

A Framework for Modular Metatheory and Verified Compilation

by

Dustin Jamner

S.B. Northeastern University, 2020.

S.M. Massachusetts Institute of Technology, 2022.

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

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2027

© 2027 Dustin Jamner. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Dustin Jamner
Department of Electrical Engineering and Computer Science
October 16, 2026

Certified by: Adam Chlipala
Arthur J. Conner (1888) Professor of Computer Science, Thesis Supervisor

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

THESIS COMMITTEE

THESIS SUPERVISOR

Adam Chlipala

*Arthur J. Conner (1888) Professor of Computer Science
Department of Electrical Engineering and Computer Science*

THESIS READERS

Rachit Nigam

*Assistant Professor of Electrical Engineering and Computer Science
Department of Electrical Engineering and Computer Science*

Zachary Tatlock

*Professor of Computer Science
Paul G. Allen School of Computer Science & Engineering, University of Washington*

A Framework for Modular Metatheory and Verified Compilation

by

Dustin Jamner

Submitted to the Department of Electrical Engineering and Computer Science
on October 16, 2026 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Common techniques for semantic reasoning are often tied to the specific structures of the languages and compilers that they support. This limitation restricts the extension and composition of semantics in these systems. Additionally, mechanized proofs are usually either highly manual or use similarly bespoke automation. As a result of these two limitations, existing compiler-verification efforts do not admit critical modes of extension or reusability.

This thesis presents a framework, *Pyrosome*¹, in the Rocq theorem prover in which verified compilers are fully extensible, meaning that to extend a language simply requires defining and verifying the compilation of the new feature, reusing the old correctness theorem for all other cases. The novel enabling idea is an inductive formulation of equivalence preservation that supports the addition of new rules to the source language, target language, and compiler.

Unfortunately, by being highly generic, such a system incurs nontrivial boilerplate. On the other hand, the same regimented structure that necessitates this overhead makes my framework particularly amenable to automation. As the second component of this thesis, I present the first formally verified, language-generic e-graph library, following the structure of the egglog fixpoint reasoning system. This verified e-graph library is of independent interest to the *Pyrosome* framework. However, its connection to *Pyrosome* is to the mutual benefit of both components. In one direction, it serves to automate the boilerplate of *Pyrosome* and even some proofs that require notable insight. In the other direction, *Pyrosome*'s metatheory serves as a high-level specification for and interface to the e-graph system.

As a case study, I present a multipass compiler from System F with simple references, through CPS translation and closure conversion. Specifically, I demonstrate how I can build such a compiler incrementally by starting with a compiler for simply typed lambda-calculus and adding natural numbers, the unit type, recursive functions, and a global heap, then extending judgments with a type environment and adding type abstraction, all while reusing the original theorems. This case study validates both key components of the thesis by showing how the modularity of *Pyrosome* allows verification to proceed incrementally and how the e-graph machinery helps to automate the resulting subgoals.

Thesis supervisor: Adam Chlipala

Title: Arthur J. Conner (1888) Professor of Computer Science

¹Pyrosomes are tiny colonial organisms that connect to each other to form tube-shaped colonies up to 60 feet in length. In my framework, compilers are similarly made up of many small, independent components.

Acknowledgments

There are too many people to thank. I attribute this fortune to both the individual efforts of my mentors, colleagues, and advisors and the dedication to nurturing the next generation that I see as a trademark of the programming-languages community. As a seventeen-year-old high school student selecting a university, I met with Jens Palsberg, Sorin Lerner, Ranjit Jhala, Mitch Wand, Olin Shivers, Matthias Felleisen, and Amal Ahmed based only on cold emails and their willingness to offer their advice. For me, these faculty represent the larger PL community, which has been a universally welcoming professional home over the years.

I have Amal to thank for managing the right balance of freedom and structure for me to grow from a precocious high school graduate into a proper researcher. She dedicated an hour or more a week to me for most of the five years I spent in her group, mentoring me on technical writing, research, and career plans throughout my time at Northeastern.

I was quite fortunate to conduct this research in Adam Chlipala's group. His reliable support and patience over the course of my degree were absolutely necessary to the completion of this thesis. I started in Fall 2020, the beginning of the pandemic, and also the start of four years of a long-distance relationship while my now-fiancée Rachel studied medicine in Chicago. To make matters even more difficult, I chose a subtle topic that required great effort, and many reviewers, in order to articulate my ideas to the community properly. Through all of these difficulties, both when I was productive and when I struggled with the combined research, world, and personal challenges, Adam remained consistently supportive and encouraging, and gave me the time and space to carry this work through to its completion.

Right when I started my PhD, I met Arthur Azevedo de Amorim through the SIGPLAN-M program. Our regular meetings through the last six years helped combat the professionally and socially isolating effects of COVID. As both an undergraduate and PhD student, I have learned from collaboration with many other researchers: Max New, Andres Erbsen, Jeremy Siek, Clément Pit-Claudel, Chris Casinghino, and many others. The members of both Amal's and Adam's groups, as well as the broader Northeastern and MIT PL groups, all helped shape my understanding, outlook, and research interests in critical ways.

Apart from my academic support, I must thank my family and friends; parents, siblings, cousins, aunts, and uncles, all of them. Every stage of my life has benefited from their support, and although at this point I can manage on my own, I still very much feel and appreciate it. The one person I could not manage without is my fiancée, Rachel, who has been with me for almost the entire journey. From my first paper submission in Amal's group until now, she has stuck with me through many "Anywhere on Earth" deadlines, despite my consistent aspirations of early submissions.

Contents

<i>List of Figures</i>	11
1 Introduction	13
1.1 Language Extension and Metatheory	14
1.2 Verified E-Graphs for Automated Proof	15
2 Background	17
2.0.1 Generalized Algebraic Theories	17
2.0.2 E-Graphs, Egg, and Egglog	18
2.1 Related Work	20
2.1.1 Alternative Generic Frameworks	20
2.1.2 Multilanguage Semantics	21
2.1.3 Existing Verified Compilers	21
2.1.4 DimSum	22
3 The Pyrosome Framework	23
3.1 Simply Typed Lambda Calculus	23
3.1.1 Substitution-Equation Generation	24
3.2 Language Extensions	25
3.2.1 Equational Theories, Contextual Equivalence, and Extensibility	26
3.3 Cross-Language Translation	27
3.3.1 CPS	28
3.3.2 Semantics Preservation	29
3.3.3 The Preserving Predicate	30
3.3.4 Closure Conversion and Vertical Composition	34
3.3.5 Recursive-Functions Compiler Extension	35
3.3.6 How Incorrect Proofs Fail	37
3.3.7 Axiomatic Equivalence and Operational Semantics	38
3.4 Optimization	38
3.5 Global State and Evaluation Contexts	39
3.6 Polymorphism and Advanced Metaprogramming	41
3.7 Linear Lambda Calculus	42
3.8 IMP	43
3.9 Future Work	44

4	A Functional Egglog Implementation	47
4.1	An Illustrated Example	47
4.2	Functional Architecture	48
4.3	Logic-Program Intermediate Representation	53
4.4	Worst-Case Optimal Join and Set Intersection	53
4.5	Shared Queries	56
4.6	Rule Optimization	57
4.6.1	Sort Pattern Optimization	58
4.7	Analysis and Extraction	59
4.8	Optimizing Pyrosome Equivalence Goals	61
4.9	Compiler Correctness by Computation	62
4.10	Equality Saturation for Type Inference	62
4.11	Equality Saturation and Pyrosome’s Case Studies	64
4.12	Implementation Limitations and Future Work	65
5	Verifying an Egglog Implementation	67
5.1	AI and Modern Formal Verification	67
5.2	Verifying Equality Saturation	68
5.2.1	Verifying Rule Translation	71
5.2.2	E-Graph Invariants	71
6	Conclusions	73
	<i>References</i>	75

List of Figures

2.1	GAT for natural numbers	17
2.2	GAT for vectors	18
2.3	Term translation from Zhang et al. [1] Figure 8	20
3.1	CBV substitution calculus (selected rules)	24
3.2	STLC	25
3.3	Recursive-function extension	27
3.4	Continuation calculus (selected rules) and CPS translation for STLC	28
3.5	Rocq definition of Preserving (selected cases)	32
3.6	Closure conversion (selected rules)	35
3.7	Compiler extension	37
3.8	STLC state extension	39
3.9	Evaluation-context extension with STLC	40
3.10	Polymorphic calculus extensions (selected rules)	42
3.11	Linear Lambda Calculus	43
3.12	IMP (excerpt)	44
4.1	Top-level functions for EqSat	50
4.2	Rebuild implementation	52
4.3	Logic-program IR	52
4.4	Trie intersection example	54
4.5	EqSat executable format	57
4.6	Language requiring sort clauses	59
4.7	Extraction Pseudocode	60
4.8	Manual proof goals	64

Chapter 1

Introduction

Compiler verification is a laborious process compared to unverified implementation. Nevertheless, numerous researchers have taken on the challenge [2–4] due to the significant benefits of formal verification [5]. Compilers are one of the most critical classes of programs to which we can apply formal verification due to their presence in almost every software pipeline. To verify a full software system requires verifying any compilers involved [6], and even in unverified software stacks, the volume of code that depends on a given compiler often warrants the work of providing strong guarantees.

When talking about compiler verification, the most important concern is what we are verifying. This thesis focuses on functional correctness, which can be described with phrases like “the source and target programs do the same thing” or “the compiler preserves the behavior of programs”, dating back to at least McCarthy and Painter [7], who proved equivalent outcomes between source programs in an arithmetic expression language and their corresponding outputs in an assembly language.

Despite the broad commonality, there are almost as many specific formulations of functional correctness as there are verified compilers. As programming languages practice evolved to define languages more operationally than denotationally, notions of equivalent outcomes had to mediate between source and target values, often through a compiler-specific cross-language value relation. Compiler specifications still often lean on transition system representations, which provide a simple and cohesive model of program behavior that is easy to compare across languages, but only applies to whole programs.

Examining the current state of compiler verification, existing efforts suffer from one or more limitations, with the end result that a few common requirements of software systems cannot be satisfied by current techniques. To begin with, existing projects’ internal mechanics are typically closely tied to their chosen source languages and often to their implementations. As a result, augmenting their results to support more permissive linking and extensibility quickly grows difficult [8,9]. While recent work has made advances in high-to-low-level and multilanguage linking [10,11], it typically does so by lowering all reasoning to a common target semantics rather than by connecting source-level reasoning directly.

In general, much of the compilation literature revolves around establishing single, all-encompassing models that form key dependencies of all steps of the verification process. This pattern drives the aforementioned limitations since correct compilers tend to be bespoke artifacts that revolve around their central models. For example, Perconti and Ahmed [12]

proved a correctness result that supported linking with arbitrary target programs, but their multilanguage approach builds all languages in the compiler into the statement of the correctness theorem, making it difficult to extend with new features. This limitation is also the root cause of another key factor in the slow adoption of verified compilation. Most major programming languages are living projects with routinely evolving specifications. State-of-the-art verification efforts view source-language specifications as single, monolithic entities. As a result, the kind of incremental improvements one might expect to see in an actively evolving language pose a serious threat to attempts at formal verification. While many such changes might be handled by effective automation and proof engineering, the cross-cutting nature of specification changes makes it difficult to estimate the cost of updating the verification since there is no hard upper bound on the necessary revisions.

My thesis. In this work, I set out to demonstrate a methodology for reasoning about programs and compilers in a way that is *preserved by extension*. My contributions cover the general design principles that enable extensible languages and compilers, a concrete framework that puts these principles to use, and substantial verified automation to automatically reason in these contexts.¹

The idea that underpins my contributions is that *extensibility is a property of positive definitions*. Specifications defined by what they allow rather than what they forbid admit extension far more smoothly. Furthermore, in the second half of this thesis, I show that such specifications are amenable to automated reasoning, which I credit to the constructive nature of positive definitions. Concretely, I implement and verify the soundness of a fixpoint reasoning system combining equality reasoning with relational rules, in the style of Zhang et al. [13]. This latter contribution successfully automates much of the reasoning necessary for verifying compilers in my extensible setting. Additionally, it demonstrates how to verify such automation and use those theorems to justify its applications. In combination, I aim to present a comprehensive characterization of the practice of extensible verification from language and compiler definitions through metatheory and automation.

1.1 Language Extension and Metatheory

The cornerstone of this thesis is a foundational study demonstrating language-agnostic metatheory and techniques that combine to enable fully extensible compiler verification, with a case study at the core-calculus level.² While I hope one day to expand my software artifact’s scope to cover the full feature set required to implement a practically usable compiler, my present aim is to lay out the definitions, metatheorems, and proof structures that support translation extensibility in the abstract and demonstrate their behavior on a small research calculus. I focus on cross-language translation passes in particular because, while intralanguage optimizations often make up the bulk of practical compilers, from a specification standpoint, a generic theory of cross-language properties is the greater challenge.

¹The repository containing the corresponding code can be found at <https://github.com/DIJamner/pyrosome>, or <https://zenodo.org/records/23340153> for a stable snapshot at the time of writing.

²This work originally appeared in Jamner et al. [14].

The particular combination of design decisions embodied in Pyrosome includes among other contributions: choosing an answer to the question of what constitutes a programming language; demarcating a well-behaved formulation of translation passes; and choosing how to formulate compiler correctness, including eschewing contextual equivalence. These three decisions come together to enable strong metatheoretical principles that support my desired notion of extensibility. I represent programming languages as Generalized Algebraic Theories (GATs) [15,16], which define program semantics via lists of judgments that generate well-formedness predicates and minimal equational theories.³ The translations I primarily consider are type- and equivalence-preserving maps (GAT morphisms) that are defined pointwise over the constructs of the source language and so are amenable to extension.

Upon this foundation, I develop metatheory, methods, and mechanized tooling for building and combining language extensions and compilers in the setting of GATs. My framework includes an inductive characterization of well-formedness for cross-language translations encoded in Pyrosome, which I prove implies both type and equivalence preservation. I also characterize extensions of these specifications and compilers and prove theorems enabling separate reasoning about and combination of such extensions. To demonstrate the key features of Pyrosome, I build a multipass compiler for System F with recursive functions, existential types, and a global heap that transforms it into CPS and then performs closure conversion, by starting from a compiler for the simply typed lambda calculus (STLC) and successively adding features via my extension theorems. I also implement a generic partial-evaluation pass to support the viability of optimizations in this context and present additional compilers from an imperative calculus to the simply typed target and from the linear lambda calculus to a corresponding linear CPS calculus.

1.2 Verified E-Graphs for Automated Proof

The second half of this thesis concerns automating compiler correctness proofs in the first place. E-graphs, data structures first described in 1980 [18] that serve as a core component of many SMT solvers, have gained popularity in recent years as a tool for performing numerous types of program reasoning and transformations [13,19–22]. Such systems typically take the equality-saturation (EqSat) approach, starting with one or more seed terms and eagerly discovering related terms via rewrite rules. The significant body of work based on their use demonstrates their utility in automating complex reasoning problems.

Given this utility, it is unsurprising that there have been attempts to integrate e-graphs into interactive-theorem-proving pipelines, where current automation is often made up of ad-hoc tactic scripts or solvers specialized to specific domains like linear arithmetic or propositional logic. However, as with most attempts to bring automated techniques to interactive provers, the most difficult challenge lies in convincing the trusted kernel that the results of EqSat hold. As existing e-graph implementations are written externally to the theorem prover, it can be quite difficult to make them generate the detailed proofs checked by the likes of Rocq or Lean. Prior work attempted to bridge the gap by instead generating tactic scripts based on the e-graph’s deductions, relying on the slack and additional inference present in the tactics’ implementations to make up the difference as often as possible [23,24]. Unfortunately, such

³As opposed to contextual equivalence, which is a maximal equational theory [17]

approaches can be brittle [25]. Recently, the Lean community has produced the grind tactic [22] based on e-graphs, which has proven much more successful at generating proofs directly. Grind is implemented as a metaprogram in Lean itself, as is common in the Lean community. Lean does not implement two-level type theory [26,27] or similar meta-level reasoning, so this metaprogram cannot be verified and must dedicate significant complexity to generating proof terms.

Instead of generating proofs, I implement and verify the soundness of my e-graphs in Rocq. While grind has been successful, I find it valuable to explore what tradeoffs exist in verified automation. Both my implementation and soundness theorem are fully modular and extensible in the sense that they are parameterized by the languages they operate over and the rewrites they apply. As a result, goals in any user-specified domain can be solved by simply invoking the soundness theorem for EqSat and then running the resulting computation in Rocq. My implementation is based on the algorithm laid out by Zhang et al. [13], who implement EqSat on top of an extended Datalog-like language. While I do not expect my implementation to compare to their performance due to the limitations of Rocq and pure functional code in general, I took care to implement performant code and the key algorithmic optimizations to the extent possible. As a result, my library is sufficiently performant for significant use on typical Rocq goals. In addition, my correctness theorem serves as the first mechanized proof of the soundness of an (asymptotically) performant equality-saturation algorithm, as well as of any (asymptotically) performant bottom-up Datalog engine.

I demonstrate my implementation’s capabilities through its application on my suite of compiler case studies, where they automate the vast majority of goals. This process even resulted in the first compiler fully automatically verified using language-agnostic procedures: the simply typed core of my primary case study, a two-pass compiler for STLC with recursive functions, let-bindings, and mutable references that transforms it into continuation-passing style and then performs closure conversion.

This portion of my thesis consists of the first formally verified e-graph engine, which can then be used as the heart of automation tactics in Rocq. It includes a reformulation of published e-graph algorithms for total, purely functional languages, as underlie theorem provers like Rocq. I combine these results with Pyrosome to produce the first expressive compiler-verification approach that achieves full automation on nontrivial translation passes, not even requiring guiding the prover toward a theorem with hints like sequences of lemmas in ACL2, evaluated on a case study inspired by “From System F to Typed Assembly Language” [28].

Chapter 2

Background

2.0.1 Generalized Algebraic Theories

Generalized Algebraic Theories (GATs) [15,16] lie at the core of Pyrosome’s metatheory. GATs are dependently typed algebraic theories, originally developed to capture dependent type theories. Like simple algebraic theories, a GAT consists of a list of judgment rules introducing function symbols and equations. However, unlike simple theories, in a GAT, each rule can depend not only on the prior function symbols but also on the prior equations to justify its well-formedness, as we will see later. While recent work has used intrinsically typed structures [16], in Pyrosome, I define all data (terms, sorts, rules, and languages) as simply typed ASTs and encode this dependent structure in separate well-formedness predicates over these constructs as Cartmell [15] did, in order to keep my computation simply typed at the Coq level.

I will use a brief example to explain the different forms of rules that make up a GAT. Figure 2.1 defines a GAT for natural numbers with addition. The first rule declares that `nat` is a sort of our nascent example theory. This theory only uses one simple sort, so it could be expressed as an untyped algebraic theory, but I will expand it shortly. The next three rules declare `0`, `S`, and `+` as 0-, 1-, and 2-argument terms of sort `nat` respectively and specify the sorts of each’s arguments above the line. The last two rules define the behavior of addition via equations on the symbols of the theory. Equality between terms in a GAT is given by the reflexive, symmetric, transitive, congruence closure of the equations included in the GAT definition.

Next I will demonstrate a GAT that makes use of dependent typing by adding more rules

$$\begin{array}{c} \frac{}{\text{nat sort}} \quad \frac{}{0 : \text{nat}} \quad \frac{n : \text{nat}}{Sn : \text{nat}} \quad \frac{m : \text{nat} \quad n : \text{nat}}{m + n : \text{nat}} \quad \frac{n : \text{nat}}{0 + n = n : \text{nat}} \\[10pt] \frac{m : \text{nat} \quad n : \text{nat}}{Sm + n = S(m + n) : \text{nat}} \end{array}$$

Figure 2.1: GAT for natural numbers

$$\begin{array}{c}
\frac{n : \text{nat}}{\text{vec } n \text{ sort}} \quad \frac{}{\text{nil} : \text{vec } 0} \quad \frac{m : \text{nat} \quad n : \text{nat} \quad v : \text{vec } n}{m :: v : \text{vec } Sn} \\
\\
\frac{m : \text{nat} \quad v : \text{vec } m \quad n : \text{nat} \quad v' : \text{vec } n}{\text{app } v \ v' : \text{vec } (m + n)} \\
\\
\frac{n : \text{nat} \quad v : \text{vec } n}{\text{app nil } v = v : \text{vec } n} \quad \frac{i : \text{nat} \quad m : \text{nat} \quad v : \text{vec } m \quad n : \text{nat} \quad v' : \text{vec } n}{\text{app } (i :: v) \ v' = i :: (\text{app } v \ v') : \text{vec } (Sm + n)}
\end{array}$$

Figure 2.2: GAT for vectors

to this theory to develop a datatype of known-length vectors (of nats for simplicity). As shown in Figure 2.2, I start with adding a second sort to our theory, the dependent sort of n -length vectors. Among the subsequently defined vector term formers, `nil`, $n :: v$, and `app`, each specifies its length as a component of its sort. Finally, I bestow `app` with computational behavior via equations that specify its action on `nil` and $n :: v$ on the left. This last rule, the reduction for `app` $(i :: v) \ v'$, demonstrates what I mentioned earlier about rules depending on prior equations: If we calculate the natural type for the left-hand side of the equation, we get $\text{vec } (Sm + n)$ as per the annotation. However, if we do the same for the right-hand side, we get $\text{vec } S(m + n)$ instead. The reason we can still use this term at my desired type is by invoking the defining equation of addition from Figure 2.1, which the GAT allows implicitly. Without the equations for addition earlier in the language, this rule would be ill-formed. While my simply typed case studies do not use this expressive strength, it is critical to supporting my polymorphic extension.

2.0.2 E-Graphs, Egg, and Egglog

Much of the architecture of my system comes from Zhang et al. [13], so I present e-graphs from an `egglog` perspective. For readers more familiar with Datalog, my system can also be seen as an implementation of an extended Datalog featuring functional dependencies, an efficient equality relation with implicit congruence rules, multiconclusion rules, and existential variables. To connect to the first half of my thesis, I define my top-level rules and specifications in Pyrosome, compiling the high-level rules to this extended Datalog.

E-Graphs as Databases

Although for most of their existence, e-graphs were not presented this way [18,21], recent work has shown that implementing an e-graph as a database has significant advantages for both the queries necessary to implement rewrites and the overall flexibility of the system [1,13]. Fundamentally, an e-graph represents a set of terms and equivalences between them. True to the “graph” portion of its name, it represents terms as a set of vertices where each vertex has an identifier and represents one AST node via a function symbol and a list of child identifiers.

Following `egglog`, I package this data together into atoms $f(\vec{x}) \rightarrow y$ with function symbol f , argument identifiers $\vec{x}$, and node identifier y to represent the output of the atom. The key observation of Zhang et al. [1,13] is that the atoms can be organized tabularly, where each function symbol of arity n corresponds to a table of $(n + 1)$ -tuples $(\vec{x}, y)$ where y has a functional dependency on $\vec{x}$, so if $f(\vec{x}) \rightarrow y$ and $f(\vec{x}) \rightarrow y'$ are both atoms in an e-graph, then $y = y'$. This structure allows us to draw on the databases literature to query the e-graph efficiently when applying rewrites [1].

E-graphs represent equivalences between terms with a union-find data structure over their identifiers. This convention ensures that each equivalence class has a single canonical identifier, so two identifiers can be compared by canonicalizing them, that is to say computing their unique representatives and then comparing those for simple equality. The union-find critically interacts with functional dependency. When an atom $f(\vec{x}) \rightarrow y'$ is added to an e-graph containing $f(\vec{x}) \rightarrow y$ where $y \neq y'$, the two values must be reconciled. The e-graph does so by merging them in the union-find. Intuitively, this merge represents the process of discovering new facts. Given the preexisting fact $f(\vec{x}) \rightarrow y$, when the e-graph discovers $f(\vec{x}) \rightarrow y'$, it learns that really y and y' must have represented the same domain object the whole time.

Congruence Closure and Rebuilding

One of the distinguishing features of e-graphs from their inception was their propagation of equivalence via congruence. To perform this operation efficiently, an e-graph typically contains a mapping from each identifier to a set of its parent atoms. When two identifiers are merged, their parents are recanonicalized. Should any newly canonicalized parent overlap with an existing atom, their output identifiers must be merged. Traditionally, these invariants were reconstructed after every rule application, but Willsey et al. [21] observed that this potentially expensive rebuilding process could safely be postponed until the end of a batch of rules, amortizing the cost. They do so by maintaining a worklist of new unions as rules are applied and processing the elements of that worklist afterward [1,13,21].

Semi-Naïve Evaluation

With rebuilding amortized, querying the e-graph becomes the most critical part of EqSat with respect to performance. A significant portion of query performance is proportional to the number of satisfying results, so optimizations that reduce the size of the set of matches return great dividends. In a naïve saturation loop, every known fact gets rediscovered on each subsequent iteration, polluting the output set with redundant tuples. Semi-naïve evaluation avoids this problem by keeping tables of all new facts, that is tuples that have been updated since the last query, and only applies a rule if at least one of its clauses matches a new fact. This optimization produces the same end behavior as naïve evaluation but with much less wasted computation.

Rule Translation

Datalog-style engines like `egglog` and this work operate on logic programs that are unaware of the higher-level term representations. Although my primary interface expects Pyrosome’s

$$\begin{aligned}
\text{Compile}(p) &= Q(\text{root}, v_1, \dots, v_k) \leftarrow \text{atoms} \\
&\quad \text{where } v_1 \dots v_k \text{ are variables in } p \\
&\quad \text{and } \text{Aux}(p) = \text{root} \sim \text{atoms} \\
\text{Aux}(f(p_1, \dots, p_k)) &= v \sim R_f(v, v_1, \dots, v_k), A_1, \dots, A_k \\
&\quad \text{where } v \text{ is fresh and } \text{Aux}(p_i) = v_i \sim A_i \\
\text{Aux}(x) &= x \sim \emptyset \text{ where } x \text{ is a pattern variable}
\end{aligned}$$

Figure 2.3: Term translation from Zhang et al. [1] Figure 8

term-based inference rules, internally my system processes rules in a Datalog-like IR. To mediate between the term and atom representations, Zhang et al. [1] defined a simple transformation from patterns to conjunctive queries, which I reproduce in Figure 2.3. This transformation works recursively on a pattern by generating a fresh variable at each syntactic form to represent the output of that atom and returning these fresh variables upward to form the arguments of their parent atoms. While my rule-compilation pipeline is more complex, it operates on the same principles.

2.1 Related Work

2.1.1 Alternative Generic Frameworks

The principal benefit of GATs [15] over most generic frameworks for programming-language metatheory [29–31] is their presentation via equational theories rather than operational semantics. Felleisen [29] gives meaning to programs by an arbitrary termination predicate and a contextual-equivalence relation defined in terms of that predicate. As discussed in subsection 3.2.1, contextual equivalence is too strong for extensible reasoning. I, by design, choose not to include every reasonable equation in a given language, since equations that may be reasonable to include, say, in a pure language interact poorly with effectful extensions. By using the smallest equational theories wherever possible, that is only including the equations at each level that application and compiler reasoning demand, users of Pyrosome increase the compatibility and extensibility of their developments.

Prior work on modular metatheory and language extensibility mechanized a system in Rocq that covered components like language interpreters, including modular soundness properties [30,31], in the style of Swierstra [32]. Allais et al. [33] also built a framework for reasoning about binding, renaming, and translation that bears some similarities to this work. There exist many other formal frameworks in the same general spirit, including a long line of work on logical frameworks [34], that are quite useful in broad domains but assume a particular structure for variable binding. However, these projects incorporate object-language binding and substitution into their frameworks, rather than implementing them as a first-class feature. While built-in binding is attractive for ease of use since it lifts substitution into the metatheory, it restricts the generality of the framework in ways that I wish to avoid in Pyrosome.

The K Framework [35] contains an extensive ecosystem of generic language-specification tooling, and their logic is powerful enough to express a variety of binders internally [36]. The

project has also recently expanded to cover certified proofs of the behavior of individual programs [37]. However, they do not address the higher-order concerns of verified compilation and compiler extensibility that I cover in this work.

The recent GATlab project [38] is based on a very similar system of GATs and GAT morphisms, implemented as an embedded domain-specific language in Julia for the purpose of algebraic modelling in scientific and engineering applications. GATlab offers a number of methods of interaction designed to facilitate exploration in applied category theory, which would likely be useful additions to Pyrosome for the purposes of GAT development. However, as it is not a theorem prover, it does not attempt to prove facts, internally or metatheoretically, as I do.

2.1.2 Multilanguage Semantics

Existing literature documents some of the uses of multilanguage semantics, including in compiler verification [12,39,40]. However, rather than use a formal framework to describe the way these works combine languages, the authors design ad-hoc multilanguages for their specific use cases. As a result, they cannot exploit the generic properties of language extension that I take advantage of. Furthermore, since they use contextual equivalence in their multilanguages in their specifications of compiler correctness [12], they cannot extend their compilers with new supported features or additional passes. Related work takes steps towards generalizing the key elements of this line of research so as to better support modular reasoning and extensibility [41] by expressing compiler cases as operational rules, but it is still limited by its use of contextual equivalence. While contextual equivalence results do provide certain guarantees with regard to security that I do not consider, I believe that a more compositional paradigm is necessary to reduce the burden of multilanguage reasoning.

Recent work on multilanguage semantics enables complex cross-language interoperation by directly defining the semantics of the source languages in terms of a target-language logical relation [11]. This technique enables a rich variety of FFI designs and multilanguage interoperation. However, since the source languages do not have their own semantics, it gives up source-level reasoning, which reduces the benefit of writing higher-level code, especially in verified settings. This approach uses a relation that generalizes the compiler as the specification of source-language behavior, so it does not adapt well to my primary concern of compiler correctness. In contrast, I define a source-level equational theory, which allows me to separate the specification of each feature’s behavior from the compiler’s implementation, verify that the former describes the latter, and thereafter reason only at the source level.

2.1.3 Existing Verified Compilers

Since Pyrosome currently is a foundational study rather than a production tool, I cannot compare it directly to the tooling built up for more mature projects like CakeML [4] and CompCert [2,10]. However, I can discuss what improvements or extensions to my theoretical results might be necessary before compilers with similar feature sets can be implemented in Pyrosome. One major hurdle is developing my story for optimization as discussed in [section 3.4](#). While prior work shows that real optimizations can be verified against equational

theories, it remains to explore the design space properly regarding how best to make them extensible.

The other significant gap between what I theoretically support and existing verified-compiler ecosystems is my lack of refinement relations since GATs by definition define language semantics in terms of equality. There are a number of ways to encode refinement or nondeterministic computation, ranging from defining the relation internally in the object language to viewing nondeterministic operations as computing sets of results. However, the most straightforward approach to incorporating refinement would be to generalize the framework itself. None of the core metatheory in Pyrosome depends on symmetry, so it should be possible to remove it as an axiom and explore the expanded space of theories. I would also need to consider how to handle translating programs from Turing-complete languages like System F to finite state machines like assembly languages, possibly by using refinement and an approach like Beck et al. [42]. CompCert also features graph-shaped IRs, which I have yet to investigate.

2.1.4 DimSum

The recent DimSum framework [43] gives a convincing accounting of language-agnostic compiler verification and cross-language linking for low-level, event-based systems. However, I see Pyrosome and DimSum as complementary. DimSum’s design addresses cross-language library interaction at preexisting language interfaces, primarily procedure calls in their examples. However, it does not provide machinery for extending the languages themselves, especially with new features that do not fit the preexisting event type. Such reasoning might be possible via some clever design pattern that expresses each language feature as an event, but this approach is clearly not an idiomatic or intended way to use their system.

Unlike DimSum, Pyrosome’s syntactic approach based on equational theories allows me to extend a language’s syntax and semantics at a fine-grained level. In addition to deriving results about the interactions of programs across languages, I enable users of Pyrosome to reason about programs, languages, and compilers in a way that is preserved by language extension as well. My syntactic core also supports both programmatic manipulation of language definitions and my extensive automation of proof goals.

In reference to prior work on multilanguage semantics [12,39,40], Sammler et al. [43] conclude that “syntactic multi-languages scale well to typed, higher-order languages. In [the DimSum] paper, we have put the focus on different kinds of languages: untyped, low-level languages comparable to C and assembly.” I largely agree with this analysis. As I discuss above in 2.1.2 and as the DimSum authors concur, prior work on constructing multilanguages suffered from difficulties with the ad-hoc nature of multilanguage construction and the limitations of contextual equivalence. Since I have addressed both of these concerns in Pyrosome, I see the methodology of Pyrosome as the preferred approach in typed and/or higher-order settings. In particular, while Pyrosome allows linking and compiler-correctness results to depend on the guarantees of the involved languages’ type systems, it is unclear how to express such properties in DimSum without a fundamental extension of their framework.

Chapter 3

The Pyrosome Framework

As described in [subsection 2.0.1](#), Pyrosome builds on Generalized Algebraic Theories (GATs) [\[15,16\]](#) to define a metatheory of programming-language extension. I chose GATs for this purpose both because they are extremely general and because it allows my case studies to match the structure of typical pen-and-paper metatheory closely, except for the necessary wrinkle of a more precise accounting of variables and binding. In this chapter, I will use my primary case study, a two-pass compiler that performs a continuation-passing-style transformation followed by closure conversion on System F with references, to explain the features of Pyrosome piece by piece.

3.1 Simply Typed Lambda Calculus

I begin my case studies with the simply typed lambda calculus. Some frameworks bake in a notion of object-language variables and substitutions [\[33,44,45\]](#) so that user languages automatically inherit the behavior of binders, which makes defining the simply typed lambda calculus (STLC) a quick task. While this convention is suitable for an important class of theories, many languages, for example linear languages or those with dynamic scope, do not fit within these models. The expressiveness of GATs allows me to define not just the expressions but also the types, contexts, and judgment forms of a programming language as terms and sorts in a GAT, so the object languages are free to define their own notions of substitution. On the other hand, since such constructs exist internally to the theories, GATs do not come equipped with a primitive notion of binding. Thus, as the cost of this freedom, to define STLC I must first define a call-by-value substitution calculus. However, once I have done so, I can reuse it as the basis for many languages.

I base the substitution calculus off of Dybjer [\[46\]](#) but modify it to represent call-by-value substitutions, with a subset of the rules shown in [Figure 3.1](#). The first five rules declare sorts for object-language contexts, types, values, expressions, and substitutions. I elide metavariable sorts of the forms $\Gamma : \text{ctx}$ and $A : \text{ty}$ in [Figure 3.1](#) when they can be inferred from the sorts of the other metavariables, although formally every metavariable in a rule has a specified sort. The syntactic form $\text{ret } v$ injects values into the sort of expressions. Its name is inspired by fine-grained call-by-value calculi [\[47\]](#), although my example remains standard call-by-value. I define an operation to append a type to a context and write de Bruijn index

$$\begin{array}{c}
\frac{}{\text{ctx } \textit{sort}} \quad \frac{}{\text{ty } \textit{sort}} \quad \frac{\Gamma : \text{ctx} \quad A : \text{ty}}{\text{val}(\Gamma, A) \textit{ sort}} \quad \frac{\Gamma : \text{ctx} \quad A : \text{ty}}{\text{exp}(\Gamma, A) \textit{ sort}} \quad \frac{\Gamma : \text{ctx} \quad \Delta : \text{ctx}}{\text{sub}(\Gamma, \Delta) \textit{ sort}} \\
\\
\frac{v : \text{val}(\Gamma, A)}{\text{ret } v : \text{exp}(\Gamma, A)} \quad \frac{\Gamma : \text{ctx} \quad A : \text{ty}}{\Gamma, A : \text{ctx}} \quad \frac{\Gamma : \text{ctx} \quad A : \text{ty}}{\underline{0} : \text{val}((\Gamma, A), A)} \\
\\
\frac{e : \text{exp}(\Gamma, A) \quad \gamma : \text{sub}(\Delta, \Gamma)}{e[\gamma] : \text{exp}(\Delta, A)} \quad \frac{v : \text{val}(\Gamma, A) \quad \gamma : \text{sub}(\Delta, \Gamma)}{v[\gamma]_v : \text{val}(\Delta, A)} \\
\\
\frac{\gamma : \text{sub}(\Delta, \Gamma) \quad v : \text{val}(\Delta, A)}{\langle \gamma, v \rangle : \text{sub}(\Delta, (\Gamma, A))} \quad \frac{v : \text{val}(\Gamma, A) \quad \gamma : \text{sub}(\Delta, \Gamma)}{(\text{ret } v)[\gamma] = \text{ret } v[\gamma]_v : \text{exp}(\Delta, A)} \\
\\
\frac{\gamma : \text{sub}(\Delta, \Gamma) \quad v : \text{val}(\Delta, A)}{\underline{0}[\langle \gamma, v \rangle]_v = v : \text{val}(\Delta, A)}
\end{array}$$

Figure 3.1: CBV substitution calculus (selected rules)

$\underline{0}$ to represent the variable at the head of a context. I also add syntactic forms for explicit substitutions [48] on expressions and values and a substitution $\langle \gamma, v \rangle$ that appends a value to the head of a substitution. I include two representative equations in Figure 3.1. The first defines the behavior of expression substitution on $\text{ret } v$: it pushes under the syntactic form to apply a value substitution to the subterm. The second equation defines part of the behavior of substitutions on de Bruijn indices, specifically the property that index $\underline{0}$ projects the head value out of any substitution applied to it. Other rules of the calculus handle features like weakening and substitution composition.

I can define STLC as an extension of this calculus by adding a few more rules, just like the vector GAT built on the naturals. Since I will be working on extensions of the substitution calculus for most of my examples, I will adopt a few fairly standard notations: I write $A : \text{ty}$ as $\vdash A$, $e : \text{exp}(\Gamma, A)$ as $\Gamma \vdash e : A$, $v : \text{val}(\Gamma, A)$ as $\Gamma \vdash_v v : A$, and $\gamma : \text{sub}(\Delta, \Gamma)$ as $\gamma : \Delta \Rightarrow \Gamma$. As before, I elide type and context sorts when they can be inferred. Additionally, I will write subsequent rules as if they were based on a named, rather than de Bruijn-indexed, substitution calculus for readability. With that preamble, the base rules of STLC are shown in Figure 3.2. I add three new terms to the syntax: the function type $A \rightarrow B$, application expressions $e \ e'$, and function values $\lambda(x : A). e$. The subsequent equations define β -reduction and substitution on applications and functions. I can (and usually should) also include an η -expansion equation. However, the benefit of extensibility is that I treat that variation as just another extension of this base calculus. It is up to the user whether they intend to support η laws, and they can be included or omitted as needed.

3.1.1 Substitution-Equation Generation

While substitution equations like those for applications and functions are numerous, they follow a prescribed form. One benefit of using a deeply embedded representation of GATs is

$$\begin{array}{c}
\frac{\vdash A \quad \vdash B}{\vdash A \rightarrow B} \quad \frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e e' : B} \quad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash_v \lambda(x : A). e : A \rightarrow B} \\
\\
\frac{\Gamma, x : A \vdash e : B \quad \Gamma \vdash_v v : A}{\Gamma \vdash (\text{ret } \lambda(x : A). e) (\text{ret } v) = e[v/x] : B} \quad \frac{\Gamma \vdash e : A \rightarrow B \quad \Gamma \vdash e' : A \quad \gamma : \Delta \Rightarrow \Gamma}{\Delta \vdash (e e')[\gamma] = e[\gamma] e'[\gamma] : B} \\
\\
\frac{\Gamma, x : A \vdash e : B \quad \gamma : \Delta \Rightarrow \Gamma}{\Delta \vdash_v (\lambda(x : A). e)[\gamma] = \lambda(x : A). e[\gamma, x/x] : A \rightarrow B}
\end{array}$$

Figure 3.2: STLC

that I can do language metaprogramming in Gallina (Rocq’s dependently typed metalanguage), which I use to generate such equations automatically rather than write them by hand. In doing so, I follow a substantial body of research. For example, the Autosubst line of work used first Ltac and then an external code generator to generate substitution functions for user-defined inductive datatypes [49,50]. I handle the related situation in my Pyrosome case studies by programmatically generating equations that define the behavior of the explicit-substitution syntax from language syntax rules. For example, the final two equations in Figure 3.2 are actually autogenerated by this procedure, not handwritten. While the resulting rules form part of the compiler’s trusted specification, they can be audited straightforwardly. I apply this machinery to all extensions of the substitution calculus and additionally port it to the corresponding constructs in the linear substitution calculus used in section 3.7.

3.2 Language Extensions

In subsection 2.0.1 and section 3.1, I informally described the vector GAT as an extension of the naturals GAT and STLC as an extension of substitution by means of adding rules to the base GAT. Fortunately, a formal accounting is intuitive: Since GATs are determined by lists of rules, I can express the addition of an extension as the concatenation of the new rules onto the old ones. The key benefit of GATs for my purposes is that all the well-formedness and equivalence judgments of a GAT are preserved by language extension. I formalize this family of properties in Theorem 3.1, where $L \subseteq L'$ denotes the unordered inclusion of all rules of L in L' . Here and in subsequent metatheory, I write $C \vdash_L Q$ to mean “ Q is derivable from C in language L ,” where C is a dependent list of metavariable sort declarations.

Theorem 3.1 (Monotonicity under language extension). *Let L and L' be languages such that $L \subseteq L'$.*

- *If $C \vdash_L s$ sort, then $C \vdash_{L'} s$ sort*
- *If $C \vdash_L t : s$, then $C \vdash_{L'} t : s$*
- *If $C \vdash_L t_1 = t_2 : s$, then $C \vdash_{L'} t_1 = t_2 : s$*

Proof. By mutual induction on all GAT judgments, noting that the cases that make use of L only check for inclusion. $\square$

It is important to remember that the order of rules matters in determining whether a language is well-formed because the well-formedness of each rule might depend on the constructors and equations defined in earlier rules, as we saw with the defining equations for the vector append operation relying on the equations for addition to justify their well-formedness. However, while the order of a language determines whether it is well-formed, it does not affect its semantics, as demonstrated by my use of list inclusion in [Theorem 3.1](#). In general, a language remains well-formed under any permutation that produces a topological sort of its dependency graph, so independent extensions can be swapped freely in a judgment via this family of properties. For example, consider an extension of the substitution calculus that adds Boolean values true and false as well as the conditional expression if e then e_1 else e_2 . Both Booleans and STLC are extensions of substitution, but neither extension depends on the other, so they can be reordered arbitrarily.

3.2.1 Equational Theories, Contextual Equivalence, and Extensibility

As I have shown, language semantics in Pyrosome are given directly by equational theories rather than by defining an operational semantics and deriving contextual equivalence from it, as is often done in the literature on verified and/or secure compilation [\[51,52\]](#). This choice is necessary if I want to have [Theorem 3.1](#). Contextual equivalence is the maximal equivalence that respects an operational semantics [\[17\]](#); it equates every pair of programs it can while still respecting reduction. Equational theories on the other hand represent smaller equivalences, namely the smallest congruent equivalences that respect the chosen rules. In practice, the axioms of a language in Pyrosome are usually just its reduction rules, possibly with the inclusion of η laws. Thus, equivalence in Pyrosome relates two programs if and only if their behavior must be equivalent in every implementation and semantic model of the language.

To demonstrate the difference, I will consider the following simple example program, which elides `ret` for convenience of presentation:

$$\begin{aligned} \text{callTwice} & : (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool} \rightarrow \text{bool} \\ \text{callTwice} & \triangleq \lambda(f : \text{bool} \rightarrow \text{bool}). \lambda(x : \text{bool}). \\ & \quad (\lambda(_ : \text{bool}). f\ x) (f\ \text{false}) \end{aligned}$$

When passed f and x , it first calls f on false and then returns $f\ x$. In STLC, *callTwice* is contextually equivalent to the identity function $\lambda(f : \text{bool} \rightarrow \text{bool}). f$ since STLC is a pure, terminating language. However, consider how this program behaves once I add the recursive-function extension in [Figure 3.3](#). This extension adds a new value, `fix` $f(x : A) := e$, that represents a recursive function that may refer to itself as f in its body e . It also adds a β -reduction rule for interacting with the existing function-application operation from STLC. With this extension, I can apply *callTwice* to functions like `fix` $f(x : \text{bool}) := \text{if } x \text{ then true else } f\ x$, which is the identity function on the input true but diverges on the input false. Since *callTwice* always starts by applying f to false, it turns this function from

$$\begin{array}{c}
\frac{\Gamma, f : A \rightarrow B, x : A \vdash e : B}{\Gamma \vdash_v \text{fix } f(x : A) := e : A \rightarrow B} \\
\\
\frac{\Gamma, f : A \rightarrow B, x : A \vdash e : B \quad \Gamma \vdash_v v : A}{\Gamma \vdash (\text{ret fix } f(x : A) := e) \text{ ret } v = e[(\text{fix } f(x : A) := e)/f, v/x] : B}
\end{array}$$

Figure 3.3: Recursive-function extension

one that terminates on one of two inputs to one that never terminates, so *callTwice* is no longer contextually equivalent to the identity function. Since language extension can so definitively break contextual equivalence, the latter is not a suitable property for Pyrosome to consider. On the other hand, [Theorem 3.1](#) tells us that any pair of functions that we can equate in the GAT for STLC will still be equivalent once we add recursion or indeed any other extension.

3.3 Cross-Language Translation

Now that I have established how languages are defined and extended, I can proceed to their compilers. I define my translations as finite maps from source-language sort and term names to target-language sorts and terms, which form strict morphisms between GATs. These finite maps are then folded over a source term, looking up each constructor, to translate it. Then, just as I can extend my languages by appending new rules, I can extend my compilers by appending new mappings. This style formalizes patterns that appear in prior work on compositional compiler correctness [\[12,53,54\]](#).

This thesis primarily addresses cross-language translations in order to focus on challenges particular to cross-language statements of correctness that can be circumvented in the intralanguage case, for example for optimization passes. Specifically, there is a central tension in the area of compiler correctness between expressiveness and modularity, which extends to what translations I consider in this work. In order to support both linking and compiler extension fully, I require compilers to operate on terms with arbitrary free metavariables. Furthermore, I want compilation to be invariant under substitution of metavariables, i.e. $\llbracket \gamma(e) \rrbracket = \llbracket \gamma \rrbracket(\llbracket e \rrbracket)$, where $\llbracket e \rrbracket$ denotes the compilation of e and $\gamma(e)$ denotes the metavariable substitution γ applied to the term e .¹

I choose to guarantee that this equation holds syntactically. While I could likely work through much of Pyrosome’s metatheory using a superficially more flexible semantic equality, it is my observation that such a change would do little to change the expressiveness of my compilers. For each term former f of arity n , such a compiler would have to define $\llbracket f \ x_1 \dots x_n \rrbracket$, where $x_1 \dots x_n$ are metavariables. Then even if the compiler were to inspect subterms deeply when compiling a more complex term $\llbracket f \ e_1 \dots e_n \rrbracket$ for subterms $e_1 \dots e_n$, it would have to produce a result semantically equivalent to $\llbracket f \ x_1 \dots x_n \rrbracket[\llbracket e_1 \rrbracket/x_1, \dots, \llbracket e_n \rrbracket/x_n]$. Since I expect

¹Unlike object-language substitution, metavariable substitutions $\gamma(e)$ are a Gallina computation on ASTs rather than another syntactic form.

$$\begin{array}{c}
\frac{\Gamma : \text{ctx}}{\text{cmp}(\Gamma) \text{ sort}} \quad \frac{}{\vdash \neg A} \quad \frac{\Gamma \vdash_v v : \neg A \quad \Gamma \vdash_v v' : A}{\Gamma \vdash_v v v'} \quad \frac{\Gamma, x : A \vdash e}{\Gamma \vdash_v \lambda(x : A). e : \neg A} \\
\\
\frac{\Gamma, x : A \vdash e \quad \Gamma \vdash_v v : A}{\Gamma \vdash (\lambda(x : A). e) v = e[v/x]} \quad \frac{\Gamma \vdash v : \neg A}{\Gamma \vdash_v (\lambda(x : A). v x) = v : \neg A} \quad \frac{\vdash A \quad \vdash B}{\vdash A \times B} \\
\\
\frac{\Gamma \vdash_v v : A \times B \quad \Gamma, x : A, y : B \vdash e}{\Gamma \vdash \text{let } \langle x, y \rangle := v \text{ in } e} \quad \frac{\Gamma \vdash_v v_1 : A \quad \Gamma \vdash_v v_2 : B}{\Gamma \vdash_v (v_1, v_2) : A \times B} \\
\\
\frac{\Gamma \vdash_v v_1 : A \quad \Gamma \vdash_v v_2 : B \quad \Gamma, x : A, y : B \vdash e}{\Gamma \vdash \text{let } \langle x, y \rangle := (v_1, v_2) \text{ in } e = e[v_1/x, v_2/y]} \\
\\
\begin{array}{l}
\llbracket \text{val}(\Gamma, A) \rrbracket \triangleq \text{val}(\llbracket \Gamma \rrbracket, \llbracket A \rrbracket) \\
\llbracket \text{exp}(\Gamma, A) \rrbracket \triangleq \text{cmp}(\llbracket \Gamma \rrbracket, k : \neg \llbracket A \rrbracket) \\
\llbracket A \rightarrow B \rrbracket \triangleq \neg(\llbracket A \rrbracket \times \neg \llbracket B \rrbracket) \\
\llbracket \text{ret } v \rrbracket \triangleq k \llbracket v \rrbracket \\
\llbracket \lambda(x : A). e \rrbracket \triangleq \lambda(p : \llbracket A \rrbracket \times \neg \llbracket B \rrbracket). \text{let } \langle x, k \rangle := p \text{ in } \llbracket e \rrbracket \\
\llbracket e e' \rrbracket \triangleq \text{bind } x := \llbracket e \rrbracket; \text{bind } y := \llbracket e' \rrbracket; x \langle y, k \rangle, \text{ where} \\
\quad \text{bind } x := e; e' \triangleq e[\lambda(x : B). e'/k] \text{ given } \Gamma, k : \neg B \vdash e
\end{array}
\end{array}$$

Figure 3.4: Continuation calculus (selected rules) and CPS translation for STLC

optimization to be handled by intralanguage passes rather than my translations anyway, I choose the simpler option of taking the above term to be the definition of compilation.

3.3.1 CPS

My case-study compilers closely follow the designs of the first two passes of Morrisett et al. [28], albeit adapted to start with a simply typed compiler before adding in polymorphism. The CPS translation targets a calculus of continuations, shown in the top half of Figure 3.4. This calculus extends the values portion of the base substitution calculus. Since CPS computations do not return, I replace the expression sort, instead extending the value-substitution calculus with a computation sort $\text{cmp}(\Gamma)$ that eliminates the return type, writing $\Gamma \vdash e$ to indicate a well-formed computation $e : \text{cmp}(\Gamma)$. I define $\neg A$ as the type of a nonreturning continuation that accepts input of type A , named to evoke the definition $\neg A \triangleq A \rightarrow \perp$ and the connection between CPS and double negation. I also include a positive product-types extension in the CPS target.

With the target out of the way, I can define my first translation. The second half of Figure 3.4 gives the translation from STLC into my CPS calculus. Formally, I encode this translation as an association list mapping term and sort formers of STLC to terms and sorts in the continuation calculus that may have free metavariables corresponding to the context of the rule associated to the term or sort former. For example, the $\text{ret } v$ case above represents

a pair in the association list containing the symbol `ret` and the output term $k \llbracket v \rrbracket$ where the $\llbracket v \rrbracket$ on the right-hand side is a metavariable that will be substituted with the result of compiling the subterm in the v position when the compiler runs.

To bridge the gap between source-language expression judgments and target-language computations, I translate expression judgments $\Gamma \vdash e : A$ to computation judgments $\llbracket \Gamma \rrbracket, k : \neg \llbracket A \rrbracket \vdash \llbracket e \rrbracket$, using the variable k for the continuation. Functions $A \rightarrow B$ map to computations $\neg(\llbracket A \rrbracket \times \neg \llbracket B \rrbracket)$, each taking as inputs the compiled argument $\llbracket A \rrbracket$ and the continuation $\neg \llbracket B \rrbracket$. Since I explicitly represent the injection from values to expressions with the syntax `ret v` in the source, the standard CPS translation of values is split in two: the translation for `ret v` is responsible for calling the continuation, and the translation for function values (and other value forms as I add them) handles mediating between source and target values.

The compiler case for function application debuts a useful macro, `bind`, that I will employ in most of my CPS-pass extensions as well. I define `bind  $x := e$ ;  $e'$` as the expression substitution that passes a function from x to e' as the continuation to e , which satisfies the monad laws implied by its name in the theory of the continuation calculus. Most importantly, $(\text{bind } x := \llbracket \text{ret } v \rrbracket; e') = e'[\llbracket v \rrbracket/x]$. Especially when there are multiple binds, as in function application, I find this notation far more readable than nested, inverted substitutions.

3.3.2 Semantics Preservation

So far I have defined a few languages and a minimal compilation pass, but I have yet to state or prove theorems. There are many interpretations of compiler correctness in the literature [9]. The original CompCert [2] work proved whole-program simulation of the source language by the target language and used that relation to show trace refinement. Simulation of closed programs is inherently vertically compositional, which allowed CompCert’s proof of correctness to be divided cleanly by pass. However, modern software relies on linking code from multiple sources, which a correctness property of this form does not cover [40]. Recent developments in the CompCert ecosystem such as CompCertO [10] support linking, and Zhang et al. [55] improve the state of affairs by properly encapsulating intermediate semantics. However, they each describe interaction through a subsuming semantic model designed with C-like languages in mind. This choice does reflect modern languages, which often interoperate with each other through a C interface. However, I view the state of FFIs as the unfortunate result of the current limitations of language specifications and interfaces, which I believe can be improved via the principles of Pyrosome. To put it another way, I believe that with proper language and compiler modularity, language interoperation can be freed of its dependence on shared models like C. Other work on verified compiler correctness solves the linking problem for programs that share a specific low-level target language [56]. It is not clear how to modify such systems to allow modular addition of new target-language features or features that extend past the design of the original semantic model.

Pyrosome is designed for proving that compilers are type- and equivalence-preserving with respect to source and target equational theories, which I express formally in the following definition:

Definition 3.2 (Semantics Preservation). *A function $\llbracket - \rrbracket$ is a semantics-preserving compiler*

from source S to target T if all of the following hold²:

- If $C \vdash_S s$ sort then $\llbracket C \rrbracket \vdash_T \llbracket s \rrbracket$ sort
- If $C \vdash_S t : s$ then $\llbracket C \rrbracket \vdash_T \llbracket t \rrbracket : \llbracket s \rrbracket$
- If $C \vdash_S t_1 = t_2 : s$ then $\llbracket C \rrbracket \vdash_T \llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket : \llbracket s \rrbracket$

I claim that these properties are the critical ones for notions of compiler correctness. While cross-language relations are more common in the literature, they typically depend on language-specific relations between observable values or states, which is less than ideal for generic metatheory. Furthermore, they can be recovered quickly from semantics preservation when desired. It turns out that cross-language specifications can readily be reduced to equivalence preservation and a standard inversion lemma.

For example, consider extending the CPS translation to compile Booleans in the source to naturals in the target. I add the rules $\llbracket \text{true} \rrbracket = \text{nv } 1$ and $\llbracket \text{false} \rrbracket = \text{nv } 0$ where nv injects a separate sort of natural numbers into the sort of values at type nat . Note that there are two reasonable cross-language value relations, which I will write as $v_s \sim v_t$, that one might choose as the specification for a compiler with these rules. Both relations agree that $\text{false} \sim \text{nv } 0$, but it would be reasonable either to decide $\text{true} \sim \text{nv } 1$ and no other naturals are allowed, or to add $\text{true} \sim \text{nv } n$ where $n \neq 0$. For this example, I will arbitrarily choose the first specification. Such decisions form an inherent, language-specific component of cross-language compiler specifications and so are unsuitable for, and do not benefit from, inclusion in agnostic scaffolding like Pyrosome.

For now, assume that the CPS translation is semantics-preserving. The components necessary to bridge to a cross-language relation are inversion and relatedness of values, precisely the language-specific parts that do not benefit from an agnostic framework. Additionally, they only deal with the basic properties of values, so lifting to expressions is entirely handled by equivalence preservation.

Theorem 3.3 (CPS Cross-Language Correctness). *If $\vdash e = \text{ret } b : \text{bool}$ in $STLC + \text{Bool}$, then $k : \neg \text{bool} \vdash \llbracket e \rrbracket = k \ n$ in $CPS + \text{Nat}$ and $b \sim n$.*

Proof. By equivalence preservation, $k : \neg \text{bool} \vdash \llbracket e \rrbracket = \llbracket \text{ret } b \rrbracket$. By inversion, since b is closed, I have two cases: $\vdash_v b = \text{true} : \text{bool}$ and $\vdash_v b = \text{false} : \text{bool}$. In the first case, $\llbracket \text{ret true} \rrbracket$ evaluates to $k \ (\text{nv } 1)$ and $\text{true} \sim \text{nv } 1$, so I conclude. The second proceeds analogously. $\square$

3.3.3 The Preserving Predicate

Now that I have established semantics preservation as my goal, it remains to establish a standard method for proving it and for doing so extensibly. A significant contribution of this work is my definition of a predicate over compilers that follows the structure of the source language and a proof that this predicate implies semantics preservation. To demonstrate why this predicate is essential to my extensibility results, consider a standard proof of equivalence

²An additional property for preservation of sort equalities is actually included in my artifact, but I do not include it here since I have yet to make use of sort equations, and so it is entailed by these three.

preservation for my CPS pass for base STLC. I assume type preservation in this example for simplicity and walk through the term-equivalence case. Given two STLC expressions e_1 and e_2 such that $\Gamma \vdash e_1 = e_2 : A$, I have to show that $\llbracket \Gamma \rrbracket, k : \neg \llbracket A \rrbracket \vdash \llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$ in the continuation calculus. I proceed by induction on the proof of $\Gamma \vdash e_1 = e_2 : A$, which requires me to consider reflexivity, transitivity, symmetry, congruence, and β -reduction. The first three hold either trivially or by the inductive hypothesis. Congruence requires that the compiler be a homomorphism with respect to substitution, which I have by the structure of translations in Pyrosome. What remains to prove is the β -reduction case, where I must prove that the right- and left-hand sides compile to equivalent terms, although I omit the proof here.

If I examine the structure of this proof sketch, the only case that depends on the rules of STLC or the definition of the CPS pass is β -reduction. The rest can be proven using invariants common across all Pyrosome languages and compilers. Since they are independent of the compiler under consideration, I can prove them generically, so I focus on equations from the source-language definition like β -reduction. Consider how the proof must be extended to add a new feature, for example product types. Intuitively, the proof case for β -reduction should remain the same, and I must add new cases for the first and second projections out of a pair. Such an extension is logically straightforward, so I would expect it to be reasonable to formulate mechanically.

However, the structure of my theorem statement, that for all STLC terms, or for all terms made of product constructs, compilation preserves equivalence, is ill-suited to such extension since it tells me nothing about terms with a mix of product operations and STLC constructs. To properly reuse my proof about STLC, I need to encapsulate it in a lemma that can be extended inside the induction on equivalence proofs.

To describe the proof obligations associated with a given source language and compiler, I define the inductive predicate $\text{Preserving}(L_t, \text{cmp}, L_s)$, which takes as input a target language L_t , compiler cmp , and source language L_s . This predicate has one constructor for each kind of rule that can be appended to L_s . Each constructor requires that the compiler satisfies Preserving for the tail of L_s plus the appropriate condition for the head depending on the kind of rule:

- For a sort constructor, the compiler must map it to a well-formed sort in the target.
- For a term constructor, the compiler must map it to a well-formed term in the target.
- For an equation, the compiler must map the left- and right-hand sides to equivalent terms in the target.

I present the formal definition of Preserving in [Figure 3.5](#), with the sort cases elided for space as they mirror the term cases. Ignoring cmp_pre for the moment, the definition closely matches the prose description. The empty compiler translates the empty language. When the top rule of a language defines a new term former named $\mathbf{n}$, the compiler must map $\mathbf{n}$ to a term $\mathbf{e}$ such that $\mathbf{e}$ is well-formed at a sort and context determined by applying the compiler to the sort and context of the rule. The rest of the compiler and rest of the language must then satisfy Preserving . In the case of an equality rule, there is no new case of the compiler. However, there is still a proof obligation: the user must show that the terms on

```

Context (cmp_pre : compiler).
Inductive preserving_compiler_ext : compiler -> lang -> Prop :=
| preserving_compiler_nil : preserving_compiler_ext [] []
...
| preserving_compiler_term : forall cmp l n c args e t,
  preserving_compiler_ext cmp l ->
  Model.wf_term (compile_ctx (cmp ++ cmp_pre) c) e (compile_sort (cmp ++ cmp_pre) t) ->
  preserving_compiler_ext ((n, term_case (map fst c) e) :: cmp)
    ((n, term_rule c args t) :: l)
| preserving_compiler_term_eq : forall cmp l n c e1 e2 t,
  preserving_compiler_ext cmp l ->
  Model.eq_term (compile_ctx (cmp ++ cmp_pre) c)
    (compile_sort (cmp ++ cmp_pre) t)
    (compile (cmp ++ cmp_pre) e1)
    (compile (cmp ++ cmp_pre) e2) ->
  preserving_compiler_ext cmp ((n, term_eq_rule c e1 e2 t) :: l).

```

Figure 3.5: Rocq definition of **Preserving** (selected cases)

the left and right of the equality, when compiled to the target language, remain equal. One significant benefit of working with terms with free metavariables is that this obligation is not quantified at the Rocq level, so proving it just requires a single target-language derivation about concrete programs.

The formal definition above features an additional parameter `cmp_pre` because it is actually a generalization of **Preserving** from a whole-pass predicate to a pass-extension predicate. Let cmp_{pre} be a compiler from source language L_{pre} to target L_t , and let L_s be an extension of L_{pre} . I will write $\text{Preserving}_{cmp_{pre}}(L_t, cmp, L_s)$ for the generalized predicate, which states that cmp is a semantics-preserving compiler extension of cmp_{pre} supporting source extension L_s . I can then define $\text{Preserving}(L_t, cmp, L_s)$ as $\text{Preserving}_{\square}(L_t, cmp, L_s)$, the case where the base language is empty.

The key to proving this property is that each case of **Preserving** only relies on earlier cases' language rules and mappings in the compiler. Since the sorts, terms, and contexts that make up each rule of a language can only reference constructs from earlier rules (those in the tail of the list), it is sufficient at each rule to be able to compile only the previous constructors. For example, when compiling STLC using the CPS compiler, the proof obligation for lambdas references a prefix of the compiler that does not include application, since the application rule comes later in the language specification. This ordering is also essential to the modularity of **Preserving** since it conversely means that proofs of earlier obligations remain valid as the compiler is extended.

Theorem 3.4 (**Preserving** implies semantics preservation). *Let L_s and L_t be well-formed languages, and let cmp be a compiler. If $\text{Preserving}(L_t, cmp, L_s)$, then cmp is a semantics-preserving compiler from L_s to L_t .*

For **Preserving** to be useful, it must give me semantics preservation. I bridge this gap in [Theorem 3.4](#), which states that the predicate described above implies the universally quantified semantic properties. The proof of this theorem is by mutual induction over all of

the judgment forms in Pyrosome, where for each judgment, I must show that it is preserved by compilation. The term-equivalence case resembles the earlier proof I sketched for the CPS pass starting from STLC, except that I generalize the β -reduction case to a lemma about the preservation of each equivalence written in the language description. This lemma relies on two related principles: weakening and monotonicity. I disallow compilers from overwriting old cases, so I can safely use weakening lemmas to extend the compilers in the hypotheses provided by **Preserving** so that they refer to the whole compiler. To finish lifting each obligation to cover the whole compiler and source language, I appeal to [Theorem 3.1](#), which states that all judgments are monotonic under language extension.

Thanks to this theorem, I prove semantic preservation for each translation in Pyrosome by way of **Preserving**. My mechanization automatically breaks down the necessary proof obligations, and the resulting goals are quite amenable to both automation and human reasoning since they feature no quantification at the Rocq level and can be proven by direct construction of either well-formedness or equivalence derivations in the target language. In fact, I am now equipped to prove equivalence preservation for my base CPS pass, which I will do in [Theorem 3.5](#) and [Theorem 3.6](#), which state respectively that the translation for the base substitution calculus and the STLC extension satisfy **Preserving**.

Theorem 3.5 ($\text{CPS}_{\text{subst}}$ is **Preserving**). *Let subst be the source substitution calculus, let $\text{cps}_{\text{subst}}$ be the portion of the CPS pass with subst as its domain, and let cont be the continuation calculus. Then we have $\text{Preserving}_{\square}(\text{cont}, \text{cps}_{\text{subst}}, \text{subst})$.*

Proof. By the definition of **Preserving** and construction of cont derivations in all cases. $\square$

Theorem 3.6 (CPS is **Preserving**). *Let $\text{cps}_{\text{subst}}$ be the portion of the CPS pass with subst as its domain, let cps be the portion with the STLC extension as its domain, and let cont be the continuation calculus. Then we have $\text{Preserving}_{\text{cps}_{\text{subst}}}(\text{cont}, \text{cps}, \text{STLC})$.*

Proof. By the definition of **Preserving** and construction of cont derivations in all cases. I show the β -reduction case here as an example (although all cases are fully automated in the mechanization). The proof obligation that **Preserving** generates corresponding to this rule requires me to show $\llbracket (\text{ret } \lambda(x : A). e) \text{ ret } v \rrbracket = \llbracket e[v/x] \rrbracket$ in the continuation calculus' equivalence relation. By evaluating the compiler and rewriting the term via target-language

rules, I do so as follows³:

$$\begin{aligned}
& \llbracket (\text{ret } \lambda(x : A). e) (\text{ret } v) \rrbracket \\
= & \text{bind } x := \llbracket \text{ret } \lambda(x : A). e \rrbracket; \text{bind } y := \llbracket \text{ret } v \rrbracket; x \langle y, k \rangle \\
= & \text{bind } x := \llbracket \text{ret } \lambda(x : A). e \rrbracket; (\lambda(y : \llbracket A \rrbracket). x \langle y, k \rangle) \llbracket v \rrbracket \\
= & \text{bind } x := \llbracket \text{ret } \lambda(x : A). e \rrbracket; x \langle \llbracket v \rrbracket, k \rangle \\
= & (\lambda(x : \llbracket A \rightarrow B \rrbracket). x \langle \llbracket v \rrbracket, k \rangle) \llbracket \lambda(x : A). e \rrbracket \\
= & \llbracket \lambda(x : A). e \rrbracket \langle \llbracket v \rrbracket, k \rangle \\
= & (\lambda(p : \llbracket A \rrbracket \times \neg \llbracket B \rrbracket). \text{let } \langle x, k \rangle := p \text{ in } \llbracket e \rrbracket) \langle \llbracket v \rrbracket, k \rangle \\
= & \text{let } \langle x, k \rangle := \langle \llbracket v \rrbracket, k \rangle \text{ in } \llbracket e \rrbracket \\
= & \llbracket e \rrbracket [\llbracket v \rrbracket / x, k / k] \\
= & \llbracket e \rrbracket [\llbracket v \rrbracket / x] \\
= & \llbracket e[v/x] \rrbracket
\end{aligned}$$

□

I am almost ready to prove semantics preservation from these facts, but first I need one more generic theorem:

Theorem 3.7 (Preserving Concatenation). *Let L_s and L_t be well-formed languages, let L'_s be a well-formed extension of L_s , and let cmp and cmp' be compilers such that $\text{Preserving}(L_t, \text{cmp}, L_s)$ and $\text{Preserving}_{\text{cmp}}(L_t, \text{cmp}', L'_s)$. Then $\text{Preserving}(L_t, \text{cmp}' + \text{cmp}, L'_s + L_s)$.*

Proof. By induction on the proof of $\text{Preserving}_{\text{cmp}}(L_t, \text{cmp}', L'_s)$. □

Now I can prove that my CPS pass is semantics-preserving in [Theorem 3.8](#). I will generally not go through these steps for future case-study components, since composing the **Preserving** theorems is entirely mechanical and indeed performed by a tactic in the mechanization.

Theorem 3.8 (CPS is Semantics-Preserving). *Let subst be the source substitution calculus, let $\text{cps}_{\text{subst}}$ be the portion of the CPS pass with subst as its domain, let cps be the portion with the STLC extension as its domain, and let cont be the continuation calculus. Then we have $\text{cps} + \text{cps}_{\text{subst}}$ is a semantics-preserving compiler from $\text{STLC} + \text{subst}$ to cont .*

Proof. By [Theorem 3.4](#) it suffices to show $\text{Preserving}(\text{cont}, \text{cps} + \text{cps}_{\text{subst}}, \text{STLC} + \text{subst})$. I apply [Theorem 3.7](#) and conclude with [Theorem 3.5](#) and [Theorem 3.6](#). □

3.3.4 Closure Conversion and Vertical Composition

For the second pass of my case study, I perform closure conversion, materializing the environment as a tuple value. In [Figure 3.6](#), I show the closure calculus that replaces my CPS calculus. To construct environment tuples in a compositional way, I compile CPS environments to closure-converted types and map CPS judgments $\Gamma \vdash e$ and $\Gamma \vdash_v v : A$ to closure-converted judgments $z : \llbracket \Gamma \rrbracket \vdash \llbracket e \rrbracket$ and $z : \llbracket \Gamma \rrbracket \vdash_v \llbracket v \rrbracket : \llbracket A \rrbracket$, using the variable z for the environment. In order to define closure conversion in the simply typed setting, I use a fused

³I skip the details of stepping through the interaction of the bind macro with the compilation of a return since it involves verbose juggling of continuation substitutions and little else.

$$\begin{array}{c}
\frac{\vdash A}{\vdash \neg A} \qquad \frac{\Gamma \vdash_v v : \neg A \quad \Gamma \vdash_v v' : A}{\Gamma \vdash v v'} \qquad \frac{z : A \times B \vdash e \quad \Gamma \vdash_v v : A}{\Gamma \vdash_v \text{clo } \langle (z : A \times B).e, v \rangle : \neg B} \\
\\
\frac{z : A \times B \vdash e \quad \Gamma \vdash_v v : A \quad \Gamma \vdash_v v' : B}{\Gamma \vdash \text{clo } \langle (z : A \times B).e, v \rangle v' = e[\langle v, v' \rangle / z]} \\
\\
\frac{z : A \vdash_v v : \neg B}{z : A \vdash_v \text{clo } \langle (z : A \times B).v[z.1/z] \ z.2, z \rangle = v : \neg B} \\
\\
\begin{array}{lcl}
[\![\cdot]\!] & \triangleq & \text{unit} \\
[\![\Gamma, x : A]\!] & \triangleq & [\![\Gamma]\!] \times [\![A]\!] \\
[\![\neg A]\!] & \triangleq & \neg [\![A]\!] \\
[\![\lambda(x : A). e]\!] & \triangleq & \text{clo } \langle (x : [\![\Gamma]\!] \times [\![A]\!]).[\![e]\!], z \rangle \\
[\![v v']]\!] & \triangleq & [\![v]\!] [\![v']]\!]
\end{array}
\end{array}$$

Figure 3.6: Closure conversion (selected rules)

closure form $\text{clo } \langle (z : A \times B).e, v \rangle$ that captures the combined behavior of the existential, pair, and function in the normal translation, in a style closely related to that of Minamide, Morrisett, and Harper [57]. I define two equations on closures, a β rule and an η rule. The β rule evaluates the body of the closure, passing in a tuple formed of its argument and its environment. The η rule states that if we have a closure v with a free variable z , then it is equivalent to a new closure that stores z as its environment and calls v in its body.

Now that I have two compiler passes, I can sequence them using function composition to get from the source to the final target. Fortunately, I can do so in one step thanks to my choice of definitions. I prove semantics preservation for closure conversion on its own analogously to CPS. Then, by transitivity (Theorem 3.9), since both passes preserve semantics, so does their composition.

Theorem 3.9 (Transitivity of Semantics Preservation). *If f is a semantics-preserving compiler from L_1 to L_2 and g is a semantics-preserving compiler from L_2 to L_3 , then $g \circ f$ is a semantics-preserving compiler from L_1 to L_3 .*

Proof. I show that $g \circ f$ maps equivalent L_1 terms to equivalent L_3 terms. The other conjuncts of semantics preservation proceed analogously. Consider $C \vdash t_1 = t_2 : s$ in L_1 . By semantics preservation of f , $f(C) \vdash f(t_1) = f(t_2) : f(s)$ in L_2 . Then, by semantics preservation of g , $g(f(C)) \vdash g(f(t_1)) = g(f(t_2)) : g(f(s))$ in L_3 , so I conclude. $\square$

3.3.5 Recursive-Functions Compiler Extension

I extended STLC with rules for recursive functions in Figure 3.3. To perform CPS translation on recursive functions, I add an analogous construct for recursive continuations to the

continuation calculus and translate from the first set of rules to the second:

$$\frac{\Gamma, f : \neg A, x : A \vdash e}{\Gamma \vdash_v \text{fix } f(x : A) := e : \neg A} \quad \frac{\Gamma, f : \neg A, x : A \vdash e \quad \Gamma \vdash_v v : A}{\Gamma \vdash (\text{fix } f(x : A) := e) v = e[(\text{fix } f(x : A) := e)/f, v/x]}$$

$$\llbracket \text{fix } f(x : A) := e \rrbracket \triangleq \text{fix } f(z : \llbracket A \rrbracket \times \neg \llbracket B \rrbracket) := \text{let } \langle x, k \rangle := z \text{ in } \llbracket e \rrbracket$$

Building on the result for the core CPS transformation, I show in [Theorem 3.10](#) that the compiler for the recursion extension satisfies **Preserving**. This proof requires fulfilling three new obligations as per the definition of **Preserving**: one to show that the compiler is type-preserving on recursive functions, one to show that reduction of recursive functions is preserved, and one for the substitution rule for recursive functions, which was autogenerated as discussed in [subsection 3.1.1](#).

Theorem 3.10 (CPS_{rec} is Preserving). *Let rec be the recursive-functions extension to STLC, let cps be the CPS pass for STLC, let cps_{rec} be the CPS extension with rec as its domain, and let cont_{rec} be the continuation calculus extended with recursive continuations. Then we have Preserving_{cps}(cont_{rec}, cps_{rec}, rec).*

Proof. By the definition of **Preserving** and construction of *cont* derivations in all cases. $\square$

Similarly to [Theorem 3.8](#), I can show that the combined compiler $\text{cps}_{\text{rec}} + \text{cps}$ is semantics-preserving by using [Theorem 3.4](#) and [Theorem 3.7](#). However, I also need one more property. I can only concatenate compilers that have the same target language, but the codomain of *cps* is the continuation calculus before I have added recursive continuations. Fortunately, [Theorem 3.11](#) states that I can lift any compiler targeting a language L_t to one targeting a superset language L'_t . Thus, I can lift *cps* to target *cont_{rec}* before appending the *cps_{rec}* compiler extension.

Theorem 3.11 (Compiler codomain embedding). *If Preserving(L_t, cmp, L_s) and $L_t \subseteq L'_t$, then it follows that Preserving(L'_t, cmp, L_s).*

Closure conversion for recursive functions requires a bit more care. The interaction between recursive functions as formulated in my first two calculi and the environment tuple generated by closure conversion is a bit complex when all of the behavior is fused into a single construct, so I separate out a fixpoint combinator in my closure-conversion language:

$$\frac{\Gamma \vdash_v v : \neg(\neg A \times A)}{\Gamma \vdash_v \text{fix } v : \neg A} \quad \frac{\Gamma \vdash_v v : \neg(\neg A \times A) \quad \Gamma \vdash_v v' : A}{\Gamma \vdash (\text{fix } v) v' = v \langle \text{fix } v, v' \rangle}$$

$$\llbracket \text{fix } f(x : A) := e \rrbracket \triangleq \text{fix clo } \langle (z : \llbracket \Gamma \rrbracket \times \neg(\neg A \times A)). \llbracket e \rrbracket [\langle \langle z.1, z.2.1 \rangle, z.2.2 \rangle / z], z \rangle$$

The translation features some verbose tuple rearrangement, but in essence it splits the recursive function into the fixpoint combinator and its argument, which is simultaneously closure-converted. The separation of the two parts means that I can apply closure laws as normal to reason about the body and cordon off the recursion so that the two concerns do not interfere with each other. Just like with the prior passes and extensions, the definition is the greatest difficulty. The three connected proof obligations are solved automatically in

the mechanization. From there, I can use my connective theorems as before to join up all relevant components.

Now that I have covered all of the ways compilers can be extended, I include [Figure 3.7](#) as a reference for all of the different ways I have extended my compiler. Solid arrows indicate translations written and verified by the user, and dashed arrows indicate translations validated by the theorems of Pyrosome that I have introduced over the course of this section.

3.3.6 How Incorrect Proofs Fail

Pyrosome avoids the failure modes typical of trying to extend most prior verified compilers, especially those with theorems based on contextual equivalence, by guaranteeing unconditional monotonicity of terms', languages', and compilers' key properties under language extension. For compiler developers used to contextual equivalence, this principle can cause some confusion since they are understandably curious about what does happen in situations where they would expect a new extension to break old equivalences.

Such situations fall into two categories. The first category covers examples like *callTwice* in [subsection 3.2.1](#) where the new language feature contradicts a property implicitly assumed by contextual equivalence.

Since semantics preservation in Pyrosome only preserves the least congruence over the explicitly given rules, I avoid this issue by not promising these implicit properties in the first place. This behavior matches the intent of language specifications in Pyrosome as representing open sets of promises rather than exhaustive, closed universes.

The second category covers situations where two explicit equations intuitively conflict. Consider a call-by-value STLC with evaluation contexts that is set up to evaluate the function first in an application; that is, it has evaluation contexts $E\ e$ and $v\ E$. Now say I have two extensions: one that supports mutation and one that adds an additional evaluation context $e\ E$, indicating that evaluation may occur on either the right- or left-hand side first. It is possible to compile the source language coherently given either one of these extensions. However, the two of them cannot work together. What goes wrong concretely in Pyrosome is that the common translation for the core language will validate the equations for at most one of the extensions. If the compilation for mutation can be proven correct, there will be no reasonable way to satisfy the demands of the evaluation-context extension and vice versa. Concretely, the obligation generated by **Preserving** for the failing one will be unprovable due to requiring that two unrelated target programs be shown equivalent. There is a single unreasonable compiler that satisfies semantics preservation by translating all programs to unit. However, a trivial check that, for example, the compiler distinguishes 0 and 1, or indeed any cross-language corollary like [Theorem 3.3](#), is adequate to rule out this case.

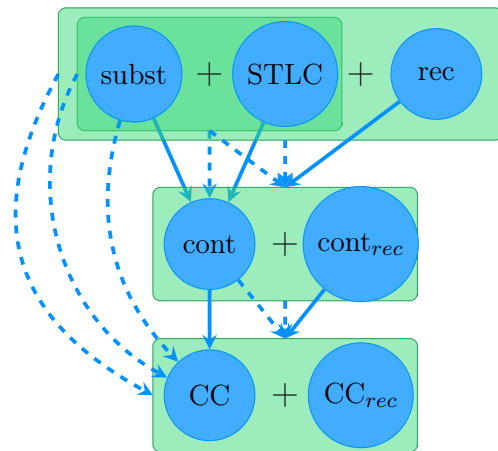


Figure 3.7: Compiler extension

3.3.7 Axiomatic Equivalence and Operational Semantics

For a language that already has a canonical operational semantics, I can bridge the gap to equational theories and still derive correctness theorems in terms of these semantics from Pyrosome’s semantics-preservation results. Generally, each step in the operational semantics is validated by a specific axiom in the equational theory, which means that reduction immediately implies equivalence. As a result, if $e \rightarrow^* v$ in the source language, then $\llbracket e \rrbracket = \llbracket v \rrbracket$ in the target where $\llbracket e \rrbracket$ is the compilation of e . There are then two ways to proceed. To conclude that the output of a compiler satisfies target-language contextual equivalence, it suffices to show that each rule of the target language’s equational theory is validated by the contextual equivalence of its external definition, for example by showing that each equation holds within a logical relation or an equivalence-reflecting denotation. For a simpler, if weaker, result, it suffices to prove that as with the source language, each step in the target operational semantics is validated by an equivalence. I can then use the following theorem to link the two:

Theorem 3.12. *Let $\llbracket - \rrbracket$ be a semantics-preserving compiler from S to T . Let $\rightarrow_S$ and $\rightarrow_T$ be subsets of the equivalence relations for S and T respectively. Let V_s and V_t be predicates on terms of languages S and T denoting the observable values of those languages such that if two observable values are equivalent, then they are syntactically equal. Let $\sim$ relate terms of language S to terms of language T such that for any observable value v_S of S , there exists an observable value v_T of T such that $\vdash_T \llbracket v_S \rrbracket = v_T$ and $v_S \sim v_T$.*

Then, if $\vdash_S e : t$, $e \rightarrow_S^* v$, and $\llbracket e \rrbracket \rightarrow_T^* v'$ where v and v' are observable values, then $v \sim v'$.

While the statement of [Theorem 3.12](#) is a bit long due to its generality, its assumptions boil down to requiring one fact each about source and target: that the operational steps are valid equations, and only one additional fact about the compiler: that the chosen cross-language relation relates every source value to a value equivalent to its compilation. As discussed in [subsection 3.3.2](#), this latter property can often be validated by simply running the compiler on the appropriate inputs.

3.4 Optimization

In my formulation, translation passes are designed for cross-language transformations rather than optimization. I focus on the cross-language case since it poses the greatest challenges with respect to formulating specifications, one of the key contributions of this work. The difference is that optimization passes typically operate within a single language, so they exist in a much broader design space since source and target code can interact. Specifically, an optimization pass o can choose to satisfy the specification $C \vdash t = o(t) : s$ for all terms $C \vdash t : s$, which implies semantics preservation.

Benton [\[58\]](#) showed that core optimizations like dead-code elimination and constant propagation can be expressed in an equational theory. While he justifies such theories by proving them sound with respect to a relational semantics, I can skip this step and consider the theories definitional, especially since intermediate languages, where optimizations typically occur, will disappear from the statement of end-to-end semantics preservation. More recent

$$\begin{array}{c}
\frac{\Gamma : \text{ctx} \quad \vdash A}{\text{config}(\Gamma, A) \text{ sort}} \qquad \frac{H : \text{heap} \quad \Gamma \vdash e : A}{\langle H, e \rangle : \text{config}(\Gamma, A)} \qquad \frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{get } e : \text{nat}} \\
\\
\frac{\Gamma \vdash E : \text{nat} \rightsquigarrow A \quad n : \mathbb{N}}{\langle H, E[\text{get } (\text{ret } (\text{nv } n))] \rangle = \langle H, E[H(n)] \rangle : \text{config}(\Gamma, A)} \qquad \frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e' : \text{nat}}{\Gamma \vdash \text{set } e := e' : \text{unit}} \\
\\
\frac{\Gamma \vdash E : \text{unit} \rightsquigarrow A \quad n : \mathbb{N} \quad m : \mathbb{N}}{\langle H, E[\text{set } (\text{ret } (\text{nv } n)) := \text{ret } (\text{nv } m)] \rangle = \langle H[n \mapsto m], E[\langle \rangle] \rangle : \text{config}(\Gamma, A)}
\end{array}$$

Figure 3.8: STLC state extension

work has used both e-graphs and more traditional tree traversals to develop language-generic, rewrite-based optimizations and analyses including uncurrying, dead-code elimination, and linear-algebra optimization [13,21,59]. Due to their rewrite-based interfaces, I expect that such tools can be adapted to my domain of equational theory-based language extensibility.

As an initial proof of concept, I implement a language-agnostic partial-evaluation pass that shares its implementation with some of my proof automation. The partial evaluator takes as input the deeply embedded representation of its host language and treats its equations as directed rewrite rules, internally producing and checking a proof of equivalence between its input and output. By filtering the input language, I can also restrict the pass to, for example, only rules that do not duplicate subterms. Thus, it is generic over all Pyrosome languages, can be freely inserted at any stage of a compilation pipeline, and permits language extension. While my partial evaluator is fairly naïve, I believe that it shows Pyrosome’s potential compatibility with more advanced rewrite-based optimizations.

3.5 Global State and Evaluation Contexts

One of my more interesting extensions adds a global mutable heap with set and get expressions, the rules for which I provide in Figure 3.8. These rules depend on other extensions not shown, one describing the behavior of global heaps and another that sets up evaluation contexts. Heaps are finite maps on natural numbers, defined with the appropriate equations. Evaluation contexts serve as an interesting example of cross-extension interaction. When defining pure reductions for features like functions and products, evaluation contexts are unnecessary since Pyrosome provides congruence. However, they are essential for the global-heap rules since they allow me to describe performing a stateful operation inside some larger program. From a formal perspective, they let me focus on a subterm of the computation while defining an equation on the whole configuration so that I can access the heap.

It would be more idiomatic in a GAT to formulate such operations with algebraic effects [60]. I nonetheless present mutation here using an explicit heap and evaluation contexts, together with algebraic equations on configurations that define the interactions between programs and heaps, to demonstrate the breadth of theories that can be encoded as GATs with reasonable naturality.

$$\begin{array}{c}
\frac{}{\Gamma \vdash [] : A \rightsquigarrow A} \qquad \frac{\Gamma \vdash E : A \rightsquigarrow B \quad \Gamma \vdash e : A}{\Gamma \vdash E[e] : B} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash [][e] = e : A} \\
\\
\frac{\Gamma \vdash E : A \rightsquigarrow B \rightarrow C \quad \Gamma \vdash e : B}{\Gamma \vdash E e : A \rightsquigarrow C} \qquad \frac{\Gamma \vdash_v v : B \rightarrow C \quad \Gamma \vdash E : A \rightsquigarrow B}{\Gamma \vdash_v v E : A \rightsquigarrow C} \\
\\
\frac{\Gamma \vdash E : A \rightsquigarrow B \rightarrow C \quad \Gamma \vdash e : B \quad \Gamma \vdash e' : A}{\Gamma \vdash (E e)[e'] = E[e'] e : C} \\
\\
\frac{\Gamma \vdash_v v : B \rightarrow C \quad \Gamma \vdash E : A \rightsquigarrow B \quad \Gamma \vdash e : A}{\Gamma \vdash (v E)[e] = v E[e] : C} \\
\\
\begin{array}{l}
\llbracket [] \rrbracket \triangleq k \ x_h \\
\llbracket E[e] \rrbracket \triangleq \text{bind } x :=; \ h \llbracket e \rrbracket \llbracket E \rrbracket \\
\llbracket E e \rrbracket \triangleq \text{bind } x := \llbracket E \rrbracket; \text{bind } y := \llbracket e \rrbracket; \ x \langle y, k \rangle \\
\llbracket v E \rrbracket \triangleq \text{bind } y := \llbracket E \rrbracket; \llbracket v \rrbracket \langle y, k \rangle
\end{array}
\end{array}$$

Figure 3.9: Evaluation-context extension with STLC

Figure 3.9 shows the core constructs of my evaluation-context extension on the first line, as well as the contexts for STLC as an example. Plug, which I write $E[e]$, and the empty context $[]$ form the core extension. I add evaluation contexts for each expression with evaluable subterms as new pieces of syntax belonging to a sort of evaluation contexts with judgment form $\Gamma \vdash E : A \rightsquigarrow B$ and define how plug acts on them. I can then use equational reasoning in my proofs to decompose terms.

As an incidental benefit of my formal, deeply embedded representation of language rules, I can write a function that generates the full typing rules and equations for evaluation contexts from short descriptions similar to those that typically appear in a language grammar. In fact, the four rules mentioning application contexts in Figure 3.9 are generated from the following description of the evaluation contexts for STLC:

$$E ::= \dots \mid E e \mid v E$$

The mechanization includes a unique identifier for each evaluation context, but I overload the syntactic form of the associated expression here, as is common. The sorts of the subterms are determined by their names. I denote evaluation contexts with names starting with E , expression subterms with e , and value subterms with v . From this description, I generate a well-formedness rule for each context and the defining equations for the plug operation $E[e]$.

Since my CPS translation uses $\text{bind } x := e; e'$ to sequence computations, I compile away evaluation contexts in my first pass. The bottom of Figure 3.9 shows the translation for the core operations as well as for STLC's evaluation contexts. This translation maps source-language judgments $\Gamma \vdash E : A \rightsquigarrow B$ to target judgments $\Gamma, k : \neg B, x_h : A \vdash \llbracket E \rrbracket$, translating evaluation contexts into computations with an extra free variable named x_h by convention. Plug then turns into a bind operation, and holes behave like return operations.

With evaluation contexts established, the translation for my heap operations is minimal. I can define the CPS pass for the set and get operations as follows:

$$\begin{aligned} \llbracket \text{set } e := e' \rrbracket &\triangleq \text{bind } x := \llbracket e \rrbracket; \text{bind } y := \llbracket e' \rrbracket; \text{set } x := y \text{ in } k \langle \rangle \\ \llbracket \text{get } e \rrbracket &\triangleq \text{bind } x := \llbracket e \rrbracket; \text{get } x \text{ as } y \text{ in } k \ y \end{aligned}$$

To match the pattern of the CPS calculus, each heap operation now takes its continuation as a subterm. This inversion is why I no longer need to worry about evaluating effects inside of a context. Since this extension does not interact directly with functions, I can reuse the CPS version with my closure-converted calculus. I still have to do a little more than write an identity compiler, however, since closure conversion changes the expected judgment form, which interacts with the binder in $\text{get } x \text{ as } v \text{ in } e$:

$$\begin{aligned} \llbracket \text{set } v := v' \text{ in } e \rrbracket &\triangleq \text{set } \llbracket v \rrbracket := \llbracket v' \rrbracket \text{ in } \llbracket e \rrbracket \\ \llbracket \text{get } x \text{ as } v \text{ in } e \rrbracket &\triangleq \text{get } x \text{ as } \llbracket v \rrbracket \text{ in } \llbracket e \rrbracket[\langle z, x \rangle / z] \end{aligned}$$

3.6 Polymorphism and Advanced Metaprogramming

Unlike other case-study extensions, polymorphism necessitates modifying the rules of my simply typed extensions to add the type environment. To do so, I develop a generic parameterization procedure that takes as input the language to be extended and a specification of which forms to parameterize, adding the appropriate metavariables. This procedure is language-agnostic and applies to more than just type environments: for example I also use it to add type information to my base substitution calculus, which allows the type substitutions and term substitutions to share a common set of underlying definitions. I prove that this parameterization function is semantics-preserving on sorts, terms, contexts, and languages. Using these facts, I further show that it preserves the **Preserving** predicate on compilers, allowing the output compilers to participate in all of my modular connections as first-class citizens.

While the high-level idea is simple, formalizing it was complex. The main challenge of this proof lay in justifying the features that do not need to be parameterized, for example the sort of natural numbers, and the position of the additional parameter in the context, when it does not appear as the first component. My parameterization procedure takes such language-specific information as inputs, so the generic proof of parameterization correctness must rely on certain properties of these declarations, like the existence of a dependency ordering such that constructs marked for parameterization cannot be dependencies of ones I do not parameterize. Fortunately, I reduce them to sound, decidable checks that can guarantee a valid parameterization without inducing additional proof obligations. The result is [Theorem 3.13](#).

Theorem 3.13 (Parameterization Preserves **Preserving**). *Let L_Δ , L_s , and L_t be well-formed languages. Let s_Δ be a well-formed sort in L_Δ , let Δ be the metavariable name for the new parameter, and let spec_s and spec_t be parameterization specifications. Let c be a compiler*

$$\begin{array}{c}
\frac{\Delta, X \vdash A}{\Delta \vdash \forall X.A} \quad \frac{\Delta, X \vdash A}{\Delta \vdash \exists X.A} \quad \frac{\Delta; \Gamma \vdash e : \forall X.A \quad \Delta \vdash B}{\Delta; \Gamma \vdash e [B] : A[B/X]} \quad \frac{\Delta \vdash \Gamma \quad \Delta, X; \Gamma \vdash e : A}{\Delta; \Gamma \vdash_v \Lambda X.e : \forall X.A} \\
\\
\frac{\Delta \vdash \Gamma \quad \Delta, X; \Gamma \vdash e : A \quad \Delta \vdash B}{\Delta; \Gamma \vdash (\text{ret } \Lambda X.e) [B] = e[B/X]} \quad \frac{\Delta; \Gamma \vdash_v v : \forall X.A}{\Delta; \Gamma \vdash_v \Lambda X.v [X] = v : \forall X.A} \\
\\
\frac{\Delta; \Gamma \vdash e : \exists X.A \quad \Delta \vdash C \quad \Delta, X; \Gamma, (x : A) \vdash e' : C}{\Delta; \Gamma \vdash \text{let } \langle X, x \rangle := e \text{ in } e' : C} \quad \frac{\Delta \vdash B \quad \Delta; \Gamma \vdash e : A[B/X]}{\Delta; \Gamma \vdash \langle B, e \rangle : \exists X.A} \\
\\
\begin{array}{l}
\llbracket \forall X.A \rrbracket \triangleq \neg \exists X. \neg \llbracket A \rrbracket \\
\llbracket \Lambda X.e \rrbracket \triangleq \lambda k. \text{let } \langle X, k \rangle := k \text{ in } \llbracket e \rrbracket \\
\llbracket e [A] \rrbracket \triangleq \text{bind } x := \llbracket e \rrbracket; x \langle \llbracket A \rrbracket, k \rangle \\
\llbracket \exists X.A \rrbracket \triangleq \exists X. \llbracket A \rrbracket \\
\llbracket \langle A, e \rangle \rrbracket \triangleq \text{bind } x := \llbracket e \rrbracket; \langle \llbracket A \rrbracket, x \rangle \\
\llbracket \text{let } \langle X, x \rangle := e \text{ in } e' \rrbracket \triangleq \text{bind } y := \llbracket e \rrbracket; \text{let } \langle X, x \rangle := y \text{ in } \llbracket e' \rrbracket \\
\text{bind } x := e; e' \triangleq e[\lambda(x : B). e'/k] \text{ given } \Gamma, k : \neg B \vdash e
\end{array}
\end{array}$$

Figure 3.10: Polymorphic calculus extensions (selected rules)

satisfying $\text{Preserving}(L_t, c, L_s)$. Then, if all syntactic checks hold on the above components, I have

$$\text{Preserving}_{id_{L_\Delta}}(P_L(\Delta, s_\Delta, \text{spec}_t, L_t), P_c(\Delta, \text{spec}_t, \text{spec}_s, c), P_L(\Delta, s_\Delta, \text{spec}_s, L_s))$$

where id_{L_Δ} is the identity compiler for L_Δ , P_L is the parameterization function for languages, and P_c is the parameterization function for compilers.

Once I have applied the parameterization procedure, instantiated with the information specific to the type-environment parameter, to my running case study, I can develop the type-abstraction and existential-types extensions to validate that the parameterization process had the intended effect. I display a portion of the rules for the source-language extension, as well as a selection of CPS-pass cases, in [Figure 3.10](#). In the target, the parameterized continuation calculus, I choose to add only existentials, as type abstractions can be compiled to existential continuations via CPS. Specifically, I translate the type $\forall X.A$ to $\neg \exists X. \neg \llbracket A \rrbracket$. The closure-conversion extension for existential types is almost an identity, with the only deviation being some boilerplate in the elimination form to address closure conversion's effect on the environment.

3.7 Linear Lambda Calculus

My primary case study builds entirely on a standard, intuitionistic substitution calculus. To demonstrate Pyrosome's expressiveness, I also verify a CPS pass analogous to my simply typed one from a separate, linear lambda calculus to a similarly linear continuation calculus.

$$\begin{array}{c}
\frac{\vdash A \quad \vdash B}{\vdash A \multimap B} \qquad \frac{\Gamma \vdash e : A \multimap B \quad \Gamma' \vdash e' : A}{\Gamma, \Gamma' \vdash e \, e' : B} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash_v \lambda(x : A). e : A \multimap B} \\
\\
\frac{\Gamma, x : A \vdash e : B \quad \Gamma' \vdash_v v : A}{\Gamma, \Gamma' \vdash (\text{ret } \lambda(x : A). e) (\text{ret } v) = e[v/x] : B} \\
\\
\frac{\Gamma \vdash e : A \multimap B \quad \Delta \vdash e' : A \quad \gamma : \Gamma' \Rightarrow \Gamma \quad \delta : \Delta' \Rightarrow \Delta}{\Gamma', \Delta' \vdash (e \, e')[\gamma, \delta] = e[\gamma] \, e'[\delta] : B} \\
\\
\frac{\Gamma, x : A \vdash e : B \quad \gamma : \Gamma' \Rightarrow \Gamma}{\Gamma' \vdash_v (\lambda(x : A). e)[\gamma] = \lambda(x : A). e[\gamma, x/x] : A \multimap B}
\end{array}$$

Figure 3.11: Linear Lambda Calculus

The linear substitution calculus is similar to the intuitionistic one, including retaining all rules in Figure 3.1, but it restricts substitutions and variables to prohibit weakening. I show the rules for the linear functions extension in Figure 3.11. These rules are similar to those of STLC, but in any rule with more than one expression or value, the context is broken up into multiple (disjoint) pieces. This separation continues in the substitution rule for application, where the left-hand substitution must break into two parts, one for e and one for e' . Just like with the intuitionistic substitution calculus, I programmatically generate the last two substitution rules for linear extensions. The CPS pass itself is actually almost identical to the CPS pass for STLC, with the only difference being the managing of explicit uses of the exchange rule as the mechanization is de Bruijn indexed.

3.8 IMP

To show that my framework is not restricted to functional languages, I also compile an imperative calculus similar to the one in Chapter 2 of Winskel [61], the grammar of which is shown in Figure 3.12, to the same target as my functional code. I reuse the heap definition from my functional heap extension to model memory, since memory locations without pointer arithmetic are equivalent to global variable names. Imperative statements translate to target-language computations that expect a continuation with unit argument, and expressions translate to computations where the continuation must be passed a natural number. To support this compiler, I add a conditional to the target language that branches based on whether the input value is 0. Since the target-language form only accepts values (including variables) in the condition, the compilation for source-level conditionals first evaluates the condition and then branches on the result. The compilation for while loops works similarly, with the added complication of a recursion construct so that it can jump back to the condition at the end of an iteration.

Note that since this compiler shares a common target with my main case study, I obtain some guarantees about the interoperation of imperative and functional code. Specifically,

$$\begin{aligned}
a, a' &::= n \mid x \mid a + a' \mid a - a' \\
s, s' &::= \text{skip} \mid \text{assign } x := a \mid s; s' \mid \text{ifz } a \text{ then } s \text{ else } s' \mid \text{while } a \{s\} \\
n, x &\in \mathbb{N}
\end{aligned}$$

$$\begin{aligned}
\llbracket \text{skip} \rrbracket &\triangleq k \langle \rangle \\
\llbracket \text{assign } x := a \rrbracket &\triangleq \text{bind } y := \llbracket a \rrbracket; \text{set } \llbracket x \rrbracket := y \text{ in } k \langle \rangle \\
\llbracket s; s' \rrbracket &\triangleq \text{bind } _ := \llbracket s \rrbracket; \llbracket s' \rrbracket \\
\llbracket \text{ifz } a \text{ then } s \text{ else } s' \rrbracket &\triangleq \text{bind } y := \llbracket a \rrbracket; \text{ifz } y \text{ then } \llbracket s \rrbracket \text{ else } \llbracket s' \rrbracket \\
\llbracket \text{while } a \{s\} \rrbracket &\triangleq \text{fix clo } \left\langle \begin{array}{l} z : \neg \text{unit} \times (\neg \text{unit} \times \text{unit}). \\ \text{bind } x := \llbracket a \rrbracket; \\ \text{ifz } x \text{ then } z.1 \langle \rangle \text{ else } \llbracket s \rrbracket[z.2.1/k] \end{array} \right\rangle, k \rangle \langle \rangle
\end{aligned}$$

Figure 3.12: IMP (excerpt)

they can be linked, directly or with target-level glue code, so long as the target types line up, and I can prove that linking a compiled imperative program $\llbracket S \rrbracket$ with a compiled functional program $\llbracket f \rrbracket$ by either method produces an equivalent program to linking some $\llbracket S' \rrbracket$ and $\llbracket f' \rrbracket$ the same way if $S = S'$ and $f = f'$ in their respective source languages.

3.9 Future Work

As-is, compilation theorems proven using Pyrosome include the initial source-language and final target-language specifications in their trusted base, in addition to Pyrosome’s core definitions. I hope to bridge the gap between existing formal semantics and ones defined in Pyrosome so that end users can benefit from the extensibility of engineering done in Pyrosome without having to trust Pyrosome’s definitions. The most important step in this process is to develop models of target-language theories implemented by verified low-level systems. Although I currently only target Pyrosome languages with my case studies, my compilers and metatheorems support any target model that validates core well-formedness and equivalence properties. I hope in the future to target established projects like CompCert [2] or Bedrock [6] so that upper levels of a compiler can use the flexibility of Pyrosome while benefiting from established codebases beneath. Target models could even be denotational in nature. For example, work using interaction trees already tends toward algebraic reasoning via corpuses of equational lemmas [62]. It may be worth investigating interaction trees as a denotation for Pyrosome languages, since the combination could leverage Pyrosome’s generic tooling to prove facts about interaction trees.

Additionally, Pyrosome was designed to support a wide range of features at both the term and type levels. Originally developed for dependent types, the theory behind Pyrosome has been shown to support a variety of interesting type-level features [15,63] in language definitions. For example, I have mechanized base Martin-Löf Type Theory as a dependent variant of my substitution language extended with dependent products and dependent sums, following the descriptions in prior work [46]. Such extensions rely on my monotonicity and compiler-extension theorems just like my case study. In the future, it would be interesting to see what

benefits Pyro might bring to the metatheory of dependent types, particularly focusing on understanding whether language extension can help create more-modular techniques for proving metatheoretic properties like canonicity and normalization.

Chapter 4

A Functional Egglog Implementation

This chapter describes my implementation of an egglog-style fixpoint reasoning system in Rocq. I leave verification to the next chapter as implementing such a system in a functional language is of independent interest, and more importantly, the work is substantial enough to warrant a chapter's worth of discussion.

4.1 An Illustrated Example

Although I said I would defer discussion of verification, I do want to motivate the connection between Pyrosome and e-graphs. To establish some baseline context, I will work through an example of how my verified EqSat procedure automates the core reasoning required to verify a compiler in Pyrosome. Consider the pass $\llbracket e \rrbracket$ that converts a program e from simply typed lambda calculus to continuation-passing style (CPS). By Pyrosome's compiler-correctness theorem, to verify a pass like the CPS pass, it suffices to show that, for each source-language judgement form, e.g. $\lambda(x : A). e$, the output of the compiler is well-typed in its target language, and for each source-language equational axiom, compiling both sides preserves the equivalence. Type preservation is typically straightforward to prove when it holds. Equivalence preservation is the critical component since it requires reasoning in the equational theory of the target language, which can easily be undecidable. The equations of STLC consist of a base theory of substitutions and variables with four added equations: substitution on application and function terms and β - and η -equivalence.

Consider the equation for call-by-value β -equivalence: $(\lambda(x : A). e) v = e[v/x]$. To satisfy Pyrosome's requirements, I must show that $\llbracket (\lambda(x : A). e) v \rrbracket = \llbracket e[v/x] \rrbracket$ in the equational theory of the CPS calculus. To prove this goal manually requires evaluating the compiler on the two programs and then applying equations from the target theory for target-language β -reduction and substitution until the left- and right-hand sides converge. I sketch that

process below, skipping most substitution equations:

$$\begin{aligned}
& \llbracket (\lambda(x : A). e) v \rrbracket \\
& \triangleq (k (\lambda p. \text{let } \langle x, k \rangle := p \text{ in } \llbracket e \rrbracket)) [\lambda z_1. (k \llbracket v \rrbracket) [\lambda z_2. (z_1 \langle z_2, k \rangle) / k] / k] \\
& = (k (\lambda p. \text{let } \langle x, k \rangle := p \text{ in } \llbracket e \rrbracket)) [\lambda z_1. ((\lambda z_2. (z_1 \langle z_2, k \rangle)) \llbracket v \rrbracket) / k] \\
& = (\lambda z_1. ((\lambda z_2. (z_1 \langle z_2, k \rangle)) \llbracket v \rrbracket)) (\lambda p. \text{let } \langle x, k \rangle := p \text{ in } \llbracket e \rrbracket) \\
& = (\lambda z_2. ((\lambda p. \text{let } \langle x, k \rangle := p \text{ in } \llbracket e \rrbracket) \langle z_2, k \rangle)) \llbracket v \rrbracket \\
& = (\lambda p. \text{let } \langle x, k \rangle := p \text{ in } \llbracket e \rrbracket) \langle \llbracket v \rrbracket, k \rangle \\
& = \text{let } \langle x, k \rangle := \langle \llbracket v \rrbracket, k \rangle \text{ in } \llbracket e \rrbracket \\
& = \llbracket e \rrbracket [\llbracket v \rrbracket / x] \\
& \triangleq \llbracket e[v/x] \rrbracket
\end{aligned}$$

Even though I skipped substitution steps in the derivation above, it shows how painfully mechanical it can be to perform equational proofs like this one manually. While for this specific case, EqSat is overkill and a reduction engine suffices, there are many such equational proofs that rely on η -equivalence or more generally on equations that do not directly define a confluent rewriting system. Furthermore, I observe that while such proofs can be onerous to perform manually, the programs involved are typically quite small from an automation perspective, and so EqSat seems an excellent fit for the domain. Using my EqSat procedure, I can prove $\llbracket (\lambda(x : A). e) v \rrbracket = \llbracket e[v/x] \rrbracket$ by invoking a single theorem, the soundness of my EqSat-based comparison function `egraph_equal`, and then running the requisite computation to see that it returns true.

4.2 Functional Architecture

This section covers the majority of the e-graph architecture but leaves details regarding e-graph rules and the internal worst-case-optimal join engine to dedicated sections. Although I follow the structure laid out by Zhang et al. [13], I do deviate in a number of ways. Most importantly, while many of the key techniques in efficiently implementing `egglog`-style EqSat, such as the idea of amortizing rebuilding, are straightforward to implement functionally (as is required to get the whole procedure running inside Rocq’s logic), a number are not. Some algorithms and data structures, like union-find, make sense to implement in the state monad, closely mirroring their imperative structure. Others, like worst-case optimal join, are easier to express in different, but conceptually connected, ways.

The most significant way in which I differ from the `egglog` implementation is due to the inherent immutable nature of my available data structures. The core table structure used in their development is fundamentally an imperative structure in a way that does not clearly translate to my setting. Instead, I use a simpler layout. An e-graph $\langle D, U, PA, E, W, A \rangle$ in my implementation is a 6-tuple of, in order, a tabular database D of the atoms in the e-graph, a union-find structure on identifiers U , a map from identifiers to their parents PA , a current epoch E , a worklist W of rebuilding actions, and a map A from identifiers to their minimal analyses. Of these, the first two should be seen as the real data of the e-graph, while the other four make up the metadata necessary to implement EqSat efficiently.

I purposefully retain a level of abstraction in the majority of the thesis by referring to identifiers rather than a more concrete type. Although in my current top-level Pyrosome

tactic, I invoke EqSat with binary numbers as identifiers using Rocq’s `positive` type, my code is parametric over the type of identifiers, so long as the chosen type is sufficiently natural number-like. Specifically, I assume identifiers support Boolean equality, have a base element, a successor operation, and an ordering in correspondence with the natural embedding of the natural numbers. In development, I took advantage of this feature to use strings as identifiers for debugging. On the other hand, I can likely improve performance by a noticeable constant factor by using machine words, although this instantiation would require small changes to check for overflow at identifier allocation as my proofs assume an infinite set of identifiers.

The database takes the form of a map from function symbols to tables, where each table is itself a map from a list of identifiers to an identifier, together with two pieces of metadata: one for tracking epochs and one for e-graph analysis. My tables take the form of maps rather than lists to enforce the functional dependence of atoms’ output identifiers on their arguments and to merge them efficiently when I add overlapping atoms to the e-graph. For example, using natural numbers as identifiers, if I have the atom $R(1, 2) \rightarrow 5$ in the e-graph and add the atom $R(1, 2) \rightarrow 7$, I must merge 5 and 7. With a map structure, I can discover the overlap without scanning the table.

I emulate the `egglog` concept of timestamping by tracking a current epoch E , a natural number that represents how many iterations of EqSat I have performed thus far. I also store the epoch each atom was last updated in the database, which allows me to implement semi-naïve evaluation by constructing indices that distinguish the newly updated entries based on their epochs.

Analysis results form the last piece of metadata in my database. My e-graphs are parameterized by an `analysis` typeclass that provides a pointed semi-lattice, that is a type with a meet operation and a default element, supporting Boolean equality and an atom-analysis function that maps an atom and a list of analysis results for its arguments to an analysis result. Although the behavior of my analyses does not affect the soundness of my EqSat theorems, for sensible results, this function should be monotone on its inputs. I tag each atom with the last-known result of atom analysis in the database and additionally record in the map A the meet of the analysis results of all atoms belonging to that identifier’s e-class. This information is useful during extraction, where it supports quick selection of the optimal atom from an equivalence class.

At the top level, my primary interface is an `egraph_equal` function that takes as input a language specification, a set of rules to apply, a maximum number of iterations to ensure termination, two terms in that language, and those terms’ expected sort. It returns `true` if it can equate those terms and `false` if it cannot. Pyrosome users who interact through this interface never even encounter an e-graph instance.

This function wraps a more general saturation procedure, `saturate_until`, that forms the core of my implementation, shown in [Figure 4.1](#). It accepts a set of rules to run, a maximum number of iterations n to ensure termination, a predicate P on e-graphs to allow for short-circuiting, and an initial e-graph. It then returns the e-graph produced by running the rule set on the initial e-graph until P is satisfied or for n iterations if it is not. In the case of `egraph_equal`, the e-graph is initialized with the two terms to be proven equivalent and their shared context, and P tests whether the identifiers of those terms have been unified. Procedures that perform more complex queries on the resulting e-graph, like extraction-based simplification, can also easily be built on `saturate_until`.

```

Context (rebuild_fuel : nat).

Definition runlitter (rs : rule_set) : state instance unit :=
  @!let tries ← build_tries rs in
    increment_epoch;
    (list_Miter (process_erule tries) rs.(compiled_rules));
    (rebuild rebuild_fuel).

Fixpoint saturate_until' rs (P : state instance bool) fuel : state instance bool :=
  match fuel with
  | 0 ⇒ Mret false
  | S fuel ⇒
    @!let done ← P in
      if (done : bool) then ret true
      else (runlitter rs);
            (saturate_until' rs P fuel)
  end.

Definition saturate_until rs (P : state instance bool) n : state instance bool :=
  @! (process_const_rules rs);
    (rebuild rebuild_fuel);
    (saturate_until' rs P n).

```

Figure 4.1: Top-level functions for EqSat

As shown in `saturate_until`, I primarily interact with the e-graph instance through the state monad, where the state consists of the current e-graph instance. I use the symbols `@!` to begin a monadic do-notation block where the notation `let x ← m in e` stands for monadic bind and, as a special case, the notation `m1; m2` stands for a bind where `m2` ignores the return result of `m1`. EqSat begins by running all constant rules and then rebuilding. I separate out constant rules to run once so as to avoid needlessly rerunning them every iteration, which also simplifies handling for all other rules as I can assume nonempty queries during the EqSat loop. The main loop, `saturate_until'`, runs up to its allotted fuel, short-circuiting if the condition P has been met. I choose to allow a monadic condition for ease-of-use, since that way it can query the e-graph using the same functions the rest of my implementation uses. If P has yet to be satisfied, I call `run1iter` to perform the steps of the inner loop. First it builds tries that serve as index structures, allowing me to query the database efficiently. It then increments the current epoch to reset the set of newly added atoms, runs each logic-program rule from the provided rule set, and then rebuilds to restore congruence invariants.

When I discover that, for any reason, two identifiers should be merged, I union them in the union-find U . Most e-graph operations that accept identifiers begin by canonicalizing them using the `find` function, which uses path compression to amortize the cost of frequent access. When two identifiers are merged, in addition to updating U , I push the pair onto the rebuilding worklist W . During rebuilding, I process each pair from the worklist using the code from my implementation in Figure 4.2. Just like Figure 4 in Willsey et al. [21], as long as the worklist is nonempty, my rebuilding function pulls the identifiers to process, canonicalizes them, and then invokes `repair` on each item. The termination argument for `rebuild` is nontrivial, and full rebuilding is not necessary for soundness, only completeness, so I simply pass in a large enough rebuilding fuel argument as an additional parameter to EqSat.

My `repair` function differs from theirs in two ways: First, it handles what I term *analysis repairs*, where the identifiers do not need to be updated, but the results of analysis can be improved. I repair these separately as the process mirrors standard repair but can terminate earlier by detecting that a parent’s analysis is unchanged. Second, for standard repairs, Willsey et al. [21] use single identifiers in their worklist and then iterate over all parents of the new union to update them. I observe that the canonical representative of the new union must be the representative of one of its inputs, so none of that input’s parents need to be updated. By recording pairs of both the demoted and maintained canonical identifiers, I can avoid iterating over the parents of the maintained identifier and instead just process the parents of the demoted identifier. Even in the case where I have to update the canonical identifier’s analysis, doing so is at most as expensive as a standard repair and typically less.

I implement this process in `repair_union`. First it removes the parents of the demoted identifier `x_old` from the parent map. For each parent in that list, it calls `repair_each`, which removes the atom from the database, canonicalizes it, and then re-adds it. This action adds the new atom as a parent to all of the identifiers it contains, including `x_canonical`, and updates the metadata used for analysis and semi-naïve evaluation. It then repairs the analysis metadata for the canonical identifier’s parents if the demoted identifier has better analysis results than the canonical one.

```

Definition repair_union x_old x_canonical (analysis_better : bool)
: state instance unit :=
let repair_each a : state instance atom :=
  @!let _ ← db_remove a in
    let a' ← canonicalize a in
    let _ ← update_entry a' in
    ret a' in
@!let old_ps ← pull_parents x_old in
  let ps1 ← list_Mmap repair_each old_ps in
  if analysis_better
  then let canon_ps ← get_parents x_canonical in
    (list_Miter repair_parent_analysis canon_ps)
  else ret tt

Definition repair e :=
match e with
| union_repair old_idx new_idx analysis_better =>
  repair_union old_idx new_idx analysis_better
| analysis_repair i =>
  @!let ps ← get_parents i in
    (list_Miter repair_parent_analysis ps)
end.

Fixpoint rebuild fuel : state instance unit :=
match fuel with
| 0 => Mret tt
| S fuel =>
  @!let todo ← pull_worklist in
    if is_empty todo then ret tt
    else let todo ← list_Mmap canonicalize_worklist_entry todo in
      (list_Miter repair (worklist_dedup todo));
      (rebuild fuel)
end.

```

Figure 4.2: Rebuild implementation

<i>Program</i>	P	$::= \vec{R}$
<i>Rule</i>	R	$::= \Gamma \vdash \Delta$
<i>Conjunction</i>	Γ, Δ	$::= \vec{c}$
<i>Clause</i>	c	$::= a \mid x = y$
<i>Atom</i>	a	$::= f(\vec{x}) \rightarrow y$
<i>Variable</i>	$x, y, \dots$	

Figure 4.3: Logic-program IR

4.3 Logic-Program Intermediate Representation

The rule set passed to `saturate_until` looks a bit different from a standard logic program, but I will defer the description of the executable rule sets until the reasons for my alternate representation become apparent. Instead, I first present the intermediate representation (IR) that compiles to them, with syntax shown in Figure 4.3, that evolved from the core `egglog` language [13]. Unlike core `egglog`, I do not support nested patterns in my IR, instead requiring them to be compiled away into conjunctions of atoms in a similar fashion to Zhang et al. [1]. On the other hand, I support single rules with multiple conclusions, making my rules sequents rather than single inferences. Such rules could be decomposed so that each conclusion belonged to a separate rule, but maintaining multiple conclusions is critical for reducing the number of queries. I also add support for explicit unification clauses $x = y$ as they are simple to support and convenient for my purposes.

Semantically, a program P represents a conjunction of rules run in parallel, where each rule $\Gamma \vdash \Delta$ represents a transformation that, for all satisfying assignments to Γ on a given database, adds the clauses given by Δ . The greatest semantic difference between my IR and core `egglog` is that any variables that appear in a rule’s conclusion Δ but not its query Γ are existentially quantified. Existential variables have been studied in Datalog settings, where they have been shown to lead to nonterminating programs when not restricted [64,65]. However, the existing expressiveness of EqSat makes such a concern moot in my case. When a rule with existential variables matches a satisfying assignment, the e-graph generates a fresh identifier for each existential variable. In the case of a logic program compiled from a term-based rewrite system, many of the output variables of atoms in the conclusion will be existential.

Supporting existentials serves a dual purpose. First, it generalizes the core `egglog` semantics for atoms without output variables. Zhang et al. [13] essentially build existential behavior into their semantics for this special form, since existential output variables are sufficient to represent term-based rewriting. The reason I choose the more general semantics, aside from increasing rule uniformity by eliminating one kind of atom, is that in conjunction with multiconclusion rules, they allow me to express the full conclusion of a rewrite rule with a single logic sequent. On the other hand, core `egglog` would require an additional rule, and thus an additional query, for each clause in the conclusion. Such a language is sufficient for exposition as it is used by Zhang et al. [13], but since my system is verified, my formalism must match my implementation precisely.

4.4 Worst-Case Optimal Join and Set Intersection

Following Zhang et al. [1], I implement a generic join procedure for computing the valid matches for each rule, with the pattern interpreted as a conjunctive query over the database. Generic join is usually presented in an imperative style that is a poor match for a functional language like Gallina. Rather than explain the traditional presentation, I will describe my implementation directly, as I believe that the functional algorithm is easier to understand. I

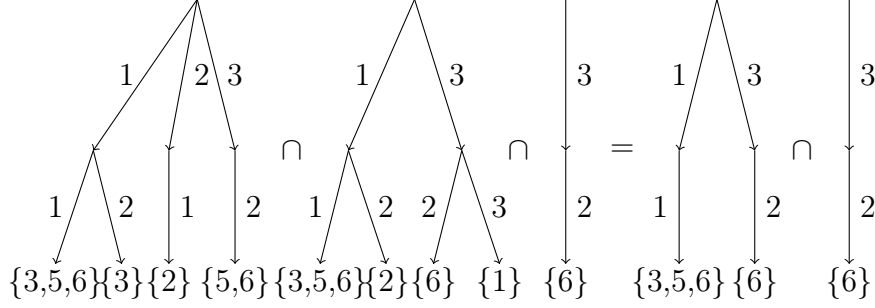


Figure 4.4: Trie intersection example

start with a very high-level, but nonetheless informative, description:

$$\begin{aligned} \text{assignments}_{DB}(Q(\vec{z}) \leftarrow R_1(\vec{x}_1), \dots, R_n(\vec{x}_n)) &\triangleq \bigcap_{i \in [1..n]} \text{trie}_{DB}(R_i(\vec{x}_i), \vec{z}) \\ \text{trie}_{DB}(R(\vec{x}), \vec{z}) &\triangleq \{\vec{v} \mid R(\vec{x})[\vec{v}/\vec{z}] \in DB\} \end{aligned}$$

Consider a conjunctive query $Q(\vec{z}) \leftarrow R_1(\vec{x}_1), \dots, R_n(\vec{x}_n)$, where $Q(\vec{z})$ means that query returns assignments to the variables $\vec{z}$ (and so each $\vec{x}_i$ is required to be a subset of $\vec{z}$). The function assignments_{DB} computes the satisfying assignments from a database DB to the query. It does so by intersecting the sets of satisfying assignments to each individual clause, given by trie_{DB} . Critically, this function takes the query-wide variable order $\vec{z}$ as an argument, allowing each set to return its tuples in the corresponding order, which in turn allows me to intersect them directly without keeping track of the clauses' variables in assignments_{DB} .

The key to generic join is constructing an index structure for each call $\text{trie}_{DB}(R_i(\vec{x}_i), \vec{z})$ in the query. This index represents the set of all assignments $\vec{v}$ for $\vec{z}$ that produce tuples in the table R_i when applied to $\vec{x}_i$. Once I have a set of assignments for each clause, I intersect them to compute the set of assignments that satisfy all clauses. The power of intersection is that if I represent the assignments as tries keyed in the order given by $\vec{z}$, I can compute the intersection of two tries in size proportional to the output, which is bounded above by the size of the smallest input. In fact, this observation generalizes to an n -way intersection, which I implement in a breadth-first manner to obtain the correct asymptotic behavior:

$$\begin{aligned} \bigcap \vec{t} &\triangleq \{x, \vec{x}' \mid \langle x, \vec{t}' \rangle \in \text{collect}(\vec{t}) \wedge \vec{x}' \in \bigcap \vec{t}'\} \\ \text{collect}(t_1, \dots, t_n) &\triangleq \{\langle x, (t'_1, \dots, t'_n) \rangle \mid \forall i \in [1, n]. \langle x, t'_i \rangle \in t_i\} \end{aligned}$$

Intersection operates recursively, calling *collect* at each level of the tries in the list and then operating on the lists of tries that make up the values of *collect*'s output. At each step, the keys of the map returned by *collect* are the set of valid assignments to the next variable in $\vec{z}$. To understand why this breadth-first style is critical for performance, consider the three-way intersection in Figure 4.4. If I intersect the two tries on the left first, as shown in the figure, I must follow the 1 branch of each and recursively compute the intersection. I then immediately discard this subtree when I intersect with the rightmost trie, since its root does not have a branch for the element 1. On the other hand, my breadth-first intersection never processes the subtrees of the 1 branch because the first call to *collect* sees that the third trie is missing the branch and so excludes it.

To build each trie, I iterate over the tuples in the table R_i ; compute the assignment, if any, to the variables $\vec{z}$ that produces that tuple; and add that assignment to the index. A tuple may fail to match a clause, getting removed from contention before I even begin the intersection process, when some variable appears more than once in $\vec{x}_i$. On the other hand, the downside of materializing the set of assignments is that, typically, many assignments satisfy a single clause because for each variable that does not appear in the clause, there is a satisfying assignment for every possible value. For example, consider the clause $\text{edge}(x, y)$ in the query $Q(x, y, z) \leftarrow \text{edge}(x, y), \text{edge}(y, z)$, where in the database, I have the table $\text{edge} \triangleq \{(0, 1), (1, 1)\}$. Since each assignment consists of a triple with one value per variable, I might have satisfying assignments like $(0, 1, 0)$, $(0, 1, 1)$, or $(0, 1, 2)$.

Technically, I could enumerate all possible values of z based on the finite pool of values in the database, but this approach is clearly infeasible on typical tables. Instead, I make a small modification to my tries so that they can compactly represent the infinite sets implied by the definition of trie_{DB} above. For each trie I maintain a list of flags indicating which variables it constrains. I then construct tries containing values only for the flagged variables. For example, the trie for $\text{edge}(x, y)$ from the earlier query would have pairs $(0, 1)$ and $(1, 1)$ as keys and the flag list $\text{true}, \text{true}, \text{false}$, corresponding to the variable list x, y, z . I interpret this trie as representing the infinite set $\{(v_x, v_y, v_z) \mid (v_x, v_y) \in \{(0, 1), (1, 1)\}\}$.

While this extension solves the problem of a compact trie representation, I must account for it during intersection. To intersect two tries where the first trie constrains the first variable and the second does not, I can intersect the second trie with each child of the first trie. Generalizing to the n -ary case, I adapt intersection to take a list of flag lists $\vec{F}$ along with the tries $\vec{t}$:

$$\begin{aligned} \bigcap(\vec{t}, \vec{F}) &\triangleq \{x, \vec{x}' \mid \langle x, \vec{t}', \vec{F}' \rangle \in \text{collect}(\vec{t}) \wedge \vec{x}' \in \bigcap(\vec{t}', \vec{F}')\} \\ \text{collect}(t_1, \dots, t_n) &\triangleq \left\{ \langle x, (t'_1, \dots, t'_n), (F'_1, \dots, F'_n) \rangle \mid \forall i \in [1, n]. \begin{array}{l} (\langle x, t'_i \rangle \in t_i \wedge F_i = \text{true}, F'_i) \\ (t'_i = t_i \wedge F_i = \text{false}, F'_i) \end{array} \right\} \end{aligned}$$

Although managing skipped variables adds complexity to my intersection algorithm, the result is functional code that efficiently computes the results of conjunctive queries. As this intersection operation is the most performance-critical component of my tool, the actual implementation¹ also includes a number of more fine-grained optimizations, but its essential process follows what I describe here.

One benefit of materializing my assignment sets is that I can share them between queries. Unlike in a traditional database implementation, where queries are assumed to be dynamic, my logic program is determined once and then executed many times. Additionally, since the rules can be thought of as evaluating in parallel, they can soundly share indices. Two rules $Q(\vec{x}) \leftarrow R_1(\vec{x}_1), \dots, R_n(\vec{x}_n)$ and $H(\vec{y}) \leftarrow P_1(\vec{y}_1), \dots, P_n(\vec{y}_n)$ can share a trie T for clauses $R_i(\vec{x}_i)$ and $P_j(\vec{y}_j)$ if $R_i = P_j$ and the variables in $\vec{x}_i$ have the same order in $\vec{x}$ as the variables in $\vec{y}_j$ do in $\vec{y}$. In theory, a logic program could necessitate a factorial number of indices per table, since for a table of arity n , there are $n!$ unique tries up to the above sharing criterion. However, in practice I have seen far fewer, indicating that this sharing is quite effective.

¹This algorithm is implemented for binary-tree-based maps and positive-number keys in the module `Utils.PosListMap` of the source code as `pt_spaced_intersect`. The lack of generality for this specific code is due to the limitations of the Rocq positivity checker.

I also take the opportunity to implement semi-naïve evaluation as part of my index-building and intersection planning. When I build an index for a relation, I actually build three: the full trie T_i^{full} , a trie T_i^{old} that only contains facts from prior epochs, and a trie T_i^{new} containing only those atoms that have been updated since the last iteration. For each query $Q(\vec{x}) \leftarrow R_1(\vec{x}_1), \dots, R_n(\vec{x}_n)$, I then compute n intersections, where we compute the i th intersection as follows:

$$T_1^{old} \cap \dots \cap T_{i-1}^{old} \cap T_i^{new} \cap T_{i+1}^{full} \cap \dots \cap T_n^{full}$$

This tripartite pattern guarantees both that every match is justified by at least one new tuple and that the n intersections do not replicate each others' work. Furthermore, since the size of the output of intersection is bounded by the size of each input, once I have built my tries, query time is bounded not by the size of the whole database but by the smaller size of the set of fresh updates.

4.5 Shared Queries

My queries can share indices due to the structure of the executable rule sets that `saturate_until` runs, shown in Figure 4.5. A rule set is a triple $\langle Q, \vec{R}, \vec{C} \rangle$ made of three components: a mapping Q from function symbols and indices to access patterns, a list $\vec{R}$ of compiled rules, and a separate list $\vec{C}$ of constant rules, rules with empty queries. A constant rule corresponds to a conjunction of clauses Δ , but I split it into three components: the list of (existential) variables $\vec{x}$, so that I do not have to compute them during iteration; the list of atom clauses $\vec{a}$; and the list of unification clauses $\vec{e}$.

The more interesting components are the compiled rules $\vec{R}$ and the shared query template map Q . For each function symbol f , Q contains the set of all access patterns used by f -clauses in the whole program. An access pattern for a function symbol f of arity n is an $n + 1$ tuple of natural-number indices, each indicating the level of the trie that represents its position in f -atoms. Numbers may appear more than once, but an access pattern must use every natural from 0 to the largest one that appears in the pattern. For example, the access pattern for the query clause $f(x, y) \rightarrow x$ would be $(0, 1) \rightarrow 0$ if the query used the variable order x, y, z and $(1, 0) \rightarrow 1$ if the query used the variable order z, y, x .

The benefit of taking an indirection through these access patterns is that a given pattern corresponds precisely to the class of indices for a function that have the same underlying trie but may skip different variables from the query. To emphasize this point with more examples, the access pattern $(0, 1) \rightarrow 0$ would also represent $f(x, y) \rightarrow x$ under the variable order z, x, y , and additionally would represent the atoms $f(z, y) \rightarrow z$ and $f(z, x) \rightarrow z$.

This ability for even clauses in the same query to share tries allows me to reduce the number of index structures I build significantly. For example, consider the logic rule that declares a relation *edge* transitive: $edge(x, z) \leftarrow edge(x, y), edge(y, z)$. Choosing the variable order x, y, z , since both clauses share the same access pattern, I only have to construct one index to support this query. My intersection algorithm would then intersect two versions of it together: one that declares its contents as x and y , skipping z ; and one that declares its contents as y and z , skipping x .

<i>Program</i>	P	$::= \langle Q, \vec{R}, \vec{C} \rangle$
<i>Query Clause Map</i>	Q	$: S \times I \xrightarrow{fin} (\vec{\mathbb{N}} \times \mathbb{N})$
<i>Rule</i>	R	$::= \langle \vec{y}, \vec{p}, \vec{x}, \vec{a}, \vec{e} \rangle$
<i>Constant Rule</i>	C	$::= \langle \vec{x}, \vec{a}, \vec{e} \rangle$
<i>Unification Clause</i>	e	$::= x = y$
<i>Atom</i>	a	$::= f(\vec{x}) \rightarrow y$
<i>Variable</i>	$x, y, \dots$	

Figure 4.5: EqSat executable format

I then store my rules with nonempty queries as 5-tuples $\langle \vec{y}, \vec{p}, \vec{x}, \vec{a}, \vec{e} \rangle$. $\vec{y}$ lists the (universally quantified) variables appearing in the rule’s query, specifying their order. I represent the clauses of the query as query pointers p , each of which has three components: the function symbol f , an identifier i , and a list of variables $\vec{z}$. During query computation, I use f and i to look up the index generated by the access pattern stored in Q with key (f, i) . I ensure that the order of the variables in $\vec{z}$ matches their order in $\vec{y}$, which allows me to see the natural numbers in the associated access pattern as indices into $\vec{z}$. The last three components serve the same purpose as in a constant rule, dictating what atoms and unifications to write on a valid assignment to the query.

I compile the more standard logic programs written in the language of Figure 4.3 to my specific representation before performing EqSat, which allows me to establish the full sharing plan once before execution rather than having to determine it on-the-fly. My compiled representation does not support unification clauses in the query. However, this convention does not reduce expressivity since on the query side of a sequent, every unification can be replaced by nonlinear pattern variables, which I do in my rule-optimization pipeline.

4.6 Rule Optimization

Given that my rule sets start as Pyrosome inference rules, compile to logical sequents, and then further transform into executable rules, efficient compilation forms a key part of my development. For example, by reducing the number of clauses that a query contains, I can eliminate redundant computation from the intersection procedure; and by converting unification clauses in a query into nonlinear patterns, I can even reduce the size of the generated indices themselves at runtime.

I take advantage of my existing e-graph machinery to optimize my logic programs as I generate them from Pyrosome. Every Pyrosome rule concludes with either the well-formedness of some term or an equation. I focus here on equational rules, although both can be compiled and run by my system. Such a rule contains a context c , which documents the types of all pattern variables in the rule, as well as any side conditions; a left-hand term e_1 ; and a right-hand term e_2 . From this rule, I will generate a left-to-right rewrite. Since the rules are symmetric, it suffices to consider one direction since I can always reverse the rule before compiling to obtain the other direction.

From c , e_1 , and e_2 , I aim to produce a sequent $\Gamma \vdash \Delta$ in my IR such that matching

Γ is equivalent to matching e_1 with values satisfying c , and accumulating the facts in Δ is equivalent to adding the resulting instantiation of e_2 to the e-graph and unifying their identifiers. To do so, I exploit a correspondence between databases and queries. If I treat equivalence as a custom, distinguished relation, my e-graphs are essentially a specialized form of database, and the left-hand sides of my sequents are conjunctive queries. One way to interpret the semantics of a query Γ is that the query itself is a database where the variables make up the data in the tables. Then the results of evaluating Γ on a database D are given by the set of assignments that map Γ to D .

The key for my use is in the other direction, since I can also recast databases as queries by interpreting their values as variables. The upshot is that to generate Γ (and respectively Δ), it suffices to make an e-graph that contains c and e_1 (and respectively e_2 , unioning its identifier with that of e_1). Since adding a term to an e-graph is one of its basic operations, I can do so quite easily.

The power of this approach is that when the components of my rule pass through the e-graph stage, I can use the same computations I already implemented for running EqSat to optimize my queries, specifically rebuilding and the natural deduplication of shared subterms. The simple translation given by Zhang et al. [1] will produce redundant clauses whenever a subterm appears more than once that must be deduplicated. Furthermore, because of the generation of fresh output variables, this deduplication has to unify the output variables as it traverses the set of clauses, or only the very leaves of the shared subterms will be unified. The hash-cons-like behavior of an e-graph solves these problems. While a standard hash-cons would handle the majority of optimizations I want, rebuilding handles some more complex cases.

4.6.1 Sort Pattern Optimization

In order to translate Pyrosome rules to e-graph rules generically, I had to include not only the left-hand side of a rewrite but also clauses based on the context c that constrain the sorts of the query variables. These clauses ensure that any matching assignment has the right signature, which is critical to proving the correctness of e-matching. However, the additional clauses have significant implications for e-graph performance. The clauses themselves are not so critical to e-matching performance as the complexity of worst-case optimal join is bounded by the size of the output anyway, but they enforce a harsh requirement: in order for a rule to fire, the e-graph must contain all of the leaf sorts of the pattern it matches.

However, some languages do not naturally produce all of the requisite sorts through rewriting. My first fix for this gap was to add the sort-former rules to the saturation loop. This solution works in the theoretical sense and indeed facilitated a few previously blocked proofs, but it had disastrous performance implications. For example, the sort of substitutions eagerly populated the database with the cross product of contexts, quickly leading to a blowup in the size of the e-graph.

The proper fix is to justify removing the sort clauses from the query. To explain why this justification is challenging, consider the language in Figure 4.6 and the pattern $h\ x$, where x is a variable to be filled. Can the assignment $[x \mapsto b]$ satisfy this query? It depends on the context. If the full rewrite that we want to compile is (the left-to-right interpretation of) $x : B \vdash h\ x = x : B$, then the assignment successfully matches the context's signature.

$$\frac{}{A \text{ sort}} \quad \frac{}{B \text{ sort}} \quad \frac{}{b : B} \quad \frac{x : B}{h \ x : B} \quad \frac{x : A}{A = B \text{ sort}}$$

Figure 4.6: Language requiring sort clauses

However, if the rule is $x : A \vdash h \ x = x : B$, then we have a problem. This rule is well-formed because when A is inhabited, it is equivalent to B , so $h \ x$ is well-formed. However, when trying to type the substitution $[x \mapsto b]$, we do not have an instance of A available, so we cannot assign it the context's typing, and it is not a valid instantiation of this rule.

I exclude complex cases like this one by defining a conservative criterion that identifies rules where the sort clauses are safe to exclude. As a simple example, this problem disappears across the board for languages with syntactic sort equality. This criterion is quite conservative, though, specifically excluding all polymorphic and linear languages, so I do better. The example above actually serves as a guide: problems only occur when there is a syntactic difference between the sorts in the context and the expected sorts of the variables in the pattern term, for example the $x : A$ in the context and the expected sort B of the argument to h , because that difference brings conversions into play.

The syntactically expected sorts of a term are trivially computable, so I statically check each variable that appears in the pattern, and if it appears at a sort syntactically equal to its context sort, I am free to drop its context-sort clauses. Under this criterion, I amend my query-generation procedure: rather than instantiating the query e-graph with all of c , I need only include the context variables that do not satisfy this condition. Following this procedure, I eliminate sort clauses for the vast majority of the variables in our rules. Most of the rest do not appear in patterns at all, which means that these represent side conditions, and neither their existence nor well-formedness is implied by the matching of e_1 . As a result of simplifying queries, I do not need the kludge of sort-former saturation.

4.7 Analysis and Extraction

In the domain of e-graphs, *extraction* deals with materializing a suitable algebraic term that represents one node of the e-graph. Often an e-graph implies that many such terms are provably equivalent, and we want to choose among the equivalence class in a principled way. For instance, we may want to choose the smallest term, or we may want to choose one predicted to run most quickly in the cost semantics of the programming language it belongs to. Though a simple approach is naive enumeration of *all* terms in the equivalence class, at which point each can be measured and the best-performing term selected, such enumeration can easily become the main bottleneck of an end-to-end reasoning flow, since e-graphs represent large equivalence classes so compactly.

While checking term equivalence does not require extraction, I make use of extraction in a variety of ways, including to implement e-graph-based type inference. Given the complex nature of e-graphs, it can also be difficult to debug negative results without it. More generally, extraction serves a key role in many potential future uses of my tool, such as

```

def optimal_nodes(g):
    nodes = {}
    for f, args, entry in g.db:
        if entry.analysis <= g.analysis[entry.ident]:
            nodes[entry.ident] = (f, args)
    return nodes

def extract(g, ident):
    nodes = optimal_nodes(g)
    memo = {}
    def extract_loop(ident):
        if memo.contains(ident): return memo[ident]
        (f, args) = nodes[ident]
        children = args.map(extract_loop)
        memo[ident] = (f, children)
        return (f, children)
    return extract_loop(g.find(ident))

```

Figure 4.7: Extraction Pseudocode

term simplification or even for turning EqSat into a stage of compilation in its own right. Unfortunately, optimal extraction is known to be NP-hard [20], which could significantly bottleneck interactive operations if implemented as-is, and additionally is hard to approximate [66]. Despite the small size of the graphs I tested with, my initial, naïve implementation failed to terminate in any reasonable length of time. Recent work has developed procedures that become efficient on sufficiently tree-like graphs [66], which covers many practical situations, but that work does not handle extraction in the presence of cycles, a critical necessity for my use cases. Goharshady, Lam, and Parreaux [66] note that their algorithm can be extended simply to handle cycles but that the complexity of the resulting procedure is impractical, negating its purpose. Additionally, their implementation depends on tree-decomposition libraries that I currently lack in Rocq.

Instead, I implement a simple, linear-time procedure that computes optimal extraction for a generalization of additive cost functions, following the approach of Willsey et al. [21]. Despite the apparent contradiction, Willsey et al. [21] do not disprove the complexity of extraction but instead find significantly preferable results from shifting a portion of computation from extraction to the running e-graph during saturation.

My implementation of extraction, like theirs, uses e-graph analysis to record the optimal costs of different e-classes. At extraction time, given an identifier, I can query for its optimal cost and then greedily select the first element of its e-class with that cost. To ensure that each e-class is only ever visited once, and therefore that the procedure is at most linear in the size of the e-graph, I memoize it. Memoization is necessary because when the optimal term has repeated subterms, it can be much larger than the size of the e-graph.

Figure 4.7 shows pseudocode for my extraction algorithm, which is quite simple once I assume the analysis system tracks minimum cost. It consists of two components: first I

traverse the e-graph tables once to gather an optimal node for each e-class. A linear scan suffices since I can simply compare each node’s recorded cost to the known optimal cost of its e-class. Once I compute the mapping from identifiers to their optimal nodes, I find the canonical identifier for my desired term and compute that term by recursively looking up the next node and computing its children. I memoize the loop by building a map from inputs to precomputed outputs as it evaluates and short-circuiting whenever I encounter an identifier in the map.

Zhao, Subotić, and Scholz [67] also implemented a similar system for the purposes of debugging Datalog. They augmented the Soufflé datalog engine with the capability to query the running engine for the minimum-height proof tree representing the provenance of a particular fact. They observed that by recording for each atom the height of its minimal proof tree, they could greedily construct such a tree by, at each level, choosing the first rule where all subtrees have lesser minimum height. In large-scale studies, they encountered an average of a $1.31\times$ runtime overhead on their logic programs [67] in exchange for fast extraction, which I believe to be quite acceptable in the context of EqSat.

4.8 Optimizing Pyrosome Equivalence Goals

As my primary use for e-graphs is in proving equivalence between pairs of Pyrosome terms, I augment `saturate_until` with some specialized machinery to support goals of this shape. There are three techniques I use to improve performance on these goals: rule scheduling, eager reduction, and opportunistic congruence.

Rule Scheduling In order for my algorithm to be a complete recognizer for equivalences in the input language, it needs to be able to run equations in both directions. This need is not just hypothetical: In the Pyrosome case studies, the η -laws must be run in reverse to expand the appropriate terms. However, running rules in reverse generates significant blowup in the size of the e-graph. To mitigate this blowup, I schedule 10 left-to-right runs of the language rules before each right-to-left run, following the broad convention that the simpler term is likely to be on the right. I could doubtless use more advanced techniques, but this heuristic served as an effective starting point.

Rule scheduling does have a subtle interaction with semi-naïve evaluation. Semi-naïve evaluation operates on the premise that the rules are constant across epochs. When the rules change between batches, unmodified semi-naïve evaluation is no longer complete. For example, in my default 10-to-1 schedule, a fact learned on iteration 5 of the left-to-right rules may need to be available as a new fact for the right-to-left rules. In order to handle this case, I generalize my evaluation to index all tuples within a window of epochs rather than just the most recent one. I compute the size of this window based on the static schedule so that the frontier contains every epoch through the last time the current rule set was run. As a result, repeated iterations of the same rule set correctly see only the most recent epoch, while a switch between rule sets sees every tuple written from the last time those rules ran onward.

Eager Reduction E-graphs suffer from a well-known challenge when manipulating terms that can reduce for many steps: Because the e-graph maintains every known term, such

reduction can also significantly clutter the e-graph in ways that compound with running expansive rules like most right-to-left rules. Following the hypothesis that most equivalences between terms factor fairly well through normal forms, I eagerly check for either of our goal terms to decrease in size, and when they do, I immediately extract both terms and restart with a fresh e-graph. This heuristic might seem wasteful, but extraction is quite cheap compared to running queries against a large e-graph.

Opportunistic Congruence In addition to my eager reduction strategy, I observe that I can also reduce the size of each e-graph by appealing to congruence. For example, should I try to prove $(a, b) = (c, d)$, it suffices to prove $a = c$ and $b = d$. This decomposed search will be faster than operating on the combined term because the pair constructors never appear in the e-graph, and each subterm can be processed independently. However, it is important not to be too eager with congruence. For example, if I want to equate two function applications $f\ v = f'\ v'$, it is absolutely not safe to appeal to congruence since f and f' , even if they are equal, may not be injective.

The choice of congruence rules to apply does not affect the soundness of the procedure, so I do not incur any proof obligations in connection to this feature, but incorrect congruences defeat the completeness of EqSat. My current approach is to generate a list of promising congruences from each language definition automatically. I do so by constructing an e-graph seeded with terms that appear in the various equations of the language and saturating it up to a small bound. I inspect this e-graph to identify non-injective operations and keep every congruence for which we do not find a counterexample. This heuristic overapproximates the true set of complete congruences but seems to work well in practice so far.

4.9 Compiler Correctness by Computation

The computational nature of my e-graph machinery lets me do something quite pleasant for all of the fully automatic compiler-correctness proofs. The only undecidable parts of compiler correctness in Pyrosome are the leaf goals, which are either well-typedness goals or equations. Furthermore, the only undecidable part of well-typedness is conversion between sorts, which means that were I to have an oracle for equivalence, type and equivalence preservation would be decidable.

I implement a verified computation that recognizes correct compilers by using my e-graph as that oracle. Although it is not guaranteed to succeed within its allotted fuel, if it does, my soundness theorems tell us that the equation must hold. Based on this observation, I implement a computation that lifts our e-graph reasoning to the whole compiler and verify its correctness. An automatic proof of compiler correctness then becomes an appeal to this lemma combined with a call to `vm_compute`, Rocq’s faster internal evaluator.

4.10 Equality Saturation for Type Inference

In addition to the equational-reasoning queries described in [section 4.1](#), I also use my e-graph implementation to perform language-agnostic type inference in Pyrosome. Type inference

for an arbitrary Pyrosome language is undecidable since GATs can encode extensional type theory and similarly difficult type systems. Type inference is also critically useful for defining languages and compilers as it would be quite unfortunate to write all of the context and type information Pyrosome requires by hand in an equation or compiler case.

Initially, I used a tactic-based unification approach to type inference built by piggybacking on Rocq’s `evvars`. However, this system’s limitations became apparent with the polymorphic and linear languages. The difficulty with these languages lies in the interaction between the implicit arguments that type inference must fill and the non-trivial type and context equations of the two languages. Since equations, as ever, are the main challenge, and I have e-graphs on hand, I use my e-graphs as a type inference engine.

The critical point here is as an egglog-style fixpoint reasoning system, my e-graph engine runs a superset of Datalog. Writing type-inference algorithms as logic programs is a well-trodden topic. In order to adapt saturation to type inference, I just need the right extraction weights and the right additional rules.

When inferring the implicit arguments in a Pyrosome term using e-graphs, I first insert a fresh variable into the position of every implicit argument. Conceptually these are unification variables, although they receive no special treatment during saturation. Instead, I choose a weighting function that assigns each unification variable infinite weight. This means that so long as saturation manages to unify every unification variable with a concrete term, extraction is guaranteed to produce a fully elaborated term.

Setting up unification variables in this way is sufficient to infer types for a few programs, but with just this much work, most programs will fail to unify their implicit arguments. To understand the problem, consider the program $(\lambda x. x) @ v$ where $v : \text{bool}$ and $@$ represents application. Pyrosome requires type annotations at every level internally, so when we insert unification variables, it ends up looking something like $(\lambda(x : ?A). x) @^{?B \rightarrow ?C} v$, where $?A$, $?B$, and $?C$ are distinct unification variables.

Because of the typing judgments for functions and function application, our e-graph will contain the equation $?A \rightarrow ?A = \text{bool} \rightarrow ?C$. However, it gives us very little to do from there. To make progress on this example, type-inference algorithms take advantage of the injectivity of the function type, which lets us unify the left- and right-hand sides of the equal function types, leaving us with $\text{bool} = ?A = ?C$.

Logically, injectivity for function types is $(A \rightarrow B) = (C \rightarrow D) \implies A = C \wedge B = D$. Unfortunately, we cannot write such a rule in Pyrosome. While it is a provable property of STLC, System F, and indeed most languages with functions, it is not monotone under language extension. Put another way, injectivity is a property of the initial model, but not every model. As a result, Pyrosome deliberately lacks the vocabulary for such implications of equalities. Instead, we write these rules directly in extended Datalog, which has no such limitation. Since they follow a rigid form, they are easy to encode. As a Datalog program, function injectivity has the following form, where we use Fn as the type former for the function type to avoid ambiguity with the syntax for atoms: $Fn(A, B) \rightarrow F, Fn(C, D) \rightarrow F \vdash A = C, B = D$.

While e-graphs would probably not be suitable as a type checker for large programs, in my context, they simply have to type the small terms used in the definitions of Pyrosome compilers. The small scale of my terms keeps saturation relatively tractable in practice. This type-inference engine is also not complete, but it seems to perform well on the current case studies. Unlike most other uses of e-graphs in Pyrosome, this one does not depend on

Component	Previously Manual	Currently Manual
STLC CPS	0	0
STLC CC	2	0
Fix CPS	0	0
Fix CC	1	0
Ref CPS	5	0
Ref CC	5	0
Poly CPS	1	1
Ex CPS	0	0
Ex CC	3	2
Imp	12	2
LinearCPS	8	1

Figure 4.8: Manual proof goals

their correctness. Instead, the extracted, annotated program gets separately checked for well-formedness by a verified procedure that can skip all of the extra type-inference rules. This divide is what allows me to add injectivity rules so easily. So long as they are admissible, they strictly grow the set of typeable terms without affecting the proof obligations. Unsound rules have the potential to cause type inference for a valid term to fail, but successful inferences are still validated by later steps.

4.11 Equality Saturation and Pyrosome’s Case Studies

I evaluate my verified e-graphs by applying them to the Pyrosome case studies. To recap, the main case study is a two-pass compiler that implements a continuation-passing-style transformation followed by closure conversion, closely inspired by the first half of the compiler from Morrisett et al. [28]. This compiler starts as a compiler for STLC and, as per the Pyrosome paradigm, incrementally adds recursive functions, mutable references, and polymorphism. For the full set of case studies, Pyrosome generates 203 proof obligations, of which 166 were solved automatically by more primitive tactics such as naïve left-to-right reduction. About half of Pyrosome’s case-study goals are equations, including almost every manually proven goal.

I replaced my prior automation for solving equations with my e-graph-based implementation, removing any formerly manual proof cases whose goals equality saturation solved. In Figure 4.8, I characterize these proof goals and find that our automation solves 197 goals, with 6 remaining. Among the goals that were not previously solved, the most common root causes were side conditions, which the prior automation could not handle; η laws, the use of which does not typically correspond to reduction; and nontrivial type equalities, which the reduction tactic also had few mechanisms for handling.

The most common culprits among the remaining manual goals are those that require expanding rewrites to solve. Multiple of the remaining goals’ proof scripts boil down to choosing the correct intermediate term, which can then reduce quickly on one side and expose sufficient structure on the other. Without explicitly introducing these terms in the proof, the

e-graph has to saturate with expensive rules that grow the term set without bound until it hits the correct intermediate term.

4.12 Implementation Limitations and Future Work

Analysis My analysis system is essentially only half of `egg`'s, specifically lacking the ability to influence the application of rewrites. This limitation renders it unsuitable for many of the purposes presented by Willsey et al. [21] but, in exchange, greatly simplifies my proofs since no theorems depend on the results of analysis. Additionally, were I to implement a full analysis system, I would hew closer to the template laid out by Zhang et al. [13] where analyses are simply additional relations and logic-program rules. To support this feature, it would suffice to generalize the data contained by relations. Currently, all values in my database are identifiers. This choice was sufficient to represent all Pyrosome rules and simplifies many facets of our implementation and proofs. However, it should be straightforward, if a bit of work, to generalize values to allow for custom semi-lattices.

Rule Scheduling I currently follow the naïve heuristic that left-to-right rules generally make execution simpler and use a constant number of iterations before running a reversed iteration for completeness. There is a rich field of work around selecting and orienting equations that would likely improve the state of affairs. It even seems reasonable to have a procedure that generates fresh equations implied by the existing ones in order to improve performance. I could also reduce the iteration of expansive rules by identifying subsets of rules that normalize and running those to a fixed point.

Chapter 5

Verifying an Egglog Implementation

In this chapter, I discuss my proof that my e-graph implementation is sound with respect to either an arbitrary model or an arbitrary Pyrosome language, depending on the component. This proof is quite extensive, but this chapter focuses on the statements of correctness and some of the critical properties.

5.1 AI and Modern Formal Verification

Before diving into the contents specific to e-graphs, I will start with some commentary on the development process and how it changed over the course of this thesis. During my work on this project, large language models (LLMs) reached a point where they were capable of making significant contributions to verification efforts. The benefit of combining an LLM with a mechanized theorem prover is that I do not have to trust the LLM, only Rocq’s kernel. I made significant use of Claude Code [68] in the completion of this proof, and so I see it as important to document both what that means for my results and what conclusions I can draw about theorem proving as a discipline going forward. I begin discussing my proof effort with this section not because it forms a significant thread across this chapter but so that I can set a common understanding and wrap up the discussion of tools at the start, leaving the rest of the chapter to focus on e-graphs. By default, readers should assume that the definitions, invariants, and theorems I describe in this chapter were designed and written by me.

AI, Development, and Trust My use of LLM assistance does not significantly reduce my confidence in my results. I make this claim because I used it adversarially: I assumed that the LLM results were untrusted and independently verified them before making claims about them. My primary methodology was simple: If the LLM only writes new proofs and proves a theorem that I wrote, then so long as the Rocq kernel approves, I can trust it. I did make some targeted edits to function definitions and theorem statements using LLMs, but when doing so I first established my own expectations for what such a change would produce and then manually checked that the resultant code matched my expectations line-for-line. I ensured control over this process by placing the LLM-driven process in a separate virtual machine that did not have permission to write to my project repository. Instead, to merge its work into my project, I would pull its work, manually inspect the diff between it and my

development’s main branch, and only merge it once satisfied that it had made no definition or top-level theorem changes I did not request and carefully audit and that its code compiled on my machine. I did not review all of the proofs it wrote, or its intermediate lemmas, so it is possible that it took advantage of a bug in the Rocq kernel, which is the only way the LLM could have subverted my results. However, its work broadly follows the path of the proof that I forecasted, so I guess that the likelihood of this is low.

Qualitative Observations and Best Practices Based on my experiences, I have a few observations that suggest development practices that benefit LLM-assisted proof development. These tools are incredibly useful, but software verification is about producing high-assurance results, and high-assurance they are not. As I indicated above, I believe that the most effective way to use them is adversarially, that is in ways such that their results are independently checkable at minimal cost. I spent an appreciable amount of time reviewing the diffs of the AI’s work. Most of this time is avoidable, especially once good definitions and theorem statements are written, so long as one separates definitions and theorem signatures from proofs at the file level.

5.2 Verifying Equality Saturation

Because I have a software stack with multiple layers (Pyrosome at the top, the logic IR in the middle, and the e-graph runtime at the bottom), I also have multiple layers of theorems and statements of correctness in my proof. At the top level, `egraph_equal` has a specification purely in terms of Pyrosome:

Theorem 5.1 (Soundness of `egraph_equal`). *Given well-formed L , c , t , e_1 , and e_2 , if `egraph_equal fuel c e1 e2 t = true`, then $c \vdash_L e_1 = e_2 : t$.*

For callers of `egraph_equal`, this theorem is all they need to know. The actual signature of `egraph_equal` in the development has a few more parameters for performance tuning, for example excluding specific rules, but the theorem quantifies over all of them unconstrainedly, and my tactics default to a standard set of inputs, so I elide them here. All implementation details and even the existence of the IR are completely hidden. On the other hand, I frame the specification of lower-level operations like `saturate_until` and `rebuild` in terms of models of my IR. Datalog-level properties, which make up the majority of my proof development, build on a stack of specification components, which I define step-by-step below.

Models of Logic Programs A standard logic program can be modelled by a carrier set D and a family of relations R_f , with arities given by signatures of the relation symbols f in the logic program. Such a model supports a rule $f_o(\vec{x}_o) \leftarrow f_1(\vec{x}_1), \dots, f_n(\vec{x}_n)$ whenever for all variable assignments γ that map each variable x to a value $d \in D$ such that $R_{f_1}(\gamma(\vec{x}_1)) \wedge \dots \wedge R_{f_n}(\gamma(\vec{x}_n))$ holds, we can deduce that $R_{f_o}(\gamma(\vec{x}_o))$ holds. Since, unlike traditional Datalog, I have functions rather than relations, I generalize the clauses to have distinguished outputs following the shape of my atoms $f(\vec{x}) \rightarrow y$. For a given pair $\langle D, R \rangle$ to be a valid model, the output must have a functional dependency in R as well, so given $R_f(\vec{v}, v_1)$ and $R_f(\vec{v}, v_2)$,

we can conclude $v_1 = v_2$. I also generalize this construct to operate over setoids to give me control over equality without resorting to exotic types for D . Thus, a model $\langle D, \sim, R \rangle$ provides a carrier set D , a partial equivalence relation $\sim$ on D , and a family of relations $R : S \times \vec{D} \rightarrow D$ where S is the set of function symbols. I let $\sim$ be a partial equivalence and R be a partial function for mechanization convenience so that I can choose a D that is a superset of the true domain, given by $\{d : D \mid d \sim d\}$, and so that R 's signature does not have to track the function symbols' arities.

The Term Model The most important model I define is the Pyrosome term model. This model is parameterized by a language L , has Pyrosome ASTs as its domain, L -equality as its partial equivalence, and maps $(f, \vec{t})$ to the term $f(\vec{t})$ in most cases. The exception relates to sort management. As a part of my translation from Pyrosome languages, which may have complex or even dependent type systems, to logic programs, I introduce an additional function symbol `sort_of` that takes one argument and returns that argument's sort. The term model's relation holds for this function symbol exactly when its output is a valid sort for its input.

E-Graph Interpretations As a way to reason about the content of the e-graph without also managing its complex internal state, I define an interpretation i in a model m as a map from the allocated identifiers of an e-graph to elements of m 's domain. I say that an interpretation is sound with respect to an e-graph e , written $e \models_m i$, when their domains agree and every pair of identifiers related by the union-find map to equivalent elements in the model and, for every atom $f(\vec{x}) \rightarrow y$ in the database, $R_f(i(\vec{x}), i(y))$ holds. These interpretations serve to separate reasoning about the logical contents of the e-graph from reasoning about its index structures and implementation details.

Specifications for E-Graph Operations Since my EqSat implementation is written using the state monad, I state the correctness properties of `saturate_until` and its subcomponents using some basic common structure. Given a computation c in the state monad, I usually state these properties as relations between the input and the paired output and return value of c . As a notational convenience, I define $c \{R\}$ to mean $\forall s. R \ s \ (c \ s)$ where R is such a relation. I use this notation to provide a bit of structure to the properties of e-graph computations. Putting together the whole stack of specification components, I state the correctness of rebuilding as follows:

Theorem 5.2 (Correctness of rebuilding). *Let m be a model, i be an arbitrary interpretation in m , and n be a natural number. Then*

$$(\text{rebuild } n) \{ \lambda e_0 \ (o, e_1). \text{egraph_ok } e_0 \implies \text{egraph_ok } e_1 \wedge \forall i. e_0 \models_m i \iff e_1 \models_m i \}$$

Rebuilding exhibits a common pattern that I take advantage of in the proof: The input and output e-graphs have the same set of interpretations. Many of the lower-level e-graph operations have the same property, which allows me largely to treat them as no-ops in terms of their effects on the soundness of the e-graph. While it may be surprising to think of

rebuilding, the signature operation of an e-graph, as having no semantic effect, it makes sense. It may be helpful to think of sound e-graph operations as discovering and recording preexisting facts about the model rather than adding new facts. Rebuilding never adds new terms to the e-graph, so the set of identifiers is invariant. The only facts rebuilding can discover are unions that record the equivalences between identifiers. If a union of two identifiers is sound, it corresponds to discovering that the domain elements corresponding to those identifiers in every input interpretation must have been equivalent the whole time.

I glue the basic e-graph operations together with helper lemmas that correspond to the basic monadic operations, most critically this one for monadic bind:

Theorem 5.3 (Correctness of monadic bind). *Let $c : \text{state } S \ A$ be a stateful computation, let $f : A \rightarrow \text{state } S \ B$, let $R_1 : S \rightarrow A \times S \rightarrow \text{Prop}$ be a relation such that $c \{R_1\}$, and let $R_2 : S \rightarrow B \times S \rightarrow \text{Prop}$ be a relation such that $\forall s_0 a. (f \ a) \{\lambda s_1 r. R_1 \ s_0 \ (a, s_1) \rightarrow R_2 \ s_0 \ r\}$. Then $(\text{let } x \leftarrow c \text{ in } f \ x) \{R_2\}$.*

I also prove corresponding lemmas for common monadic operations like monadic map and fold over lists. This library of lemmas proved quite effective both in deconstructing composite computations and in guiding me toward the right specifications.

The correctness of `saturate_until` is more complex because it is highly parameterized, but roughly speaking, the correctness of `saturate_until` states that for any valid interpretation of the input e-graph, the output e-graph has an interpretation that extends it:

Theorem 5.4 (Correctness of `saturate_until`). *Let m be a model, i be an arbitrary interpretation in m , n be a natural number, rs be a rule set, and $p : \text{state instance bool}$. If rs is sound with respect to m and we have*

$$p \{\lambda e_0 (o, e_1). \text{egraph_ok } e_0 \implies \text{egraph_ok } e_1 \wedge \forall i. e_0 \models_m i \implies e_1 \models_m i\}$$

then:

$$\text{saturate_until } rs \ p \ n \\ \{\lambda e_0 (o, e_1). \text{egraph_ok } e_0 \implies \text{egraph_ok } e_1 \wedge \forall i_0. e_0 \models_m i_0 \implies \exists i_1 \supseteq i_0. e_1 \models_m i_1\}$$

While such a theorem would not be sufficient to prove completeness of the procedure, it gives us everything we need for soundness. Using the term model and `egraph_equal` as an example, this theorem guarantees that the identifiers of the input terms in the starting e-graph point to the same terms in the resulting e-graph. Then, if those identifiers are related by the union-find, their model elements must be equivalent, which gives us the conclusion of [Theorem 5.1](#).

Although [Theorem 5.4](#) is more complex than [Theorem 5.2](#), it also represents a weaker property because it allows for the existential extension of the interpretation. For [Theorem 5.4](#), this weakening is necessary because `saturate_until` does not satisfy `rebuild`'s stronger property since it adds new terms to the e-graph. I initially proved a similar theorem about `rebuild` instead of [Theorem 5.2](#), which would have been sufficient for proving [Theorem 5.4](#). However, I also use `rebuild` in rule compilation, where interpretations correspond to satisfying assignments of a query. In order to verify that process, I must apply [Theorem 5.2](#) contravariantly, which necessitates the bidirectional implication.

5.2.1 Verifying Rule Translation

Verifying rule translation requires engaging with the full gamut of specifications and abstraction levels present in my system. At a high level, translation promises that, given a set of Pyrosome rules from a language l , it produces an executable rule set such that for any e-graph that is sound with respect to the term model and an interpretation i that is well-formed with respect to the term model, for any rule in the set whose query matches on i , every clause of its output is sound with respect to the term model and therefore safe to add to i .

I decompose this property into smaller steps. For pattern terms e , I prove the following theorem about the function that generates an e-graph from them:

Theorem 5.5 (Correctness of `add_open_term`). *Let i be an arbitrary interpretation in the term model for a language l , e be a term such that it has sort t under context c in l , s be a closing c -substitution, and r be a mapping from the pattern variables of e to the domain of i such that for each typed pattern variable $x : t_x \in c$, r maps x to some z such that $i(z) = s(x)$. Then:*

$$\text{add_open_term } r \ e \left\{ \lambda e_0 \ (o, e_1). \begin{array}{l} \text{egraph_ok } e_0 \wedge e_0 \models_{\text{TERM}_l} i \implies \\ \text{egraph_ok } e_1 \wedge \exists i' \supseteq i. e_1 \models_{\text{TERM}_l} i' \wedge i'(o) = e[s] \end{array} \right\}$$

I optimize the pattern in this representation by applying `rebuild`, which preserves the output interpretation precisely and so appears transparent from a soundness perspective. I then build a sequent $\Gamma \vdash \Delta$ in my IR from two e-graphs: one for the assumptions of the Pyrosome rule and one for its conclusion. This sequent must satisfy the property that for all interpretations i' such that a renaming exists from the identifiers in Γ to the identifiers of i' that satisfies the clauses of Γ , the clauses of Δ are also satisfiable. There is a correspondence between the context variables of the rule under compilation and the identifiers of i' . Specifically, when we add a pattern into the e-graph, every generated identifier occupies a return position of a function tuple except for the identifiers that map to the context variables. This invariant ensures that the interpretations of the e-graph are in bijection with the well-formed substitutions at that context since the interpretation's values at the identifiers corresponding to variables determine the rest of the interpretation up to equivalence through functional dependencies. As a result, I can equivalently state my goal as requiring that, for every substitution that matches the left-hand side of a rewrite, the result must be equal to the right-hand side with the same substitution applied. This property follows directly from the validity of the rule under compilation.

5.2.2 E-Graph Invariants

Although I hide them inside the interface to my EqSat procedure, the correctness of my algorithms depends on an array of invariants that apply to the various components of my e-graphs. I begin with the union-find data structure, which consists of three computational components: the primary identifier map that relates each identifier to its parent, a rank map that bounds the depth of the tree rooted at each identifier, and the next fresh identifier to be used for allocation. My implementation also tracks the maximum rank as a separate field

so that my `find` function can be structurally recursive. Formally, I maintain the following invariants and prove that `find`, `union`, and `alloc` preserve them:

1. The parent map is a forest with a list l of canonical elements as its roots. I define and reason about this property using separation logic, although it is wholly contained by the interface to union-find.
2. Every identifier in the parent map has a rank.
3. Every noncanonical identifier has rank strictly less than its direct parent.
4. The maximum rank is an upper bound on the ranks.
5. The next identifier is strictly greater than all current identifiers.

I define `egraph_ok` to hold on an e-graph when the above properties hold on its union-find for some l and it additionally satisfies three more invariants:

1. Every pair of identifiers in the worklist is related by the union-find.
2. Every parent atom in the map from identifiers to their parents is in the database, up to union-find equivalence.
3. For every atom in the database, all of its identifiers are in the union-find.

The empty e-graph satisfies these properties and I prove that every operation I define preserves them. The worklist and parent-map invariants required careful statements to preserve. In particular, I initially stated both as properties relative to the interpretation, rather than the e-graph's concrete collection of facts. However, this weaker requirement created challenges in my proofs that easily resolved when I directly related the metadata of the e-graph to its tables. At the same time, I only maintain the parent-atom invariant up to union-find equivalence. While it might seem that every parent atom should literally appear in the database, it is not so during rebuilding. Since I do not eagerly update the parent atoms, they may temporarily contain noncanonical identifiers until we finish rebuilding congruence closure.

Chapter 6

Conclusions

A Retrospective If I were to pick a single lesson that I hope may carry forward from my work in this thesis, it is that in order to achieve modular and extensible design, we must state our specifications constructively, in terms of what behavior we require, rather than exclusionarily, in terms of what behavior we forbid. Furthermore, a language or software system defined in this way is more likely to be compatible with automated reasoning and metatheoretic facilities innately.

While I also take pride in my specific technical results regarding verified compilers and e-graphs, I believe their greatest value is as a substantial existential witness to the success of extensible reasoning under this discipline. From a mathematical point of view, I am in a sense just arguing for initial models, but it is an argument I believe worth making, given how common exclusion-based semantics like contextual equivalence are in both the historic and modern literature. To be clear, such semantics have important roles to play in our field, especially relating to concerns around security. However, almost by design, contextual equivalence is anti-extensional: It relates every pair of programs that cannot be told apart in the present language fragment, leaving no room for future behavioral extensions to differentiate them.

Looking Forward This work deliberately avoids secure compilation and sticks to functional correctness for precisely this reason. I do not mean to claim that security and extensibility are exclusive. With the right security properties expressed internally within a language design, I believe it may be possible to have both. However, common metatheoretic tools like contextual equivalence and equivalence reflection achieve security precisely by codifying a closed-world assumption: that there shall be no more observations in the future than exist today, exactly the antithesis of extensibility. I see positive specification of safety properties as a key challenge in the future of language-based security. This is not a new idea: the community has already developed techniques like information flow type systems for encoding specific security properties. However, their applications to broad properties like secure compilation rather than specific properties like secrete leakage has yet to be fully explored.

Regarding the more specific, technical contributions of this thesis, I hope to continue positioning Pyrosome as a testbed and toolbox for programming-language and compilers metatheory. The scaffolding I have built is substantial and is just reaching what I consider general (research) usability as I conclude this portion of the work. Far too much effort in our

field goes into repeated reasoning over pools of routine language features. Each paper presents its one new concept but must build a whole calculus around it in order to demonstrate its functionality, and little of that work gets effectively reused. Extensible metatheory, whether Pyrosome itself or any future successor, promises to advance the velocity of research by enabling proper reuse and recombination of existing components.

References

- [1] Y. Zhang, Y. R. Wang, M. Willsey, and Z. Tatlock. “Relational e-matching.” *Proc. ACM Program. Lang.*, **6**(POPL), Jan. 2022. DOI: [10.1145/3498696](https://doi.org/10.1145/3498696). URL: <https://doi.org/10.1145/3498696>.
- [2] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. “CompCert – A Formally Verified Optimizing Compiler.” In: *ERTS 2016*. SEE, 2016.
- [3] A. Chlipala. “A Verified Compiler for an Impure Functional Language.” In: *POPL’10: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Madrid, Spain, Jan. 2010. URL: <http://adam.chlipala.net/papers/ImpurePOPL10/>.
- [4] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. “CakeML: A Verified Implementation of ML.” In: *POPL 2014*. POPL ’14. San Diego, California, USA: ACM, 2014.
- [5] X. Yang, Y. Chen, E. Eide, and J. Regehr. “Finding and Understanding Bugs in C Compilers.” In: *PLDI 2011*. San Jose, California, USA: ACM, 2011.
- [6] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala. “Integration verification across software and hardware for a simple embedded system.” In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 604–619.
- [7] J. McCarthy and J. Painter. “Correctness of a compiler for arithmetic expressions.” *Mathematical aspects of computer science*, **19**, 1967, pp. 33–41.
- [8] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. “Lightweight Verification of Separate Compilation.” In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 178–190. ISBN: 9781450335492. DOI: [10.1145/2837614.2837642](https://doi.org/10.1145/2837614.2837642). URL: <https://doi.org/10.1145/2837614.2837642>.
- [9] D. Patterson and A. Ahmed. “The next 700 compiler correctness theorems (functional pearl).” *Proceedings of the ACM on Programming Languages*, **3**(ICFP), 2019, pp. 1–29.
- [10] J. Koenig and Z. Shao. “CompCertO: compiling certified open C components.” In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 1095–1109. ISBN: 9781450383912. DOI: [10.1145/3453483.3454097](https://doi.org/10.1145/3453483.3454097). URL: <https://doi.org/10.1145/3453483.3454097>.

- [11] D. Patterson, N. Mushtak, A. Wagner, and A. Ahmed. “Semantic Soundness for Language Interoperability.” In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 609–624. ISBN: 9781450392655. DOI: [10.1145/3519939.3523703](https://doi.org/10.1145/3519939.3523703). URL: <https://doi.org/10.1145/3519939.3523703>.
- [12] J. Perconti and A. Ahmed. “Verifying an open compiler using multi-language semantics.” In: *European Symposium on Programming Languages and Systems*. Springer. 2014, pp. 128–148.
- [13] Y. Zhang, Y. R. Wang, O. Flatt, D. Cao, P. Zucker, E. Rosenthal, Z. Tatlock, and M. Willsey. “Better together: Unifying Datalog and equality saturation.” *Proceedings of the ACM on Programming Languages*, **7**(PLDI), 2023, pp. 468–492.
- [14] D. Jamner, G. Kammer, R. Nag, and A. Chlipala. “Pyrosome: Verified Compilation for Modular Metatheory.” **9**(OOPSLA2), Oct. 2025. DOI: [10.1145/3763052](https://doi.org/10.1145/3763052). URL: <https://doi.org/10.1145/3763052>.
- [15] J. Cartmell. “Generalised algebraic theories and contextual categories.” *Annals of Pure and Applied Logic*, **32**, 1986, pp. 209–243. ISSN: 0168-0072. DOI: [https://doi.org/10.1016/0168-0072\(86\)90053-9](https://doi.org/10.1016/0168-0072(86)90053-9). URL: <https://www.sciencedirect.com/science/article/pii/0168007286900539>.
- [16] A. Kaposi, A. Kovács, and T. Altenkirch. “Constructing quotient inductive-inductive types.” *Proc. ACM Program. Lang.*, **3**(POPL), Jan. 2019. DOI: [10.1145/3290315](https://doi.org/10.1145/3290315). URL: <https://doi.org/10.1145/3290315>.
- [17] G. D. Plotkin. “LCF considered as a programming language.” *Theoretical computer science*, **5**(3), 1977, pp. 223–255.
- [18] G. Nelson and D. C. Oppen. “Fast Decision Procedures Based on Congruence Closure.” *J. ACM*, **27**(2), Apr. 1980, pp. 356–364. ISSN: 0004-5411. DOI: [10.1145/322186.322198](https://doi.org/10.1145/322186.322198). URL: <https://doi.org/10.1145/322186.322198>.
- [19] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. “Equality saturation: a new approach to optimization.” In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2009, pp. 264–276.
- [20] M. Stepp, R. Tate, and S. Lerner. “Equality-based translation validator for LLVM.” In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 737–742.
- [21] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. “Egg: Fast and Extensible Equality Saturation.” *Proc. ACM Program. Lang.*, **5**(POPL), Jan. 2021. DOI: [10.1145/3434304](https://doi.org/10.1145/3434304). URL: <https://doi.org/10.1145/3434304>.
- [22] T. L. P. Community. *Grind*. 2025. URL: https://leanprover-community.github.io/mathlib4_docs/Init/Grind/Tactics.html.
- [23] S. Gruetter and T. Bourgeat. *E-Graphs to Help Writing Coq Proofs*. Talk presented at New England Systems Verification Day, Cambridge, Massachusetts, USA. 2022.

- [24] T. Köhler, A. Goens, S. Bhat, T. Grosser, P. Trinder, and M. Steuwer. “Guided Equality Saturation.” *Proc. ACM Program. Lang.*, **8**(POPL), Jan. 2024. DOI: [10.1145/3632900](https://doi.org/10.1145/3632900). URL: <https://doi.org/10.1145/3632900>.
- [25] M. Rossel. “An Equality Saturation Tactic for Lean.” Master’s thesis. Technische Universität Dresden, July 2024. URL: https://cfaed.tu-dresden.de/files/Images/people/chair-cc/theses/2407_Rossel_MA.pdf.
- [26] M. E. Maietti. “A minimalist two-level foundation for constructive mathematics.” *Annals of pure and applied logic*, **160**(3), 2009, pp. 319–354.
- [27] A. Kovács. “Staged compilation with two-level type theory.” *Proc. ACM Program. Lang.*, **6**(ICFP), Aug. 2022. DOI: [10.1145/3547641](https://doi.org/10.1145/3547641). URL: <https://doi.org/10.1145/3547641>.
- [28] G. Morrisett, D. Walker, K. Crary, and N. Glew. “From System F to typed assembly language.” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **21**(3), 1999, pp. 527–568.
- [29] M. Felleisen. “On the expressive power of programming languages.” *Science of computer programming*, **17**(1-3), 1991, pp. 35–75.
- [30] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. “Meta-theory à La Carte.” In: *POPL (2013)*. Rome, Italy: ACM, 2013, pp. 207–218. ISBN: 978-1-4503-1832-7.
- [31] B. Delaware, S. Keuchel, T. Schrijvers, and B. C. Oliveira. “Modular Monadic Meta-theory.” In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 319–330. ISBN: 978-1-4503-2326-0. DOI: [10.1145/2500365.2500587](https://doi.org/10.1145/2500365.2500587). URL: <http://doi.acm.org/10.1145/2500365.2500587>.
- [32] W. Swierstra. “Data Types à La Carte.” *J. Funct. Program.*, **18**(4), July 2008, pp. 423–436. ISSN: 0956-7968. DOI: [10.1017/S0956796808006758](https://doi.org/10.1017/S0956796808006758). URL: <http://dx.doi.org/10.1017/S0956796808006758>.
- [33] G. Allais, R. Atkey, J. Chapman, C. McBride, and J. McKinna. “A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs.” *Proc. ACM Program. Lang.*, **2**(ICFP), July 2018, 90:1–90:30. ISSN: 2475-1421. DOI: [10.1145/3236785](https://doi.org/10.1145/3236785). URL: <http://doi.acm.org/10.1145/3236785>.
- [34] R. Harper, F. Honsell, and G. Plotkin. “A framework for defining logics.” *Journal of the ACM (JACM)*, **40**(1), 1993, pp. 143–184.
- [35] G. Roşu and T. F. Şerbănuţă. “An overview of the K semantic framework.” *The Journal of Logic and Algebraic Programming*, **79**(6), 2010, pp. 397–434.
- [36] X. Chen and G. Roşu. *A General Approach to Define Binders Using Matching Logic*. Tech. rep. <http://hdl.handle.net/2142/106608>. University of Illinois at Urbana-Champaign, June 2020.
- [37] X. Chen, Z. Lin, M.-T. Trinh, and G. Roşu. “Towards a Trustworthy Semantics-Based Language Framework via Proof Generation.” In: *Proceedings of the 33rd International Conference on Computer-Aided Verification*. ACM, July 2021.

- [38] O. Lynch, K. Brown, J. Fairbanks, and E. Patterson. “GATlab: Modeling and Programming with Generalized Algebraic Theories.” *Electronic Notes in Theoretical Informatics and Computer Science*, **Volume 4-Proceedings of the Fortieth Conference on the Mathematical Foundations of Programming Semantics**, Dec. 2024. ISSN: 2969-2431. DOI: [10.46298/entics.14666](https://doi.org/10.46298/entics.14666). URL: <http://dx.doi.org/10.46298/entics.14666>.
- [39] J. Matthews and R. B. Findler. “Operational Semantics for Multi-language Programs.” *ACM Trans. Program. Lang. Syst.*, **31**(3), Apr. 2009, 12:1–12:44. ISSN: 0164-0925. DOI: [10.1145/1498926.1498930](https://doi.org/10.1145/1498926.1498930). URL: <http://doi.acm.org/10.1145/1498926.1498930>.
- [40] A. Ahmed. “Verified Compilers for a Multi-Language World.” In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 15–31. ISBN: 978-3-939897-80-4. DOI: [10.4230/LIPIcs.SNAPL.2015.15](https://doi.org/10.4230/LIPIcs.SNAPL.2015.15). URL: <http://drops.dagstuhl.de/opus/volltexte/2015/5013>.
- [41] W. J. Bowman. “Compilation as Multi-Language Semantics.” In: *Workshop on Principles of Secure Compilation*. New York, NY, USA: ACM, 2021.
- [42] C. Beck, I. Yoon, H. Chen, Y. Zakowski, and S. Zdancewic. “A two-phase infinite/finite low-level memory model: Reconciling integer–pointer casts, finite space, and undef at the llvm ir level of abstraction.” *Proceedings of the ACM on Programming Languages*, **8**(ICFP), 2024, pp. 789–817.
- [43] M. Sammler, S. Spies, Y. Song, E. D’Osualdo, R. Krebbers, D. Garg, and D. Dreyer. “DimSum: A Decentralized Approach to Multi-Language Semantics and Verification.” *Proc. ACM Program. Lang.*, **7**(POPL), Jan. 2023. DOI: [10.1145/3571220](https://doi.org/10.1145/3571220). URL: <https://doi.org/10.1145/3571220>.
- [44] M. Fiore and D. Szamozvancev. “Formal metatheory of second-order abstract syntax.” *Proc. ACM Program. Lang.*, **6**(POPL), Jan. 2022. DOI: [10.1145/3498715](https://doi.org/10.1145/3498715). URL: <https://doi.org/10.1145/3498715>.
- [45] J. C. Blanchette, L. Gheri, A. Popescu, and D. Traytel. “Bindings As Bounded Natural Functors.” *Proc. ACM Program. Lang.*, **3**(POPL), Jan. 2019, 22:1–22:34. ISSN: 2475-1421. DOI: [10.1145/3290335](https://doi.org/10.1145/3290335). URL: <http://doi.acm.org/10.1145/3290335>.
- [46] P. Dybjer. “Internal type theory.” In: *International Workshop on Types for Proofs and Programs*. Springer. 1995, pp. 120–134.
- [47] P. Levy, J. Power, and H. Thielecke. “Modelling environments in call-by-value programming languages.” *Information and computation*, **185**(2), 2003, pp. 182–210.
- [48] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. “Explicit substitutions.” In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 31–46.
- [49] S. Schäfer, T. Tebbi, and G. Smolka. “Autosubst: Reasoning with de Bruijn terms and parallel substitutions.” In: *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6*. Springer. 2015, pp. 359–374.

- [50] K. Stark, S. Schäfer, and J. Kaiser. “Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions.” In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2019, pp. 166–180.
- [51] A. Ahmed and M. Blume. “An equivalence-preserving CPS translation via multi-language semantics.” In: *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. 2011, pp. 431–444.
- [52] D. Devriese, M. Patrignani, and F. Piessens. “Fully-abstract compilation by approximate back-translation.” In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 164–177.
- [53] M. S. New, W. J. Bowman, and A. Ahmed. “Fully abstract compilation via universal embedding.” In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, pp. 103–116.
- [54] P. Mates, J. Perconti, and A. Ahmed. “Under control: Compositionally correct closure conversion with mutable state.” In: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. 2019, pp. 1–15.
- [55] L. Zhang, Y. Wang, J. Wu, J. Koenig, and Z. Shao. “Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules.” *Proc. ACM Program. Lang.*, **8**(POPL), Jan. 2024. DOI: [10.1145/3632914](https://doi.org/10.1145/3632914). URL: <https://doi.org/10.1145/3632914>.
- [56] P. Wang, S. Cuellar, and A. Chlipala. “Compiler Verification Meets Cross-Language Linking via Data Abstraction.” In: *OOPSLA’14: Proceedings of the 2014 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Portland, OR, USA, Oct. 2014. URL: <http://adam.chlipala.net/papers/CitoOOPSLA14/>.
- [57] Y. Minamide, G. Morrisett, and R. Harper. “Typed closure conversion.” In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 1996, pp. 271–283.
- [58] N. Benton. “Simple relational correctness proofs for static analyses and program transformations.” *ACM SIGPLAN Notices*, **39**(1), 2004, pp. 14–25.
- [59] J. M. Li and A. W. Appel. “Deriving Efficient Program Transformations from Rewrite Rules.” *Proc. ACM Program. Lang.*, **5**(ICFP), Aug. 2021. DOI: [10.1145/3473579](https://doi.org/10.1145/3473579). URL: <https://doi.org/10.1145/3473579>.
- [60] G. Plotkin and J. Power. “Notions of computation determine monads.” In: *International Conference on Foundations of Software Science and Computation Structures*. Springer. 2002, pp. 342–356.
- [61] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, Feb. 1993. ISBN: 9780262291453. DOI: [10.7551/mitpress/3054.001.0001](https://doi.org/10.7551/mitpress/3054.001.0001). URL: <https://doi.org/10.7551/mitpress/3054.001.0001>.
- [62] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. “Interaction trees: representing recursive and impure programs in Coq.” *Proceedings of the ACM on Programming Languages*, **4**(POPL), 2019, pp. 1–32.

- [63] J. Sterling. “Algebraic Type Theory and Universe Hierarchies.” <https://arxiv.org/pdf/1902.08848>. 2019.
- [64] N. Leone, M. Manna, G. Terracina, and P. Veltri. “Efficiently Computable Datalog[∃] programs.” In: *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*. KR’12. Rome, Italy: AAAI Press, 2012, pp. 13–23. ISBN: 9781577355601.
- [65] A. Cali, G. Gottlob, and T. Lukasiewicz. “A general Datalog-based framework for tractable query answering over ontologies.” In: *Proceedings of the Twenty-Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’09. Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 77–86. ISBN: 9781605585536. DOI: [10.1145/1559795.1559809](https://doi.org/10.1145/1559795.1559809). URL: <https://doi.org/10.1145/1559795.1559809>.
- [66] A. K. Goharshady, C. K. Lam, and L. Parreaux. “Fast and Optimal Extraction for Sparse Equality Graphs.” *Proc. ACM Program. Lang.*, **8**(OOPSLA2), Oct. 2024. DOI: [10.1145/3689801](https://doi.org/10.1145/3689801). URL: <https://doi.org/10.1145/3689801>.
- [67] D. Zhao, P. Subotić, and B. Scholz. “Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy.” *ACM Trans. Program. Lang. Syst.*, **42**(2), Apr. 2020. ISSN: 0164-0925. DOI: [10.1145/3379446](https://doi.org/10.1145/3379446). URL: <https://doi.org/10.1145/3379446>.
- [68] Anthropic. *Claude Code*. 2025. URL: <https://docs.anthropic.com/en/docs/claude-code> (visited on 06/24/2026).