[1]This is verbatim MANDATE output except that, in actual output, (1) the `connect` statements are in no particular order, and (2) the actual return value is `([tIn], [tOut])`, as, in general, AST nodes such as conditionals may have multiple final CFG nodes.

4.3 From Operational Semantics to Abstract Machines

The first step in our algorithm is to convert the structural operational semantics for a language into an abstract machine which has a clear notion of control-flow. Surprisingly, no prior algorithm for this exists (see discussion in Section 5.3). This section hence presents **the first algorithm to convert SOS to AM**. Our algorithm is unique in its use of a new style of semantics as an intermediate form, the phased abstract machine, which simulates a recursive program running the SOS rules. We believe this formulation is particularly elegant and leads to simple proofs, while being able to scale to the realistic languages Tiger and MITScript.

There is a lot of notation required first. Section 4.3.1 gives a notation for all programming languages, while Section 4.3.2 gives an alternate notation for structural operational semantics, one more amenable to inductive transformation. Section 4.3.3 and Section 4.3.4 describe the phased and orthodox abstract machines, while Section 4.3.5 and Section 4.3.6 give the conversion algorithm. Correctness results are be provided in Appendix B.1.

4.3.1 Terms and Languages

Our presentation requires a uniform notation for terms in all languages. Figure 4-7 gives this notation, describing both concrete terms as well as patterns used in rewrite rules. Throughout this chapter, we use the notation $\bar{\cdot}$ to represent lists, so that, e.g.: $\overline{\text{term}}$ represents the set of lists of terms.

A typical presentation of operational semantics will have variable names like v_1 and e', where v_1 implicitly represents a *value* which cannot be reduced, while e' represents a *nonvalue* which can. We formalize this distinction by marking each node either value or nonvalue, and giving each variable a *match type* controlling whether it may match only values, only nonvalues, or either.

Each variable is specialized with one of three **match types**. Variables of the form

$$
\begin{array}{rcll}
\text{const} & ::= & n \in \mathsf{Int} \mid \text{str} \in \mathsf{String} & \\
\text{sym} & ::= & +, <, \textbf{if}, \ldots \in \mathsf{Symbol} & \\
& & & \\
\text{mt} & ::= & \mathsf{Val} \mid \mathsf{NonVal} \mid \mathsf{All} & \text{(Match Types)} \\
& & a, b, c, \ldots \in \mathsf{Var} & \text{(Raw Vars)} \\
x & ::= & a_{\text{mt}} & \text{(Pattern Vars)} \\
& & & \\
\text{term} & ::= & \text{nonval}(\text{sym}, \overline{\text{term}}) & \\
& \mid & \text{val}(\text{sym}, \overline{\text{term}}) & \\
& \mid & \text{const} & \\
& \mid & x & \\
& & & \\
s & ::= & \mathsf{State}_l & \text{(Reduction State)} \\
c & ::= & (\text{term}, s) & \text{(Configurations)}
\end{array}
$$

Figure 4-7: Universe of terms.

a_{Val}, a_{NonVal}, and a_{All} are called **value**, **nonvalue**, and **general** variables respectively. Value variables may only match val constructors and constants; nonvalue variables match only nonval constructors; general variables match any.

We use a shorthand to mimic typical presentations of semantics. We will use e to mean a nonvalue variable, v or n to mean a value variable, and t or x for a general variable. But variables in a right-hand side will always be general variables unless said otherwise. For instance, in $e \rightsquigarrow e'$, e is a nonvalue variable, while e' is a general variable.

Each internal node is tagged either *val* or *nonval*. For example, $1+1$ is shorthand for the term $\text{nonval}(+, \text{val}(\text{IntLit}, 1), \text{val}(\text{IntLit}, 1))$. The IMP statement $x := 1$ may ambiguously refer to either the concrete term $\text{nonval}(:=, \text{“}x\text{”}, \text{val}(\text{IntLit}, 1))$ or to the pattern $\text{nonval}(:=, x_{\mathsf{All}}, \text{val}(\text{IntLit}, 1))$, but should be clear from context. Others' presentations commonly have a similar ambiguity, using the same notation for patterns and terms.

Each language l is associated with a **reduction state** State_l containing all extra information used to evaluate the term. For example, $\mathsf{State}_{\text{IMP}}$ is the set of *environ-*

$$
\begin{array}{rcl}
\text{rule} & ::= & c \rightsquigarrow \text{rhs} \\
\text{rhs} & ::= & c \\
& | & \textbf{let } [c \rightsquigarrow c] \textbf{ in } \text{rhs} \\
& | & \textbf{let } c = \mathsf{semfun}(\overline{c}) \textbf{ in } \text{rhs}
\end{array}
$$

Figure 4-8: Notation for SOS

ments, mapping variables to values. Formally:

$$\text{env} \quad ::= \quad \varnothing \mid \text{env}[\text{str} \to v]$$

Environments are matched using associative-commutative-idempotent patterns [7], so that *e.g.:* the pattern $x[\text{“}y\text{”} \to v]$ matches the environment $\varnothing[\text{“}y\text{”} \to 1][\text{“}z\text{”} \to 2]$. The latter environment is abbreviated $[z \mapsto 2, y \mapsto 1]$. The environment-extension operator itself is not commutative. Specifically, it is right biased, e.g.: $\varnothing[\text{“}z\text{”} \to 1][\text{“}z\text{”} \to 2] = \varnothing[\text{“}z\text{”} \to 2]$.

Finally, the basic unit of reduction is a *configuration*, defined $\mathsf{Conf}_l = \mathsf{term} \times \mathsf{State}_l$. We say that configuration $c = (t, s)$ is a value if t is a value.

4.3.2 Straightened Operational Semantics

Rules in structural operational semantics are ordinarily written like logic programs, allowing them to be used to run programs both forward and backwards, and allowing premises to be proven in any order. However, in most rules, there are dependences between the variables that effectively permit only one ordering. In this section, we give an **alternate syntax** for small-step operational semantics rules, which makes this order explicit. This is essentially the conversion of the usual notation into A-normal form [58].

The most immediate benefit of the Straightened Operational Semantics notation is that it orders the premises of a rule. One can also gain the ordering property by imposing a restriction on rules without a change in notation: Ibraheem and Schimdt's “L-attributed semantics” is exactly this, but for big-step semantics [79]. But there is an additional advantage of this new notation: it has an inductive structure, which allows defining recursive algorithms over rules.

Figure 4-8 defines a grammar for SOS rules. These rules collectively define the step-to relation for a language l, $\rightsquigarrow_l$. These rules are relations rather than functions, as they may fail and may have multiple matches. $R(A)$ denotes the image of a relation, i.e.: $R(x)$ refers to any y such that $x\ R\ y$.

A rule matches on a configuration, potentially binding several pattern variables. It then executes a right-hand side. Rule right-hand sides come in three alternatives. The two primary ones are that a rule's right-hand side may construct a new configuration from the bound variables, or may recursively invoke the step-to relation, matching the result to a new pattern. For example, the AssnCong rule from Section 4.2.1 would be rendered as:

$$(x := e, \mu) \rightsquigarrow \mathbf{let}\ [(e, \mu) \rightsquigarrow (e', \mu')]\ \mathbf{in}\ (x := e', \mu')$$

As a third alternative, it may invoke an external semantic function. Semantic functions are meant to cover everything in an operational semantics that is not pure term rewriting, e.g.: arithmetic operations. Each language has its own set of allowed semantic functions.

Definition 4.3.1. *Associated with each language l, there is a set of* ***semantic functions*** semfun_l*. Each is a relation*[2] *R of type $\overline{\mathsf{Conf}_l} \times \mathsf{Conf}_l$, subject to the restriction that (1) if $\overline{c}\ R\ d$, then $\overline{c}$ and d are values, and (2) for each $\overline{c} \in \mathsf{Conf}_l$, there are only finitely many d such that $\overline{c}\ R\ d$.*

For instance, there are two semantic functions for the IMP language: $\mathsf{semfun}_{\mathrm{IMP}} = \{+, <\}$. Both are partial functions, only defined on number/environment pairs, i.e.: arguments of the form $((n_1, \mu_1), (n_2, \mu_2))$. Since these functions only act on their term arguments and ignore their μ arguments, we invoke them using the abbreviated

[2]We define semantic "functions" to actually be relations (i.e.: partial nondetermistic functions) so that this definition can be reused for abstract interpretation in Section 4.5.2.

notation

$$\textbf{let } n_3 = +(n_1, n_2) \textbf{ in } \text{rhs}$$

which is short for **let** $(n_3, \mu_3) = +((n_1, \mu_1), (n_2, \mu_2))$ **in** rhs, where μ_1, μ_2 are dummy values, and μ_3 is an otherwise unused variable.

Semantic functions give straightened-operational-semantics notation a lot of flexibility. They can be used to encode side-conditions, e.g.: "**let true** = isvalid(x) **in** c" would fail to match if x is not valid. They may include external effects such as I/O. As an example using semantic functions, the rule

$$\frac{n = v_1 + v_2}{(v_1 + v_2, \mu) \rightsquigarrow (n, \mu)} \; \mathit{AddEval}$$

would be rendered as

$$(v_1 + v_2, \mu) \rightsquigarrow \textbf{let } n = +(v_1, v_2) \textbf{ in } (n, \mu)$$

where the first occurrence of "+" refers to a nonval node of the object language, and the second occurrence refers to a semantic function of the meta-language.

There are two extra non-syntactic requirements on SOS rules. The first makes the value/term distinction meaningful, and is needed in the proofs:

Assumption 4.3.2 (Sanity of Values)**.** *The following must hold:*

- *For a rule, $(t, s) \rightsquigarrow rhs$, the pattern t must not match a value.*
- *If t is a nonvalue, there are s, t', s' such that $(t, s) \rightsquigarrow (t', s')$.*

The second is necessary because nondeterministic languages lack a clear notion of control-flow.

Assumption 4.3.3 (Determinism)**.** *For any t, s, there is at most one t', s' such that $(t, s) \rightsquigarrow (t', s')$.*

This assumption is not necessary for converting the semantics to PAM, which can faithfully emulate any structural operational semantics. It is necessary for conversion

$$\begin{array}{rcll}
\text{frame} & ::= & [c \rightarrow \text{rhs}] & \\
K & ::= & \textbf{emp} & \text{(Contexts)} \\
 & | & K \circ \text{frame} & \\
 & | & k & \text{(Context Vars)} \\
\updownarrow & ::= & \uparrow \mid \downarrow & \text{(Phase)} \\
\text{pamState} & ::= & \langle c \mid K \rangle \updownarrow & \\
\text{pamRhs} & ::= & \text{pamState} & \\
 & | & \textbf{let } c = \mathsf{semfun}(\bar{c}) \textbf{ in } \text{pamRhs} & \\
\text{pamRule} & ::= & \text{pamState} \hookrightarrow \text{pamRhs} &
\end{array}$$

Figure 4-9: The Phased Abstract Machine

to an abstract machine, as abstract machines have difficulty expressing behaviors with top-level nondeterminism. Intuitively, this is desirable. Nondeterminism means that consecutive steps may occur in distant parts of a term: evaluating $((a*b)+(c-d))+(e/f)$ may e.g.: evaluate first the multiplication on the left, then the division on the right, and then the subtraction on the left. It is difficult to imagine a notion of control-flow for such a space of executions.

Overall, this notation gives a simple inductive structure to SOS rules. We will see in Section 4.3.5 how it makes the SOS-to-PAM conversion straightforward. And it loses very little generality from the conventional SOS notation: it essentially only assumes that the premises can be ordered. (And converting this notation back into the conventional form is easy: turn all RHS fragments into premises.)

4.3.3 The Phased Abstract Machine

In an SOS, a single step may involve many rules, and each rule may perform multiple computations. The phased abstract machine (PAM) breaks each of these into distinct steps. In doing so, it simulates how a recursive functional program would interpret the operational semantics.

A PAM state takes the form $\langle c \mid K \rangle \updownarrow$. The configuration c is the same as for operational semantics. K is a *context* or *continuation*, which represents the remainder of an SOS right-hand side. The phase is the novel part. Each PAM state is either

$$\mathrm{C} \quad ::= \quad \Box \quad | \quad \textbf{let } c = \mathsf{semfun}(\bar{c}) \textbf{ in } C$$

Figure 4-10: Abstract Machine: RHS contexts

in the evaluating ("down") phase $\downarrow$, or the returning ("up") phase $\uparrow$; an arbitrary phase is given the variable name "$\updownarrow$". A down state $\langle c \mid K \rangle\downarrow$ can be interpreted as an intention for the PAM to evaluate c in a manner corresponding to one step of the operational semantics, yielding $\langle c' \mid K \rangle\uparrow$. In fact, in Appendix B.1, we prove that a single step $c_1 \rightsquigarrow c_2$ in the operational semantics perfectly corresponds to a sequence of steps $\langle c_1 \mid K \rangle\downarrow \hookrightarrow^* \langle c_2 \mid K \rangle\uparrow$ in the PAM.

The full syntax of PAM rules is in Figure 4-9. A PAM rule steps a left-hand state into a right-hand state, potentially after invoking a sequence of semantic functions. A PAM state contains a configuration, context, and phase. A context is a sequence of frames, terminating in **emp**.

Note that a pamRhs consists of a sequence of RHS fragments which terminate in a $\mathsf{pamState}$ $\langle c \mid K \rangle\updownarrow$. Figure 4-10 captures this into a notation for RHS contexts, so that an arbitrary PAM rule may be written $\langle c_1 \mid K_1 \rangle\updownarrow_1 \hookrightarrow C[\langle c_2 \mid K_2 \rangle\updownarrow_2]$.

Finally, as alluded to in Section 4.2.1, a frame like $[(t', \mu') \to (x := t', \mu')]$ can be abbreviated to $[(x := \Box_t, \Box_\mu)]$, since t' and μ' are variables (i.e.: no destructuring).

A PAM rule $\langle c_p \mid K_p \rangle\updownarrow_p \hookrightarrow_l \mathrm{rhs}_p$ for language l is executed on a PAM state $\langle c_1 \mid K_1 \rangle\updownarrow_1$ as follows:

1. Find a substitution σ such that $\sigma(c_p) = c_1$, $\sigma(K_p) = K_1$. Fail if no such σ exists, or if $\updownarrow_p \neq \updownarrow_1$.

2. Recursively evaluate rhs_p as follows:

   - If $\mathrm{rhs}_p = \textbf{let } c_{\mathrm{ret}} = \mathrm{func}(\overline{c_{\mathrm{args}}}) \textbf{ in } \mathrm{rhs}'_p$, with $\mathrm{func} \in \mathsf{semfun}_l$, pick $r \in \mathrm{func}(\sigma(\overline{c_{\mathrm{args}}}))$, and extend σ to σ' s.t. $\sigma'(c_{\mathrm{ret}}) = r$ and $\sigma'(x) = \sigma(x)$ for all $x \in \mathrm{dom}(\sigma)$. Fail if no such σ' exists. Then recursively evaluate rhs'_p on σ'.

   - If $\mathrm{rhs}_p = \langle c'_p \mid K'_p \rangle\updownarrow'_p$, return the new PAM state $\langle \sigma(c'_p) \mid \sigma(K'_p) \rangle\updownarrow'_p$.

$$\begin{array}{rcl}\text{amState} & ::= & \langle c \,|\, K \rangle \\ \text{amRhs} & ::= & \text{amState} \mid \textbf{let } c = \mathsf{semfun}(\overline{c}) \textbf{ in } \text{amRhs} \\ \text{amRule} & ::= & \text{amState} \to \text{amRhs}\end{array}$$

Figure 4-11: Abstract Machines

Let us give some example PAM rules. (An example execution will be in Section 4.3.6.) The AssnCong and AssnEval rules from Section 4.2.1 get transformed into the following three rules:

$$\begin{array}{rcl}\big\langle (x := e, \mu) \,\big|\, k \big\rangle{\downarrow} & \hookrightarrow & \big\langle (e, \mu) \,\big|\, k \circ [(x := \square_t, \square_\mu)] \big\rangle{\downarrow} \\ \big\langle (t, \mu) \,\big|\, k \circ [(x := \square_t, \square_\mu)] \big\rangle{\uparrow} & \hookrightarrow & \big\langle (x := t, \mu) \,\big|\, k \big\rangle{\uparrow} \\ \big\langle (x := v, \mu) \,\big|\, k \big\rangle{\downarrow} & \hookrightarrow & \big\langle (\textbf{skip}, \mu[x \to v]) \,\big|\, k \big\rangle{\uparrow}\end{array}$$

The AssnCong rule becomes a pair of mutually-inverse PAM rules. One is a **down rule** which steps a down state to a down state, signaling a recursive call. The other is an **up rule**, corresponding to a return. This is a distinguishing feature of congruence rules, and is an important fact used when constructing the final abstract machine. Evaluation rules, on the other hand, typically become **down-up** rules. Note also that frames may be able to store information: here, the frame $[(x := \square_t, \square_\mu)]$ stores the variable to be assigned, x.

The PAM and operational semantics share a language's underlying semantic functions. The AddEval rule of, for instance, Section 4.3.2 becomes the following PAM rules:

$$\begin{array}{rcl}\big\langle (v_1 + v_2, \mu) \,\big|\, k \big\rangle{\downarrow} & \hookrightarrow & \textbf{let } n = +(v_1, v_2) \textbf{ in } \big\langle (n, \mu) \,\big|\, k \big\rangle{\downarrow} \\ \big\langle (v, \mu) \,\big|\, k \big\rangle{\downarrow} & \hookrightarrow & \big\langle (v, \mu) \,\big|\, k \big\rangle{\uparrow}\end{array}$$

The latter rule becomes redundant upon conversion to an abstract machine, which drops the phase.

4.3.4 Abstract Machines

Finally, the AM is similar to the PAM, except that an AM state does not contain a phase. Figure 4-11 gives a grammar for AM rules. We gave example rules for the AM for IMP in Section 4.2.1.

Summary of Notation This section has introduced three versions of semantics, each defining their own transition relation. Mnemonically, as the step-relation gets closer to the abstract machine, the arrow "flattens out." They are:

1. $c_1 \rightsquigarrow c_2$ ("squiggly arrow") denotes one SOS step (Section 4.3.2).

2. $c_1 \hookrightarrow c_2$ ("hook arrow") denotes one PAM step (Section 4.3.3).

3. $c_1 \rightarrow c_2$ ("straight arrow") denote one AM step (Section 4.3.4).

The conversion between PAM and AM introduces a fourth system, the *unfused abstract machine*, which is identical to the AM except that some rules of the AM correspond to several rules of the unfused AM. We thus do not distinguish it from the orthodox AM, except in one of the proofs, where its transition relation is given the symbol $\longrightarrow$ ("long arrow").

4.3.5 Splitting the SOS

In this section, we present our algorithm for converting an operational semantics to PAM. Figure 4-12 defines the sosToPam function, which computes this transformation.

The algorithm generates PAM rules for each SOS rule. For each SOS rule $c \rightsquigarrow \text{rhs}$, it begins in the state $\langle c \mid k \rangle \downarrow$, the start state for evaluation of c. It then generates rules corresponding to each part of rhs. For a semantic function, it transitions to a down state, and begins the next rule in the same down state, so that they may match in sequence. For a recursive invocation, it transitions to a down state $\langle c \mid k' \rangle \downarrow$, but begins the next rule in the state $\langle c \mid k' \rangle \uparrow$, so that other PAM rules must evaluate c before proceeding with computations corresponding to this SOS rule. Finally, upon

$$\boxed{\text{sosToPam}(\overline{\text{rules}})}$$

$$\begin{aligned}
\text{sosToPam}(\overline{\text{rules}}) \quad = \quad & \left(\bigcup_{r \in \overline{\text{rules}}} \text{sosRuleToPam}(r) \right) \\
& \cup \left\{ \left\langle (t, s) \,\middle|\, \textbf{emp} \right\rangle\!\uparrow \;\hookrightarrow\; \left\langle (t, s) \,\middle|\, \textbf{emp} \right\rangle\!\downarrow \right\} \\
& \textbf{where } t,\ s \text{ are fresh variables}
\end{aligned}$$

$$\boxed{\text{sosRuleToPam}(\text{rule})}$$

$$\begin{aligned}
\text{sosRuleToPam}(c \rightsquigarrow \text{rhs}) \quad = \quad & \text{sosRhsToPam}(\langle c \,|\, k \rangle \downarrow, k, \text{rhs}) && (1) \\
& \textbf{where } k \text{ is a fresh variable}
\end{aligned}$$

$$\boxed{\text{sosRhsToPam}(\text{pamState}, K, \text{rhs})}$$

$$\begin{aligned}
\text{sosRhsToPam}(s, k, c) \quad &= \quad \left\{ s \hookrightarrow \langle c \,|\, k \rangle\!\uparrow \right\} && (2) \\
\text{sosRhsToPam}(s, k, \textbf{let } [c_1 \rightsquigarrow c_2] \textbf{ in } \text{rhs}) \quad &= \quad \left\{ s \hookrightarrow \langle c_1 \,|\, k' \rangle\!\downarrow \right\} && (3) \\
& \quad\ \cup \text{sosRhsToPam}(\langle c_2 \,|\, k' \rangle\!\uparrow, k, \text{rhs}) \\
& \quad\ \textbf{where } k' = k \circ [c_2 \to \text{rhs}] \\
\text{sosRhsToPam}(s, k, \textbf{let } c_2 = \text{f}(\overline{c_1}) \textbf{ in } \text{rhs}) \quad &= \quad \left\{ s \hookrightarrow \textbf{let } c_2 = \text{f}(\overline{c_1}) \textbf{ in } \langle c_2 \,|\, k' \rangle\!\downarrow \right\} && (4) \\
& \quad\ \cup \text{sosRhsToPam}(\langle c_2 \,|\, k' \rangle\!\downarrow, k, \text{rhs}) \\
& \quad\ \textbf{where } k' = k \circ [c_2 \to \text{rhs}]
\end{aligned}$$

Figure 4-12: The SOS-to-PAM algorithm. Labels are used in the proofs of Appendix B.1.

encountering the end of the step c, it transitions to a state $\langle c \,|\, k \rangle\!\uparrow$, returning c up the stack.

For each step, it also pushes a frame containing the remnant of the SOS rhs onto the context, both to help ensure rules may only match in the desired order, and because the rhs may contain variables bound in the left-hand side, which must be preserved across rules.

After the algorithm finishes creating PAM rules for each of the SOS rules, it adds one special rule, called the **reset rule**: $\langle (t, s) \,|\, \mathbf{emp} \rangle\!\uparrow \hookrightarrow \langle (t, s) \,|\, \mathbf{emp} \rangle\!\downarrow$.

The reset rule takes a state which corresponds to completing one step of SOS evaluation, and changes the phase to $\downarrow$ so that evaluation may continue for another step. Note that it matches using a nonvalue-variable t so that it does not attempt to evaluate a term after termination. Note also that the LHS and RHS of the reset rule differ only in the phase. The translation from PAM to abstract machine hence removes this rule, as, upon dropping the phases, this rule would become a self-loop. It is also removed in the proof of Theorem 4.4.1.

4.3.6 Cutting PAM

The PAM evaluates a term in lockstep with the original SOS rules. Yet, in both, each step of computation always begins at the root of the term, rather than jumping from one subterm to the next. By optimizing these extra steps away, our algorithm will create the abstract machine from the PAM.

Consider how the PAM evaluates the term $(1 + (1 + 1)) + 1$, shown in Figure 4-13. Notice how lines 4.6–4.7 mirror lines 4.8–4.9. After evaluating $1 + 1$ deep within the term, the PAM walks up to the root, and then back down the same path. Knowing this, an optimized machine could jump directly from line 4.6 to 4.10.

This insight is similar to the one behind Danvy's *refocusing*, which converts a reduction semantics to an abstract machine [42]. But in the setting of PAM, the necessary property becomes particularly simple and mechanical:

Definition 4.3.4. *An up-rule* $\langle c_1 \,|\, K_1 \rangle\!\uparrow \hookrightarrow C[\, \langle c_2 \,|\, K_2 \rangle\!\uparrow \,]$ *is* ***invertible*** *if, for any* c_1

$$\big\langle ((1+(1+1))+1, \varnothing) \,\big|\, \mathbf{emp} \big\rangle\!\downarrow \tag{4.1}$$
$$\hookrightarrow \big\langle (1+(1+1), \varnothing) \,\big|\, \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \big\rangle\!\downarrow \tag{4.2}$$
$$\hookrightarrow \big\langle (1+1, \varnothing) \,\big|\, \mathbf{emp} \circ [(\square_t + 1, \square_\mu) \circ [(1 + \square_t, \mu)]] \big\rangle\!\downarrow \tag{4.3}$$
$$\hookrightarrow \big\langle (2, \varnothing) \,\big|\, \mathbf{emp} \circ [(\square_t + 1, \square_\mu) \circ [(1 + \square_t, \square_\mu)]] \big\rangle\!\downarrow \tag{4.4}$$
$$\hookrightarrow \big\langle (2, \varnothing) \,\big|\, \mathbf{emp} \circ [(\square_t + 1, \square_\mu) \circ [(1 + \square_t, \square_\mu)]] \big\rangle\!\uparrow \tag{4.5}$$
$$\hookrightarrow \big\langle (1+2, \varnothing) \,\big|\, \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \big\rangle\!\uparrow \tag{4.6}$$
$$\hookrightarrow \big\langle ((1+2)+1, \varnothing) \,\big|\, \mathbf{emp} \big\rangle\!\uparrow \tag{4.7}$$
$$\hookrightarrow \big\langle ((1+2)+1, \varnothing) \,\big|\, \mathbf{emp} \big\rangle\!\downarrow \tag{4.8}$$
$$\hookrightarrow \big\langle (1+2, \varnothing) \,\big|\, \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \big\rangle\!\downarrow \tag{4.9}$$
$$\hookrightarrow \big\langle (3, \varnothing) \,\big|\, \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \big\rangle\!\downarrow \tag{4.10}$$
$$\hookrightarrow \big\langle (3, \varnothing) \,\big|\, \mathbf{emp} \circ [(\square_t + 1, \square_\mu)] \big\rangle\!\uparrow \tag{4.11}$$
$$\hookrightarrow \big\langle (3+1, \varnothing) \,\big|\, \mathbf{emp} \big\rangle\!\uparrow \tag{4.12}$$
$$\hookrightarrow \big\langle (3+1, \varnothing) \,\big|\, \mathbf{emp} \big\rangle\!\downarrow \tag{4.13}$$
$$\hookrightarrow \big\langle (4, \varnothing) \,\big|\, \mathbf{emp} \big\rangle\!\downarrow \tag{4.14}$$
$$\hookrightarrow \big\langle (4, \varnothing) \,\big|\, \mathbf{emp} \big\rangle\!\uparrow \tag{4.15}$$

Figure 4-13: Example PAM derivation.

nonvalue, $\langle c_2 \,|\, K_2 \rangle\!\downarrow \; \hookrightarrow^* \; \langle c_1 \,|\, K_1 \rangle\!\downarrow$.

If an up-rule and its corresponding down-rules do not invoke any semantic functions, invertibility can be checked automatically via a reachability search. When all up-rules are invertible, we can show that $\langle c \,|\, K \rangle\!\uparrow \; \hookrightarrow^* \; \langle c \,|\, K \rangle\!\downarrow$ whenever c is a nonvalue, meaning that these transitions may be skipped (Lemma B.1.5). This means that all up-rules are redundant unless the LHS is a value, and hence they can be specialized to values. The phases now become redundant and can be removed, yielding the first abstract machine.

The last requirement for an abstract machine to be valid is not having any up-down rules, rules of the form $\langle c \,|\, k \rangle\!\uparrow \; \hookrightarrow C[\, \langle c' \,|\, k' \rangle\!\downarrow \,]$. An up-down rule follows from any SOS rule which simultaneously steps multiple subterms, and are not found in typical semantics. An example SOS rule which does result in an up-down rule is this lockstep-composition rule:

$$\frac{e_1 \rightsquigarrow e_1' \quad e_2 \rightsquigarrow e_2'}{e_1 \parallel e_2 \rightsquigarrow e_1' \parallel e_2'} \; \textit{LockstepComp}$$

The LockstepComp rule differs from normal parallel composition, in that both components must step simultaneously. Up-down rules like this break the locality of the transition system, meaning that whether one subterm can make consecutive steps may depend on different parts of the tree. Correspondingly, it also means that a single step of the program may step multiple parts of the tree, making it difficult to have a meaningful notion of program counter.

Most of the time, the presence of an up-down rule will also cause some up-rules to not be invertible, making a prohibition on up-down rules redundant. However, there are pathological cases where this is not so. For example, consider the expression $e_1 \parallel e_2$ with the LockstepComp rule. The LockstepComp rule splits into 3 PAM rules, of which the third is an up-rule, $\langle e_2' \,|\, k \circ [e_1' \parallel \Box] \rangle\!\uparrow \; \hookrightarrow \; \langle e_1' \parallel e_2' \,|\, k \rangle\!\uparrow$. If it is possible for e_1' to be a value but not e_2', then this rule is not invertible. However, if $e_1 \rightsquigarrow e_1$ and $e_2 \rightsquigarrow e_2$ for all e_1, e_2, then this rule is invertible. Hence, the PAM-to-AM algorithm includes an additional check that there are no up-down rules save the reset rule.

Algorithm: PAM to Unfused Abstract Machine

1. Check that all up-rules for l are invertible. Fail if not.
2. Check that there are no up-down rules other than the reset rule. Fail if not.
3. Remove the reset rule.
4. For each up-rule with LHS $\langle (t, s) \,|\, K \rangle\uparrow$, unify t with a fresh value variable. The resulting t' will either have a value node at the root, or will consist of a single value variable. If t fails to unify, remove this rule.
5. Remove all rules of the form $\langle c \,|\, K \rangle\downarrow \hookrightarrow \langle c \,|\, K \rangle\uparrow$, which would become self-loops.
6. Drop the phase $\updownarrow$ from the pamState's in all rules

Fusing the Abstract Machine This unfused abstract machine still takes more intermediate steps than a normal abstract machine. The final abstract machine is created by *fusing* successive rules together. A rule $\langle c_1 \,|\, K_1 \rangle \to C_1[\langle c_1' \,|\, K_1' \rangle]$ is fused with a rule $\langle c_2 \,|\, K_2 \rangle \to C_2[\langle c_2' \,|\, K_2' \rangle]$ by unifying (c_1', K_1') with (c_2, K_2), and replacing them with the new rule $\langle c_1 \,|\, K_1 \rangle \to C_1[C_2[\langle c_2' \,|\, K_2' \rangle]]$.

Property 4.3.5 (Fusion)**.** *Consider two AM rules F and G, and let their fusion be FG. Then* $\langle c \,|\, K \rangle \xrightarrow{F} \langle c' \,|\, K' \rangle \xrightarrow{G} \langle c'' \,|\, K'' \rangle$ *if and only if* $\langle c \,|\, K \rangle \xrightarrow{FG} \langle c'' \,|\, K'' \rangle$.

There are two cases where rules should be fused. First, considering Figure 4-12, rules which invoke a semantic function always have only one possible successor rule, and should be fused. Without this, the abstract machine for AddEval would have an extra state for after it invokes the semantic computation $+(n_1, n_2)$, but before it plugs the result into a term. Second, up-rules should be fused with all possible successors. Without this, computing $e_1 + e_2$ would have an extra state where, after evaluating e_1, it revisits $e_1 + e_2$, rather than jumping straight into evaluating e_2. Both steps are strictly optional. However, doing so generates abstract machine rules which match the standard versions (as in e.g.: [54]), and also generate more intuitive control-flow graphs.

For example, here are the final rules for assignment:

$$\begin{aligned}\big\langle (x := e, \mu) \,\big|\, k \big\rangle &\to \big\langle (e, \mu) \,\big|\, k \circ [(x := \square_t, \square_\mu)] \big\rangle \\ \big\langle (v, \mu) \,\big|\, k \circ [(x := \square_t, \square_\mu)] \big\rangle &\to \big\langle (\mathbf{skip}, \mu[x \to v]) \,\big|\, k \big\rangle\end{aligned}$$

And here are the final rules for addition:

$$\begin{aligned}\big\langle (e_1 + e_2, \mu) \,\big|\, K \big\rangle &\to \big\langle (e_1, \mu) \,\big|\, K \circ [(\square_t + e_2, \square_\mu)] \big\rangle \\ \big\langle (v, \mu) \,\big|\, K \circ [(\square_t + e, \square_\mu)] \big\rangle &\to \big\langle (e, \mu) \,\big|\, K \circ [(v + \square_t, \square_\mu)] \big\rangle \\ \big\langle (v_2, \mu) \,\big|\, K \circ [(v_1 + \square_t, \square_\mu)] \big\rangle &\to \mathbf{let}\ n = +(v_1, v_2)\ \mathbf{in}\ \big\langle (n, \mu) \,\big|\, K \big\rangle\end{aligned}$$

4.4 Correctness

This section provides the correspondence between the operational semantics and abstract machine. We present only the high-level theorems here, with the proofs available in Appendix B.1.

The core idea of the correspondence is simple: The PAM emulates the SOS because each PAM rule was explicitly constructed to correspond to an RHS fragment of the SOS:

Theorem 4.4.1. *$c_1 \leadsto_l^* c_2$ if and only if, for all contexts K, $\langle c_1 \,|\, K \rangle\!\downarrow \hookrightarrow_l^* \langle c_2 \,|\, K \rangle\!\uparrow$*

The PAM and AM are equivalent because the AM merely removes redundant steps from the PAM, and because fused rules in the AM each correspond to several rules in the PAM. However, a PAM derivation may have “false starts" corresponding to a partially-applied SOS rule, and so we first give some technical definitions that determine which states are included in the correspondences.

Stuck States The first kind of “false start" comes from steps that cannot be completed.

Definition 4.4.2. *A configuration/context pair* (c, K) *is* ***non-stuck*** *if* $\langle c \mid K \rangle\!\uparrow \hookrightarrow_l^* \langle c' \mid \textbf{\textit{emp}} \rangle\!\uparrow$ *for some* c'*.*

Because each PAM rule corresponds to part of an SOS rule, our definition of non-stuckness is different from the usual one: it is intended to exclude terms which correspond to a partial match on an SOS rule. A single step $c_1 \rightsquigarrow c_2$ in the SOS corresponds to a sequence $\langle c_1 \mid \textbf{emp} \rangle\!\downarrow \hookrightarrow^* \langle c_2 \mid \textbf{emp} \rangle\!\uparrow$ in the PAM, so a state is non-stuck if it can complete the current step. Stuck states result from SOS rules which only partially match a term. For example, the SOS rule

$$(a.b := v, \mu) \rightsquigarrow \textbf{let } (r, \mu') = \text{Lookup}((a, \mu)) \textbf{ in let false} = \text{ContainsField}(r, b) \textbf{ in } (\textbf{error}, \mu)$$

decomposes into 3 PAM rules. If Lookup succeeds, the first brings $\langle (a.b := v, \mu) \mid K \rangle\!\downarrow$ into the state

$$\langle (r, \mu') \mid K \circ [\textbf{let false} = \text{ContainsField}(\Box_t, b) \textbf{ in } (\textbf{error}, \mu)] \rangle\!\downarrow$$

If **false** $\neq$ ContainsField(r, b), then this will be a stuck state.

Working Steps As seen in the example in Section 4.3.6, many steps get removed when converting from PAM to AM. This causes the second form of “false start."

Definition 4.4.3. *An* ***inversion sequence*** *beginning at* $\langle c \mid K \rangle\!\uparrow$ *is a sequence of transitions* $\langle c \mid K \rangle\!\uparrow \hookrightarrow^* \langle c \mid K \rangle\!\downarrow$ *which contains at most one application of the reset rule.*

The idea of an inversion sequence partitions a derivation $\langle c_1 \mid K_1 \rangle\!\downarrow \hookrightarrow^* \langle c_2 \mid K_2 \rangle\!\downarrow$ into two parts: the inversion sequences, which do redundant work, and the remainder, which we call the **working steps**. A PAM state inside an inversion sequence might not correspond to any AM state.

Definition 4.4.4. *A reduction* $\langle c_1 \mid K_1 \rangle\!\updownarrow_1 \hookrightarrow \langle c_2 \mid K_2 \rangle\!\updownarrow_2$ *within a derivation is a* ***working step*** *if the derivation cannot be extended so that* $\langle c_1 \mid K_1 \rangle\!\updownarrow_1$ *is part of an inversion sequence.*

PAM-AM Correspondence The PAM and AM correspond as follows, via the Unfused AM.

Theorem 4.4.5 (PAM-Unfused AM: Forward)**.** *Suppose* $\langle c \,|\, K \rangle\!\downarrow \; \hookrightarrow_l^* \; \langle c' \,|\, K' \rangle\!\downarrow$*,* $\langle c' \,|\, K' \rangle\!\downarrow$ *is non-stuck, and the derivation's last step is working. Then there is a derivation* $\langle c \,|\, K \rangle \longrightarrow_l^* \langle c' \,|\, K' \rangle$*.*

Theorem 4.4.6 (PAM-Unfused AM: Backward)**.** *If* $\langle c \,|\, K \rangle \longrightarrow_l^* \langle c' \,|\, K' \rangle$*, then there are phases* $\updownarrow_c$ *and* $\updownarrow_{c'}$ *such that* $\langle c \,|\, K \rangle\!\updownarrow_c \; \hookrightarrow_l^* \; \langle c' \,|\, K' \rangle\!\updownarrow_{c'}$ *.*

Reductions in the Unfused AM correspond to reductions in the normal AM unless the last rules used in the Unfused AM have been fused away.

Theorem 4.4.7 (Unfused AM-AM)**.** $\langle c \,|\, K \rangle \to_l^* \langle c' \,|\, K' \rangle$ *if and only if* $\langle c \,|\, K \rangle \longrightarrow_l^* \langle c' \,|\, K' \rangle$ *by a sequence of rules whose last rule is not fused away.*

4.5 Control-flow Graphs as Abstractions

The abstract machine is a transition system describing all possible executions of a program. Applying an *abstract interpretation* shrinks this to a finite one. Some abstractions yield a graph resembling a traditional CFG (Section 4.5.3).

Yet to compute on abstract states, one must also abstract the transition rules, and this traditionally requires manual definitions. Fortunately, we will find that the desired families of CFG can be obtained by a specific class of *syntactic abstraction* which allow the transition rules to be abstracted automatically, via *abstract rewriting* (Section 4.5.1–Section 4.5.2). After constructing the abstract transition graph, more control can be obtained by further combining states using a *projection function* (Section 4.5.4). A final choice of control-flow graph is then obtained by an abstraction/projection pair (α, π).

The development of abstract-rewriting in this section is broadly similar to that of Bert et al [14], but departs greatly in details to fit our formalism of terms and machines. In Section 4.6, we explore connections to an older technique called *narrowing.*

4.5.1 Abstract Terms, Abstract Matching

Our goal is to find a notion of abstract terms flexible enough to allow us to express desired abstraction functions, but restricted enough that we can find a way to automatically apply the existing abstract machine rules to them. We accomplish this by defining a set of generalized terms $\mathsf{term}\star$, satisfying $\mathsf{term} \subset \mathsf{term}\star$, where some nodes have been replaced with $\star$ nodes which represent any term.

$$\begin{aligned} \mathsf{term}\star \quad ::= \quad & \mathrm{nonval}(\mathrm{sym}, \overline{\mathrm{term}\star}) \mid \mathrm{val}(\mathrm{sym}, \overline{\mathrm{term}\star}) \\ & \mid \mathrm{const} \mid x \mid \star_{\mathrm{mt}} \end{aligned}$$

Here, mt is a match type (Figure 4-7), so that the allowed $\star$ nodes are $\star_{\mathsf{Val}}$, $\star_{\mathsf{NonVal}}$, and $\star_{\mathsf{All}}$. Formally, we define an ordering $\prec$ on $\mathsf{term}\star$ as the reflexive, transitive, congruent closure of an ordering $\prec'$, defined by the following relations for all $t, s, \bar{t}$:

$$\mathrm{nonval}(s, \bar{t}) \prec' \star_{\mathsf{NonVal}} \qquad \mathrm{val}(s, \bar{t}) \prec' \star_{\mathsf{Val}} \qquad t \prec' \star_{\mathsf{All}}$$

A join operator $t_1 \sqcup t_2$ then follows as the least upper bound of t_1 and t_2. For instance, $(x := 1 + 1) \sqcup (y := 2 + 1) = (\star_{\mathsf{Val}} := \star_{\mathsf{Val}} + 1)$. We can then define the set of concrete terms represented by $\widehat{t} \in \mathsf{term}\star$ as:

$$\gamma\left(\widehat{t}\right) = \left\{t \in \mathsf{term} | t \prec \widehat{t}\right\}$$

The power of this definition of $\mathsf{term}\star$ is that it allows *abstract matching*, which allows the rewriting machinery behind abstract machines to be automatically lifted to abstract terms.

Definition 4.5.1 (Abstract Matching). *A pattern $t_p \in \mathsf{term}$ matches an abstract term $\widehat{t} \in \mathsf{term}\star$ if there is at least one $t \in \gamma(\widehat{t})$ and substitution σ_t such that $\sigma_t(t_p) = t$. The witness of the abstract match is a substitution $\widehat{\sigma}$ defined:*

$$\widehat{\sigma}(x) = \bigsqcup \left\{\sigma_t(x) | t \in \gamma\left(\widehat{t}\right) \wedge \sigma_t(t_p) = t\right\}$$

For example, the abstract term $\star_{\mathsf{All}}$ matches the pattern $v_1 + v_2$ with a witness $\widehat{\sigma}$ with $\widehat{\sigma}(v_1) = \widehat{\sigma}(v_2) = \star_{\mathsf{Val}}$. We are now ready to state the main property of abstract matching.

Property 4.5.2 (Abstract Matching (for terms))**.** *Let t_p be a pattern, and $t \in \textsf{term}$ be a matching term, so that there is a substitution σ with $\sigma(t_p) = t$. Consider a $t' \in \textsf{term}\star$ such that $t' \succ t$. Then t' matches t_p with witness σ', where σ' satisfies $\sigma(x) \prec \sigma'(x)$ for all $x \in dom(\sigma) = dom(\sigma')$, and $t \prec \sigma'(t_p)$.*

We assume there is some external definition of abstract reduction states $\widehat{\mathsf{State}_l}$ (discussed in Section 4.5.2). After doing so, the definitions of abstract terms and states can be lifted to abstract configurations $\mathsf{Conf}\star_l$, lists of abstract configurations $\overline{\mathsf{Conf}\star}$, contexts $\mathsf{Context}\star$, abstract machine states $\mathsf{amState}\star$, and abstract semantic functions $\mathsf{semfun}\star_l$, etc by transitively replacing all instances of term and State_l in their definitions with $\mathsf{term}\star$ and $\widehat{\mathsf{State}_l}$. Abstract matching and the Abstract Matching Property are lifted likewise. To define abstract rewriting, we need a few more preliminaries.

4.5.2 Abstract Rewriting

Abstract rewriting works by taking the (P)AM execution algorithm of Section 4.3.3, and using abstract matching in place of regular matching. Doing so effectively simulates the possible executions of an abstract machine on a large set of terms. To define it, we must extend $\prec$ to other components of an abstract machine state.

First, we assume there is some externally-defined notion of abstract reduction states $\widehat{\mathsf{State}_l} \supseteq \mathsf{State}_l$ with ordering $\prec$. There must also be a notion of substitution satisfying $\sigma_1(s) \prec \sigma_2(s)$ for $s \in \widehat{\mathsf{State}_l}$ if $\sigma_1(x) \prec \sigma_2(x)$ for all $x \in \mathrm{dom}(\sigma_1)$. Finally, there must be an abstract matching procedure for states satisfying the Abstract Matching Property. In the common case where State_l is the set of environments mapping variables to terms, this all follows by extending the normal definitions above to associative-commutative-idempotent terms.

We can now abstract matching and the $\prec$ ordering over abstract contexts $\mathsf{Context}\star$, abstract configurations $\mathsf{Conf}\star_l$, and lists of abstract configurations $\overline{\mathsf{Conf}\star_l}$ via congruence. We extend $\prec$ over sets of configurations $\mathbb{P}(\mathsf{Conf}\star_l)$ via the multiset ordering [48] , and extend $\prec$ over semantic functions pointwise, i.e.: for $f_1, f_2 \in \mathsf{semfun}_l$, $f_1 \prec f_2$ iff $f_1(x) \prec f_2(x)$ for all $x \in \overline{\mathsf{Conf}\star_l}$.

Because normal AM execution may invoke an external semantic function, we need some way to abstract the result of semantic functions. We assume there is some externally-defined set $\widehat{\mathsf{semfun}_l}$. Abstract rewriting will be hence parameterized over a "base abstraction" $\beta : \mathsf{semfun}_l \to \widehat{\mathsf{semfun}_l}$, satisfying the following property:

$$\beta(f)(\widehat{\overline{c}}) \succ \bigcup\{f(\overline{c}) | \overline{c} \in \gamma(\widehat{\overline{c}})\}$$

Informally, this condition means that running the abstracted function $\beta(f)$ over a list of abstract configurations must produce an output which abstracts all possible outputs from running f on the concrete configurations $\gamma(\widehat{\overline{c}})$. For example, let f be the addition function $f((t_1, \mu_1), (t_2, \mu_2)) = \{(t_1 + t_2, \varnothing)\}$ when t_1, t_2 are integers, and $\{\}$ otherwise. Define $\beta(f)(c_1, c_2) = \{(\star_{\mathsf{Val}}, \varnothing)\}$. Then $\beta(f)$ satisfies this condition: the possible concrete states are $S = \{\dots, (-1, \varnothing), (0, \varnothing), (1, \varnothing), \dots\}$, and $\{(\star_{\mathsf{Val}}, \varnothing)\} \succ S$ due to our use of the multiset ordering in the construction of $\succ$ (i.e.: $(\star_{\mathsf{Val}}, \varnothing) \succ s$ for all $s \in S$).

We are now ready to present abstract rewriting. An AM rule $\langle c_p \mid K_p \rangle \to_l \mathrm{rhs}_p$ for language l is abstractly executed on an AM state $\langle c_1 \mid K_1 \rangle$ using base abstraction β as follows:

1. Compute the abstract match of $\langle c_p \mid K_p \rangle$ and $\langle c_1 \mid K_1 \rangle$, giving witness $\widehat{\sigma}$; fail if they do not abstractly match.

2. Recursively evaluate rhs_p as follows:

   - If $\mathrm{rhs}_p = \textbf{let } c_{\mathrm{ret}} = \mathrm{func}(\overline{c_{\mathrm{args}}}) \textbf{ in } \mathrm{rhs}_p'$, with func $\in \mathsf{semfun}_l$, then pick $r \in \beta(\mathrm{func})(\widehat{\sigma}(\overline{c_{\mathrm{args}}}))$ and compute $\widehat{\sigma_r}$ as the witness of abstractly matching c_{ret} against r. Define $\widehat{\sigma'}$ by $\widehat{\sigma'}(x) = \widehat{\sigma}(x)$ for $x \in \mathrm{dom}(\widehat{\sigma})$, and $\widehat{\sigma'}(y) = \widehat{\sigma}_{\mathrm{r}}(y)$ for $y \in \mathrm{dom}(\widehat{\sigma}_{\mathrm{r}})$. Fail if no such $\widehat{\sigma'}$ exists. Then recursively evaluate rhs_p'

using $\widehat{\sigma'}$ as the new witness.

- If $\mathrm{rhs}_p = \langle c'_p \mid K'_p \rangle$, return the new abstract AM state $\langle \widehat{\sigma}(c'_p) \mid \widehat{\sigma}(K'_p) \rangle$.

If $\left\langle \widehat{c_1} \mid \widehat{K_1} \right\rangle$ steps to $\left\langle \widehat{c_2} \mid \widehat{K_2} \right\rangle$ by abstractly executing an AM rule with base abstraction β, we say that $\left\langle \widehat{c_1} \mid \widehat{K_1} \right\rangle \underset{\beta}{\widehat{\rightarrow}} \left\langle \widehat{c_2} \mid \widehat{K_2} \right\rangle$. Here is the fundamental property relating abstract and concrete rewriting:

Lemma 4.5.3 (Lifting Lemma)**.** *If* $\langle c_1 \mid K_1 \rangle \prec \left\langle \widehat{c_1} \mid \widehat{K_1} \right\rangle$*, and* $\langle c_1 \mid K_1 \rangle \rightarrow \langle c_2 \mid K_2 \rangle$ *by rule* F*, then, for any* β*, there is a* $\left\langle \widehat{c_2} \mid \widehat{K_2} \right\rangle$ *such that* $\langle c_2 \mid K_2 \rangle \prec \left\langle \widehat{c_2} \mid \widehat{K_2} \right\rangle$ *and* $\left\langle \widehat{c_1} \mid \widehat{K_1} \right\rangle \underset{\beta}{\widehat{\rightarrow}} \left\langle \widehat{c_2} \mid \widehat{K_2} \right\rangle$ *by* F*.*

Note that, if β is the identity function, then $\underset{\beta}{\widehat{\rightarrow}}$ is the same as $\rightarrow$. Hence, the Lifting Lemma theorem follows from a more general statement.

Lemma 4.5.4 (Generalized Lifting Lemma)**.** *Let* β_1, β_2 *be base abstractions where* β_1 *is pointwise less than* β_2*, i.e.:* $\beta_1(f)(\overline{c}) \prec \beta_2(f)(\overline{c})$ *for all* f, c*. Suppose* $\langle c_1 \mid K_1 \rangle \prec \left\langle \widehat{c_1} \mid \widehat{K_1} \right\rangle$*, and also* $\langle c_1 \mid K_1 \rangle \underset{\beta_1}{\widehat{\rightarrow}} \langle c_2 \mid K_2 \rangle$ *by rule* F*. Then there is a* $\left\langle \widehat{c_2} \mid \widehat{K_2} \right\rangle$ *such that* $\langle c_2 \mid K_2 \rangle \prec \left\langle \widehat{c_2} \mid \widehat{K_2} \right\rangle$ *and* $\left\langle \widehat{c_1} \mid \widehat{K_1} \right\rangle \underset{\beta_2}{\widehat{\rightarrow}} \left\langle \widehat{c_2} \mid \widehat{K_2} \right\rangle$ *by rule* F*.*

Proof. See Appendix B.2. □

4.5.3 Machine Abstractions

This section finally ties the knot on the adage "CFGs are an abstraction of control-flow" by defining the relation between a program's actual control flow (the concrete state transition graph) and its finite CFG. Intuitively, this relation involves skipping intermediate steps and combining states that correspond to the same program point. Yet getting the details right is tricky.

The simple function call `f()` exemplifies the difficulties of defining this relationship. Running `f()` may evaluate arbitrary code. (In the language of abstract rewriting, without context on `f`, it evaluates to an arbitrary program $\star_{\mathsf{NonVal}}$.) An intraprocedural CFG generator must generate a small fixed number of nodes (typically, 1 evaluation node or part of a basic-block node) to represent the call to `f()`; any additional nodes depending on the definition of `f` makes it no longer intraprocedural. It must then choose to either (1) draw edges through these nodes continuing onwards, or (2) when `f()` provably does not terminate, it may optionally draw edges into but not out of these nodes. Under a naive "combine states and skip steps" definition of valid abstraction, an intraprocedural CFG-generator must first perform an interprocedural termination analysis to be sound. And if `f()` only conditionally diverges, then both options are incorrect.

We shall present a relation which does indeed justify abstracting `f()` to $\star_{\mathsf{Val}}$, using a definition with subtle treatment of nontermination and branching. The upshot of this work is the ability to write control-flow abstractions with formal guarantees whose actual implementation is trivial; abstracting function calls as described here is but a small modification to the 5 lines of Figure 4-6. And, in spite of the definition's complexity, actually showing a candidate abstraction meets the definition is usually trivial.

Over the next paragraphs, we develop the abstraction preorder $a \sqsubseteq b$ between abstract states. From this, we obtain our first formal definition of a CFG: when G is the concrete transition graph for some program P, and H is a finite abstract transition graph, every state in G is $\sqsubseteq$ some state in H, and every state in H is $\sqsupseteq$ some state in G, then H is a valid CFG for P.

The $(\sqsubseteq)$ relation is built from three smaller relations. Clearly, $(\sqsubseteq)$ must contain the $(\prec)$ ordering, so that a single CFG node may describe concrete states that differ only in values. It should be able to ignore some steps of computation (e.g.: desugaring a while-loop), and so should involve $(\underset{\beta}{\hookrightarrow})$. It also must be able to skip over loops regardless of termination. We define the $(\lhd)$ relation to skip over infinite loops. Intuitively, $a = \left\langle (\widehat{t}, \widehat{\mu}) \,\middle|\, \widehat{K} \right\rangle \lhd \left\langle (\star_{\mathsf{Val}}, \top) \,\middle|\, \widehat{K}' \right\rangle$ if $\top$ is the "any" state (i.e.: overapproximates all possible effects) and all executions of a get "trapped" in some part of the program, with $\widehat{K}'$ never getting popped off the stack.

Definition 4.5.5 (Nontermination-cutting ordering). *Let $\top_l$ be a maximal element of $\widehat{\mathsf{State}_l}$. Consider a state $a = \left\langle (\widehat{t}, \widehat{s}) \,\middle|\, \widehat{K} \right\rangle \in \mathsf{amState}\star$, and let $\widehat{K}'$ be a subcontext of $\widehat{K}$. Suppose that, for all $\left\langle (t, s) \,\middle|\, K \right\rangle \prec a$, and for all derivations of the form*

$\langle (t,s) \,|\, K \rangle \to^* \langle (t',s') \,|\, K' \rangle$, *either K is a subterm of K', or there is a subderivation of the form* $\langle (t,s) \,|\, K \rangle \to^* \langle (t'',s'') \,|\, K \rangle$ *such that* $t'' \prec \star_{\mathsf{Val}}$*. Then* $a \lhd \left\langle (\star_{\mathsf{Val}}, \top_l) \,\middle|\, \widehat{K'} \right\rangle$*. If a does not satisfy this condition, then* $\nexists a'. a \lhd a'$*.*

We now combine the $(\prec), (\lhd)$, and $(\underset{\beta}{\widehat{\to}})$ orderings to fully describe what an abstraction may do. The naive approach would be to take the transitive closure of $(\prec) \cup (\lhd) \cup (\underset{\beta}{\widehat{\to}})$. But this would permit abstracting $\langle \textbf{if } \star_{\mathsf{Val}} \textbf{ then } A \textbf{ else } B \,|\, K \rangle$ to $\langle A \,|\, K \rangle$! Instead:

Definition 4.5.6 (Ordering $\sqsubseteq$ of $\mathsf{amState}\star$)**.** *The relation $\sqsubseteq$ is defined inductively as follows: $a \sqsubseteq b$ if any of the following hold:*

1. *$a = b$*
2. *For some c, $a \prec c$ and $c \sqsubseteq b$*
3. *For some c, $a \lhd c$ and $c \sqsubseteq b$*
4. *For all c such that $a \underset{\beta}{\widehat{\to}} c$, $c \sqsubseteq b$*

We can now define an abstract machine abstraction to be a pair (α, β), where β is a base abstraction and $\alpha : \mathsf{amState}\star \to \mathsf{amState}\star$ is a function which is an upper closure operator under the $\sqsubseteq$ ordering, meaning it is monotone and satisfies $x \sqsubseteq \alpha(x)$ and $\alpha(\alpha(x)) = \alpha(x)$. It is well-known that such an upper closure operator establishes a Galois connection between $\mathsf{amState}\star$ and the image of α, $\alpha(\mathsf{amState}\star)$ [124]. We will assume that every α is associated with a unique β, and will abbreviate the machine abstraction (α, β) as just α. Appendix B.3 adds a few additional technical restrictions on α. These restrictions have no bearing on interpreted-mode graph generation, but do rule out some pathological cases in the correctness proofs for compiled-mode graph generation.

We now define the abstract transition relation $\underset{\alpha}{\widehat{\to}}$ for α as: if $a \underset{\beta}{\widehat{\to}} b$, then $a \underset{\alpha}{\widehat{\to}} \alpha(b)$. We now state the fundamental theorem of abstract transitions, proved in Appendix B.2.

Theorem 4.5.7 (Abstract Transition)**.** *For $a, b \in \mathsf{amState}$, if α is an abstraction with base abstraction β, and $a \to b$, then either $b \sqsubseteq \alpha(a)$, or $\exists g \in \mathsf{amState}\star \,.\, \alpha(a) \underset{\alpha}{\widehat{\to}} g \wedge b \sqsubseteq g$.*

Following are some example abstractions.

Abstraction: Value-Irrelevance The value-irrelevance abstraction (code in Figure 4-6) maps each node $\text{val}(\text{sym}, t)$ and each constant to $\star_{\mathsf{Val}}$, and each semantic function to the constant $x \mapsto \star_{\mathsf{Val}}$. Combining this with the abstract machine for IMP yields an expression-level control-flow graph, as in Figure 4-2b. Tiger and MITScript use a modified version described at the beginning of this section, which also abstracts all function calls to $\star_{\mathsf{Val}}$.

Abstraction: Expression-Irrelevance This abstraction is like value-irrelevance, but it also "skips" the evaluation of expressions by mapping any expression under focus to $\star_{\mathsf{Val}}$. In doing so, it overapproximates modifications to the state from running e; the easiest implementation is to add the mapping $[\star_{\mathsf{Val}} \mapsto \star_{\mathsf{Val}}]$. Combining this with an abstract machine yields a statement-level control-flow graph.

The Boolean-Tracking Abstraction This abstraction is similar to the value-irrelevance abstraction, except that it preserves some **true** and **false** values. It differs in two ways: (1) boolean-valued semantic functions such as $<$ nondeterministically return $\{\textbf{true}, \textbf{false}\}$, (2) for a configuration (t, μ), it preserves all **true** and **false** values in t, as well as the value of $\mu(v)$ for each v in a provided set of *tracking variables V*. Including the variable "b" in the tracked set, and combining this with the basic-block projection (Section 4.5.4) yields the path-sensitive control-flow graph seen in Figure 4-2c.

As a limitation, note that this abstraction (nor any other abstraction of program state) does not give an account of path-sensitive analyses, which may assign different program points to states with identical values and identical continuations. This could be rectified by including information about the computation history as part of the program state, as in the history machines we developed in a different work [97].

4.5.4 Projections

A **projection**, also called a **quotient map**, is a function $\pi : \mathsf{amState}\star \to \mathsf{amState}\star$. They are used after constructing the initial CFG to merge together extra nodes, resulting in a blown-down graph.

The definition below comes from §5.4 of Manolios [114], which presented it in the context of bisimulations. It differs slightly from the standard graph-theoretic definitions, in that it has an extra condition to prevent spurious self-loops.

Definition 4.5.8. *Let (V, E) be a graph with $V \subseteq \mathsf{amState}\star$, $E \subseteq (\mathsf{amState}\star^2)$. Let π be a projection. Then the* ***projected graph*** *(or* ***quotient graph****) is the graph (V', E') satisfying:*

1. *$V' = \pi(V)$*
2. *For $a \neq b$, $(a, b) \in E'$ iff there is $(c, d) \in E$ such that $(a, b) = (\pi(c), \pi(d))$.*
3. *$(a, a) \in E'$ iff, for all $b \in V$ such that $\pi(b) = a$, there is $c \in V$ such that $(b, c) \in E$.*

Many of the uses of projections could be accomplished by instead using a coarser abstraction. However, projections have the advantage that they have no additional requirements to prove: they can be any arbitrary function. The addition of projections gives our final definition of a CFG: when G is the concrete transition graph for some program P, and H is a finite abstract transition graph, and every state in G is $\sqsubseteq$ some state in H and vice versa, then for any projection π, the projected graph of H under π is a valid CFG for P. In short, a CFG is **a projection of the transition graph of abstracted abstract machine states**.

Projections can be defined either manually, or automatically by the graph-pattern code generator (Section 4.6), and are most often used to hide internal details of a language's semantics, such as in the graph pattern of Figure 4-3, which merges away the internal steps of a while-loop which are mere artifacts of the SOS rules. But one important projection goes further:

Projection: Basic Block The basic-block projection inputs $\langle c \,|\, K \rangle$ and removes all but the last top-level sequence-nodes from c and K, essentially identifying each statement of a basic-block with the last statement in the block. In combination with the expression-irrelevance abstraction, this yields the classic basic-block control-flow graph, as in Figure 4-2a.

The correspondence between operational semantics and basic-block control-flow graphs is then given by Definition 4.5.8 and Theorems 4.4.1, 4.4.5, 4.4.6, and 4.5.7.

4.5.5 Termination

Will the interpreted-mode CFG-generation algorithm terminate in a finite control-flow graph? If the abstraction used is the identity, and the input program may have infinitely many concrete states, the answer is a clear no. If the abstraction reduces everything to $\star_{\mathsf{Val}}$, the answer is a clear yes. In general, it depends both on the abstraction as well as the rules of the language.

If CFG-generation terminates for a program P, that means there are only finitely-many states reachable under the $\underset{\alpha}{\widehat{\rightarrow}}$ relation from P's start state. Term-rewriting researchers call this property "global finiteness," and have proven it is usually equivalent to another property, "quasi-termination" [48]. While the literature on these properties can help, there must still be a separate proof for the termination of CFG -generation for every language/abstraction pair. Such a proof may be a tedious one of showing that e.g.: every while-loop steps to **if** e **then** s; **while** e **do** s **else skip**, which eventually steps back to **while** e **do** s, and never grows the stack.

Fortunately, for abstractions which discard context, the graph-pattern generation algorithm of Section 4.6 does this analysis automatically. Because the transitions discovered by abstract rewriting for a specific program are a subset of the union of graph patterns for all AST nodes in that program, we discuss in Section 4.6.1 and prove in Appendix B.3 that, if graph-pattern generation for a given language and abstraction terminates, then so does interpreted-mode CFG generation for all programs.

4.6 Syntax-Directed CFG Generators

Although it may sound like a large leap to go from generating CFGs for specific programs to statically inferring the control-flow of every possible node, it is actually only little more than running the CFG-generation algorithm of Section 4.5 on a term with variables. Indeed, the core implementation in MANDATE is only 22 lines of code. Hence, the description in Section 4.2.3 was already mostly complete, and we have only a few details to add. Correctness results are given in Appendix B.3.

There are two primary points of simplification in Section 4.2.3. First, it did not explain how to run AM rules on terms with variables. Second, it ignored match types.

Executing Terms with Variables Abstract rewriting can discover that $\star_{\mathsf{NonVal}} + \star_{\mathsf{NonVal}}$ steps to a state that executes a $\star_{\mathsf{NonVal}}$, but it cannot tell you which $\star_{\mathsf{NonVal}}$ it executes first. Yet there is a simple way to determine if an AM state with variables could be instantiated to match the LHS of an AM rule: if they unify. The result is then the RHS of the rule with the substitution applied. This shows that, from the term $a_{\mathsf{NonVal}} + b_{\mathsf{NonVal}}$, a_{NonVal} is evaluated first.

This operation is called *narrowing*, used since 1975 in decision procedures [103] and functional-logic programming. We present a variant of narrowing which makes it suitable for abstract interpretation of semantics: we say that $f \rightsquigarrow g$ if, for some rule $x \to y$, f and x unify by substitution σ, and $g = \sigma(y)$. The resulting relation $\underset{\alpha}{\widehat{\rightsquigarrow}}$ is defined identically to the development of $\underset{\alpha}{\widehat{\rightarrow}}$ given in Section 4.5.2 and Section 4.5.3, except that the witness $\widehat{\sigma}$ is computed by unification instead of by matching.

Note that the abstract-rewriting of Section 4.5 can be viewed as an overapproximation of our version of narrowing that follows each narrowing step with an abstraction that replaces each occurrence of the same with distinct fresh variables (i.e.: $\star$ nodes), along with extensions to handle match types and semantic functions.

Our abstract rewriting differs from conventional narrowing in an important way. Conventional narrowing is actually a ternary relation. Example: the rule $f(f(x)) \to x$ enables the derivation $f(y) \rightsquigarrow_{[y \mapsto f(y')]} y'$, with the unifying substitution as the third component of the relation. We turn it into a binary relation by applying the substitution on the right, and ignoring it on the left.

This small tweak changes the interpretation of narrowing while preserving its ability to overapproximate rewriting. In conventional narrowing, $[v_1 \mapsto v_2]$ represents an environment with a single, to-be-determined key/value pair. v_1 may unify with both the names “x” and “y”, but only in parallel universes. In abstract rewriting, $[v_1 \mapsto v_2]$ represents arbitrary environments, where any name maps to any value.

Graph-Pattern Generation (Now with Match-Types) The final requirement to generate graph patterns for a language l is that the ordering for State_l has a maximum value $\top_l$. Then, graph patterns are generated as follows: For each node type N, generate the abstract transition graph by narrowing from the start state $S = \left\langle (N(\overline{x^i_{\mathsf{NonVal}}}), \top_l) \,\middle|\, k \right\rangle$, where the x^i are arbitrary non-value variables, and k is fresh. Any time a state of the form $\langle (e_{\mathsf{NonVal}}, \mu) \,|\, K \rangle$ is encountered, instead of narrowing, add a transitive edge to $\langle (\star_{\mathsf{Val}}, \top_l) \,|\, K \rangle$. Halt at any state $\left\langle (v, \mu) \,\middle|\, k \right\rangle$, where k is the same context variable as S, and v, μ are an arbitrary value and reduction state.

From this, we see that the graph pattern in Figure 4-3 was slightly simplified. The real one has each starting variable annotated with NonVal and replaces each $\star$ node with $\star_{\mathsf{Val}}$.

Code-Generation

The code generator traverses the graph pattern, identifying subterm enter and exit states, and greedily merges unrecognized intermediate states with adjacent ones if one dominates the other, essentially building a custom projection. The output projection will have the property that each equivalence class of nodes forms a connected loop-free subgraph of the entire graph pattern. In the end, for an AST-node with k children, there will be up to $2k + 2$ graph nodes, an enter and exit CFG-node for the outer

AST node and each child. Once all abstract states have been merged into these graph nodes, the algorithm identifies the edges between the nodes, and outputs code like in Figure 4-3, with one `connect` statement per edge in the projected graph pattern.

The code generator guarantees that the generated code will generate a valid CFG (i.e.: a valid projection of the abstract transition graph). However, there are cases where a projection of the desired form does not exist, and hence the search terminates with failure. This occurs in cases where a single step in the source program corresponds to an arbitrary number of steps in the internal semantics (as in Java method resolution, which must traverse the class table). This scenario does not occur in TIGER and MITSCRIPT; it successfully generates code for both.

One subtle fact is that not every AST node will have a distinct exit node. For example, in all three languages, the graph pattern for **if** e **then** s_1 **else** s_2 will actually terminate in the exit nodes of s_1 and s_2, which both must be directly connected to whatever statement follows the if. The generated CFG-generators hence actually treat lists of enter/exit nodes as the atomic unit, rather than single nodes. Note that, if the language designer did want to have a distinct "join" node terminating the CFG for an if-statement, they could accomplish this by changing the semantics of if-statements to introduce an extra step after the body is evaluated.

4.6.1 An Automated Termination-Prover

As the normal abstract state transition graph overapproximates all concrete executions of a program, a graph pattern for a language construct overapproximates the relevant fragment of all abstract state transition graphs.

Is it possible to have finite graph patterns but infinite abstract transition graphs, i.e.: for the compiled-mode CFG generator to terminate, but not the interpreted-mode one? With some light assumptions, we can show this is not the case. Hence, while the output of the compiled-mode CFG-generator will always be less precise than that of an interpreted-mode generator, it turns out running the compiled-mode generator once per language will prove that the interpreted-mode generator terminates on all programs in that language.

```
name "assn—cong" $
mkRule5 (\x e e' mu mu' —>
 let (gx, ne, ge') = (GVar x, NVar e, GVar e')
 in StepTo (conf (Assign gx ne) mu)
     (LetStepTo (conf ge' mu') (conf ne mu)
      (Build $ conf (Assign gx ge') mu')))
```

Figure 4-14: Encoding of the AssnCong rule from Section 4.3.2

Table 4.1

	Interpreted			Compiled	
	E	S	P	E	S
IMP	✓	✓	✓	✓	✓
Tiger	✓	N/A	×	✓	N/A
MITScript	✓	✓	×	✓	✓

Theorem 4.6.1. *Let $a \in \mathsf{amState}_l$ and α be a machine abstraction. If the graph patterns under abstraction α for all nodes in a are finite, then only finitely many states are reachable from a under the $\underset{\alpha}{\widehat{\rightarrow}}$ relation.*

This proof requires a few more technical assumptions and a refinement to the definition of $\underset{\alpha}{\widehat{\rightsquigarrow}}$. The details are in Appendix B.3.

4.7 Deriving Control from a Mandate

We have implemented our approach in a tool called Mandate. Mandate takes as input an operational semantics for a language as an embedded Haskell DSL, and generates a control-flow graph generator for that language for every abstraction/projection supplied. It can then output a generated CFG to the graph-visualization language DOT. Mandate totals approximately 9600 lines of Haskell: 4100 lines in the core engine, and 5500 lines for our language definitions and example analyzers. 1350 of those lines define the 80 SOS rules for Tiger and 60 rules for MITScript, using the DSL depicted in Figure 4-14. 550 of those lines are automatically-generated CFG-generation code.

Table 4.1 lists the CFG-generators we have generated using Mandate. The

Table 4.2: Example analyzers

	LOC	IMP	TIGER	MIT
Constant-prop	115	✓	✓	✓
Paren-balancing	49	✓	×	×

columns E, S, and P correspond to the expression-level, statement-level, and path-sensitive CFGs from Section 4.1, and which are generated by the three abstractions from Section 4.5.3. The expression- and statement-level CFG-generators come in interpreted-mode and compiled-mode flavors. TIGER lacks a statement-level CFG-generator, because everything in TIGER is an expression. Section 4.7.1 explains the structure of heaps in TIGER and MITSCRIPT, and the implementation that would be required for MANDATE to support the boolean-tracking abstraction for them.

The high readability of the generated CFG-generators allows for easy inspection and comparison to intuitive control-flow. But, to further test the usefulness of the generated CFGs, we built two example analyzers, summarized in Table 4.2. The first is a simple constant-propagation analysis on expression-level CFGs, supporting assignments and integer arithmetic. The second is the parenthesis-balancing analyzer described in Section 4.1, built atop the path-sensitive CFG. Even though MANDATE was built as a demonstration of theory rather than as a practical tool, the simplicity of this exercise is further evidence that MANDATE's output does indeed correspond to conventional hand-written CFG-generators, while their brevity reinforces our thesis that **having the appropriate kind of CFG-generator greatly simplifies tool construction**.

In the remainder of this section, we demonstrate the power of Mandate by showing how it generates concise, readable code even in the face of complicated language constructs.

4.7.1 Control-Flow Graphs for TIGER and MITSCRIPT

Previous sections used IMP as the running language, which, in our implementation, has only 20 SOS rules, with low complexity. In this section, we explain how

```
(ConsFrame (HeapAddr 0) NilFrame,
 JustSimpMap $ SimpEnvMap $ Map.fromList
  [ (HeapAddr 0,
     ReducedRec
     $ RedRecCons (RedRecPair (Name "print")
                   (RefVal $ HeapAddr 1))
     $ RedRecCons (RedRecPair (Name "read")
                   (RefVal $ HeapAddr 2))
     $ RedRecCons (RedRecPair (Name "intcast")
                   (RefVal $ HeapAddr 3))
     $ Parent $ HeapAddr $ −1)
  , (HeapAddr 1, builtinPrint)
  , (HeapAddr 2, builtinRead)
  , (HeapAddr 3, builtinIntCast)
  ])
```

Figure 4-15: Starting state of MITSCRIPT programs

our techniques work when applied to two larger languages, TIGER and MITSCRIPT, which have 80 and 60 rules, respectively. It turns out that these do not introduce fundamental new challenges, although they do impose more stress on MANDATE's term-rewriting engine.

Reduction State The main difference between IMP and the larger languages is in the structure of their heap. In IMP, the reduction state was a simple map of variable names to values. In TIGER and MITSCRIPT, the reduction state must allow for stack frames, pointers, and closures. This reduction state is merely **a particularly-shaped pair of environment and term**, and involves **no extension** to the mechanics already used in IMP. Both TIGER and MITSCRIPT use the same design for their reduction state, designed as follows:

$$\begin{array}{lcl}
\mathsf{State} & ::= & (\mathrm{Stack}, \mathrm{Heap}) \\
\mathsf{HeapAddr} & ::= & \mathsf{Int} \\
\mathsf{Stack} & ::= & \mathrm{ConsFrame}(\mathsf{HeapAddr}, \mathsf{Stack}) \mid \mathrm{NilFrame} \\
\mathsf{Heap} & ::= & \mathsf{HeapAddr} \rightharpoonup \mathsf{Record} \\
\mathsf{Record} & ::= & (\mathsf{Symbol} \rightharpoonup \mathsf{Value}, \mathsf{HeapAddr}?)
\end{array}$$

The state consists of a stack and a heap. The stack is a list of heap addresses. The heap is a map of heap addresses to records, where each record contains a map of symbols to values. These records are used to store not only stack frames but also arrays and objects in the language. Each record optionally also contains a pointer to a parent record: looking up a symbol in a record will traverse the parent record if not found.

For an example heap, Figure 4-15 gives the starting heap of all MITScript programs, which contains hardcoded mappings of several strings to builtin functions. These functions are ordinary MITScript values defined elsewhere, whose bodies contain the special `Builtin` node, whose evaluation invokes an appropriate semantic function.

A limitation of the current Mandate implementation is that it does not support associative-commutative matching for nested maps. Hence, each heap record is implemented as an assoc-list rather than a map, with record lookup implemented as a semantic function. This is the reason why the boolean-tracking abstraction is not implemented for Tiger or MITScript.

Examples Our first example is MITScript if-statements, which illustrate how Mandate can generate multiple CFG-generators from the same semantics. When run in compiled-mode with the value-irrelevance abstraction, Mandate generates the code in Figure 4-16a for if-statements:

Note how the generator returns multiple exit-nodes for the if-statement. This means that the graph-generator will draw edges from both nodes to whatever comes

```
genCfg t@(Node "If" [a, b, c]) =
  do (tIn, tOut) <- makeInOut t
     (aIn, aOut) <- genCfg a
     (bIn, bOut) <- genCfg b
     (cIn, cOut) <- genCfg c
     connect tIn aIn
     connect aOut cIn
     connect aOut bIn
     return (inNodes [tIn],
             outNodes [bOut,cOut])
```

(a)

```
genCfg t@(Node "If" [_, a, b]) =
  do (tIn, tOut) <- makeInOut t
     (bIn, bOut) <- genCfg b
     (aIn, aOut) <- genCfg a
     connect tIn bIn
     connect tIn aIn
     return (inNodes [tIn],
             outNodes [bOut,aOut])
```

(b)

Figure 4-16

```
genCfg t@(Node "ForExp" [a, b, c, d]) =
  do (tIn, tOut) <- makeInOut t
     (bIn, bOut) <- genCfg b
     (cIn, cOut) <- genCfg c
     (dIn, dOut) <- genCfg d
     (aIn, aOut) <- genCfg a
     connect tIn bIn; connect dOut tOut
     connect dOut dIn; connect dOut tOut
     connect cOut dOut; connect bOut cIn
     return (inNodes [tIn], outNodes [tOut])
```

Figure 4-17: Genenerated Tiger for-loop CFG generator

after the if-statement. When run with the expression-irrelevance abstraction MANDATE produces code generating smaller graphs which do not have nodes to represent the evaluation of the condition, code shown in Figure 4-16b.

We next show our most impressive example, showcasing MANDATE's ability to cut through syntactic sugar to determine the control-flow behavior of a node type: TIGER for-loops. TIGER for-loops have more parts than for-loops in most languages. A for-loop in TIGER looks like this.

```
for a = b to c do
  d
```

The semantics are given by a single rule which desugars the above to:

```
let a := b
    __hi := c
in
  while (a <= __hi) do
    (d; a := a + 1)
```

MANDATE generates a graph pattern containing 46 nodes. MANDATE's code-generator projects these into just 8 states, for the enter and exit nodes of b, c, d, and the entire loop, yielding the code in Figure 4-17.

MANDATE has blown that giant graph down into just 5 edges. While MANDATE does not sort the `connect` statements, and does output one edge twice, it is still easy to see what the code is doing: it states that control first evaluates b, then c. The `connect cOut dOut` line is the most interesting one: it says that, after evaluating c (the upper boundary of the loop), control flows to the thing that happens after d is evaluated, namely the condition of the `while` loop, from which control flows either to the body of d or to the end of the entire loop.

It is impressive that MANDATE can generate short code for this construct with no reference to these internal computations, particularly considering that `while` is defined by expansion into `if`. **This code is generated completely automatically** from the TIGER semantics and the (function-skipping variant of the) value-irrelevance abstraction. The user need not even provide a projection; one is generated automatically by the code generator, by greedily merging nodes as described in Section 4.6.

Notes on Designing the Semantics Consider the following rule for evaluating the l-value of a field assignment:

$$(a.n := e, \mu) \rightsquigarrow \textbf{let } [(a, \mu) \rightsquigarrow (a', \mu')] \textbf{ in } ((a').n := e, \mu')$$

Upon running MANDATE in compiled-mode, rules like this produce graph-patterns which are too coarse. That's because MANDATE generates graph patterns for all assignments $x := e$, which could match this rule as well as the rules for assignments of variables or array indices. While we could modify MANDATE to produce separate graph patterns for $a.n := e$ vs. $a[i] := e$ or $x := e$, we opted instead to change the semantics. We instead modified the rule to this:

$$(l := e, \mu) \rightsquigarrow \textbf{let } [(l, \mu) \rightsquigarrow l', \mu')] \textbf{ in } (l' := e, \mu')$$

We then ensured there were separate node types for l-values vs. expressions of the form $a.n$, and defined rules to evaluate each kind of l-value.

The overall lesson is that MANDATE's compiled mode works best when rules match only on the topmost node. We have not yet needed to make MANDATE more sophisticated to overcome this problem because it is easier to modify the language semantics.

4.8 Conclusion

This work presented both an algorithm for constructing CFGs from first principles and the world's first CFG-generator generator. Yet our work also furthers three larger goals.

First, we have provided an answer to "what is a control-flow graph?" beyond the vague "a CFG is an abstraction of control-flow:" A CFG is a projection of the transition graph of abstracted abstract machine states. This fulfills our original impetus for this work, that of needing to create static analyzers with exotic notions of "program point."

Second, we have introduced abstract rewriting as a simple yet powerful technique for deriving tools from a language's semantics. We are excited by the idea of using it to derive other artifacts from language semantics, such as a symbol-table generator from the typing abstract-machine [152].

Third, we have used a language's semantics to derive a tool entirely unlike a semantics. Though it's long been known that a semantics can be executed to obtain an interpreter or even a symbolic-executor [148], we see our contribution as qualitatively different, and an important step towards the dream of being able to write down a language's syntax and semantics and automatically derive all desired tools.

Chapter 5

Related Work

5.1 Modular Language Tooling

Cubix is most directly based on the data types à la carte approach to modular syntax [160], and its extensions in work on compositional data types by [9]. The extension to multi-sorted terms was introduced in [185]. Other approaches to modular syntax include tagless-final [89], object algebras [186], and modular reifiable matching [38]. All these works share the same limitation: supporting a language requires building it from scratch in terms of special components. We overcame this limitation by using sort-injections to intermix a generic representation with one from existing frontends.

This work on modular syntax is joined by work on modular semantics, such as modular monadic semantics [108] and its cousin modular monadic meta-theory [47], as well as modular SOS [120] and its successor work on funcons [28]. These are used to build and verify interpreters for multiple languages, and will likely be necessary to extend our work to verifying multi-language transformations.

2003 saw a Dutch grant on language-parametric refactoring [167], building on a prototype by Lämmel [98, 76]. Their approach was to parameterize a transformation on (1) a fixed number of (language-specific) sorts used by the transformation, and (2) a set of primitive transformation operations, given as functions over these sorts. In this approach, the ASTs are opaque to the generic code, and hence essentially all computation happens in the language-specific functions. Conversely, in our approach, the generic code can manipulate the generic portions of a tree directly, which allows large chunks of a language-parametric transformation to be written similarly to a normal single-language rewrite.

But then in 2019, after the publication of Cubix, GitHub announced their `semantic` project [66], which bore an uncanny resemblance to Cubix: a multi-language program analysis toolkit based on (single sorted) data types à la carte, featuring a strategy combinator library with an interface similar to `compstrat`. Unlike Cubix, however, it lossily replaced all nodes in language ASTs with generic nodes, and was thus unsuitable for source-to-source transformation, being intended instead to provide symbol table creation to power GitHub's "jump to definition" feature. We spoke to several employees on the `semantic` team to compare notes. Despite the initial convergent evolution, because `semantic`'s goals differed substantially from those of Cubix, it soon diverged, soon replacing the DLC approach with a set of incompatible ASTs but with some common interface, reminiscent of Lämmel et al's language-parametric refactoring approach.

Sort injections are an instance of the concept of *feature interactions* from the field of software product lines [168]. A similar idea is seen in the TruffleVM [70] to allow language runtimes to exchange messages.

The past decade has seen extensive work in *language workbenches*, which are designed to make it easy to implement languages and transformations on them. They include Spoofax and its component Stratego [87], Rascal [91], TXL [36], Semantic Designs DMS [12], and JetBrains MPS [174]. These were extensively surveyed in Erdweg et al [53]. All these share the limitation that, while they make it easy to define languages and write transformations, the resulting transformations can only run on one representation of one language. At best they can be used to implement the "Clang-style" common representation, discussed in Section 2.1.1.

One recent work that echoes our work on Cubix is Brown et al's [23] work using *island grammars* [119] to write static analyzers for multiple languages. They show that they only need to represent fragments of a language to construct an analyzer. Their analyzers are still built for a single language, and they resort to cloning code to implement them for others. They do not address transformation.

Incremental concrete syntax [51] is a technique using island grammars to construct parsers. It focuses on concrete syntax (parsing), whereas ours focuses on abstract syntax (representation).

In spite of the promise of statistical approaches for producing language-agnostic tools, few multi-language tools have been created by such methods in domains where correct code output is required. For example, though several groups have experimented with neural program repair, ENCORE [112] bills itself as the first automated program-repair tool to address multiple languages, achieving a 33% fix rate on QuixBugs, our own multi-language program repair benchmark [1].

5.2 Control-Flow

CFG Generation There have been a few works on techniques for constructing control-flow graphs. FLOWSPEC [157], a component of the SPOOFAX language workbench [87], contains a DSL for specifying control-flow graphs in terms of single-pushout graph rewrite rules [111] reminiscent of the GrGen graph-rewriting language [63]. Though it yields compact definitions, it has no support for generic programming. Semantic Designs DMS [12] has a DSL based on attribute grammars for constructing CFGs. The results are quite verbose; their Java CFG-generator is over 5000 lines.

CFG-based Transformation Despite extensive work on advanced techniques in program transformation [172], there has been a paucity on work which uses a graph to aid in transformation. We only know of two works in this category. The more well-known one is Coccinelle [130]. The capabilities of Coccinelle can be succinctly described as: simultaneous associative matching/rewriting of multiple tree patterns, connected by arbitrary control-flow paths. In a personal conversation, one of the Coccinelle creators summarized “We use the CFG in matching, but not in rewriting." Though it provides an ellipsis (“...") primitive which matches arbitrary sets of control-flow paths, each end of a path is ordinary tree rewriting. Coccinelle would still need many cases to handle the examples of Section 2.7.1, though "..." may help with `continue` statements. More relevant and more obscure is a technique of Griswold [72] used in the world's first refactoring tool. Though based on program-dependence graphs [55] rather than CFGs, it shares the characteristic of more closely tying the graph and rewriting, by providing simultaneous updates on the AST and PDG.

Control-Flow Analysis Many papers have been written on control-flow analysis [116, 156, 85, 80]. Older research tries to manually construct a complicated analysis of programs with highly-dynamic control flow. Our work on MANDATE automatically constructs CFG-generators from first principles. Our goal in MANDATE is not to analyze complex programs, but to match the work of hand-written CFG-generators with minimal user input.

As such, we do not consider MANDATE to be part of the literature on control-flow analysis. Owing to their different emphasis, these works uniformly have three limitations that make them unsuitable for automatically deriving CFG-generators:

1. While they explain how a human could define a new analysis for different languages, their analyses are ultimately manually defined for each language. They further repeat this manual construction for every abstraction used.

2. They check that their result safely approximates executions, but pay no attention to the shape of the graph.

3. Most importantly, they manually partition program states into equivalence classes. That is, they manually annotate the program with labels or program-points, using these as CFG nodes. This is a hindrance to both automation and theory, as most type theories do not contain labels. This was particularly important for MANDATE, which was originally motivated by a planned project on combining analyses with different notions of program point (i.e.: which partition program states differently), which makes not hardcoding them especially important.

We now present a brief overview of research on control-flow analysis. CFA research typically focuses on functional languages, with an emphasis on overapproximating the potential targets of anonymous function calls. Perhaps the most famous such work is the k-CFA analysis of Shivers [156]. This and many other works frame their analysis as an abstract interpretation of executions, though there are too many approaches to describe here; Midtgaard [116] gives an extensive survey.

There are two works from this field which deserve special mention for relevance to Mandate. The first is Jones [85], which is the first to define control-flow as an abstract interpretation of executions. Through a modern lens, it is also the first to do so by abstracting abstract machines: it presents an abstraction of a custom abstract machine resembling a CC machine [54]. It shares the limitations mentioned above, although its representation of program points is subtle: it uses a set of tokens representing the different function applications of the source program.

The second is Jagannathan and Weeks [80], due to its focus on generality, explicit construction of graphs, and attempt to relate their construction to an operational semantics. It is also the most extreme in its use of manual program-point annotations, going as far as to design an abstract machine with an explicit program counter.

5.3 Tools from Semantics

Work with Related Goals Our work on Mandate joins a small-but-growing body of work on mechanizing the generation of programming tools. Others include generating static analyzers via multiple executions of interpreters or semantics [44, 153, 43, 17], using an executable language semantics as a tool (by the K Framework [148]), and our own work in the language-parametric construction of programming tools. Our work is similar to tools built with the K Framework in that both start with a semantics; ours differs by transforming the semantics into an applied tool, whereas K is limited to applications directly based on equational reasoning, namely interpreters and symbolic executors. Bodin et al [17] briefly mention having a 0-CFA for the lambda calculus generated from their “skeletal semantics;" in personal communication, they described it as “working code, but not a principled approach" and “We don’t do a CFG-generator."

Transformation of Semantics There are a few projects that transform semantics for one language into semantics for a related language. Examples include transforming rules to support gradual types [29, 30], and deriving new rules for types and scoping of syntactic sugar [136, 137].

More closely related are projects that transform semantics presented in one formalism into identical semantics in a different formalism, mostly done by Danvy and his students [39, 42, 184, 4, 41, 40, 15], with a few by others. Hannan and Miller [74] is perhaps the most known older work. But it is very limited: They give a few examples of manually deriving an abstract machine from big-step semantics, using a sequence of up to 6 hand-proven transformations. Ager [3] continues this work, providing a simpler and automated approach, though with some additional limitations, such as needing nondeterminism to execute if-statements. Other works in this area include Poulsen and Mosses [139], Huizing et al [78], and Vesely and Fisher [171].

When we first began work on Mandate, we planned to simply look up an algorithm to convert SOS to abstract machines, but surprisingly found none existed. Almost all work in this space is by Danvy and associates, and, while their papers focus on individual formalisms, Danvy personally told us that he is not interested in small-step semantics because it would not require new techniques over his prior work, and that he is similarly uninterested in creating a general algorithm, when he’s already sketched how to do the transformation for a single example [39]. So, although Danvy may personally know how to do the transformation, **there is no prior published algorithm for converting SOS to abstract machines**.

Our next plan was to convert the SOS to reduction semantics and then apply Danvy’s algorithm for converting reduction semantics to an abstract machine [42]. However, we found the assumptions of this algorithm too limiting, as it prohibits any use of external semantic functions or state. Rather than extend this algorithm, we found it much easier to use PAM as an intermediate step, because its nature as a modified term-rewriting system gives us access to unification-based techniques for analyzing and transforming it. For instance, while proving up-rules invertible is a reachability search taking 20 lines of code, proving the equivalent property for reduction semantics, unique decomposition, took a 20-page paper [184].

To our knowledge, the TIGER and MITSCRIPT languages in this thesis are by far the largest languages to undergo automated conversion between two forms of semantics; prior work focuses on simple lambda calculi. And Danvy told us the largest example of a *mechanized* (hand-guided) transformation from his group is a toy model of Prolog [16]; ours are far larger.

Abstracting Abstract Machines (AAM) The first known work explicitly on abstract interpretation of abstract machines was by Midtgaard and Jensen [118, 117], though a modern reader may also consider Jones [85] to be an earlier example. This was followed by the "abstracting abstract machines" (AAM) line of work began by Van Horn and Might [169, 179, 67, 68].

These take a substantially different flavor from our own, due to their focus on higher-order analyses of functional languages. A key technical difference is the choice of abstraction operator: they use variants of store-bounding, whereas we use syntactic abstractions designed to make the abstract state space resemble a conventional CFG. The use of these syntactic abstractions allows our algorithm to automatically abstract the transition rules of an abstract machine via abstract rewriting. Their paper's approach only automatically abstracts reads and writes to an abstract store; the abstract transition steps are still manually defined. For example, the algorithm in this thesis can take in a rule like $\langle \mathbf{true} \mid k \circ [(\mathbf{if}\ \square_t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, \square_k)] \rangle \rightarrow \langle s_1 \mid k \rangle$ and the corresponding rule for **false**, and deduce that if-statements may step into either branch because both rules match a single $\star$ state. The approach in their paper can at best nondeterministically evaluate an expression to separate $\{\mathbf{true}, \mathbf{false}\}$ states and then match both if-rules, which produces an un-CFG-like graph where the branching happens prior to the if-statement.

Abstract Rewriting and Syntactic Abstraction Abstract rewriting was introduced by Bert and Echahed in the early 90's [14, 13] and has received little attention since. We can thus only compare to their work. While the details differ substantially owing to their different focus (approximating the possible normal forms of a term), it has some common elements with our development: they split nodes into "constructors" and "completely-defined operators," resembling our value/nonvalue split, and use a $\top$ node with similar semantics to our $\star$. A major point of departure in their development is that, in their system, each abstract step must overapproximate all concrete transitions from an abstract term. A newer related technique from a different lineage is rewriting modulo SMT [147], which operates on (numeric) symbolic terms constrained by an SMT formula (e.g.: linear arithmetic). We are interested in future work combining these techniques to automatically derive more-precise analyses.

Outside of Bert & Echahed, there are a few works which share minor details or terminology with our development of abstract rewriting. In the broadest sense, abstract unification means taking some algorithm that uses unification, and replacing the unification with some other operation. Prior work in this sense comes from work on the abstract interpretation of logic programs [88, 37]. These algorithms commonly replace unification with a simple operation such as tracking sharing and linearity, for the purpose of, e.g.: eliminating backtracking or the occurs-check (see §10 of [37] for a literature review).

In contrast, the abstract matching procedure in our work (and in Bert & Echahed) is an extension of normal matching to a set of abstract terms, consisting of normal terms augmented with a set of "blown-down" terms. This technique appears absent from the literature on static analysis of logic programs, for it is not useful for traditional static analysis by abstract interpretation, as the domain of abstract terms is infinite (meaning: our algorithm cannot compute a single control-flow graph which usefully describes all possible programs).

Apart from Bert & Echahed, we have found a couple papers that use syntactic abstraction, albeit in a different form. An early use is the "star abstraction" of Codish et al [32, 33], which merges identical subterms of a tree, and is unrelated to the similarly-named abstraction in this thesis. Schmidt [150] uses this to merge identical processes in a CCS-like system, bounding executions to aid in model-checking. Although it is not discussed in the paper, Glaze [68] does some syntactic abstraction in the implementation, representing all numbers as identical number nodes.

The term "abstract matching" also has an unrelated meaning in the model-checking community, where it refers to finding equivalences between abstract states [133].

The ARM abstract rewriting machine [86] provides a compact instruction set for executing term-rewriting systems efficiently. It handles ordinary rewriting, rather than abstract rewriting in our sense.

Chapter 6

Conclusion

Programmers create immaterial wealth for the world in the form of useful software, and so programmer productivity is the first derivative of software wealth. Work on building tools contributes to the second derivative of wealth, and meta-metaprogramming to the third derivative. Yet because tools are themselves software, tool development promises to create a positive feedback loop giving exponential growth in the rate of software wealth creation so long as problems remain amenable to tooling, and so advances in meta-metaprogramming will hasten this exponential.

In Chapter 1, we described two major technical barriers making tools laborious to build, the linguistic complexity problem and the heterogeneity problem, the latter being subdivided into the problems of closedness, type invariance, and bidirectionality. This thesis has dealt several blows against these problems barriers. What stands between now and victory? What more must be done to achieve the programming utopia tantalized by the union of tools papers, where a magic button to assist with virtually any programming task in any language of interest is but a credit-card swipe away?

With Mandate, we have performed an automated conversion of semantics on by far the largest language, and ultimately automatically derived a tool from its semantics. Yet the semantics formalism it acts on, though expressive, is still far too limited to scale to a full semantics of C or Java (c.f.: [75, 20]), and a CFG-generator is still a tiny piece of a practical tool. ECTAs unlock a new category of problems that can be given to a general solver, but their success still requires future generations of researchers to use it for ever-more applications, just as SMT and e-graphs took many years to begin reaching their full potential. Cubix and Yogo are closest to commercial impact. With Cubix we have built the world's first multi-language program transformations, including ones developed simultaneously for 5 languages, while we used Yogo to find a performance bug in a 1.2 million line codebase that had been missed by a handwritten analyzer designed for that kind of bugs. Yet Yogo still gives its output in difficult-to-read text dumps, and Cubix only supports 5 languages and has a number of engineering limitations. For instance, Cubix still cannot effectively handle multiple dialects of a language such as GCC vs. Microsoft C or Java 7 vs. Java 8 — not for any good reason, but simply because generalizing certain constraints hits idiosyncratic performance bugs in the Haskell compiler.

In short, we have knocked a large hole through the barrier of type invariance, and made large dents in the problems of closedness and linguistic complexity, and a small one in bidirectionality. Yet the barriers all still stand.

This thesis has also not touched at all on the *usability* and *discoverability* of tools. The usability problem is almost self-explanatory: the larger a task a tool assists or automates, the more input or interaction it needs with the user to ensure it is carrying out their will. As for discoverability: given the sheer variety of distinct programming problems tools address, many of which went unnamed prior to a tool being built, there will be a huge problem of connecting programmers to the right tool for a given moment. We can envision browsing source code with some advanced representation where an appropriate tool can be suggested in a context menu, or perhaps even an AI that watches a programmer type or reads programming communications and guesses their larger task. These are problems that must be addressed by future researchers, using techniques foreign to this thesis such as software visualization, HCI, and NLP.

Yet the many works discussed in Chapter 5 show that we are not alone. In the not-so-distant future, achieving breakthroughs in the ease of tool construction will come not from novel techniques, but from the immense knowledge and design needed to integrate much of this literature into an integrated system or framework.

Fixing the Y2K bug cost a large fraction of the world's output. Its scarier successor, the 2038 problem, is coming near. Yet the increasing sophistication of software tools, and our increasing ability to build them as furthered by this thesis, gives hope that rectifying Y2038 will not be such a world stopping effort, and similar for the many smaller problems that stymie software creation and change worldwide, such as the Facebook and Dropbox problem of Chapter 2 that dominated both companies for a time. With greater ease of scaling, the research prototypes of today can become the practical tools of tomorrow. Software victory shall be attained.

Bibliography

[1] QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge, author=Lin, Derrick and Koppel, James and Chen, Angela and Solar-Lezama, Armando. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56. ACM, 2017.

[2] Michael D Adams and Matthew Might. Restricting Grammars with Tree Automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–25, 2017.

[3] Mads Sig Ager. From Natural Semantics to Abstract Machines. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 245–261. Springer, 2004.

[4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM, 2003.

[5] Ferran Alet, Javier Lopez-Contreras, James Koppel, Maxwell Nye, Armando Solar-Lezama, Tomas Lozano-Perez, Leslie Kaelbling, and Joshua Tenenbaum. A Large-Scale Benchmark for Few-Shot Program Induction and Synthesis. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 175–186. PMLR, 18–24 Jul 2021.

[6] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[7] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge university press, 1999.

[8] Leo Bachmair and Nachum Dershowitz. Equational Inference, Canonical Proofs, and Proof Orderings. *Journal of the ACM (JACM)*, 41(2):236–276, 1994.

[9] Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 83–94, 2011.

[10] Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. The Emptiness Problem for Tree Automata with Global Constraints. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 263–272. IEEE, 2010.

[11] Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. Decidable Classes of Tree Automata Mixing Local and Global Constraints Modulo Flat Theories. *Logical Methods in Computer Science*, 9, 02 2013.

[12] Ira D Baxter, Christopher Pidgeon, and Michael Mehlich. DMS®: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.

[13] Didier Bert and Rachid Echahed. Abstraction of Conditional Term Rewriting Systems. In *Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, USA, December 4-7, 1995*, pages 162–176, 1995.

[14] Didier Bert, Rachid Echahed, and Bjarte M Østvold. Abstract Rewriting. In *International Workshop on Static Analysis*, pages 178–192. Springer, 1993.

[15] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, Department of Computer Science, Aarhus University, Aarhus, Denmark, 2006.

[16] Dariusz Biernacki and Olivier Danvy. From Interpreter to Logic Engine by Defunctionalization. In *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*, pages 143–159, 2003.

[17] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and Their Interpretations. *PACMPL*, 3(POPL):44:1–44:31, 2019.

[18] Bruno Bogaert, Franck Seynhaeve, and Sophie Tison. The Recognizability Problem for Tree Automata with Comparisons Between Brothers. In *International Conference on Foundations of Software Science and Computation Structure*, pages 150–164. Springer, 1999.

[19] Bruno Bogaert and Sophie Tison. Equality and Disequality Constraints on Direct Subterms in Tree Automata. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 159–171. Springer, 1992.

[20] Denis Bogdanas and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 445–456, 2015.

[21] Niklas Broberg. language-java: Manipulating Java Source. `http://hackage.haskell.org/package/language-java-0.2.8`, November 2015.

[22] Frederick P Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Pearson Education, 1995.

[23] Fraser Brown, Andres Nötzli, and Dawson Engler. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–157. ACM, 2016.

[24] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *CAV*, 2011.

[25] Yuandao Cai, Peisen Yao, and Charles Zhang. Canary: Practical Static Detection of Inter-Thread Value-Flow Bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1126–1140, 2021.

[26] Michael Carbin and Armando Solar-Lezama. MITScript Language Specification. `http://6.s081.scripts.mit.edu/sp18/handout-pdfs/specification.pdf`, 2018.

[27] Adam Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 143–156, 2008.

[28] Martin Churchill, Peter D Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable Components of Semantic Specifications. In *Transactions on Aspect-Oriented Software Development XII*, pages 132–179. Springer, 2015.

[29] Matteo Cimini and Jeremy G. Siek. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 443–455, 2016.

[30] Matteo Cimini and Jeremy G. Siek. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 789–803, 2017.

[31] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José F Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[32] Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 451–464, 1993.

[33] Michael Codish, Moreno Falaschi, and Kim Marriott. Suspension Analysis for Concurrent Logic Programs. In *Logic Programming, Proceedings of the Eigth International Conference, Paris, France, June 24-28, 1991*, pages 331–345, 1991.

[34] Aurelien Coet. Staticfg. `https://github.com/coetaur0/staticfg/tree/9948ab8574c254f69564da46c0ca30e5ac0c35a5`, 2020.

[35] Hubert Comon. Tree Automata Techniques and Applications. *http://www. grappa. univ-lille3. fr/tata*, 1997.

[36] James R Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.

[37] Patrick Cousot and Radhia Cousot. Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming*, 13(2-3):103–179, 1992.

[38] Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. Modular Reifiable Matching: A List-of-Functors Approach to Two-level Types. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 82–93, 2015.

[39] Olivier Danvy. Defunctionalized Interpreters for Programming Languages. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 131–142, 2008.

[40] Olivier Danvy and Jacob Johannsen. Inter-Deriving Semantic Artifacts for Object-Oriented Programming. *Journal of Computer and System Sciences*, 76(5):302–323, 2010.

[41] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On Inter-Deriving Small-Step and Big-Step Semantics: A Case Study for Storeless Call-by-Need Evaluation. *Theoretical Computer Science*, 435:21–42, 2012.

[42] Olivier Danvy and Lasse R Nielsen. Refocusing in Reduction Semantics. *BRICS Report Series*, 11(26), 2004.

[43] David Darais, Nicholas Labich, Phúc C Nguyen, and David Van Horn. Abstracting Definitional Interpreters (Functional Pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):12, 2017.

[44] David Darais, Matthew Might, and David Van Horn. Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 552–571, 2015.

[45] Max Dauchet. Rewriting and Tree Automata. In *French School on Theoretical Computer Science*, pages 95–113. Springer, 1993.

[46] Max Dauchet, Anne-Cécile Caron, and Jean-Luc Coquidé. Automata for reduction properties solving. *Journal of Symbolic Computation*, 20(2):215–233, 1995.

[47] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C. d. S. Oliveira. Modular Monadic Meta-Theory. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 319–330, 2013.

[48] Nachum Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1-2):69–115, 1987.

[49] David Detlefs, Greg Nelson, and James B Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM*, 52(3):365–743, 2005.

[50] Isil Dillig, Thomas Dillig, and Alex Aiken. SAIL: Static Analysis Intermediate Language with a Two-level Representation. *Stanford University Technical Report*, 2009.

[51] Tom Dinkelaker, Michael Eichberg, and Mira Mezini. Incremental Concrete Syntax for Embedded Languages with Support for Separate Compilation. *Science of Computer Programming*, 78(6):615–632, 2013.

[52] Michael Dory, Allison Parrish, and Brendan Berg. *Introduction to Tornado: Modern Web Applications with Python.* O'Reilly Media, Inc., 2012.

[53] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The State of the Art in Language Workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[54] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* Mit Press, 2009.

[55] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[56] John Feser, Sam Madden, Nan Tang, and Armando Solar-Lezama. Deductive Optimization of Relational Data Storage. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[57] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasoning*, 33(3):341–383, 2004.

[58] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993.

[59] Charles L Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[60] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):17–es, 2007.

[61] Python Software Foundation. CPython Test Suite. Version 3.7.0a0. `https://docs.python.org/devguide/runtests.html`, October 2016.

[62] FSF. C Language Testsuites: "C-torture". Revision 240758. `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`, October 2016.

[63] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *International Conference on Graph Transformation*, pages 383–397. Springer, 2006.

[64] Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. On Tree Automata that Certify Termination of Left-Linear Term Rewriting Systems. *Information and Computation*, 205(4):512–534, 2007.

[65] Andy Gill. A Haskell Hosted DSL for Writing Transformation Systems. In *Domain-Specific Languages*, pages 285–309. Springer, 2009.

[66] GitHub, Inc. semantic. `https://github.com/github/semantic`, 2019.

[67] Dionna Amalie Glaze and David Van Horn. Abstracting Abstract Control. In *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 11–22, 2014.

[68] Dionna Amalie Glaze, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing Abstract Abstract Machines. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 443–454, 2013.

[69] Sebastian Graf, Simon Peyton Jones, and Ryan G Scott. Lower your guards: A compositional pattern-match coverage checker. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, 2020.

[70] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 78–90, 2015.

[71] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Inc., 2014.

[72] William G. Griswold. Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool. In *SIGSOFT '93, Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering, Los Angeles, California, USA, December 7-10, 1993*, pages 42–55, 1993.

[73] Sumit Gulwani. Automating String Processing in Spreadsheets using Input-Output Examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[74] John Hannan and Dale Miller. From Operational Semantics to Abstract Machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[75] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 336–345, 2015.

[76] Jan Heering and Ralf Lämmel. Generic Software Transformations. In *Proceedings of the Software Transformation Systems Workshop*, 2004.

[77] Benedikt Huber. language-c: Analysis and Generation of C Code. `http://hackage.haskell.org/package/language-c`, 2016.

[78] Cornelis Huizing, Ron Koymans, and Ruurd Kuiper. A Small Step for Mankind. In *Concurrency, Compositionality, and Correctness*, pages 66–73. Springer, 2010.

[79] Husain Ibraheem and David A Schmidt. Adapting Big-Step Semantics to Small-Step Style: Coinductive Interpretations and "Higher-Order" Derivations. *Electronic Notes in Theoretical Computer Science*, 10:121, 1997.

[80] Suresh Jagannathan and Stephen Weeks. A Unified Treatment of Flow Analysis in Higher-Order Languages. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 393–407, 1995.

[81] Michael B James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for Fold: Synthesis-Aided API Discovery for Haskell. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–27, 2020.

[82] JDK Bug System. javac crash when local from enclosing context is captured multiple times. `https://bugs.openjdk.java.net/browse/JDK-8169345`, 2016.

[83] Julian Jensen. ast-flow-graph. `https://github.com/julianjensen/ast-flow-graph/tree/0c669d1dad54fa28004741a7e9cf82eee8d683e2`, 2020.

[84] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S Foster, and Armando Solar-Lezama. Synthesizing Framework Models for Symbolic Execution. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 156–167. IEEE, 2016.

[85] Neil D Jones. Flow Analysis of Lambda Expressions. In *International Colloquium on Automata, Languages, and Programming*, pages 114–128. Springer, 1981.

[86] Jasper FT Kamperman and Humphrey Robert Walters. ARM Abstract Rewriting Machine. 1993.

[87] Lennart CL Kats and Eelco Visser. *The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs*, volume 45. ACM, 2010.

[88] Andy King and Mark Longley. Abstract Matching Can Improve on Abstract Unification. 1995.

[89] Oleg Kiselyov. Typed Tagless Final Interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

[90] Csongor Kiss, Matthew Pickering, and Nicolas Wu. Generic Deriving of Generic Traversals. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018.

[91] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 222–289. Springer, 2009.

[92] Edward A. Kmett. lens: Lenses, Folds and Traversals. `http://hackage.haskell.org/package/lens-4.19.2`, April 2020.

[93] James Koppel. Version Space Algebras are Acyclic Tree Automata, 2021.

[94] James Koppel, Jackson Kearl, and Armando Solar-Lezama. Automatically Deriving Control-Flow Graph Generators from Operational Semantics, 2020.

[95] James Koppel, Sreenidhi Nair, and Armando Solar-Lezama. One CFG-Generator to Rule them All. `http://www.jameskoppel.com/files/papers/cubix_cfg.pdf`.

[96] James Koppel, Varot Premtoon, and Armando Solar-Lezama. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[97] James Koppel, Gabriel Scherer, and Armando Solar-Lezama. Capturing the Future by Replaying the Past (Functional Pearl). *Proceedings of the ACM on Programming Languages*, 2(ICFP):76, 2018.

[98] Ralf Lämmel. Towards Generic Refactoring. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, pages 15–28. ACM, 2002.

[99] Ralf Lämmel and Simon Peyton Jones. *Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming*, volume 38. ACM, 2003.

[100] Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic Programming Meets Adaptive Programming. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 168–177, New York, NY, USA, 2003. ACM.

[101] Ralf Lämmel and Joost Visser. Typed Combinators for Generic Traversal. In *International Symposium on Practical Aspects of Declarative Languages*, pages 137–154. Springer, 2002.

[102] Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In *International Symposium on Practical Aspects of Declarative Languages*, pages 357–375. Springer, 2003.

[103] Dallas S Lankford. *Canonical Inference.* University of Texas, Department of Mathematics and Computer Sciences, 1975.

[104] Thomas D LaToza and Brad A Myers. Hard-to-Answer Questions about Code. In *Evaluation and Usability of Programming Languages and Tools*, pages 1–6. 2010.

[105] Thomas David LaToza. *Answering Reachability Questions.* PhD thesis, Carnegie Mellon University, 2012.

[106] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[107] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. Programming by Demonstration Using Version Space Algebra. *Machine Learning*, 53(1):111–156, 2003.

[108] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343. ACM, 1995.

[109] Jay P Lim, Mridul Aanjaneya, John Gustafson, and Santosh Nagarakatte. An Approach to Generate Correctly Rounded Math Libraries for New Floating Point Variants. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–30, 2021.

[110] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition.* Pearson Education, 2014.

[111] Michael Löwe. Algebraic Approach to Single-Pushout Graph Transformation. *Theoretical Computer Science*, 109(1-2):181–224, 1993.

[112] Thibaud Lutellier, Lawrence Pang, Hung Viet Pham, Moshi Wei, and Lin Tan. ENCORE: ensemble learning using convolution neural machine translation for automatic program repair. *CoRR*, abs/1906.08691, 2019.

[113] Sam Madden. 6.830 Lab 1: SimpleDB. `http://db.csail.mit.edu/6.830/assignments/lab1.html`, 2017.

[114] Panagiotis Manolios. *Mechanical Verification of Reactive Systems.* PhD thesis, University of Texas at Austin, 2001.

[115] Krzysztof Maziarz, Tom Ellis, Alan Lawrence, Andrew Fitzgibbon, and Simon Peyton Jones. Hashing Modulo Alpha-Equivalence. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 960–973, 2021.

[116] Jan Midtgaard. Control-flow Analysis of Functional Programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, June 2012.

[117] Jan Midtgaard and Thomas P. Jensen. Control-flow Analysis of Function Calls and Returns by Abstract Interpretation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming.*

[118] Jan Midtgaard and Thomas P. Jensen. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, pages 347–362, 2008.

[119] Leon Moonen. Generating Robust Parsers Using Island Grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22. IEEE, 2001.

[120] Peter D. Mosses. Modular Structural Operational Semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[121] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing Structured CAD models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–44, 2020.

[122] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[123] Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, 1980.

[124] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2015.

[125] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL (T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[126] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. Polyglot: An Extensible Compiler Framework for Java. In *International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[127] Bruno CdS Oliveira and William R Cook. Functional Programming with Structured Graphs. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 77–88, 2012.

[128] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 511–525, 2021.

[129] Ömer Sinan Ağacan and Eric Mertens. language-lua: Lua Parser and Pretty-Printer. `http://hackage.haskell.org/package/language-lua-0.10.0`, August 2016.

[130] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. *Acm sigops operating systems review*, 42(4):247–260, 2008.

[131] Pavel Panchekha and Emina Torlak. Automated Reasoning for Web Page Layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 181–194, 2016.

[132] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *ACM SIGPLAN Notices*, volume 50, pages 346–356. ACM, 2015.

[133] Corina S Păsăreanu, Radek Pelánek, and Willem Visser. Concrete Model Checking with Abstract Matching and Refinement. In *International Conference on Computer Aided Verification*, pages 52–66. Springer, 2005.

[134] Joshua Pollock and Altan Haan. E-Graphs Are Minimal Deterministic Finite Tree Automata (DFTAs) · Discussion #104 · egraphs-good/egg, 2021.

[135] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

[136] Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. Inferring Scope through Syntactic Sugar. *Proceedings of the ACM on Programming Languages*, 1(ICFP):44, 2017.

[137] Pombrio, Justin and Krishnamurthi, Shriram. Inferring Type Rules for Syntactic Sugar. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 812–825. ACM, 2018.

[138] Bernard James Pope. language-python: Parsing and Pretty Printing of Python Code. http://hackage.haskell.org/package/language-python-0.5.4, July 2016.

[139] Casper Bach Poulsen and Peter D Mosses. Deriving Pretty-Big-Step Semantics from Small-Step Semantics. In *European Symposium on Programming Languages and Systems*, pages 270–289. Springer, 2014.

[140] Varot Premtoon. Multi-Language Code Search. Master's thesis, Massachusetts Institute of Technology, 2019.

[141] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1066–1082, 2020.

[142] LLVM project authors. Clang. https://github.com/llvm/llvm-project/tree/eed333149d178b69fdaf39b9419b7ca032520182, 2020.

[143] Polyglot project authors. Polyglot. https://github.com/polyglot-compiler/polyglot/tree/6235855368ce3b0ab27cb29cd117ca5d0fba54e7, 2020.

[144] PUC-Rio. Lua: Test suites. Version 5.3.3. https://www.lua.org/tests/, 2016.

[145] Andreas Reuß and Helmut Seidl. Bottom-up Tree Automata with Term Constraints. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 581–593. Springer, 2010.

[146] Charles Rich and Richard C Waters. The Programmer's Apprentice. *Computer*, 21j(11):10–25, 1988.

[147] Camilo Rocha, José Meseguer, and César Muñoz. Rewriting Modulo SMT and Open System Analysis. *Journal of Logical and Algebraic Methods in Programming*, 86(1):269–297, 2017.

[148] Grigore Roşu and Traian Florin Şerbănută. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[149] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian G. Elbaum. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 191–201, 2015.

[150] David A Schmidt. Abstract Interpretation of Small-Step Semantics. In *LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, pages 76–99. Springer, 1996.

[151] Semantic Designs, Inc. Test Coverage tools. `http://www.semanticdesigns.com/Products/TestCoverage/`, 2005.

[152] Ilya Sergey and Dave Clarke. From Type Checking by Recursive Descent to Type Checking with an Abstract Machine. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*, page 2. ACM, 2011.

[153] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic Abstract Interpreters. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 399–410, 2013.

[154] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 390–405, 2021.

[155] Tim Sheard and Simon Peyton Jones. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 1–16. ACM, 2002.

[156] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[157] Jeff Smits and Eelco Visser. FlowSpec: Declarative Dataflow Analysis Specification. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pages 221–231, 2017.

[158] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Demanded Abstract Interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 282–295, 2021.

[159] Bjarne Stroustrup. *The C++ Programming Language.* Pearson Education, 2013.

[160] Wouter Swierstra. Data Types à la Carte. *Journal of Functional Programming*, 18(04):423–436, 2008.

[161] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. IncA: A DSL for the Definition of Incremental Program Analyses. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 320–331. IEEE, 2016.

[162] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: A New Approach to Optimization. In *Proceedings of the Symposium on Principles of Programming Languages*, Savannah, GA, 2009.

[163] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 264–276, 2009.

[164] ECMA TC39. Test262: ECMAScript Language Conformance Test Suite. Version 5.1. `http://test262.ecmascript.org`, 2014.

[165] Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. Example-Guided Synthesis of Relational Queries. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1110–1125, 2021.

[166] Danny van Bruggen. JavaParser: Process Java Code Programmatically. `http://javaparser.org`, 2016.

[167] Mark van de Brand, Jan Heering, Paul Klint, Ralf Lämmel, and Christian Verhoef. Language-Parametric Program Restructuring. `http://www.cs.vu.nl/lppr/abstract/abstract.html`, 2003.

[168] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.

[169] David Van Horn and Matthew Might. Abstracting Abstract Machines. In *15th ACM SIGPLAN International Conference on Functional Programming, ICFP'10*, 2010.

[170] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–30, 2021.

[171] Ferdinand Vesely and Kathleen Fisher. One Step at a Time. In *European Symposium on Programming*, pages 205–231. Springer, Cham, 2019.

[172] Eelco Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[173] Sebastiaan Visser, Erik Hesselink, Chris Eidhof, and Sjoerd Visscher. fclabels: First Class Accessor Labels Implemented as Lenses. `http://hackage.haskell.org/package/fclabels-2.0.5`, May 2020.

[174] Markus Voelter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1449–1450. IEEE, 2012.

[175] Philip Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313. ACM, 1987.

[176] Philip Wadler. The Expression Problem. *Java-genericity mailing list*, 1998.

[177] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of Data Completion Scripts using Finite Tree Automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

[178] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program Synthesis using Abstraction Refinement. *Proc. ACM Program. Lang.*, 2(POPL):63:1–63:30, 2018.

[179] Guannan Wei, James Decker, and Tiark Rompf. Refunctionalization of Abstract Abstract Machines: Bridging the Gap Between Abstract Abstract Machines and Abstract Definitional Interpreters (Functional Pearl). *Proceedings of the ACM on Programming Languages*, 2(ICFP):105, 2018.

[180] Stefan Wellek. *Testing Statistical Hypotheses of Equivalence and Noninferiority*. CRC Press, 2010.

[181] Wikipedia contributors. Duff's device — Wikipedia, the free encyclopedia, 2021. [Online; accessed 25-August-2021].

[182] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[183] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 187–204, 2013.

[184] Yong Xiao, Amr Sabry, and Zena M. Ariola. From Syntactic Theories to Interpreters: Automating the Proof of Unique Decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.

[185] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 233–244, 2009.

[186] Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. Scrap Your Boilerplate with Object Algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 127–146, 2015.

[187] Shaowei Zhu and Zachary Kincaid. Termination Analysis without the Tears. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1296–1311, 2021.

[188] Alan Zimmerman. language-javascript: Parser for JavaScript. `http://hackage.haskell.org/package/language-javascript-0.6.0.9`, November 2016.

Appendix A

The ADT Modularization Transformation

This appendix gives a formal definition for the ADT modularization transformation implemented in `comptrans`, described in Section 2.3.4. This algorithm transforms a syntax definition given as a family of mutually recursive ADTs into an equivalent definition in the sum-of-signatures representations, expressed as a collection of GADT definitions combined with an explicit sum and fixpoint.

Figure A-1 gives a syntax for GADTs. GADTs and ADTs are given as a set of constructors with a given type. We assume that there is a corresponding set of type constructors. We assume that ADTs have monomorphic types, and their associated constructors have the base kind. Conversely, GADTs may have polymorphic types with equality constraints, and their constructors may have higher kinds. We use the syntax $\forall \overline{\nu} : \overline{k}.\overline{D} \Rightarrow \sigma$ as sugar for nested forall types. The language includes two "container" functors, lists and pairs, to give an example of how the transformation deals with containers embedded in syntax trees. Figure A-2 gives the kinding rules for ADTs and GADTs which check if the constructor types are well-formed. Γ is a local context storing all type variables in scope, while we assume Φ has been populated with the types of all constructors declared.

To close a set of GADT definitions into a sum-of-signatures representation of a syntax, the language must be extended with sum and recursive types, and with the ability to instantiate a polymorphic type. Figure A-3 gives this extension and the corresponding typing rules. Note that the sum types in this language are of polymorphic kind. To avoid the need to track constraints with variables, the rule for polymorphic type application recurses into the left-hand side until the constraint can be checked syntactically. This has the unfortunate consequence that kind-checking may become circular. These typing rules should hence be interpreted with greatest fixed point semantics, meaning that circularly-defined judgments hold.

Figure A-5 gives the transformation algorithm. The transformation replaces every ADT constructor with a type of kind $*$ with a GADT constructor with a type constructor of kind $(* \to *) \to * \to *$. More importantly, these GADT type constructors do not refer to any other type definitions, with the exception of the "label" types, which are purely nominal and uninhabited. It makes use of three auxiliary functions. newcon(con) returns a fresh name for a GADT constructor. newconType(ν) does similar for the corresponding type constructors. lab(C) maps each constructor C to a corresponding "label" constructor of kind $*$.

Of final note, in order to inhabit terms of sort List γ or Pair $\gamma\ \iota$, there are three specially-defined constructors, given in Figure A-4. These are given Curry-style types, meaning their types contain an unbound variable, and they may be given a type for any instantiation of that variable.

We are now ready to state the property this transformation was designed to satisfy: For a family of mutually recursive ADTs defined by $\overline{con^{\sigma}}$ with root type C, C is equivalent to the sum of the generated GADTs at sort lab(C). Formally, if Φ contains the types for all declared ADT constructors and $\Phi \vdash \overline{con^{\sigma}}$ okay, then

$$\Phi \vdash C \equiv (\mu\alpha : * \to *.(\mathrm{PairF} + \mathrm{ListF} + \sum_{s \in \overline{\sigma}} \mathrm{transType}(s))\ \alpha)\ \mathrm{lab}(C)$$

Here, $\equiv$ denotes the classical notion of a type isomorphism, i.e.: the presence of a pair of mutually inverse functions that convert from one to the other. This is still an informal statement, albeit in formal notation, as we have not fully defined the language of terms which is needed to make this statement fully rigorous. The description in this appendix is meant only to unambiguously describe the algorithm.

Type variables	$\alpha, \beta, \ldots$
Predefined constructors	C
Kinds	$k ::= * \mid k \rightarrow k$
Primitive types	$P ::= \mathrm{Int} \mid \mathrm{Bool} \mid \ldots$
Container functors	$F ::= \mathrm{List} \mid \mathrm{Pair}$
Base types	$\nu ::= \alpha \mid C \mid F \mid P \mid \nu\nu$
Monotypes	$\tau ::= \nu \mid \nu \rightarrow \tau$
Constraints	$D ::= \cdot \mid \alpha \sim \nu$
Polytypes	$\sigma ::= \tau \mid \forall \alpha : k.D \Rightarrow \sigma$
Constructors	$c ::= \mathrm{con}^{\sigma}$

Figure A-1: A syntax for GADTs

$$\frac{}{\Gamma; \Phi \vdash P : *}\ \mathit{PRIM}$$

$$\frac{}{\Gamma; \Phi \vdash \mathrm{List} : * \rightarrow *}\ \mathit{LIST}$$

$$\frac{}{\Gamma; \Phi \vdash \mathrm{Pair} : * \rightarrow * \rightarrow *}\ \mathit{PAIR}$$

$$\frac{\alpha : k \in \Gamma}{\Gamma; \Phi \vdash \alpha : k}\ \mathit{VAR}$$

$$\frac{C : k \in \Phi}{\Gamma; \Phi \vdash C : k}\ \mathit{CON}$$

$$\frac{\Gamma; \Phi \vdash \nu : * \quad \Gamma; \Phi \vdash \tau : *}{\Gamma; \Phi \vdash \nu \rightarrow \tau : *}\ \mathit{ARR}$$

$$\frac{\Gamma, \alpha : k_1; \Phi \vdash \sigma : k_2}{\Gamma; \Phi \vdash \forall \alpha : k_1.D \Rightarrow \sigma : k_1 \rightarrow k_2}\ \mathit{FORALL}$$

$$\frac{\Gamma; \Phi \vdash \nu_1 : k_1 \rightarrow k_2 \quad \Gamma; \Phi \vdash \nu_2 : k_1}{\Gamma; \Phi \vdash \nu_1\nu_2 : k_2}\ \mathit{APP}$$

$$\frac{\cdot; \Phi \vdash \sigma : *}{\Phi \vdash \mathrm{con}^{\sigma}\ \mathrm{okay}}$$

Figure A-2

$$\sigma ::= \ldots \mid \sigma + \sigma \mid \mu\alpha : k.\ \sigma \mid \sigma\sigma$$

$$\frac{\Gamma;\Phi \vdash \sigma_1 : k \quad \Gamma;\Phi \vdash \sigma_2 : k}{\Gamma;\Phi \vdash \sigma_1 + \sigma_2 : k}\ POLY-SUM$$

$$\frac{\Gamma, \alpha : k;\Phi \vdash \sigma : k}{\Gamma;\Phi \vdash \mu\alpha : k.\sigma : k}\ REC$$

$$\frac{\Gamma,\Phi \vdash \sigma_1 + \sigma_2 : k' \quad \Gamma,\Phi \vdash \sigma_1\sigma_3 : k}{\Gamma,\Phi \vdash (\sigma_1 + \sigma_2)\sigma_3 : k}\ POLY-APP-SUM-LEFT$$

$$\frac{\Gamma,\Phi \vdash \sigma_1 + \sigma_2 : k' \quad \Gamma,\Phi \vdash \sigma_2\sigma_3 : k}{\Gamma,\Phi \vdash (\sigma_1 + \sigma_2)\sigma_3 : k}\ POLY-APP-SUM-RIGHT$$

$$\frac{\Gamma,\Phi \vdash (\mu\alpha : k.\sigma) : k \quad \Gamma,\Phi \vdash (\sigma[(\mu\alpha : k.\ \sigma)/\alpha])\sigma' : k'}{\Gamma,\Phi \vdash (\mu\alpha : k.\ \sigma)\sigma' : k'}\ POLY-APP-REC$$

$$\frac{\Gamma,\Phi \vdash \nu : k' \quad \Gamma,\Phi \vdash \sigma[\nu/\alpha] : k}{\Gamma,\Phi \vdash (\forall\alpha : k'.\cdot \Rightarrow \sigma)\nu : k}\ POLY-APP$$

$$\frac{\Gamma,\Phi \vdash \nu : k' \quad \Gamma,\Phi \vdash \sigma[\nu/\alpha] : k}{\Gamma,\Phi \vdash (\forall\alpha : k'.\alpha \sim \nu \Rightarrow \sigma)\nu : k}\ POLY-APP-CONSTRAINT$$

Figure A-3

ConsF : $\forall\,(\alpha : * \to *)\,(\gamma : *).\gamma \sim \mathit{List}\,\iota \Rightarrow \alpha\,\iota \;\to\; \alpha\,\gamma \;\to \text{ListF}\,\alpha\,\gamma$

(for any ι)

NilF : $\forall\,(\alpha : * \to *)\,(\gamma : *).\gamma \sim \mathit{List}\,\iota \Rightarrow \text{ListF}\,\alpha\,\gamma$

(for any ι)

PairF : $\forall(\alpha : * \to *)\,(\gamma : *).\gamma \sim (\text{Pair}\;\iota\,\kappa) \Rightarrow \alpha\,\iota \to\; \alpha\,\kappa \to \text{PairF}\,\alpha\,\gamma$

(for any ι, κ)

Figure A-4

$$\boxed{\text{trans}(\text{con}^\tau)}$$

$$\text{trans}(\text{con}^\tau) = \text{newcon}(\text{con})^{\text{transTypeTop}(\tau)}$$

$$\boxed{\text{transTypeTop}(\tau)}$$

$$\text{transTypeTop}(\tau) = \forall(\alpha : * \to *).\ \cdot\ \Rightarrow (\forall\, (\gamma : *).\ \gamma \sim \text{getSort}(\tau)\ \Rightarrow \text{transType}(\tau, \alpha, \gamma))$$

$$\boxed{\text{getSort}(\tau)}$$

$$\begin{aligned} \text{getSort}(\nu \to \tau) &= \text{getSort}(\tau) \\ \text{getSort}(C) &= \text{lab}(C) \end{aligned}$$

$$\boxed{\text{transType}(\tau, \alpha, \gamma)}$$

$$\begin{aligned} \text{transType}(\nu \to \tau, \alpha, \gamma) &= \text{transTypeBase}(\nu, \alpha, \gamma) \to \text{transType}(\tau, \alpha, \gamma) \\ \text{transType}(C, \alpha, \gamma) &= \text{newconType}(C)\ \alpha\ \gamma \end{aligned}$$

$$\boxed{\text{transTypeBase}(\nu, \alpha, \gamma)}$$

$$\begin{aligned} \text{transTypeBase}(P, \alpha, \gamma) &= P \\ \text{transTypeBase}(F\overline{\nu}, \alpha, \gamma) &= \alpha\ (F\ \overline{\text{transTypeBase}(\nu)}) \\ \text{transTypeBase}(C, \alpha, \gamma) &= \alpha\ \text{lab}(C) \end{aligned}$$

Figure A-5

Appendix B

Proofs for Mandate

B.1 Correctness of SOS-AM Translation

This section proves the correspondence between the operational semantics and abstract machine. We begin by proving the correspondence between operational semantics and PAM. We then prove an important consequence of invertible up rules, and then prove the correspondence between PAM and the abstract machine.

The core idea of the correspondence is simple: The PAM emulates the SOS because each PAM rule was explicitly constructed to correspond to an RHS fragment of the SOS. The PAM and AM are equivalent because the AM merely removes some redundant steps from the PAM, and because the fused rules in the AM each correspond to several rules in the PAM. However, a PAM derivation may have some "false starts" corresponding to a partially-applied SOS rule, and so there are some additional technical details to explain which states are included in the correspondences.

B.1.1 SOS-PAM Correspondence

The goal of this section is to prove the main theorem of SOS-PAM correspondence:

Theorem 4.4.1. *$c_1 \rightsquigarrow_l^* c_2$ if and only if, for all contexts K, $\langle c_1 \mid K \rangle{\downarrow} \hookrightarrow_l^* \langle c_2 \mid K \rangle{\uparrow}$*

The forward direction is easy, because the PAM rules were designed to follow in lockstep with each component of the SOS rules. The reverse-direction appears harder, but is rendered easy by two important facts:

Observation B.1.1. *From the LHS of each PAM rule, it is possible to identify the arguments s, K, rhs of* sosRhsToPam *that generated it.*

This is because each case generates rules in a distinct form, and each generated rule contains all of the parameters of sosRhsToPam. (Note that LHSs originating on line (1) of Figure 4-12 must be non-values, while those from line (4) must be a value.)

Property B.1.2 (Sanity of Phase)**.** *The following three properties hold:*

1. *If $\langle c \,|\, K \rangle {\updownarrow_c} \hookrightarrow \langle c' \,|\, K' \rangle {\updownarrow_{c'}}$ and K' contains strictly more stack frames than K, then ${\updownarrow_c} = {\updownarrow_{c'}} = {\downarrow}$, and $K' = K \circ f$ for some f.*

2. *If $\langle c \,|\, K \rangle {\updownarrow_c} \hookrightarrow \langle c' \,|\, K' \rangle {\updownarrow_{c'}}$ and K' contains strictly fewer stack frames than K, then ${\updownarrow_{c'}} = {\uparrow}$, and $K = K' \circ f$ for some f.*

3. *If the PAM rules for language l have no up-down rules, and $\langle c \,|\, K \rangle {\uparrow} \hookrightarrow_l \langle c' \,|\, K' \rangle {\updownarrow_{c'}}$ without using the reset rule, then ${\updownarrow_{c'}} = {\uparrow}$, and $K = K' \circ f$ for some f.*

These properties follow by inspection of the possible rules. We now prove Theorem 4.4.1 as a corollary of a stronger result:

Lemma B.1.3. *$c_1 \rightsquigarrow_l c_2$ if and only if, for all K, there is a derivation $\langle c_1 \,|\, K \rangle {\downarrow} \hookrightarrow_l^* \langle c_2 \,|\, K \rangle {\uparrow}$ which does not use the reset rule. This derivation must use the same sequence of rules regardless of K.*

Proof. We address each direction.

($\Rightarrow$): Let K be an arbitrary context, and consider a derivation of $c_1 \rightsquigarrow c_2$. By induction on the last SOS rule applied, we prove there exists a derivation $\langle c_1 \,|\, K \rangle {\downarrow} \hookrightarrow^* \langle c_2 \,|\, K \rangle {\uparrow}$.

Let the last SOS rule used take the form $c_1 \rightsquigarrow C_1[\dots C_n[c_2]]$, where each C_i is a single RHS fragment. We define R to be the remaining fragments, $R = C_{i+1}[\dots C_n[c_2]]$. We induct again on i to show that $\langle c_1 \,|\, K \rangle {\downarrow} \hookrightarrow^* s_i$, where:

1. If $i = 0$ (base case), then $s_i = \langle c_1 \mid K \rangle\!\downarrow$.

2. If C_i is of the form **let** $[c \rightsquigarrow c']$ **in** $\square$, then

$$s_i = \langle c' \mid K \circ [c' \to R] \rangle\!\uparrow$$

3. If C_i is of the form **let** $c' = f(\bar{c})$ **in** $\square$, then

$$s_i = \langle c' \mid K \circ [c' \to R] \rangle\!\downarrow$$

and further, the PAM system contains rules generated by $\textsc{sosRhsToPam}(s_i, k, R)$ for some context variable k. We handle each case:

1. Satisfied by the empty transition sequence and definition of $\textsc{sosRuleToPam}$.

2. Then there is a rule that $s \hookrightarrow \langle c \mid K[c' \to R] \rangle\!\downarrow$. By inversion of the SOS derivation, we must have that $c \rightsquigarrow c'$. Then, by the outer induction hypothesis, $\langle c \mid K[c' \to R] \rangle\!\downarrow \hookrightarrow^* \langle c' \mid K[c' \to R] \rangle\!\uparrow$. The rest follows by line (3) of the definition of $\textsc{sosRhsToPam}$.

3. Then there is a rule that $s \hookrightarrow \langle c' \mid K[c' \to R] \rangle\!\downarrow$. The rest follows by line (4) of the definition of $\textsc{sosRhsToPam}$.

The inner induction hypothesis for C_n tells us that a rule $s \hookrightarrow \langle c_2 \mid K \rangle\!\uparrow$ must exist, finishing the proof of the outer induction.

($\Leftarrow$): We proceed by a strong induction on all derivations of the form $\langle c_1 \mid K \rangle\!\downarrow \hookrightarrow_l^* \langle c_2 \mid K \rangle\!\uparrow$. Consider the first PAM rule of the derivation. Because it may not depend on K, it must have an LHS generated on line (1) of Figure 4-12. By Observation B.1.1, we can hence reconstruct the entire originating SOS rule, $c_1 \rightsquigarrow C_1[\dots C_n[c_2]]$. We show that $c_1 \rightsquigarrow c_2$ by this rule.

The proof proceeds similarly to the forward direction, so we omit more details. We induct over i, and show that there must be a prefix of the derivation $\langle c_1 \mid K \rangle\!\downarrow \hookrightarrow^* s_i$, where $s_0 = \langle c_1 \,|\, K \rangle\!\downarrow$ and s_i is a state corresponding to the computation of C_i. Each s_i matches the LHS of the PAM rule used in the derivation; Observation B.1.1 tells us this rule must be the one generated for C_{i+1}. The only interesting case is for recursive steps; there, $s_i = \langle c \,|\, K' \rangle\!\downarrow$, and the Sanity of Phase properties dictate there must be a later state in the derivation $\langle c' \,|\, K' \rangle\!\uparrow$; applying the outer inductive hypothesis finishes this case.

□

B.1.2 Invertibility

Our goal is to show that, if all up-rules are invertible (Definition 4.3.4), then, for any K and non-value c, $\langle c \,|\, K \rangle\!\uparrow \hookrightarrow^* \langle c \,|\, K \rangle\!\downarrow$, thus justifying the optimizations of Section 4.3.6, and characterizing which PAM states are and are not removed when translating a derivation to AM. However, there are a few technical restrictions on this.

Definition B.1.4. *A configuration/context pair (c, K) is **non-stuck** if $\langle c \,|\, K \rangle\!\uparrow \hookrightarrow^* \langle c' \,|\, \boldsymbol{emp} \rangle\!\uparrow$ for some c'.*

Because each PAM rule corresponds to part of an SOS rule, our definition of non-stuckness is different from the usual one: it is intended to exclude terms which correspond to a partial match on an SOS rule. A single step $c_1 \rightsquigarrow c_2$ in the SOS corresponds to a sequence $\langle c_1 \,|\, \mathbf{emp} \rangle\!\downarrow \hookrightarrow^* \langle c_2 \,|\, \mathbf{emp} \rangle\!\uparrow$ in the PAM, so a state is non-stuck if it can complete the current step. Stuck states result from SOS rules which only partially match a term. For example, the SOS rule

$$(a.b := v, \mu) \rightsquigarrow \mathbf{let}\ (r, \mu') = \text{Lookup}((a, \mu))\ \mathbf{in}$$
$$\mathbf{let\ false} = \text{ContainsField}(r, b)\ \mathbf{in}\ (\mathbf{error}, \mu)$$

decomposes into 3 PAM rules. Assuming Lookup succeeds, the first PAM rule brings $\langle (a.b := v, \mu) \,|\, K \rangle\!\downarrow$ into the state

$$\left\langle (r, \mu') \,\middle|\, K \circ [\textbf{let false} = \text{ContainsField}(\Box_t, b) \textbf{ in } (\textbf{error}, \mu)] \right\rangle\downarrow$$

If **false** $\neq$ ContainsField(r, b), then this will be a stuck state.

Excluding stuck states is enough to prove the general Invertibility Lemma:

Lemma B.1.5 (Invertibility)**.** *If all up-rules for l are invertible, and there are no up-down rules for l other than the reset rule, then, for any non-stuck non-value (c, K), $\langle c \,|\, K\rangle\uparrow \hookrightarrow^* \langle c \,|\, K\rangle\downarrow$.*

Proof. Consider a derivation $\langle c \,|\, K\rangle\uparrow \hookrightarrow^* \langle c' \,|\, \textbf{emp}\rangle\uparrow$. Because there are no up-down rules, each step must follow from an up-rule. Hence, each step is invertible. Applying each inverted step gives a new derivation $\langle c' \,|\, \textbf{emp}\rangle\downarrow \hookrightarrow^* \langle c \,|\, K\rangle\downarrow$.

This requires that the term in the RHS of each step is a non-value, which also follows because each rule is invertible, and hence the RHS can be reduced. $\Box$

This motivates the definition of an **inversion sequence**. We add the condition about the reset rule to prevent the definition from including arbitrarily large subsequences of a nonterminating execution.

Definition B.1.6. *An* ***inversion sequence*** *which begins at $\langle c \,|\, K\rangle\uparrow$ is a sequence of transitions $\langle c \,|\, K\rangle\uparrow \hookrightarrow^* \langle c \,|\, K\rangle\downarrow$ which contains at most one application of the reset rule.*

This idea of an inversion sequence partitions a derivation $\langle c_1 \,|\, K_1\rangle\downarrow \hookrightarrow^* \langle c_2 \,|\, K_2\rangle\downarrow$ into two parts: the inversion sequences, which do redundant work, and the remainder, which we call the **working steps**. Sometimes a derivation must be extended to contain a complete inversion sequence, which is then eliminated upon the conversion to an abstract machine.

Definition B.1.7. *A reduction $\langle c_1 \,|\, K_1\rangle\updownarrow_1 \hookrightarrow \langle c_2 \,|\, K_2\rangle\updownarrow_2$ within a derivation is a* ***working step*** *if the derivation cannot be extended so that $\langle c_1 \,|\, K_1\rangle\updownarrow_1$ is part of an inversion sequence.*

Observation B.1.8. *If (c, K) is a non-stuck non-value and all up-rules are invertible, then, by the Determinism Assumption, all sequences $\langle c \mid K\rangle\!\uparrow \hookrightarrow^* \langle c' \mid K'\rangle\!\updownarrow$ may be extended to contain an inversion sequence starting at $\langle c \mid K\rangle\!\uparrow$.*

Corollary B.1.9. *If a reduction $\langle c \mid K\rangle\!\uparrow \hookrightarrow \langle c' \mid K'\rangle\!\uparrow$ cannot be extended to contain an inversion sequence starting at $\langle c \mid K\rangle\!\uparrow$, then either (c, K) is stuck or c is a value.*

With these extra properties, we are now ready to exactly state the PAM-AM correspondence.

B.1.3 PAM-AM Correspondence

Intuitively, a derivation in the AM $\langle c_1 \mid K_1\rangle \rightarrow^* \langle c_2 \mid K_2\rangle$ is the same as a derivation in the PAM, but with the inversion sequences cut out and with some consecutive steps merged into one. We prove this in steps. First, we show that if $\langle c_1 \mid K_1\rangle\!\downarrow \hookrightarrow^* \langle c_2 \mid K_2\rangle\!\downarrow$, and the last transition is a working step, then $\langle c_1 \mid K_1\rangle \longrightarrow^* \langle c_2 \mid K_2\rangle$. Next we prove that every derivation in the Unfused AM corresponds to a derivation in the fused AM, but with some consecutive steps merged. We also prove the reverse theorem, which is easier to state, as every state in an AM derivation has a corresponding state in the PAM.

The forward direction comes first. This version of the theorem starts with transitions between down-states, to simplify consideration of which states may be eliminated on conversion to AM.

Theorem 4.4.5 (PAM-Unfused AM: Forward)**.** *Suppose $\langle c \mid K\rangle\!\downarrow \hookrightarrow_l^* \langle c' \mid K'\rangle\!\downarrow$, $\langle c' \mid K'\rangle\!\downarrow$ is non-stuck, and the derivation's last step is working. Then there is a derivation $\langle c \mid K\rangle \longrightarrow_l^* \langle c' \mid K'\rangle$.*

Proof. First, recall that, if $\longrightarrow_l$ is defined, then all up-rules for l are invertible.

Consider a derivation $\langle c \mid K\rangle\!\downarrow \hookrightarrow_l^* \langle c' \mid K'\rangle\!\downarrow$. Remove all maximal inversion sequences. Then remove all phases from the PAM states, resulting in AM states. This means that an inversion sequence $\langle c_1 \mid K_1\rangle\!\uparrow \hookrightarrow^* \langle c_1 \mid K_1\rangle\!\downarrow$ is replaced with a single state $\langle c_1 \mid K_1\rangle$.

If we can show that all PAM rules for all remaining steps of the derivation have a corresponding rule in the unfused abstract machines, then we will be done. Note that the only PAM rules without a corresponding rule in the AM are those of the form $\langle c \,|\, K \rangle\!\downarrow \hookrightarrow \langle c \,|\, K \rangle\!\uparrow$, and those of the form $\langle c_1 \,|\, K_1 \rangle\!\uparrow \hookrightarrow \langle c_2 \,|\, K_2 \rangle\!\uparrow$ where c_1 is a non-value. The former correspond to stutter steps in the AM and may be ignored. For the latter, because of the Determinism Assumption and Corollary B.1.9, all such transitions must be part an inversion sequence, and were hence removed. □

In the next two proofs, we use the notation $\langle c \,|\, K \rangle \xrightarrow{F} \langle c' \,|\, K' \rangle$ to denote that $\langle c \,|\, K \rangle$ steps to $\langle c' \,|\, K' \rangle$ by rule F.

Theorem 4.4.6 (PAM-Unfused AM: Backward). *If $\langle c \,|\, K \rangle \longrightarrow_l^* \langle c' \,|\, K' \rangle$, then there are phases $\updownarrow_c$ and $\updownarrow_{c'}$ such that $\langle c \,|\, K \rangle\!\updownarrow_c \hookrightarrow_l^* \langle c' \,|\, K' \rangle\!\updownarrow_{c'}$.*

Proof. Consider a derivation $\langle c \,|\, K \rangle \longrightarrow_l^* \langle c' \,|\, K' \rangle$. For each rule of the unfused abstract machine used in this derivation, consider the corresponding PAM rule that generated it.

Let $\updownarrow_c$ be the phase of the LHS of the first such rule. We will show that there is $\updownarrow_{c'}$ such that $\langle c \,|\, K \rangle\!\updownarrow_c \hookrightarrow_l^* \langle c' \,|\, K' \rangle\!\updownarrow_{c'}$, obtained by replacing each AM rule with its corresponding PAM rule and by inserting inversion sequences. We proceed by induction on the derivation.

Consider the last step of the derivation $\langle c'' \,|\, K'' \rangle \longrightarrow_l \langle c' \,|\, K' \rangle$. Consider the AM rule of this last step, and let G be the PAM rule from which it originated, $\langle c_G^1 \,|\, K_G^1 \rangle\!\updownarrow_G^1 \overset{G}{\hookrightarrow}_l C_G[\langle c_G^2 \,|\, K_G^2 \rangle\!\updownarrow_G^2]$. If G matches, it would finish the proof. In the base case where there is only one step, taking $\updownarrow_c = \updownarrow_G^1$ and $\updownarrow_{c'} = \updownarrow_G^2$ suffices to make it match.

If there is more than one step in the derivation, then, by the induction hypothesis, there are phases $\updownarrow_c$ and $\updownarrow_{c''}$ such that $\langle c \,|\, K \rangle\!\updownarrow_c \hookrightarrow_l^* \langle c'' \,|\, K'' \rangle\!\updownarrow_{c''}$. Consider the second-to-last step of the AM derivation $\langle c''' \,|\, K''' \rangle \longrightarrow_l \langle c'' \,|\, K'' \rangle$ and its generating AM rule, and let the corresponding PAM rule be F. If the RHS of F matches the LHS of G, we would be done. We hence must consider all PAM rules F, G such that the RHS of F and LHS of G match except for the phase, meaning they would erroneously match upon conversion to AM rules. Call such an (F, G) a *confused pair.* We perform case analysis on Figure 4-12 to find all possible confused pairs, and for each find a derivation $\langle c'' \mid K'' \rangle \updownarrow_{c''} \hookrightarrow_l^* \langle c' \mid K' \rangle \updownarrow_{c'}$.

Figure 4-12 gives 3 possible forms of PAM RHSs, generated on lines (2), (3), and (4), and 3 possible LHSs, generated on lines (1), (3), and (4). Note that some of these only match values/non-values, and that, because of the prohibition on up-down rules, all rules using the LHS from line (3) will be restricted to only match values. This leaves only 3 possible forms for a confused pair: using the RHS/LHS generated on lines (2)/(1), (2)/(4), and (4)/(3). We analyze each in turn. We find that the first case is desirable, as it results from removing inversion sequences, while the other two are benign, as another rule must exist that does match.

- (2)/(1): In this case, the RHS of a rule F, $s \overset{F}{\hookrightarrow}_l \langle c_F \mid K_F \rangle \uparrow$, matches the LHS of a rule G, $\langle c_G \mid K_G \rangle \downarrow \overset{G}{\hookrightarrow}_l t$, where $c_F = c_G$ upon matching with c''. $c_F = c_G$ must be a non-value because c_G originates from a SOS rule $c_G \rightsquigarrow r$, and by the Sanity of Values assumption. The invertibility lemma finishes this case.

- (2)/(4): In this case, the RHS of a rule F, $s \overset{F}{\hookrightarrow}_l \langle c_F \mid K_F \rangle \uparrow$, matches the LHS of a rule G, $\langle c_G \mid K_G \rangle \downarrow \overset{G}{\hookrightarrow}_l t$. Further, after unifying with the current state $\langle c'' \mid K'' \rangle$, $\langle c_F \mid k_F \rangle = \langle c_G \mid k_G \rangle$, and $k_F = k_G$ can be written $k_F = k_G = k \circ [c_F \to \text{rhs}]$. By the induction hypothesis, there is a derivation $\langle c \mid K \rangle \updownarrow_c \hookrightarrow_l^* \langle c_F \mid K_F \rangle \uparrow$. Extend the last transition of this derivation to a maximal sequence $\langle c_F'' \mid K_F'' \rangle \downarrow \overset{H}{\hookrightarrow}_l \langle c_F' \mid K_F \rangle \downarrow \hookrightarrow_l * \langle c_F \mid K_F \rangle \uparrow$ which does not use the reset rule; by the Sanity of Phase properties, this must exist, and Rule H must have been created by line (3). By Observation B.1.1, we know that rule G was created by an invocation matching $\textsc{sosRhsToPam}(\langle c_F \mid K_F \rangle \downarrow, k, \text{rhs})$, while rule H was created by an invocation matching

$$\textsc{sosRhsToPam}(\langle c_F'' \mid k \rangle \downarrow, k, \textbf{let } [x \rightsquigarrow c_F] \textbf{ in } rhs)$$

which means there is also a rule J created by an invocation

$$\textsc{sosRhsToPam}(\langle c_F \mid K_F\rangle\!\uparrow, k, \mathrm{rhs})$$

whose RHS is t. Using rule J completes this case.

- (4)/(3): In this case, the RHS of a rule F, $s \overset{F}{\hookrightarrow}_l \langle c_F \mid k_F\rangle\!\downarrow$, matches the LHS of a rule G, $\langle c_G \mid k_G\rangle\!\uparrow \overset{G}{\hookrightarrow}_l C_G[t]$. Further, after unifying with the current state $\langle c'' \mid k''\rangle$, $\langle c_F \mid k_F\rangle = \langle c_G \mid k_G\rangle$, and $k_F = k_G$ can be written $k_F = k_G = k \circ [x \to \mathrm{rhs}]$. By Observation B.1.1, we know that rule F was created by an invocation

  $$\textsc{sosRhsToPam}(s, k, \textbf{let } [c_F \rightsquigarrow x] \textbf{ in } \mathrm{rhs})$$

  and there is hence a rule H created by an invocation of the form

  $$\textsc{sosRhsToPam}(\langle c_F \mid k_F\rangle\!\downarrow, k, \mathrm{rhs})$$

  As rule G was created by an invocation $\textsc{sosRhsToPam}(\langle c_G \mid k_G\rangle\!\uparrow, k, \mathrm{rhs})$, rule H hence takes the form $\langle c_F \mid k_F\rangle\!\downarrow \overset{H}{\hookrightarrow}_l C_G[t]$. Using rule H completes this case.

□

Finally, to show the correspondence between the Unfused AM and the normal AM, we must show that fusing rules does not substantially alter the transition relation. This is very simple, thanks to the Fusion Property.

Lemma B.1.10. *Let M be an abstract machine whose transition relation is $\to_M$, containing a rule F. Let M' be M with Rule F fused with all possible successors, and let its transition relation be $\to_{M'}$. Then, for any state $\langle c \mid K\rangle$, $\langle c \mid K\rangle \to_{M'} \langle c' \mid K'\rangle$ if and only if $\langle c \mid K\rangle \to_M \langle c' \mid K'\rangle$, or $\langle c \mid K\rangle \overset{F}{\to}_M \langle c'' \mid K''\rangle \overset{G}{\to}_M \langle c' \mid K'\rangle$ for some rule $F \neq G$.*

Proof. By Property 4.3.5. □

Theorem 4.4.7 (Unfused AM-AM)**.** *$\langle c \,|\, K \rangle \rightarrow_l^* \langle c' \,|\, K' \rangle$ if and only if $\langle c \,|\, K \rangle \longrightarrow_l^* \langle c' \,|\, K' \rangle$ by a sequence of rules whose last rule is not fused away.*

Proof. Corollary of Lemma B.1.10. □

Finally, we state some useful properties which are analogues of Sanity of Phase.

Property B.1.11 (Sanity of Frame)**.** *The following properties hold:*

1. *If $\langle c \,|\, K \rangle \rightarrow \langle c' \,|\, K' \rangle$, then either $K = K' \circ f$ for some f, $K' = K \circ f$ for some f, or there are f, f', K'' such that $K = K'' \circ f$ and $K' = K'' \circ f'$.*
2. *If c is a nonvalue, and $\langle c \,|\, K \rangle \rightarrow \langle c' \,|\, K' \rangle$, then K is contained in K'.*

Why Didn't We Use Bisimulations? A common alternative way of stating results like Theorems 4.4.5 and 4.4.6 is by giving a stuttering bisimulation between the PAM and the AM. However, between any two (terminating) transition systems, without giving additional labels on the states, there always exists a trivial stuttering bisimulation. The interesting information lies in giving a specific stuttering bisimulation. We decided that dealing with the additional machinery of stuttering bisimulations would add to the background needed to understand the proof, without much benefit.

B.2 Proofs of Abstract Rewriting Theorems

Lemma 4.5.4 (Generalized Lifting Lemma)**.** *Let β_1, β_2 be base abstractions where β_1 is pointwise less than β_2, i.e.: $\beta_1(f)(\overline{c}) \prec \beta_2(f)(\overline{c})$ for all f, c. Suppose $\langle c_1 \,|\, K_1 \rangle \prec \left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle$, and also $\langle c_1 \,|\, K_1 \rangle \underset{\beta_1}{\rightrightarrows} \langle c_2 \,|\, K_2 \rangle$ by rule F. Then there is a $\left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ such that $\langle c_2 \,|\, K_2 \rangle \prec \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ and $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle \underset{\beta_2}{\rightrightarrows} \left\langle \widehat{c_2} \,\middle|\, \widehat{K_2} \right\rangle$ by rule F.*

Proof. We show a correspondence between the applications abstract rewriting algorithms for both β_1 and β_2. We know that the LHS of rule F matches $\langle c_1 \,|\, K_1 \rangle$ with witness σ; hence, by the Abstract Matching Property, it also matches $\left\langle \widehat{c_1} \,\middle|\, \widehat{K_1} \right\rangle$ with some witness $\sigma' \succ \sigma$. Let the RHS of f be rhs_p. Then:

- If $\mathrm{rhs}_p = \mathbf{let}\ c_{\mathrm{ret}} = \mathrm{func}(\overline{c_{\mathrm{args}}})\ \mathbf{in}\ \mathrm{rhs}'_p$, then the concrete rewriting of $\langle c_1 \mid K_1 \rangle$ must have picked some $r \in \beta_1(\mathrm{func})(\sigma(\overline{c_{\mathrm{args}}}))$, where r matches c_{ret} with witness σ_r. Since β_1 is a base abstraction and $\widehat{\sigma}(\overline{c_{\mathrm{args}}}) \succ \sigma(\overline{c_{\mathrm{args}}})$, there is an $\widehat{r} \in \beta_2(\mathrm{func})(\widehat{\sigma}(\overline{c_{\mathrm{args}}}))$ with $\widehat{r} \succ r$. Then, by the abstract matching property, $\widehat{r}$ matches c_{ret} with witness $\widehat{\sigma_r} \succ \sigma_r$. The abstract rewriting algorithm then proceeds to recursively evaluate rhs_p with some $\widehat{\sigma'}$ and σ' respectively; by their definition and the previous argument, we must have $\widehat{\sigma'} \succ \sigma'$.

- If $\mathrm{rhs}_p = \langle c'_p \mid K'_p \rangle$, then the result is $\langle \widehat{\sigma}(c'_p) \mid \widehat{\sigma}(K'_p) \rangle$. Since $\widehat{\sigma} \succ \sigma$, we have $\langle \widehat{\sigma}(c'_p) \mid \widehat{\sigma}(K'_p) \rangle \succ \langle \sigma(c'_p) \mid \sigma(K'_p) \rangle = \langle c_2 \mid K_2 \rangle$ by the Abstract Matching Property, finishing the proof.

□

Theorem 4.5.7 (Abstract Transition)**.** *For $a, b \in$ amState, if α is an abstraction with base abstraction β, and $a \to b$, then either $b \sqsubseteq \alpha(a)$, or $\exists g \in$ amState$\star$. $\alpha(a) \underset{\alpha}{\widehat{\to}} g \wedge b \sqsubseteq g$.*

The proof uses the following observation:

Observation B.2.1. *The nontermination-cutting relation ($\lhd$) satisfies the following properties:*

1. *If $a \prec b$ and $b \lhd c$, then $a \lhd c$.*

2. *If $a \to b$ and $a \lhd c$, then $b \lhd c$.*

We now prove the Abstract Transition Theorem:

Proof. Because ($\prec$) is reflexive and transitive, there must be a *canonical derivation* of $a \sqsubseteq \alpha(a)$ of one of the following three forms:

- Case (1): $a \prec \alpha(a)$. Then, by the lifting lemma, there is a g with $b \prec g$ and $\alpha(a) \underset{\beta}{\widehat{\to}} g$.

- Case (2): There are x_1, x_2 such that $a \prec x \lhd x_2 \sqsubseteq \alpha(a)$. Then, by Observation B.2.1, $a \lhd x_2$, and hence $b \lhd x_2$, and hence $b \sqsubseteq x_2 \sqsubseteq \alpha(a)$.

- Case (3): There is an x_1 such that $a \prec x_1$, and, for all x_2 such that $x_1 \underset{\beta}{\widehat{\rightarrow}} x_2$, $x_2 \sqsubseteq \alpha(a)$. First, by the lifting lemma, there is an x_1' with $b \prec x_1'$ and $x_1 \underset{\beta}{\widehat{\rightarrow}} x_1'$. But, by assumption, $x_1' \sqsubseteq \alpha(a)$. Hence, $b \prec x_1' \sqsubseteq \alpha(a)$, so $b \sqsubseteq \alpha(a)$.

□

B.3 Correctness of Graph Patterns

We first note that, by starting abstract execution on a node whose immediate children are all non-value variables, this algorithm assumes that the initial state of the program must contain no value nodes. This is not a real restriction; one can transform any language to meet this criterion by replacing each value-node $V(\overline{x})$ in initial program states with a non-value node $MkV(\overline{x})$ and adding the rule $MkV(\overline{x}) \rightsquigarrow V(\overline{x})$. So, pedantically speaking, the graph-patterns produced by the algorithm are not actually graph patterns of the original language, but rather of this normalized form.

We now build the setting of our proofs. In this develpment, let $\mathsf{term}_{\mathrm{Var}}$, $\mathsf{amState}_{\mathrm{Var}}$, etc be variants of $\mathsf{term}\star$, $\mathsf{amState}\star$, etc that may contain free variables as well as $\star$ nodes. We extend the $\prec$ ordering to include the subsumption ordering, and with the relation $x_{\mathrm{mt}} \prec \star_{\mathrm{mt}}$ for any variable x, so that $a \prec b$ if a may be obtained from b by specializing a match type, expanding a star node, *or* substituting a variable.

As mentioned in Section 4.5.3, we add a few technical conditions to the definition of an abstraction α. The first condition prevents an antagonistically-chosen α from doing something substantially different when encountering a more abstract term.

Assumption B.3.1. *On terms without variables, α must be monotone in the $\prec$ ordering. As the analogue for terms with variables, if there is a substitution σ such that $b = \sigma(a)$, then there must be a substitution σ' extending σ such that $\alpha(b) = \sigma'(\alpha(a))$.*

The next condition is stronger than we need, but greatly simplifies the discovery of the correspondence between graph patterns and interpreted-mode graphs. In plain words, it states that abstractions may not drop stack frames: they may skip over the execution of a subterm entirely, but may not skip over only the latter part of a computation. All abstractions discussed in this chapter satisfy it.

Definition B.3.2. *Define the stack length of a state $s = \langle c \,|\, K \rangle$ as:*

- *$stacklen(\langle c \,|\, \boldsymbol{emp} \rangle) = 0$*
- *$stacklen(\langle c \,|\, K \circ f \rangle) = 1 + stacklen(\langle c \,|\, K \rangle)$*

Assumption B.3.3. *For all $a \in \mathsf{amState}_{Var}$, we require that*

$$stacklen(a) \leq stacklen(\alpha(a))$$

Further, there must be a derivation of $a \sqsubseteq \alpha(a)$ where none of the intervening states c, as in Definition 4.5.6, satisfy $stacklen(c) < stacklen(a)$.

We now begin the proofs. We need a new version of the Generalized Lifting Lemma for narrowing. Mimicking the proof, and using the relation between matching and unification, gives the following:

Lemma B.3.4 (Lifting Lemma (Narrowing))**.** *Let $a \in \mathsf{amState}_{\star}, b \in \mathsf{amState}_{Var}$, $a \prec b$, and let β be a base abstraction. Suppose $a \underset{\beta}{\widehat{\rightarrow}} a'$. Then there exists b' such that $b \underset{\beta}{\widehat{\rightsquigarrow}} b'$ and $a' \prec b'$.*

We now prove the correspondence between a graph pattern and the relevant subgraph of an abstract transition graph. To isolate the relevant subgraphs, we use the concept of *hammocks* from graph theory, which are commonly used in the analysis of control-flow graphs (e.g.: [55]). A hammock of a control-flow graph is a single-entry single-exit subgraph. We use the modified term *weak hammock* to refer to a single-entry multiple-exit subgraph.

Definition B.3.5. *Let $N^+(n)$ be the out-neighborhood of n in a graph G. Then the weak hammock of G bounded by entry node n and exit node-set $\mathcal{T}$ is the subgraph of G induced by the node set given by the least-fixed-point of Q, where*

$$Q(S) = \{n\} \cup \bigcup_{m \in (S \setminus \mathcal{T})} N^+(m)$$

Our goal now is to, given the abstract transition graph of a program, discover the fragment that corresponds to the control-flow of a single node. We will then prove the correspondence between these fragments and the relevant graph pattern.

Definition B.3.6. *Let N be a non-value node type and α an abstraction, and consider some configuration $S = \left\langle (N(\overline{e_i}), \mu) \,\middle|\, K \right\rangle$. Let T be the abstract transition graph T of $(\underset{\alpha}{\widehat{\rightarrow}})$ starting from S. Let $E = \{t \in T | t \neq S \wedge \mathit{stacklen}(t) \leq \mathit{stacklen}(S)\}$. Then the CFG fragment for S is defined inductively as follows:*

- *If none of the e_i are non-values, then the CFG fragment for S is the weak hammock of T bounded by S and E.*
- *Otherwise, the CFG fragment for S is the weak hammock of T bounded by S and E, minus the edges of the CFG fragments for each non-value e_i, minus also the nodes which then become unreachable from both S and E.*

We say there is a transitive edge *from the start state of each sub-CFG-fragment to its end states. By default when discussing the edges of a CFG fragment, we do not include the transitive edges.*

Lemma B.3.7. *Let $\widehat{e} \prec \star_{\mathsf{NonVal}}$. Then, for any $\widehat{s}, \widehat{K}$, $\left\langle (\widehat{e}, \widehat{s}) \,\middle|\, \widehat{K} \right\rangle \lhd \left\langle (\star_{\mathsf{Val}}, \top_l) \,\middle|\, \widehat{K} \right\rangle$.*

Proof. Consider $\langle c \,|\, K \rangle \in \gamma(\left\langle (\widehat{e}, \widehat{s}) \,\middle|\, \widehat{K} \right\rangle)$. By the Sanity of Frame properties, for any derivation $\langle c \,|\, K \rangle \rightarrow^* \langle c' \,|\, K' \rangle$, either K' contains K or there is a subderivation of the form $\langle c \,|\, K \rangle \rightarrow^* \langle c' \,|\, K \rangle$. □

Theorem B.3.8 (Correctness of Graph Patterns)**.** *Let N be a non-value node type, and P be its graph pattern under $(\underset{\alpha}{\widehat{\rightarrow}})$. For an abstraction α, consider the abstract transition graph T of $(\underset{\alpha}{\widehat{\rightarrow}})$ from some start state $S = \left\langle (N(\overline{e_i}), \mu) \,\middle|\, K \right\rangle$. Let F be the CFG fragment for S in T. Let σ be the substitution resulting from unifying the start state of P with S. Then, for every edge $a \underset{\alpha}{\widehat{\rightarrow}} b$ in F, there are $a', b' \in P$ with $a \prec \sigma(a')$, $b \prec \sigma(b')$ such that $a' \underset{\alpha}{\widehat{\rightarrow}} b'$ is in P.*

Proof. For any $a \in V(F), a' \in V(P)$ with $a \prec \sigma(a')$, this is true for all edges reachable from a by the Lifting Lemma and by Assumption B.3.1. We hence must show that every node in F is reachable from some node x satisfying $\exists x' \in P, x \prec \sigma(x')$. But note that every node in F is reachable from either S or by the exit nodes of the CFG fragment for one of the e_i. By construction, S is equal to the start state of P'. It remains to show this condition for the exit nodes of the CFG fragments for the e_i, i.e.: the nodes which are the target of a transitive edge.

Pick such a node $x = \langle c_x \,|\, K_x \rangle$, and note that, by construction and by Sanity of Frame, c_x must be a value. Then, using the Sanity of Frame properties and Assumption B.3.3, the source of the corresponding transitive edge(s) must have the form $\langle (e_i, \mu) \,|\, K_x \rangle$. By induction, we must have e'_i, μ', K'_x with $e_i \prec \sigma(e'_i)$ and $K_x \prec \sigma(K'_x)$ with $\langle (e'_i, \mu') \,|\, K'_x \rangle \in P$. This node has a transitive edge to $\langle (\star_{\mathsf{Val}}, \top) \,|\, K'_x \rangle$ in P, which satisfies $\langle c_x \,|\, K_x \rangle \prec \sigma(\langle (\star_{\mathsf{Val}}, \top) \,|\, K'_x \rangle)$, finishing the proof. $\square$

One advantage of the compiled-mode is that it's done once per language/abstraction pair, so any corner case that causes an infinite-loop would be exposed up front. Yet interpreted-mode CFG-generation is always more precise, albeit slower and less predictable. Can we at least guarantee that interpreted-mode generation will terminate, i.e.: result in a finite graph?

It's easy to construct examples where interpreted-mode CFG-generation does not terminate (e.g.: the identity abstraction, and any non-terminating program), so there's no universal termination theorem. But here's a cool result: if compiled-mode generation terminates, then so does interpreted-mode. Intuitively, this is so because graph-pattern generation does the kind of case analysis that's needed to prove termination of interpreted-mode CFG-generation for a single language/abstraction pair

Theorem B.3.9 (Finiteness)**.** *Consider a node type N, its graph pattern P, any state of the form $S = \langle (N(\overline{e_i}), \mu) \,|\, K \rangle$, and the CFG fragment F for S in the abstract transition graph of S. If P is finite, then so is F.*

Proof. Assume P is finite, and suppose F is infinite. By König's Lemma, either F contains a node of infinite outdegree, or an infinite path. However, because all semantic functions func have the property that $\text{func}(\overline{c})$ is finite for all $\overline{c}$, the only possibility is for F to contain an infinite path.

Theorem B.3.8 establishes a relation M between F and P. Since P is finite, this implies that there is a path $f_1 \to f_2 \to \cdots \to f_n$ in F where each f_i is distinct, where the corresponding path in P $p_1 \to \cdots \to p_{n-1} \to p_1$ is a cycle, where each $\to$ is either a transitive edge, $(\underset{\alpha}{\widehat{\to}})$ (in F), or $(\underset{\alpha}{\widehat{\leadsto}})$ (in P).

However, by Theorem B.3.8, there is σ such that $\sigma(p_1) = f_1$, and there is σ' extending σ such that $\sigma'(p_1) = f_n$. Since $\sigma(p_1)$ must be ground, that means $\sigma(p_1) = \sigma'(p_1)$, and hence $f_1 = f_n$, contradicting the assumption. Hence, F must be finite. $\square$

As a corollary, we obtain our automated proof of termination for the interpreted-mode CFG generators.

Theorem 4.6.1. *Let $a \in \mathsf{amState}_l$ and α be a machine abstraction. If the graph patterns under abstraction α for all nodes in a are finite, then only finitely many states are reachable from a under the $\underset{\alpha}{\widehat{\to}}$ relation.*