A Framework for Modular, Extensible, Equivalence-Preserving Compilation

by

Dustin Jamner

B.S., Northeastern University (2020)

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 11, 2022

Certified by .....
Adam Chlipala
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

A Framework for Modular, Extensible, Equivalence-Preserving Compilation

by

Dustin Jamner

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

I present Pyrosome[1], a generic framework for the verification of extensible, compositional compilers in Coq. Current techniques for proving compiler correctness are generally tied to the specific structures of the languages and compilers that they support. This limits the extent to which these systems can be extended and composed. In Pyrosome, verified compilers are fully extensible, meaning that to add a new feature to a language simply requires defining and verifying the compilation of that single feature, reusing the old correctness theorem to cover all other cases. This is made possible by an inductive formulation of equivalence preservation that supports the addition of new rules to the source language, target language, and compiler.

Pyrosome defines a formal, deeply embedded notion of programming languages with semantics given by sorted equational theories, so all compiler-correctness proofs boil down to type-checking and equational reasoning. My work supports vertical composition of any compilers expressed in Pyrosome in addition to feature extension. Since my design requires compilers to support open programs, my correctness guarantees support linking with any target code of the appropriate type. As a case study, I present a multipass compiler from STLC through CPS translation and closure conversion, and show that natural numbers, the unit type, recursive functions, and a global heap can be added to this compiler while reusing the original proofs.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Electrical Engineering and Computer Science

[1]Pyrosomes 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.

Contents

List of Figures

Chapter 1

Introduction

Compiler verification is a laborious process compared to unverified implementation. Nevertheless, numerous researchers have taken on the challenge [Leroy et al., 2016; Chlipala, 2010; Kumar et al., 2014] due to the significant benefits of formal verification [Yang et al., 2011]. 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 [Erbsen et al., 2021], and even in unverified software stacks, the volume of code that depends on a given compiler often warrants the work of providing strong guarantees.

Despite the clear desirability of formally verified compilers, the overwhelming majority of compilers in use today are unverified. Examining the current state of compiler verification, existing efforts suffer 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 [Kang et al., 2016; Patterson and Ahmed, 2019]. When verified compilers support linking, it often is in the form of separate compilation, meaning that they can only link programs that were compiled from the same source language and sometimes only by the same compiler. While a useful result, separate compilation does not permit, for example, linking with specialized low-level code written directly

$$
\begin{array}{rcl}
\lfloor A \to B \rfloor & \triangleq & \neg(\lfloor A \rfloor \times \neg \lfloor B \rfloor) \\
\lfloor x \rfloor & \triangleq & k\ x \\
\lfloor \lambda(x : A).\ e \rfloor & \triangleq & k\ \lambda(x : \lfloor A \rfloor, k' : \neg \lfloor B \rfloor).\ \lfloor e \rfloor [k'/k], \\
 & & \text{where } e : B \\
\lfloor e\ e' \rfloor & \triangleq & \text{bind } x := \lfloor e \rfloor;\ \text{bind } y := \lfloor e' \rfloor;\ x\ \langle y, k \rangle \\
\text{bind } x := e;\ e' & \triangleq & e[\lambda x.\ e'/k]
\end{array}
$$

Figure 1-1: CPS translation for STLC

in the target language. Furthermore, the state of the art in verification is ill-equipped to remedy this. Correct compilers tend to be bespoke artifacts and, like most verification efforts, can be difficult to extend. Perconti and Ahmed [2014] 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.

Pyrosome is a language-agnostic framework for compiler verification that addresses these concerns via a formal system of language and compiler modularity and extension. Pyrosome provides a straightforward method of proving type and equivalence preservation that is monotone under such extension. This allows us to add new rules to a language, augment its compiler, and reverify it by only considering those new rules. Similarly, the work required to update a language feature in a verified compiler is limited to the set of dependent features.

As an informal example, consider the compiler $\lfloor - \rfloor$ in Figure 1-1. It transforms

the simply typed lambda calculus (STLC) into continuation-passing style (CPS), with the convention that the continuation is bound to variable k. This compiler may be proven type-preserving without making use of Pyrosome by induction on the input, with one proof case per syntactic form in the source language. To cover the variable case, given $\Gamma \vdash x : A$ in STLC, it suffices to derive that $k\ x$ is well-typed under the target context $\lfloor \Gamma \rfloor, k : \neg \lfloor A \rfloor$. However, equivalence preservation is typically more complicated. Existing work on compiler verification often utilizes contextual equivalence, which relates each pair of terms that cannot be distinguished by any context in a given language. Unfortunately, contextual equivalence is difficult to reason about [Patrignani et al., 2019]. In particular, it plays poorly with language extension since it can vary based on the inclusion of language features unrelated to the terms under consideration. Language semantics in Pyrosome are defined via equational theories, eliminating the dependency on arbitrary contexts. In practice, many rules in these theories look similar to small-step operational semantics. For example, to characterize the equivalence of terms in STLC requires two rules in the language specification, beta and eta equivalence:

$$(\lambda(x : A).\ e)\ v = e[v/x] \tag{1.1}$$

$$(\lambda(x : A).\ f\ x) = f \tag{1.2}$$

Pyrosome turns these rules into an equivalence relation by taking their reflexive, transitive, symmetric, congruence closure. I prove a generic theorem that allows users of Pyrosome to prove equivalence preservation for a given compiler analogously to an inductive proof of type preservation, with one proof case per equation in the source-language specification. Consider Equation 1.1, the rule for beta equivalence. When verifying the CPS translation, the proof obligation that Pyrosome generates corresponding to this rule requires us to show $\lfloor (\lambda(x : A).\ e)\ v \rfloor = \lfloor e[v/x] \rfloor$ in the target language's equivalence relation. By evaluating the compiler and rewriting the

$$
\begin{array}{rcl}
\lfloor n \rfloor & \triangleq & k\ n, \text{ where } n \in \mathbb{N} \\
\lfloor \text{set } e := e' \rfloor & \triangleq & \text{bind } x := \lfloor e \rfloor;\ \text{bind } y := \lfloor e' \rfloor; \\
 & & \text{set } x := y \text{ in } k\ \langle\rangle \\
\lfloor \text{get } e \rfloor & \triangleq & \text{bind } x := \lfloor e \rfloor;\ \text{get } x \text{ as } y \text{ in } k\ y
\end{array}
$$

Figure 1-2: CPS mutation extension

term via target-language rules, we can do so roughly as follows:

$$
\begin{array}{rl}
 & \lfloor (\lambda(x : A).\ e)\ v \rfloor \\
= & \text{bind } x := \lfloor \lambda(x : A).\ e \rfloor;\ \text{bind } y := \lfloor v \rfloor;\ x\ \langle y, k \rangle \\
= & \text{bind } x := k\ \lambda(x : \lfloor A \rfloor, k' : \neg \lfloor B \rfloor).\ \lfloor e \rfloor [k'/k]; \\
 & \text{bind } y := \lfloor v \rfloor;\ x\ \langle y, k \rangle \\
= & (\lambda x.\ \text{bind } y := \lfloor v \rfloor;\ x\ \langle y, k \rangle)\ \lambda(x : \lfloor A \rfloor, k' : \neg \lfloor B \rfloor). \\
 & \lfloor e \rfloor [k'/k] \\
= & \text{bind } y := \lfloor v \rfloor;\ (\lambda(x : \lfloor A \rfloor, k' : \neg \lfloor B \rfloor).\ \lfloor e \rfloor [k'/k])\ \langle y, k \rangle \\
= & (\lambda(x : \lfloor A \rfloor, k' : \neg \lfloor B \rfloor).\ \lfloor e \rfloor [k'/k])\ \langle \lfloor v \rfloor, k \rangle \\
= & \lfloor e \rfloor [\lfloor v \rfloor / x] \\
= & \lfloor e[v/x] \rfloor
\end{array}
$$

To extend a language and compiler with new rules, it suffices to prove the appropriate conditions for the new rules just as we did for the base language. For example, we might want to add a global store to our compiler from Figure 1-1. In Figure 1-2, I sketch an extension to the compiler to add features for interacting with a global heap to the source language by compiling them to CPS-compatible versions of the same operations in the target. We can extend our original compiler, which did not feature mutation in the source or target, with this addition by first embedding its output into an extended CPS calculus with effectful operations, utilizing monotonicity to guarantee well-typedness and equivalence preservation, then proving the appropriate fact for each new case of the compiler and each additional source axiom.

1.1 Contributions

Concretely, I present the following novel contributions, with code in the attached supplement:

- I mechanize an existing formalism for programming-language specification [Sterling, 2019] in Coq.
- I define an inductive characterization of well-formedness for compilers between languages encoded in Pyrosome and prove that it implies both type and equivalence preservation.
- I characterize extensions of these specifications and compilers and prove theorems enabling separate reasoning about and combination of such extensions.
- I present as a case study a multipass compiler starting with STLC that transforms it into CPS and then performs closure conversion.
- I extend this compiler with natural numbers, the unit type, recursive functions, and a global heap using the original correctness theorem to justify the correctness of the existing components.

Chapter 2

Concepts of Pyrosome

I begin by outlining the high-level functionality of Pyrosome via additional informal examples before I go into the technical details of its implementation. Figure 2-1 shows the grammars for STLC and various extensions. I use ellipses in each of the extensions to indicate that it inherits all of the rules of the grammar it is extending. For example, if we were to consider the language formed by extending STLC with naturals, then we would have two listed value forms: naturals n and lambdas $\lambda(x : A).\ e$. I will use $+$ to indicate language extension, writing STLC + Naturals to indicate the extension of STLC with the natural-numbers feature.

Pyrosome's greatest benefit is laying out a modular procedure for proving type and equivalence preservation for compilers based on results about their constituent components. I define a predicate $\mathit{Preserving}(L_t, \mathit{cmp}, L_s)$ for this purpose, described in more detail in section 3.3, which implies that *cmp* is a type- and equivalence-preserving compiler from L_s to L_t. This predicate can be proven modularly. In fact, proving $\mathit{Preserving}(L_t, \mathit{cmp}, L_s)$ requires exactly one goal per syntactic form and one per equation of L_s. As mentioned in chapter 1, this amounts to proving well-typedness of each case of the compiler, just as if we were inducting on its input, and proving that each rule $t_1 = t_2$ in the source language corresponds to a derivable equality $\lfloor t_1 \rfloor = \lfloor t_2 \rfloor$ in the target where I use $\lfloor - \rfloor$ to indicate compilation by *cmp*.

If $L_s = L_1 + L_2$, for example STLC + Recursion, we can prove the necessary cases for features from STLC and from Recursion separately. The Recursion cases

STLC:

$$\begin{array}{rcl} A, B & ::= & \ldots \mid A \to B \\ v, v' & ::= & \ldots \mid \lambda(x : A).\ e \\ e, e' & ::= & \ldots \mid e\ e' \end{array}$$

Naturals:

$$\begin{array}{rcl} A, B & ::= & \ldots \mid \text{nat} \\ v, v' & ::= & \ldots \mid n \\ e, e' & ::= & \ldots \mid e + e \mid e * e \\ n & \in & \mathbb{N} \end{array}$$

Recursion:

$$\begin{array}{rcl} A, B & ::= & \ldots \\ v, v' & ::= & \ldots \mid \text{fix } f(x : A) := e \\ e, e' & ::= & \ldots \end{array}$$

State:

$$\begin{array}{rcl} A, B & ::= & \ldots \\ v, v' & ::= & \ldots \\ e, e' & ::= & \ldots \mid \text{set } e := e' \mid \text{get } e \end{array}$$

Figure 2-1: Language grammars

will depend on the definition of the STLC compiler since Recursion is an extension of STLC, but the STLC cases do not have to be aware of Recursion or other potential extensions, and all cases can be proven independently. To extend the CPS compiler to support State, we can add the cases from Figure 1-2 and prove the additional obligations in the same manner. While the syntax of State is simple, its equations depend on a few features not described here: the addition of heaps H as finite maps from $\mathbb{N}$ to $\mathbb{N}$, configurations $\langle H, e\rangle$ that pair computations with heaps, and evaluation contexts E, including a plug operation $E[e]$. I leave the details of these constructs to section 4.2. For now, we consider the proof obligation corresponding to the source-language rule $\langle H, E[\text{get } n]\rangle = \langle H, E[H(n)]\rangle$, which describes evaluation of get expressions, to demonstrate how they come into play. To prove that this equation is preserved in the target, we must show that the compilation of the left-hand term is equivalent to the compilation of the right-hand term, which we have by the following

equational reasoning:

$$
\begin{aligned}
& \lfloor \langle H, E[\text{get } n] \rangle \rfloor \\
= \; & \langle H, \lfloor E[\text{get } n] \rfloor \rangle \\
= \; & \langle H, \text{bind } x := \lfloor \text{get } n \rfloor;\ \lfloor E \rfloor[x] \rangle \\
= \; & \langle H, \text{bind } x := (\text{bind } x := \lfloor n \rfloor; \text{ get } x \text{ as } y \text{ in } k\ y); \lfloor E \rfloor[x] \rangle \\
= \; & \langle H, \text{bind } x := \text{get } n \text{ as } y \text{ in } k\ y;\ \lfloor E \rfloor[x] \rangle \\
= \; & \langle H, \text{get } n \text{ as } y \text{ in } (\lambda(x : \text{nat}).\ \lfloor E \rfloor[x])\ y \rangle \\
= \; & \langle H, \text{get } n \text{ as } y \text{ in } \lfloor E \rfloor[y] \rangle \\
= \; & \langle H, \lfloor E \rfloor[H(n)] \rangle \\
= \; & \langle H, \text{bind } x := H(n);\ \lfloor E \rfloor[x] \rangle \\
= \; & \lfloor \langle H, E[H(n)] \rangle \rfloor
\end{aligned}
$$

Close observation of this proof and the earlier one for beta reduction will show that they amount to computing normal forms for both the left- and right-hand sides, if we think of our equations as operational rules going from left to right. In this case, we have $\langle H, \lfloor E \rfloor[H(n)] \rangle$, third from the bottom, as a common normal form. While this pattern does not apply to all cases, I automate it in the mechanization to solve the vast majority of goals in my case study.

One benefit of working with equivalence preservation is that compiler correctness is transitive. For example, we could choose to include fix $f(x : A) := e$ in our base calculus instead of $\lambda(x : A).\ e$ and implement $\lambda(x : A).\ e$ as an extension. We would then define our base CPS transform on Recursion instead of STLC, as shown in Figure 2-2. Later, we could add nonrecursive functions to the source language by means of a preliminary pass defined by adding the rule $\lfloor \lambda(x : A).\ e \rfloor = \text{fix } f(x : \lfloor A \rfloor) := \lfloor e \rfloor$ to a compiler that otherwise leaves syntactic forms intact. We can prove separately that this compiler and the CPS compiler for Recursion preserve equivalences and then compose the results to verify the 2-stage compiler. Should we ever replace our CPS compiler with a different phase, we can simply compose the desugaring of lambda with that new pass.

Since our compilers operate on open terms, equivalences preserved by Pyrosome

$$
\begin{array}{rcl}
\lfloor A \to B \rfloor & \triangleq & \neg(\lfloor A \rfloor \times \neg \lfloor B \rfloor) \\
\lfloor x \rfloor & \triangleq & k\ x \\
\lfloor \text{fix } f(x : A) := e \rfloor & \triangleq & k \text{ fix } f(x : \lfloor A \rfloor, k' : \neg \lfloor B \rfloor) := \lfloor e \rfloor [k'/k], \\
 & & \text{where } e : B \\
\lfloor e\ e' \rfloor & \triangleq & \text{bind } x := \lfloor e \rfloor;\ \text{bind } y := \lfloor e' \rfloor;\ x\ \langle y, k \rangle \\
\text{bind } x := e;\ e' & \triangleq & e[\lambda x.\ e'/k]
\end{array}
$$

Figure 2-2: CPS for recursion

compilers support arbitrary linking with target programs. Specifically, if $t_1 = t_2$ in L_s, then for any target substitution γ with the appropriate domain, $\gamma(\lfloor t_1 \rfloor) = \gamma(\lfloor t_2 \rfloor)$ where $\lfloor - \rfloor$ denotes compilation by any type- and equivalence-preserving compiler. Consider the following program:

$$
\begin{array}{rcl}
\mathit{callTwice} & \triangleq & \lambda(f : \text{nat} \to \text{nat}).\ \lambda(x : \text{nat}). \\
 & & (\lambda(\_ : \text{nat}).\ f\ x)\ (f\ 0)
\end{array}
$$

When passed f and x, it first calls f on 0 and then returns $f\ x$. Let $\mathit{natFun} \triangleq \neg(\text{nat} \times \neg\text{nat})$. If we run the compiler from Figure 1-1 on *callTwice*, we get a term of type $\neg(\mathit{natFun} \times \neg \mathit{natFun})$ passed to some continuation k. I use these target functions, which use a halt primitive typed $\neg$nat, to illustrate compiled behavior:

$$
\begin{array}{rcl}
\mathit{haltEarly} & \triangleq & \lambda(x : \text{nat}, k : \neg\text{nat}).\ \text{halt}\ x \\
\mathit{callWithOne} & \triangleq & \lambda(f : \mathit{natFun}).\ f\ \langle 1, \text{halt} \rangle
\end{array}
$$

If we substitute

$$
\begin{array}{rcl}
\mathit{cont} & \triangleq & \lambda(c : \neg(\mathit{natFun} \times \neg \mathit{natFun})). \\
 & & c\ \langle \mathit{haltEarly}, \mathit{callWithOne} \rangle
\end{array}
$$

for k in $\lfloor \mathit{callTwice} \rfloor$, then running the resultant program will call *haltEarly* on 0 first. It will then halt with result 0 and drop the continuation containing *callWithOne*. By the guarantees of Pyrosome, we therefore know that for any term f such that $f = \mathit{callTwice}$ in STLC, $\lfloor f \rfloor [\mathit{cont}/k] = \text{halt } 0$ in the theory of the target language.

This example illustrates the importance of using equational theories to capture a language's intended semantics as opposed to contextual equivalence. In pure STLC, *callTwice* is contextually equivalent to $\lambda(f : \text{nat} \to \text{nat}).\ f$, but compiling the latter with this continuation would produce a program that halts with result 1. Equational theories allow us to specify which equalities we expect to hold for a given calculus, whereas contextual equivalence implicitly includes all equations that are consistent with the observations available in that calculus, whether intentional or incidental. In fact, due to the modular nature of equational theories, all judgments in Pyrosome are monotonic under language extension, so it even supports linking with programs written in any extension of the target language.

Language designers could abuse this modular nature to lay out incompatible sets of rules if they choose, but since each source-language equation generates a compiler-verification obligation, invalid extensions will simply be unimplementable, other than by trivial compilers. To pick a concrete example, consider beta reduction again: $(\lambda(x : A).\ e)\ v = e[v/x]$. Notice that we limit this equation to call-by-value, rather than allowing full beta reduction. This restriction prevents us from relating *callTwice* and $\lambda(f : \text{nat} \to \text{nat}).\ f$ in our theory since we cannot reduce the internal application in *callTwice* before we have evaluated its argument, $f\ 0$. If we instead allowed full beta reduction, these two terms would be equivalent in our source-language semantics. However, we would run into a problem in the process of proving compiler correctness for our CPS transformation since the compiler fixes an order of evaluation due to the order it passes terms their continuations.

Chapter 3

Formalism and Metatheory

3.1 Language Specifications

One of the keys to my approach is a formal mechanism for programming-language extension that solves the well-known expression problem [Wadler, 1998; Krishnamurthi et al., 1998] for the particular goals of compiler correctness. Standard presentations of programming languages using inference rules are informally modular in that it is straightforward to write down another rule in the language, but like functional programs over syntax [Krishnamurthi et al., 1998], formal properties of languages generated by inference rules do not easily generalize upon the addition of new constructs. I use a formal description of programming languages and their extensions to draw on existing ideas in the literature about the expression problem as it relates to functional programs, extensible syntax, and logical statements [Swierstra, 2008; Delaware et al., 2013a,b; Allais et al., 2018].

One goal of Pyrosome is for researchers and developers to be able to specify their languages and compilers as closely as possible to the way they would on paper with inference rules. Generalized algebraic theories (GATs) [Sterling, 2019] represent this style of definition well. Prior work describes various other generic formalizations of what characterizes a programming language [Felleisen, 1991]. However, I find GATs to be well-structured for the purpose of extensibility. For a comparison to some alternatives, see section 5.1. In Pyrosome, program terms are n-ary trees of

syntax, with the available syntactic constructs determined by the object language under consideration. Each term has a sort that serves as a combination of its syntactic class and a well-formedness judgment for that class. Language modules define both terms and sorts, as well as equational axioms, but I will begin by focusing on term definitions. I use the following notation to represent terms, where ellipses represent space-delimited sequences of 0 or more:

$$\begin{array}{lcl} x, c & \in & \mathit{string} \\ \mathit{term} & ::= & \texttt{\#}c\ \mathit{term}... \mid x \mid (\mathit{term}) \end{array}$$

Terms come in two forms: object-language syntax written #c $term...$ where c is the name of the syntactic form and $term...$ are subterms, and metavariables x. One key detail to the design of this framework is the distinction between object-language variables and framework-level metavariables. Programming-languages research typically treats metavariables in language definitions as ranging over concrete terms. GATs make this concept explicit by including metavariables as part of the syntax, allowing us to represent such patterns directly as ASTs. Object-language binding structures are defined separately as a user-defined language module, so they do not affect my handling of metavariables. In the mechanization I use de Bruijn indices for object-language variables. For example, I write the term $\lambda(x : A).\ \langle x, x\rangle$ in the Coq formalization as `#"lambda" "A" (#"pair" #"hd" #"hd")` where `"A"` is a metavariable, `#"hd"` is de Bruijn index 0[1], and all other strings represent the expected object-language syntactic forms. However, I present named variables in the rest of this thesis for readability.

A language specification in Pyrosome is built up as a list of inference rules that each either declare a new syntactic form and the conditions under which it is well-formed or a new equation that relates two terms. As an example, consider the definition of STLC in Figure 3-1. The first three rules define STLC's term syntax and its associated typing judgments, and the last rule declares β-equivalence. We encode these rules in Pyrosome in Figure 3-2. They retain the same structure, with two

[1]The name refers to the head of the environment viewed as a list.

$$\frac{\vdash A\ type \qquad \vdash B\ type}{\vdash A \to B\ type} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A).\ e : A \to B} \qquad \frac{\Gamma, A \vdash e : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda(x : A).\ e)\ v = e[v/x] : B}$$

Figure 3-1: STLC

notable exceptions: First, every metavariable must be given an explicit sort above the line, as a basic tenet of my framework. Second, to maintain a value/expression distinction, we define separate sorts for the two and use an explicit return operator to inject values into the sort of expressions.

We will use the rules in Figure 3-2 to illustrate some important points about using Pyrosome. The first three rules are term rules that declare new syntactic forms and assign them sorts given the sorts of their subterms. In this thesis, I use "term" to refer to constructions in the object-language syntax, which in this case includes not only expressions and values but also types, and I use "sort" to refer to the syntactic classes and their judgment forms that categorize my terms. Thus, the rule declaring `#"->" "A" "B"` to be a type is a term rule that assigns this syntax the sort `#"ty"`. The rules for application and lambda both demonstrate an important feature of Pyrosome, which is the ability to manage implicit subterms. From the perspective of Pyrosome's theory, each metavariable declared above the horizontal line is a subterm that must be included in the AST. This would be infeasible to write by hand, so term rules declare which arguments they expect to be written down by listing them after the name of the new syntactic form below the line. In the case of application, I expect the user to provide `"e"` and `"e'"`, while I assume that the environment and types can be inferred. Pyrosome does not commit to a particular inference strategy, instead leaving inference up to tactics run during proof of term well-formedness. I discuss this in section 4.5. For lambda, I choose to require that the input type `"A"` be written down. This decision makes the standard tradeoff of simplifying inference at the expense of brevity. I could just as easily leave lambda's

```
Definition stlc : lang := {[l/subst
[:| "A" : #"ty", "B": #"ty"
    ------------------------------------------------
    #"->" "A" "B" : #"ty"                          ];

[:| "G" : #"env",
    "A" : #"ty",
    "B" : #"ty",
    "e" : #"exp" "G" (#"->" "A" "B"),
    "e'" : #"exp" "G" "A"
    ------------------------------------------------
    #"app" "e" "e'" : #"exp" "G" "B"               ];

[:| "G" : #"env",
    "A" : #"ty",
    "B" : #"ty",
    "e" : #"exp" (#"ext" "G" "A") "B"
    ------------------------------------------------
    #"lambda" "A" "e" : #"val" "G" (#"->" "A" "B") ];

[:= "G" : #"env",
    "A" : #"ty",
    "B" : #"ty",
    "e" : #"exp" (#"ext" "G" "A") "B",
    "v" : #"val" "G" "A"
    ------------------------------------------------ ("beta")
    #"app" (#"ret" (#"lambda" "A" "e")) (#"ret" "v")
    = #"exp_subst" (#"snoc" #"id" "v") "e"
    : #"exp" "G" "B"                                ]]}.
```

Figure 3-2: Definition of STLC in Pyrosome

$$\frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{get } e : \text{nat}} \qquad \frac{\Gamma \vdash e : \text{nat} \qquad \Gamma \vdash e' : \text{nat}}{\Gamma \vdash \text{set } e := e' : \text{unit}}$$

$$\frac{\Gamma \vdash E : \text{nat} \rightsquigarrow A}{\Gamma \vdash \langle H, E[\text{get } n]\rangle = \langle H, E[H(n)]\rangle : A}$$

$$\frac{\Gamma \vdash E : \text{unit} \rightsquigarrow A}{\Gamma \vdash \langle H, E[\text{set } n := m]\rangle = \langle H[n \mapsto m], E[\langle\rangle]\rangle : A}$$

Figure 3-3: STLC state extension

`"A"` implicit in Figure 3-2, but it would require more powerful inference to instantiate. The last rule in Figure 3-2 declares equivalence between an applied lambda on the left and the substitution of its argument into its body on the right. I defer an explanation of the substitution calculus used here to section 4.3 and use standard notations as in Figure 3-1 in the rest of the thesis.

Now that we have a base calculus in mind, we can discuss extensions to this language. In fact, the rules for STLC already extend our substitution calculus, but we will use a new example to avoid getting sidetracked by the details of substitution. Figure 3-3 gives the rules in traditional notation of an extension that provides a global store both containing and indexed by natural numbers. As mentioned in chapter 2, this extension also depends on heap and evaluation-context extensions, not shown here. Our goal is then to construct a language that contains all of the rules from both Figure 3-1 and Figure 3-3 (as well as the elided rules they depend on) such that we can type-check terms like $\lambda(x : \text{nat}).$ set $x := 42$, a function that takes in a number and sets the cell at that index to 42. Fortunately, since languages in Pyrosome are lists of rules, we construct this language by concatenating the lists.

3.2 Compilers and Correctness

There is a central tension in the area of compiler correctness between expressivity and modularity, and this extends to what compilers I consider in this work. In order

to fully support both linking and compiler extension, I require compilers to operate on terms with arbitrary free metavariables. Furthermore, I want compilation to be invariant under substitution of metavariables, i.e. $\lfloor \gamma(e) \rfloor = \lfloor \gamma \rfloor(\lfloor e \rfloor)$, where $\gamma(e)$ denotes the metavariable substitution γ applied to the term e. I therefore define my compilers via finite maps from source-language sort and term names to target-language sorts and terms. These finite maps are then folded over a source term, looking up each constructor, to compile it. Then, just as we can extend our languages by appending new rules, we can extend our compilers by appending new mappings. This style formalizes patterns that appear in prior work on compositional compiler correctness [Perconti and Ahmed, 2014; New et al., 2016; Mates et al., 2019]. However, it does impose some limitations, especially around optimization. See section 5.3 for a discussion of how I anticipate optimization will fit into Pyrosome.

Recall the CPS transformation outlined in chapter 1, in particular the global-store extension from Figure 1-2. In Pyrosome, this compiler would be represented as a table indexed by the constructors on the left of the $\triangleq$ and containing the terms on the right of the $\triangleq$. This is straightforward for constants like naturals but requires a little insight for syntax with subterms like get e and set $e := e'$. Since we include metavariables in our term representation, we can represent the terms on the right by plugging a metavariable in for each recursive call to the compiler. In other words, our compilation table would map the label for "get" to the term "bind $x := e$; get x as y in $k\ y$" where e is a framework metavariable. To compile a term using such a compilation table, we traverse its syntax tree from the leaves up and replace each syntactic form with the corresponding term in the range of the table, filling that term's metavariables with the appropriate compiled subterms. Metavariables in a term undergoing compilation remain constant. For example, to compile the term `#"get" "v"`, we proceed as follows: As `"v"` is a metavariable, it is compiled to itself. We then look up `"get"` in our table and replace `"e"` with `"v"` in the associated term, giving us "bind $x :=$ v; get x as y in $k\ y$".

3.3 Framework Proof Structure

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 type and equivalence preservation. To demonstrate why this predicate is essential to our extensibility results, consider a standard proof of equivalence preservation for our CPS pass for STLC before we extend it. We assume type preservation in this example for simplicity and walk through the term equivalence case. Given two terms e_1 and e_2 such that $\Gamma \vdash_{STLC} e_1 = e_2 : A$, we have to show that $\lfloor \Gamma \rfloor \vdash_{CPS} \lfloor e_1 \rfloor = \lfloor e_2 \rfloor : \lfloor A \rfloor$. We proceed by induction on the proof of $\Gamma \vdash_{STLC} e_1 = e_2 : A$, which requires us to consider reflexivity, transitivity, symmetry, congruence, and beta 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 we have by the structure of compilers in Pyrosome. This leaves us the beta-reduction case, where we must prove that the right- and left-hand sides compile to equivalent terms, although we omit the proof here.

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

However, the structure of our 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 us nothing about terms with a mix of product operations and STLC constructs. To properly reuse our proof about STLC, we 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 $\mathit{Preserving}(L_t, \mathit{cmp}, L_s)$, which takes as input a target language L_t, compiler cmp, and source language L_s. This predicate has one case for each kind of rule that can be appended to L_s. Each case requires that the compiler satisfies *Preserving* for the tail of L_s and the appropriate condition for the head depending on the kind of rule:

- For a new sort rule, the compiler must map it to a well-formed sort in the target.
- For a new term rule, 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.

Since *Preserving* is defined pointwise over the rules of the input language, it is invariant under language extension, as shown in Theorem 1. If two compilers cmp_1 and cmp_2 map L_1 and L_2 respectively to a shared target T, we can append them to produce a compiler from $L_1 + L_2$ to T. Theorem 1 generalizes this fact to account for an arbitrary shared prefix in the case that L_1 and L_2 are extensions of some existing language.

Theorem 1 (Compiler extension)**.** *Let $L_{pre} + L_1$ and $L_{pre} + L_2$ be well-formed languages such that the rule names in L_1 and L_2 are disjoint. If Preserving($L_t, \mathit{cmp}_{pre} + \mathit{cmp}_1, L_{pre}+L_1$) and Preserving($L_t, \mathit{cmp}_{pre}+\mathit{cmp}_2, L_{pre}+L_2$), then Preserving($L_t, \mathit{cmp}_{pre}+\mathit{cmp}_1 + \mathit{cmp}_2, L_{pre} + L_1 + L_2$).*

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, it is sufficient at each rule to be able to compile only the previous rules. 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 key to the modularity of *Preserving* since it means that proofs of earlier obligations remain valid as the compiler is extended. In addition, we can use a weakening principle on the proof of *Preserving* for one extension so that it correctly fits on top of the other extension thanks to monotonicity.

We bridge the gap between these well-structured proof obligations and type and equivalence preservation in Theorem 2, 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, we must show that it is preserved by compilation. The term equivalence case resembles the earlier proof we sketched for the CPS pass starting from STLC, except that we generalize the beta-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. We disallow compilers from overwriting old cases, so we 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, we appeal to the fact that all judgments in Pyrosome are monotonic under language extension.

Theorem 2 (*Preserving* implies semantic preservation)**.** *Let L_s and L_t be well-formed languages, and let cmp be a compiler. If Preserving(L_t, cmp, L_s), then cmp is a type- and equivalence-preserving compiler from L_s to L_t.*

Thanks to this theorem, we prove semantic preservation for each compiler in Pyrosome by way of *Preserving*. Our 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 Coq level and can be proven by direct construction of either well-formedness or equivalence derivations in the target language. In chapter 2 I show how to solve one such equational-reasoning

obligation for the state extension to the CPS compiler.

We can also embed the output of our compilers into larger target languages using Theorem 3. I use $L_t \subseteq L_t'$ in the theorem in the literal sense that all rules in L_t are also in L_t'. Recall the CPS-transformation extension in chapter 1 that added global state. It compiled a source-language extension adding stateful operations to a related target-language extension that was not present in the target of the core CPS transformation. We expand the target language of the core compiler with Theorem 3 so that we can verify the extension.

Theorem 3 (Compiler codomain embedding)**.** *If we know Preserving*(L_t, cmp, L_s) *and* $L_t \subseteq L_t'$*, then it follows that Preserving*(L_t', cmp, L_s)*.*

3.4 Why Equivalence Preservation

There are many interpretations of compiler correctness in the literature [Patterson and Ahmed, 2019]. The original CompCert [Leroy et al., 2016] work proved whole-program simulation of the source language by the target language and used that 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 [Ahmed, 2015].

Some work on verified compiler correctness solves the linking problem for programs that share a specific low-level target language [Wang et al., 2014]. It is not clear how to modify such a framework to allow modular addition of new target-language features.

Pyrosome is designed for proving that compilers are type- and equivalence-preserving with respect to source and target equational theories. I claim that these properties are the critical ones for notions of compiler correctness. Equivalence preservation is often discussed in terms of contextual equivalence [Ahmed and Blume, 2011; Devriese et al., 2016]. However, as our *callTwice* example in chapter 2 showed, contextual equivalence can in fact be too strong to admit desirable language extensions. For example, in source languages with nondeterministic concurrency, a compiler might

want to pick a particular schedule for some portion of the code, thereby rendering it equivalent to a sequential program.

In this setting, there is one important potential blind spot: an equivalence-preserving compiler has the liberty to equate unintended values. At the most basic level, the trivial compiler that maps all programs to one that returns 0 is equivalence-preserving. On a similar note, both equivalence-preserving and fully abstract compilers can exchange true and false, or perform other isomorphic but potentially undesirable transformations. Fortunately, both of these issues can be resolved by fixing an expected mapping from observable values in the source to ones in the target, for example mapping all naturals and Booleans to themselves when they exist in both languages. These properties can be checked on top of my framework, say by reasoning about the effect of the compiler on an expression that is just a metavariable standing for a constant.

Chapter 4

Case Study

To validate Pyrosome, I present as a case study a 3-pass compiler for STLC that converts it to CPS and then performs closure conversion. I then extend this compiler to support natural numbers, the unit type, recursive functions, and mutable state to demonstrate the reusability of Pyrosome compilers. Figure 4-1 displays the rules of our STLC calculus. This figure is largely the same as Figure 3-1, but I make the value-expression distinction explicit. I use ret v, a construct from the substitution calculus, to embed values into the sort of expressions.

Our CPS translation targets a calculus of continuations, shown in the top half of Figure 4-2. Since our computations do not return, we eliminate the return type from their judgment form, writing $\Gamma \vdash e$ to indicate a well-formed computation and $\neg A$ as the type of a continuation that accepts input of type A. We still retain the judgment form $\Gamma \vdash v : A$ for values, however. To bridge the gap between source-language expression judgments and target-language computations, we translate expression judgments

$$\frac{\vdash A\ type \qquad \vdash B\ type}{\vdash A \to B\ type} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash_v \lambda(x : A).\ e : A \to B} \qquad \frac{\Gamma, x : A \vdash e : B \qquad \Gamma \vdash_v v : A}{\Gamma \vdash \text{ret}\ (\lambda(x : A).\ e)\ (\text{ret}\ v) = e[v/x] : B}$$

Figure 4-1: STLC

$$\frac{\vdash A\ type}{\vdash \neg A\ type} \qquad \frac{\Gamma \vdash v : \neg A \qquad \Gamma \vdash v' : A}{\Gamma \vdash v\ v'} \qquad \frac{\Gamma, x : A \vdash e}{\Gamma \vdash \lambda(x : A).\ e : \neg A}$$

$$\frac{\Gamma, x : A \vdash e \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda(x : A).\ e)\ v = e[v/x]} \qquad \frac{\Gamma \vdash v : \neg A}{\Gamma \vdash (\lambda(x : A).\ v\ x) = v : \neg A}$$

$$\begin{array}{rcl}
\lfloor A \to B \rfloor & \triangleq & \neg(\lfloor A \rfloor \times \neg \lfloor B \rfloor) \\
\lfloor \text{ret } v \rfloor & \triangleq & k\ \lfloor v \rfloor \\
\lfloor \lambda(x : A).\ e \rfloor & \triangleq & \lambda(p : \lfloor A \rfloor \times \neg \lfloor B \rfloor).\ \text{let } \langle x, k \rangle := p \text{ in } \lfloor e \rfloor \\
\lfloor e\ e' \rfloor & \triangleq & \text{bind } x := \lfloor e \rfloor;\ \text{bind } y := \lfloor e' \rfloor;\ x\ \langle y, k \rangle \\
\text{bind } x := e;\ e' & \triangleq & e[\lambda(x : B).\ e'/k], \text{ where } \Gamma, k : \neg B \vdash e
\end{array}$$

Figure 4-2: Continuation calculus and CPS for STLC

$\Gamma \vdash e : A$ to computation judgments $\lfloor \Gamma \rfloor, k : \neg \lfloor A \rfloor \vdash \lfloor e \rfloor$, using the variable k for the continuation. The second half of Figure 4-2 gives the translation from STLC into our CPS calculus, which also utilizes products in the target language. We prove equivalence preservation for this compiler via my predicate described in section 3.3. For each extension to each pass, we prove a theorem of the same form as Theorem 4, and then we rely on the theorems in section 3.3 to connect them and prove type and equivalence preservation. We prove similar properties for all languages and extensions mentioned in this section but elide most of them to avoid repetition. Section 4.5 discusses the statements and proofs of these theorems in Coq.

Theorem 4 (The CPS translation for STLC satisfies *Preserving*)**.** *The CPS translation satisfies Preserving with STLC as the source language and our CPS calculus as the target.*

For the second pass of my case study, I perform closure conversion, materializing the environment as a tuple value. In Figure 4-3, I show the closure calculus that replaces the CPS calculus. To construct environment tuples in a compositional way, we compile CPS environments to closure-converted types and map CPS judgments $\Gamma \vdash e$ and $\Gamma \vdash_v v : A$ to closure-converted judgments $z : \lfloor \Gamma \rfloor \vdash \lfloor e \rfloor$ and $z : \lfloor \Gamma \rfloor \vdash_v \lfloor v \rfloor : \lfloor A \rfloor$, using the variable z for the environment. Since I do not address type variables in this case study, I use a fused closure form clo $\langle (z : A \times B).e, v \rangle$ that

$$\frac{\vdash A\ type}{\vdash \neg A\ type} \qquad \frac{\Gamma \vdash v : \neg A \qquad \Gamma \vdash v' : A}{\Gamma \vdash v\ v'} \qquad \frac{z : A \times B \vdash e \qquad \Gamma \vdash v : A}{\Gamma \vdash \text{clo}\ \langle (z : A \times B).e, v\rangle : \neg B}$$

$$\frac{z : A \times B \vdash e \qquad \Gamma \vdash v : A \qquad \Gamma \vdash v' : B}{\Gamma \vdash \text{clo}\ \langle (z : A \times B).e, v\rangle\ v' = e[\langle v, v'\rangle / z]}$$

$$\frac{z : A \vdash_v v : \neg B}{z : A \vdash_v \text{clo}\ \langle (z : A \times B).v[z.1/z]\ z.2, z\rangle = v : \neg B}$$

$$\begin{array}{rcl} \lfloor \cdot \rfloor & \triangleq & \text{unit} \\ \lfloor \Gamma, x : A \rfloor & \triangleq & \lfloor \Gamma \rfloor \times \lfloor A \rfloor \\ \lfloor \neg A \rfloor & \triangleq & \neg \lfloor A \rfloor \\ \lfloor \lambda (x : A).\ e \rfloor & \triangleq & \text{clo}\ \langle (x : \lfloor A \rfloor).\lfloor e \rfloor, z\rangle \\ \lfloor v\ v' \rfloor & \triangleq & \lfloor v \rfloor\ \lfloor v' \rfloor \end{array}$$

Figure 4-3: Closure conversion (excerpt)

captures the combined behavior of the existential, pair, and function in the normal translation. I define two equations on closures, a beta rule and an eta rule. The beta rule evaluates the body of the closure, passing in a tuple formed of its argument and its environment. The eta 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.

4.1 Recursive Functions

We extend STLC with recursive functions, written fix $f(x : A) := e$, as a new value of function type, as shown below:

$$\frac{\Gamma, f : A \to B, x : A \vdash e : B}{\Gamma \vdash \text{fix}\ f(x : A) := e : A \to B}$$

$$\frac{\Gamma, f : A \to B, x : A \vdash e : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\text{fix}\ f(x : A) := e)\ v = e[(\text{fix}\ f(x : A) := e)/f, v/x] : B}$$

Since we assign this construct the same type as lambdas, we can use the same application form as well by providing a reduction rule for application of a recursive function. To perform CPS translation on recursive functions, we add an analogous construct for recursive continuations to the CPS calculus and translate from the first set of rules to the second:

$$\frac{\Gamma, f : \neg A, x : A \vdash e}{\Gamma \vdash \text{fix } f(x : A) := e : \neg A}$$

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

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

Building on the result for the core CPS transformation, we show that the compiler for the recursion extension satisfies *Preserving*. Since we have already proven Theorem 4, this proof requires fulfilling three new obligations: 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 substitution, which we discuss in section 4.3.

Theorem 5 (The CPS translation for recursion satisfies *Preserving*)**.** *The extended CPS translation satisfies Preserving with STLC + Recursion as the source language and our CPS calculus extended with recursive continuations as the target.*

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

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

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

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 we 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.

4.2 Global State and Evaluation Contexts

We provided the rules of our source-level global-state extension in Figure 3-3. As we discussed, these rules depend on two other extensions, one describing the behavior of global heaps and another that sets up evaluation contexts. Global heaps are given their own sort, with the basic axioms of finite maps. 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 us to describe performing a stateful operation inside some larger program. From a formal perspective, they let us focus on a subterm of the computation while defining an equation on the whole configuration so that we can access the heap. Figure 4-4 shows the core constructs of our evaluation-context extension on the first line, as well as the contexts for STLC as an example. Plug and the empty context are common to all languages with evaluation contexts, so they form the core extension. We add evaluation contexts in for each construct 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 to produce filled expressions. We can then use equational reasoning in our proofs to decompose terms and find their redices.

Since our CPS translation uses bind $x := e;\ e'$ to sequence computations, we actually compile away evaluation contexts in our first pass. The bottom part of Figure 4-4 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

$$\frac{}{\Gamma \vdash [\,] : A \rightsquigarrow A} \qquad \frac{\Gamma \vdash E : A \rightsquigarrow B \qquad \Gamma \vdash e : A}{\Gamma \vdash E[e] : B} \qquad \frac{\Gamma \vdash e : A}{\Gamma \vdash [\,][e] = e : A}$$

$$\frac{\Gamma \vdash E : A \rightsquigarrow B \to C \qquad \Gamma \vdash e : B}{\Gamma \vdash E\ e : A \rightsquigarrow C} \qquad \frac{\Gamma \vdash v : B \to C \qquad \Gamma \vdash E : A \rightsquigarrow B}{\Gamma \vdash v\ E : A \rightsquigarrow C}$$

$$\frac{\Gamma \vdash E : A \rightsquigarrow B \to C \qquad \Gamma \vdash e : B \qquad \Gamma \vdash e' : A}{\Gamma \vdash (E\ e)[e'] = E[e']\ e : C}$$

$$\frac{\Gamma \vdash v : B \to C \qquad \Gamma \vdash E : A \rightsquigarrow B \qquad \Gamma \vdash e : A}{\Gamma \vdash (v\ E)[e] = v\ E[e] : C}$$

$$\begin{array}{rcl} \lfloor [\,] \rfloor & \triangleq & k\ x_h \\ \lfloor E[e] \rfloor & \triangleq & \text{bind } x := \lfloor e \rfloor;\ \lfloor E \rfloor \\ \lfloor E\ e \rfloor & \triangleq & \text{bind } x := \lfloor E \rfloor;\ \text{bind } y := \lfloor e \rfloor;\ x\ \langle y, k \rangle \\ \lfloor v\ E \rfloor & \triangleq & \text{bind } y := \lfloor E \rfloor;\ \lfloor v \rfloor\ \langle y, k \rangle \end{array}$$

Figure 4-4: Evaluation-context extension with STLC

judgments $\Gamma, k : \neg B, x_h : A \vdash \lfloor E \rfloor$, translating evaluation contexts into computations with an extra free variable we call x_h by convention. Plug then turns into a bind operation, and holes behave similarly to return operations. Since evaluation contexts disappear in the CPS translation, they do not affect closure conversion.

With evaluation contexts established, the translation for our heap operations is minimal. Recall the CPS pass from Figure 1-2, restated here:

$$\begin{array}{rcl} \lfloor \text{set } e := e' \rfloor & \triangleq & \text{bind } x := e; \text{bind } y := e'; \text{set } x := y \text{ in } k\langle\rangle \\ \lfloor \text{get } e \rfloor & \triangleq & \text{bind } x := e; \text{get } x \text{ as } y \text{ in } k\ y \end{array}$$

To match the pattern of the CPS calculus, the heap operations now each take their continuation as a subterm. This inversion is why we no longer need to worry about evaluating effects inside of a context. Since this extension does not interact directly with functions, we can reuse the CPS version with our closure-converted calculus. We 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

$$\frac{}{\Gamma, A \vdash_v \underline{0} : A} \qquad \frac{}{\vdash \uparrow \ : \Gamma, A \Rightarrow \Gamma} \qquad \frac{\vdash \gamma : \Gamma \Rightarrow \Gamma' \qquad \Gamma' \vdash_v v : A}{\Gamma \vdash_v \gamma(v) : A}$$

$$\frac{\vdash \gamma : \Gamma \Rightarrow \Gamma' \qquad \Gamma' \vdash e : A}{\Gamma \vdash \gamma(e) : A} \qquad \frac{\Gamma \vdash_v v : A}{\Gamma \vdash \text{ret } v : A} \qquad \frac{\vdash \gamma : \Gamma \Rightarrow \Gamma' \qquad \Gamma' \vdash_v v : A}{\Gamma \vdash \gamma(\text{ret } v) = \text{ret } \gamma(v) : A}$$

$$\frac{}{\vdash \text{id} \ : \Gamma \Rightarrow \Gamma} \qquad \frac{\vdash \gamma : \Gamma \Rightarrow \Gamma' \qquad \Gamma \vdash_v v : A}{\vdash \langle \gamma, v \rangle : \Gamma \Rightarrow \Gamma', A} \qquad \frac{\vdash \gamma : \Gamma \Rightarrow \Gamma' \qquad \Gamma \vdash_v v : A}{\Gamma \vdash_v \langle \gamma, v \rangle(\underline{0}) = v : A}$$

$$\frac{\vdash \gamma : \Gamma \Rightarrow \Gamma' \qquad \gamma' : \Gamma' \Rightarrow \Gamma''}{\Gamma \vdash \gamma \circ \gamma' : \Gamma \Rightarrow \Gamma''} \qquad \frac{\vdash \gamma : \Gamma \Rightarrow \Gamma' \qquad \Gamma \vdash_v v : A}{\Gamma \vdash \langle \gamma, v \rangle \circ \uparrow \ = \gamma : \Gamma \Rightarrow \Gamma'}$$

Figure 4-5: Substitution calculus (excerpt)

get x as v in e:

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

4.3 Substitution

So far, I have assumed the existence of variables and substitution operations as convenient. I outline the formal structure of these constructs below. In summary, I use an explicit substitution calculus, implemented as a language module within Pyrosome. This calculus is a derivative of the one found in Sterling [2019], but simply typed rather than dependently typed to fit within the bounds of my current proof automation. In addition, this calculus makes a distinction between values and expressions. Since substitutions can only contain values, this calculus naturally supports call-by-value language features. Figure 4-5 shows some of the rules of this calculus.

Variables are actually repeated applications of the weakening substitution $\uparrow$ to the index $\underline{0}$, acting as de Bruijn indices. When we apply a nonempty substitution to $\underline{0}$, it grabs the first value, whereas composing the same substitution with $\uparrow$ discards that value. Notably missing here is how substitutions interact with the constructs in our extensions. However, this is to be expected. If we were to mechanize the

case study outside of Pyrosome, we would have to define a substitution function that operates over each language's syntax, with one case per syntactic form in each language feature. Instead, we take advantage of the systematic form of language definitions in Pyrosome to programmatically generate corresponding equations from language syntax rules. While the resulting rules form part of the compiler's trusted specification, they can be straightforwardly audited. For example, we generate the following rules for STLC:

$$\frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A \qquad \vdash \gamma : \Gamma' \Rightarrow \Gamma}{\Gamma \vdash \gamma(e\ e') = \gamma(e)\ \gamma(e') : B}$$

$$\frac{\Gamma, A \vdash e : B \qquad \vdash \gamma : \Gamma' \Rightarrow \Gamma}{\Gamma \vdash_v \gamma(\lambda A.\ e) = \lambda A.\ \langle \gamma, \underline{0} \rangle(e) : A \to B}$$

Since we implement substitution internally to Pyrosome just like any other feature, different languages can adapt their own notions of binding as necessary. Indeed, our CPS calculus replaces the rules for expressions that have return types with rules for nonreturning computations. Thanks to the modularity provided by Pyrosome, we reuse the behavior of substitutions on values rather than fully implementing substitution twice. Closure conversion targets the same substitution calculus as CPS, so we have little duplication.

4.4 Summary of Case-Study Implementation

While my case-study compiler itself is simple, the range of language extensions I include shows that Pyrosome allows me to add multiple effects to my language and verify them independently of each other and STLC's pure reasoning. I combine all of the separate correctness proofs of my components to show correctness of the amalgamated multipass compiler using Theorem 2 and my modularity theorems:

Theorem 6 (Equivalence preservation of the case-study compiler)**.** *The case-study components form an equivalence-preserving compiler from STLC + Naturals + Re-*

Definitions	Theorem Statements	Proofs
1054	259	112

Figure 4-6: Case-study lines of code by purpose

cursion + Unit + State to the closure-conversion calculus augmented with a fixpoint combinator, state operations, the unit type, and product types.

To assess the work required to use Pyrosome, I categorize the lines of code required for proving the *Preserving* predicate for all of my compilers by purpose in Figure 4-6. I designate three categories: language and compiler definitions, theorem statements, and proofs. Compiler definitions include any helper functions and shorthand I define. I leave out import statements, hints, and blank lines. While it may seem that I have many lines of definitions and theorem statements for a rather modest compiler, the code in these categories is quite sparse since I take advantage of vertical space to improve its formatting. Theorem statements have a fixed format, so although they represent a reasonable proportion of the code, they should not pose a burden to users of Pyrosome. Each theorem consists of a list of dependencies followed by the language or compiler to be verified. Most remarkably for this case study, I manage to prove the majority of my theorems using single tactics. These tactics are generic over language extensions in Pyrosome, as I discuss in section 4.5. While they are currently limited to simple types, this case study demonstrates that Pyrosome has the potential to substantially lower the burden of proof engineering for its users.

4.5 Inference and Automation

As mentioned in section 3.1 and section 4.4, I rely on automation for type inference and to do most of the heavy lifting in my proofs. To support type inference, I define a separate elaboration judgment in parallel with each well-formedness judgment of Pyrosome, including term well-formedness, language well-formedness, and *Preserving*. These versions take in an extra parameter, the preelaboration syntax, and ensure that the term, language, or compiler under scrutiny lines up with it. I then derive the full term, language, or compiler with tactics that generate correct-by-construction

```
Derive fix_cc
  SuchThat (elab_preserving_compiler
    (cc ++ prod_cc_compile ++ subst_cc)
    (fix_cc_lang ++ ... ++ value_subst)
    fix_cc_def fix_cc fix_cps_lang)
  As fix_cc_preserving.
Proof.
  auto_elab_compiler.
  cleanup_elab_after
    (reduce;
     term_cong; try term_refl;
     eapply eq_term_trans;
     [eapply eq_term_sym;
      eredex_steps_with cc_lang "clo_eta"|];
     by_reduction).
Qed.
```

Figure 4-7: Coq proof for recursive continuations

elaborations. The tactics I developed are fairly general and bear no ties to any of the languages in my case study, including to the substitution calculus. For example, they would likely do just as well on languages with dynamic scope or linearity. Their primary limitation at present is with regard to languages that have type equations, for example to handle substitution of type variables. They can handle certain instances of type equalities but are not yet advanced enough to automate polymorphism or dependent typechecking. It is still possible to model such languages in Pyrosome, but they currently require far more manual proofs. I discuss this further in section 5.5. Since all terms in the case study are simply typed, however, I fully automate all term and language elaboration and well-formedness proofs in my examples. I also largely automate proofs of equivalence preservation using a tactic to normalize both sides of the equation. To do so, I rely on the convention that the equations specified in each language can be read left-to-right as reduction rules. This approach turned out to be very effective, solving almost all of the equivalence-preservation goals in my case study.

While I have exhibited two of my compiler-correctness theorems in chapter 4 using English, I provide one example in Figure 4-7 of the direct Coq statement

and proof to convey the detailed workings of my framework. I use Coq's `Derive` command to indicate that this theorem constructs an elaborated language `fix_cc` while proving that `fix_cc` is an elaboration of `fix_cc_def` and satisfies the predicate *Preserving*(L_t, cmp_{pre}+`fix_cc`, $L_{pre}+L_s$), where L_t is the language passed as the second argument to the predicate, cmp_{pre} is the compiler passed as the first argument, and L_{pre} is the source language of cmp_{pre}. I use `++`, Coq's built-in append function on lists, to build L_{pre} and cmp_{pre} from smaller components. In this proof, my main tactic `auto_elab_compiler` only solves 2 out of 3 of this theorem's proof obligations, and the remaining obligation requires a little finesse. I must show that the behavior of substitution on CPS-calculus recursive functions is preserved by the translation. To do so, I use eta expansion for closures in the proof, so I cannot solve it solely by beta reduction. I first reduce both sides of the equation as much as possible. This leaves us with two instances of the fix v combinator, whose arguments I must equate. I go under the combinator using congruence, apply the eta law on the left-hand side in the appropriate orientation, and solve the goal by my reduction tactic.

Chapter 5

Related and Future Work

5.1 Alternative Generic Frameworks

The principle benefit of GATs [Sterling, 2019] over most generic frameworks for programming-language metatheory [Felleisen, 1991; Delaware et al., 2013a,b] is its presentation via equational theories rather than operational semantics. Felleisen [1991] gives meaning to its programs by an arbitrary termination predicate and a contextual-equivalence relation defined in terms of that predicate. As discussed in chapter 2, 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 minimal equational theories wherever possible, users of Pyrosome increase the compatibility and extensibility of their developments. Additionally, since I do not rely on evaluation, I can reason about fully normalizing languages and extensions in the same way as Turing-complete ones.

Prior work on modular metatheory and language extensibility mechanized a system in Coq that covered components like language interpreters, including modular soundness properties [Delaware et al., 2013a,b], in the style of Swierstra [2008]. Allais et al. [2018] also built a framework for reasoning about binding, renaming, and translation that bears some similarities to this work. However, both projects incorporate object-language binding and substitution into their frameworks, rather than

implementing them as a first-class feature. While this is attractive for ease of use since it lifts substitution reasoning into the metatheory, it restricts the generality of the framework.

The K Framework [Roşu and Şerbănuţă, 2010] has built an extensive ecosystem of generic language-specification tooling, and their logic is powerful enough to express a variety of binders internally [Chen and Roşu, 2020]. The project has also recently expanded to cover certified proofs of the behavior of individual programs [Chen et al., 2021]. However, they do not address the higher-order concerns of verified compilation and compiler extensibility that I cover in this work.

5.2 Multilanguage Semantics

Existing literature documents some of the uses of multilanguage semantics, including in compiler verification [Matthews and Findler, 2009; Perconti and Ahmed, 2014; Ahmed, 2015]. 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 [Perconti and Ahmed, 2014], 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 [Bowman, 2021] by expressing compiler cases as operational rules, but it is still limited by its use of contextual equivalence.

5.3 Optimization

My current formulation of compilers precludes any real attempt at an optimizing compiler since I maintain the invariant $\lfloor \gamma(e) \rfloor = \lfloor \gamma \rfloor(\lfloor e \rfloor)$ syntactically. This forces my compilers to generate the same code for each occurrence of a syntactic form,

even when some of them could be optimized. However, I still see a way to make optimization work within Pyrosome. While for extensibility's sake, my translations from one language to another cannot optimize, intralanguage optimization seems much more feasible. Consider an arbitrary function $o : \mathit{term} \to \mathit{term}$ such that for any well-formed term $e : t$ in some language, $e = o(e) : t$ in that language. I should be able to insert such a function into a vertically composed sequence of translations and still derive semantic equivalence preservation of the composition. In this way, optimization passes would have the freedom to manipulate the program tree arbitrarily so long as they respect the semantics. The key to this approach is that since optimization occurs within one language unlike translation, it can directly relate the input to the output. The technique described above is not perfect. Working with an arbitrary function in the middle of the compiler gives up on the extensibility of whatever language it operates on. I hope to explore the design space of optimizations in future work to see if there is a middle ground that can both express standard optimizations and retain some level of extensibility.

5.4 Modeling Pyrosome's Equational Theories

As-is, theorems proven about compilation using Pyrosome include the initial source-language and final target-language specifications in their trusted base, in addition to the core definitions of the framework. 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. I hope in the future to target established projects like CompCert [Leroy et al., 2016] and Bedrock [Erbsen et al., 2021] so that upper levels of a compiler can use the flexibility of Pyrosome while benefitting from established codebases beneath.

5.5 Advanced Type Systems

Pyrosome was designed to support a wide range of features at both the term and type levels. The current theory should support a variety of interesting type-level features, such as polymorphism, linearity, or dependent types. For example, I have defined a variant of my substitution language that includes type-level substitutions so as to support polymorphic or dependent types. However, in this thesis I chose to focus on simple types as they were adequate to present the features of the framework and allowed for effective proof automation. More complex type systems should fit naturally within Pyrosome with either additional manual effort or improved automation techniques.

Bibliography

A. Ahmed. Verified Compilers for a Multi-Language World. In T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15–31, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-80-4. doi: 10.4230/LIPIcs.SNAPL.2015.15. URL `http://drops.dagstuhl.de/opus/volltexte/2015/5013`.

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*, pages 431–444, 2011.

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):90:1–90:30, July 2018. ISSN 2475-1421. doi: 10.1145/3236785. URL `http://doi.acm.org/10.1145/3236785`.

W. J. Bowman. Compilation as multi-language semantics. In *Workshop on Principles of Secure Compilation*, 2021.

X. Chen and G. Roşu. A general approach to define binders using matching logic. Technical Report http://hdl.handle.net/2142/106608, University of Illinois at Urbana-Champaign, June 2020.

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.

A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–106, 2010.

B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Meta-theory à la carte. In *POPL (2013)*, pages 207–218. ACM, 2013a. ISBN 978-1-4503-1832-7.

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, pages 319–330, New York, NY, USA, 2013b.

ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500587. URL http://doi.acm.org/10.1145/2500365.2500587.

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*, pages 164–177, 2016.

A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala. Integration verification across software and hardware for a simple embedded system. In *PLDI*, volume 21, page 2021, 2021.

M. Felleisen. On the expressive power of programming languages. *Science of computer programming*, 17(1-3):35–75, 1991.

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, page 178190, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837642. URL https://doi.org/10.1145/2837614.2837642.

S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113. Springer, 1998.

R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *POPL 2014*, POPL '14, New York, NY, USA, 2014. ACM.

X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. Compcert – a formally verified optimizing compiler. In *ERTS 2016*. SEE, 2016.

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*, pages 1–15, 2019.

J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3):12:1–12:44, Apr. 2009. ISSN 0164-0925. doi: 10.1145/1498926.1498930. URL http://doi.acm.org/10.1145/1498926.1498930.

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*, pages 103–116, 2016.

M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys (CSUR)*, 51(6):1–36, 2019.

D. Patterson and A. Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

J. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming Languages and Systems*, pages 128–148. Springer, 2014.

G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

J. Sterling. Algebraic type theory and universe hierarchies. `http://www.jonmsterling.com/pdfs/algebraic-universes.pdf`, 2019.

W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008. ISSN 0956-7968. doi: 10.1017/S0956796808006758. URL `http://dx.doi.org/10.1017/S0956796808006758`.

P. Wadler. The expression problem. 1998.

P. Wang, S. Cuellar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 675–690, 2014.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI 2011*. ACM, 2011.