Pyrosome: Verified Compilation for Modular Metatheory

DUSTIN JAMNER, Massachusetts Institute of Technology, USA
GABRIEL KAMMER, Intel, USA
RITAM NAG, Massachusetts Institute of Technology, USA
ADAM CHLIPALA, Massachusetts Institute of Technology, USA

We present Pyrosome, a generic framework for modular language metatheory that embodies a novel approach to extensible semantics and compilation, implemented in Coq. Common techniques for semantic reasoning are often tied to the specific structures of the languages and compilers that they support. Contextual equivalence is difficult to work with directly, and both logical relations and transition system-based approaches typically fix a specific notion of effect globally. While modular transition systems have been effective in imperative settings, they are suboptimal for functional code. These limitations restrict the extension and composition of semantics in these systems. In Pyrosome, 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.

Pyrosome defines a formal, deeply embedded notion of programming languages with semantics given by dependently sorted equational theories, so all compiler-correctness proofs boil down to type-checking and equational reasoning. We support vertical composition of any compilers expressed in our framework in addition to feature extension. Since our design requires compilers to support open programs, our correctness guarantees support linking with any target code of the appropriate type. As a case study, we present a multipass compiler from System F with simple references, through CPS translation and closure conversion. Specifically, we demonstrate how we 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. We also present a linear version of the simply typed CPS pass and compile a small imperative language to the simply typed target to show how Pyrosome handles substructural typing and imperative features.

CCS Concepts: • **Software and its engineering → Compilers**; **Formal language definitions**; **Formal software verification**.

Additional Key Words and Phrases: compiler verification, extensibility, modular metatheory

ACM Reference Format:
Dustin Jamner, Gabriel Kammer, Ritam Nag, and Adam Chlipala. 2025. Pyrosome: Verified Compilation for Modular Metatheory. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 274 (October 2025), 28 pages. https://doi.org/10.1145/3763052

1 Introduction

Compiler verification is a laborious process compared to unverified implementation. Nevertheless, numerous researchers have taken on the challenge [12, 25, 26] due to the significant benefits of

Authors' Contact Information: Dustin Jamner, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, dijamner@mit.edu; Gabriel Kammer, Intel, USA, gkammer@mit.edu; Ritam Nag, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, rnag@mit.edu; Adam Chlipala, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, adamc@csail.mit.edu.

ACM 2475-1421/2025/10-ART274
https://doi.org/10.1145/3763052

formal verification [50]. 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 [17], 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 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 [22, 35]. While recent work has made advances in high-to-low-level and multilanguage linking [24, 36], 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 [37] 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.

In this work, we present 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. While we hope one day to expand our software artifact's scope to cover the full feature set required to implement a practically usable compiler, our 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. We 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.

The particular combination of design decisions embodied in Pyrosome includes among other things: 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 our desired notion of extensibility. We represent programming languages as Generalized Algebraic Theories (GATs) [9, 23], which define program semantics via lists of judgments that generate well-formedness predicates and minimal equational theories.[1] The translations we primarily consider are type- and equivalence-preserving

[1]As opposed to contextual equivalence, which is a maximal equational theory. [39]

$$\frac{}{\text{nat } \mathit{sort}} \qquad \frac{}{0 : \text{nat}} \qquad \frac{n : \text{nat}}{Sn : \text{nat}} \qquad \frac{m : \text{nat} \qquad n : \text{nat}}{m + n : \text{nat}} \qquad \frac{n : \text{nat}}{0 + n = n : \text{nat}}$$

$$\frac{m : \text{nat} \qquad n : \text{nat}}{Sm + n = S(m + n) : \text{nat}}$$

Fig. 1. GAT for natural numbers

maps (GAT morphisms) that are defined pointwise over the constructs of the source language and so are amenable to extension. Our overall contributions are as follows:

- We develop metatheory, methods, and mechanized tooling for building and combining language extensions and compilers in the setting of GATs.
- We define an inductive characterization of well-formedness for cross-language translations encoded in Pyrosome and prove that it implies both type and equivalence preservation.
- We characterize extensions of these specifications and compilers and prove theorems enabling separate reasoning about and combination of such extensions.
- We implement a generic partial-evaluation pass to support the viability of optimizations in this context.
- We 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 STLC and successively adding features via our extension theorems.

2 Generalized Algebraic Theories

Generalized Algebraic Theories (GATs) [9, 23] 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 [23], in Pyrosome, we 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 [9] did in order to keep our computation simply typed at the Coq level.

We will use a brief example to explain the different forms of rules that make up a GAT. Figure 1 defines a GAT for natural numbers with addition. The first rule declares that nat is a sort of our theory. This theory only uses one simple sort, so it could be expressed as an untyped algebraic theory, but we 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 we will demonstrate a GAT that makes use of dependent typing by adding more rules to this theory to develop a datatype of known-length vectors (of nats for simplicity). As shown in Figure 2, we 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, we bestow app with computational behavior via equations that specify its action on nil and $n :: v$ on the left. This last rule demonstrates what we mentioned earlier

$$\frac{n : \text{nat}}{\text{vec } n \; \mathit{sort}} \qquad \frac{}{\text{nil} : \text{vec } 0} \qquad \frac{m : \text{nat} \qquad n : \text{nat} \qquad v : \text{vec } n}{m :: v : \text{vec } Sn}$$

$$\frac{m : \text{nat} \qquad v : \text{vec } m \qquad n : \text{nat} \qquad v' : \text{vec } n}{\text{app } v \; v' : \text{vec } (m + n)} \qquad \frac{n : \text{nat} \qquad v : \text{vec } n}{\text{app nil } v = v : \text{vec } n}$$

$$\frac{i : \text{nat} \qquad m : \text{nat} \qquad v : \text{vec } m \qquad n : \text{nat} \qquad v' : \text{vec } n}{\text{app } (i :: v) \; v' = i :: (\text{app } v \; v') : \text{vec } (Sm + n)}$$

Fig. 2. GAT for vectors

about rules depending on prior equations. If we calculate the natural type for the left-hand side of the equation, we get vec $(Sm + n)$ as per the annotation. However, if we do the same for the right-hand side, we get vec $S(m + n)$ instead. The reason we can still use this term at our desired type is by invoking the defining equation of addition from Figure 1, which the GAT allows implicitly. While our simply typed case studies do not use this expressive strength, it is critical to supporting our polymorphic extension.

3 Simply Typed Lambda Calculus

We begin our case studies with the simply typed lambda calculus. Some frameworks bake in a notion of object-language variables and substitutions [4, 7, 19] so that user languages automatically inherit the behavior of binders, which makes defining 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 us 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 we must first define a call-by-value substitution calculus. However, once we have done so, we can reuse it as the basis for many languages.

We base the substitution calculus off of Dybjer [16] but modify it to represent call-by-value substitutions, with a subset of the rules shown in Figure 3. The first five rules declare sorts for object-language contexts, types, values, expressions, and substitutions. We elide metavariable sorts of the forms $\Gamma : \text{ctx}$ and $A : \text{ty}$ in Figure 3 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 ret v injects values into the sort of expressions. Its name is inspired by fine-grained call-by-value calculi [27], although our example remains standard call-by-value. We define an operation to append a type to a context and write de Bruijn index $\underline{0}$ to represent the variable at the head of a context. We also add syntactic forms for explicit substitutions [1] on expressions and values and a substitution $\langle \gamma, v \rangle$ that appends a value to the head of a substitution. We include two representative equations in Figure 3. The first defines the behavior of expression substitution on 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.

$$\frac{}{\text{ctx } \mathit{sort}} \qquad \frac{}{\text{ty } \mathit{sort}} \qquad \frac{\Gamma : \text{ctx} \qquad A : \text{ty}}{\text{val}(\Gamma, A)\ \mathit{sort}} \qquad \frac{\Gamma : \text{ctx} \qquad A : \text{ty}}{\text{exp}(\Gamma, A)\ \mathit{sort}} \qquad \frac{\Gamma : \text{ctx} \qquad \Delta : \text{ctx}}{\text{sub}(\Gamma, \Delta)\ \mathit{sort}}$$

$$\frac{v : \text{val}(\Gamma, A)}{\text{ret}\ v : \text{exp}(\Gamma, A)} \qquad \frac{\Gamma : \text{ctx} \qquad A : \text{ty}}{\Gamma, A : \text{ctx}} \qquad \frac{\Gamma : \text{ctx} \qquad A : \text{ty}}{\underline{0} : \text{val}((\Gamma, A), A)} \qquad \frac{e : \text{exp}(\Gamma, A) \qquad \gamma : \text{sub}(\Delta, \Gamma)}{e[\gamma] : \text{exp}(\Delta, A)}$$

$$\frac{v : \text{val}(\Gamma, A) \qquad \gamma : \text{sub}(\Delta, \Gamma)}{v[\gamma]_v : \text{val}(\Delta, A)} \qquad \frac{\gamma : \text{sub}(\Delta, \Gamma) \qquad v : \text{val}(\Delta, A)}{\langle \gamma, v \rangle : \text{sub}(\Delta, (\Gamma, A))}$$

$$\frac{v : \text{val}(\Gamma, A) \qquad \gamma : \text{sub}(\Delta, \Gamma)}{(\text{ret}\ v)[\gamma] = \text{ret}\ v[\gamma]_v : \text{exp}(\Delta, A)} \qquad \frac{\gamma : \text{sub}(\Delta, \Gamma) \qquad v : \text{val}(\Delta, A)}{\underline{0}[\langle \gamma, v \rangle]_v = v : \text{val}(\Delta, A)}$$

Fig. 3. CBV substitution calculus (selected rules)

$$\frac{\vdash A \qquad \vdash B}{\vdash A \to B} \qquad \frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash e\ e' : B} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash_v \lambda(x : A).\ e : A \to B}$$

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

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

Fig. 4. STLC

We 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 we will be working on extensions of the substitution calculus for most of our examples, we will adopt a few fairly standard notations: we 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, we elide type and context sorts when they can be inferred. Additionally, we 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 4. We add three new terms to the syntax: the function type $A \to B$, application expressions $e\ e'$, and function values $\lambda(x : A).\ e$. The subsequent equations define β-reduction and substitution on applications and functions. We can (and usually should) also include an η-expansion equation. However, the benefit of extensibility is that we 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 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 that we can do language metaprogramming in Gallina (Coq's dependently typed metalanguage), which we use to generate such equations automatically rather than write them by hand. In doing so, we 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 [42, 43]. We handle the related situation in our 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 4 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. We 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 9.

4 Language Extensions

In section 2 and section 3, we 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, we can express the addition of an extension as the concatenation of the new rules onto the old ones. The key benefit of GATs for our purposes is that all the well-formedness and equivalence judgments of a GAT are preserved by language extension. We formalize this family of properties in Theorem 4.1, where $L \subseteq L'$ denotes the unordered inclusion of all rules of L in L'. Here and in subsequent metatheory, we 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 4.1 (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. □

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 our use of list inclusion in Theorem 4.1. 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.

4.1 Equational Theories, Contextual Equivalence, and Extensibility

As we 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 [3, 15]. This choice is necessary if we want to have Theorem 4.1. Contextual equivalence is the maximal equivalence that respects an operational semantics [39]; 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

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

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

Fig. 5. Recursive-function extension

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, we will consider the following simple example program, which elides ret for convenience of presentation:

$$\begin{array}{lcl} \mathit{callTwice} & : & (\text{bool} \to \text{bool}) \to \text{bool} \to \text{bool} \\ \mathit{callTwice} & \triangleq & \lambda(f : \text{bool} \to \text{bool}).\ \lambda(x : \text{bool}). \\ & & (\lambda(\_ : \text{bool}).\ f\ x)\ (f\ \text{false}) \end{array}$$

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 $\lambda(f : \text{bool} \to \text{bool}).\ f$ since STLC is a pure, terminating language. However, consider how this program behaves once we add the recursive-function extension in Figure 5. 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, we can apply *callTwice* to functions like fix $f(x : \text{bool}) :=$ if x 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 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 4.1 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.

5 Cross-Language Translation

Now that we have established how languages are defined and extended, we can proceed to their compilers. We define our 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 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 [30, 34, 37].

This paper 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 we consider in this work. In order to fully support both linking and compiler extension, we require compilers to operate on terms with arbitrary free metavariables. Furthermore, we want compilation

$$\frac{\Gamma : \mathrm{ctx}}{\mathrm{cmp}(\Gamma)\ \mathit{sort}} \qquad \frac{\vdash A}{\vdash \neg A} \qquad \frac{\Gamma \vdash_v v : \neg A \qquad \Gamma \vdash_v v' : A}{\Gamma \vdash v\ v'} \qquad \frac{\Gamma, x : A \vdash e}{\Gamma \vdash_v \lambda(x : A).\ e : \neg A}$$

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

$$\frac{\Gamma \vdash_v v : A \times B \qquad \Gamma, x : A, y : B \vdash e}{\Gamma \vdash \mathrm{let}\ \langle x, y\rangle := v\ \mathrm{in}\ e} \qquad \frac{\Gamma \vdash_v v_1 : A \qquad \Gamma \vdash_v v_2 : B}{\Gamma \vdash_v (v_1, v_2) : A \times B}$$

$$\frac{\Gamma \vdash_v v_1 : A \qquad \Gamma \vdash_v v_2 : B \qquad \Gamma, x : A, y : B \vdash e}{\Gamma \vdash \mathrm{let}\ \langle x, y\rangle := (v_1, v_2)\ \mathrm{in}\ e = e[v_1/x, v_2/y]}$$

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

Fig. 6. Continuation calculus (selected rules) and CPS translation for STLC

to be invariant under substitution of metavariables, i.e. $[\![\gamma(e)]\!] = [\![\gamma]\!]([\![e]\!])$, where $[\![e]\!]$ denotes the compilation of e and $\gamma(e)$ denotes the metavariable substitution γ applied to the term e.[2]

We choose to guarantee that this equation holds syntactically. While we could likely work through much of Pyrosome's metatheory using a superficially more flexible semantic equality, it is our observation that such a change would do little to change the expressiveness of our compilers. For each term former f of arity n, such a compiler would have to define $[\![f\ x_1 ... x_n]\!]$, where $x_1 ... x_n$ are metavariables. Then even if the compiler were to inspect subterms deeply when compiling a more complex term $[\![f\ e_1 ... e_n]\!]$ for subterms $e_1 ... e_n$, it would have to produce a result semantically equivalent to $[\![f\ x_1 ... x_n]\!][[\![e_1]\!]/x_1, ..., [\![e_n]\!]/x_n]$. Since we expect optimization to be handled by intralanguage passes rather than our translations anyway, we choose the simpler option of taking the above term to be the definition of compilation.

5.1 CPS

Our case-study compilers closely follow the designs of the first two passes of Morrisett et al. [33], 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 6. This calculus extends the values portion of the base substitution calculus. Since CPS computations do not return, we replace the expression sort, instead extending the value-substitution calculus with a computation sort $\mathrm{cmp}(\Gamma)$ that eliminates the return type, writing $\Gamma \vdash e$ to indicate a well-formed computation $e : \mathrm{cmp}(\Gamma)$. We define $\neg A$ as the type of a nonreturning continuation that accepts input of type A,

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

named to evoke the definition $\neg A \triangleq A \longrightarrow \bot$ and the connection between CPS and double negation. We also include a positive product-types extension in the CPS target.

With the target out of the way, we can define our first translation. The second half of Figure 6 gives the translation from STLC into our CPS calculus. Formally, we 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 ret v case above represents a pair in the association list containing the symbol ret and the output term $k\ [\![v]\!]$ where the $[\![v]\!]$ on the right-hand side is a metavariable that will be substituted with the result of compiling the subterm in the v position when we run the compiler.

To bridge the gap between source-language expression judgments and target-language computations, we translate expression judgments $\Gamma \vdash e : A$ to computation judgments $[\![\Gamma]\!], k : \neg[\![A]\!] \vdash [\![e]\!]$, using the variable k for the continuation. Functions $A \longrightarrow B$ map to computations $\neg([\![A]\!] \times \neg[\![B]\!])$, each taking as inputs the compiled argument $[\![A]\!]$ and the continuation $\neg[\![B]\!]$. Since we 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 we add them) handles mediating between source and target values.

The compiler case for function application debuts a useful macro, bind, that we will employ in most of our CPS-pass extensions as well. We 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, $(\text{bind } x := [\![\text{ret } v]\!];\ e') = e'[[\![v]\!]/x]$. Especially when there are multiple binds, as in function application, we find this notation far more readble than nested, inverted substitutions.

5.2 Semantics Preservation

So far we have defined a few languages and a minimal compilation pass, but we have yet to state or prove theorems. There are many interpretations of compiler correctness in the literature [35]. The original CompCert [26] 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 [2]. Recent developments in the CompCert ecosystem such as CompCertO [24] support linking, and Zhang et al. [51] 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, we view the state of FFIs as the unfortunate result of the current limitations of language specifications and interfaces, which we believe can be improved via the principles of Pyrosome. To put it another way, we 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 [46]. 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 we express formally in the following definition:

Definition 5.1 (Semantics Preservation). A function $[\![-]\!]$ is a *semantics-preserving* compiler from source S to target T if all of the following hold[3]:

- If $C \vdash_S s$ *sort* then $[\![C]\!] \vdash_T [\![s]\!]$ *sort*
- If $C \vdash_S t : s$ then $[\![C]\!] \vdash_T [\![t]\!] : [\![s]\!]$
- If $C \vdash_S t_1 = t_2 : s$ then $[\![C]\!] \vdash_T [\![t_1]\!] = [\![t_2]\!] : [\![s]\!]$

We 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. We add the rules $[\![\text{true}]\!] = \text{nv } 1$ and $[\![\text{false}]\!] = \text{nv } 0$ 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 we 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 false $\sim$ nv 0, but it would be reasonable either to decide true $\sim$ nv 1 and no other naturals are allowed, or to add true $\sim$ nv n where $n \neq 0$. For this example, we 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 5.2 (CPS Cross-Language Correctness). *If* $\vdash e = \mathit{ret}\ b : \mathit{bool}$ *in STLC + Bool, then* $k : \neg \mathit{bool} \vdash [\![e]\!] = k\ n$ *in CPS + Nat and* $b \sim n$.

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

5.3 The Preserving Predicate

Now that we have established semantics preservation as our goal, it remains to establish a standard method for proving it and for doing so extensibly. A significant contribution of this work is our 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 our extensibility results, consider a standard proof of equivalence preservation for our CPS pass for base STLC. We 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 $\Gamma \vdash e_1 = e_2 : A$, we have to show that $[\![\Gamma]\!], k : \neg[\![A]\!] \vdash [\![e_1]\!] = [\![e_2]\!]$ in the continuation calculus. We proceed by induction on the proof of $\Gamma \vdash e_1 = e_2 : A$, which requires us 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 we have by the structure of translations in Pyrosome. What remains to prove is the β-reduction case,

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

```
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).
```

Fig. 7. Coq definition of Preserving (selected cases)

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 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, we can prove them generically, so we 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 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, we define the inductive predicate Preserving(L_t, 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.

We present the formal definition of Preserving in Figure 7, 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 `n`, the compiler must map `n` to a term `e` such that `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 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 Coq 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}. We 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. We can then define $\text{Preserving}(L_t, cmp, L_s)$ as $\text{Preserving}_{[]}(L_t, cmp, L_s)$, 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 5.3 (Preserving implies semantics 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 semantics-preserving compiler from L_s to L_t.*

For Preserving to be useful, it must give us semantics preservation. We bridge this gap in Theorem 5.3, 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 β-reduction case to a lemma about the preservation of each equivalence written in the language description. This lemma relies on 2 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 Theorem 4.1, which states that all judgments are monotonic under language extension.

Thanks to this theorem, we prove semantic preservation for each translation 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 fact, we are now equipped to prove equivalence preservation for our base CPS pass, which we will do in Theorem 5.4 and Theorem 5.5, which state respectively that the translation for the base substitution calculus and the STLC extension satisfy Preserving.

Theorem 5.4 (CPS_{subst} is Preserving). *Let subst be the source substitution calculus, let cps_{subst} be the portion of the CPS pass with subst as its domain, and let cont be the continuation calculus. Then we have $\text{Preserving}_{[]}(cont, cps_{subst}, subst)$.*

Proof. By the definition of Preserving and construction of *cont* derivations in all cases. □

Theorem 5.5 (CPS is Preserving). *Let cps_{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 Preserving$_{cps_{subst}}$(cont, cps, STLC).*

Proof. By the definition of Preserving and construction of *cont* derivations in all cases. We 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 us to show $[\![(\text{ret } \lambda(x : A).\ e)\ \text{ret } v]\!] = [\![e[v/x]]\!]$ in the continuation calculus' equivalence relation. By evaluating the compiler and rewriting the term via target-language rules, we do so as follows[4]:

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

□

We are almost ready to prove semantics preservation from these facts, but first we need one more generic theorem:

Theorem 5.6 (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 Preserving(L_t, cmp, L_s) and Preserving$_{cmp}(L_t, cmp', L_s')$. Then Preserving$(L_t, cmp' + cmp, L_s' + L_s)$.*

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

Now we can prove that our CPS pass is semantics-preserving in Theorem 5.7. We 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 5.7 (CPS is Semantics-Preserving). *Let subst be the source substitution calculus, let cps_{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 cps + cps_{subst} is a semantics-preserving compiler from STLC + subst to cont.*

Proof. By Theorem 5.3 it suffices to show Preserving$(cont, cps + cps_{subst}, STLC + subst)$. We apply Theorem 5.6 and conclude with Theorem 5.4 and Theorem 5.5. □

5.4 Closure Conversion and Vertical Composition

For the second pass of our case study, we perform closure conversion, materializing the environment as a tuple value. In Figure 8, we show the closure calculus that replaces our 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 $z : [\![\Gamma]\!] \vdash [\![e]\!]$

[4]We 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.

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

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

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

Fig. 8. Closure conversion (selected rules)

and $z : [\![\Gamma]\!] \vdash_v [\![v]\!] : [\![A]\!]$, using the variable z for the environment. In order to define closure conversion in the simply typed setting, we use a fused 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 et al. [32]. We 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 we have two compiler passes, we can sequence them using function composition to get from the source to the final target. Fortunately, we can do so in one step thanks to our choice of definitions. We prove semantics preservation for closure conversion on its own analogously to CPS. Then, by transitivity (Theorem 5.8), since both passes preserve semantics, so does their composition.

Theorem 5.8 (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. We 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 we conclude. □

5.5 Recursive-Functions Compiler Extension

We extended STLC with rules for recursive functions in Figure 5. To perform CPS translation on recursive functions, we add an analogous construct for recursive continuations to the continuation calculus and translate from the first set of rules to the second:

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

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

Building on the result for the core CPS transformation, we show in Theorem 5.9 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.

Theorem 5.9 (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. □

Similarly to Theorem 5.7, we can show that the combined compiler cps_{rec} + cps is semantics-preserving by using Theorem 5.3 and Theorem 5.6. However, we also need one more property. We can only concatenate compilers that have the same target language, but the codomain of *cps* is the continuation calculus before we have added recursive continuations. Fortunately, Theorem 5.10 states that we can lift any compiler targeting a language L_t to one targeting a superset language L_t'. Thus, we can lift *cps* to target $cont_{rec}$ before appending the cps_{rec} compiler extension.

Theorem 5.10 (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).*

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 generated by closure conversion is 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:

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

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

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. 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 other than a single η expansion in one case. From there, we can use our connective theorems as before to join up all relevant components.

Now that we have covered all of the ways compilers can be extended, we include Figure 9 as a reference for all of the different ways we have extended our compiler. Solid arrows indicate translations written and verified by the user, and dashed arrows indicate translations validated by the theorems of Pyrosome that we have introduced over the course of this section.

5.6 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 4.1 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 explictly given rules, we 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.

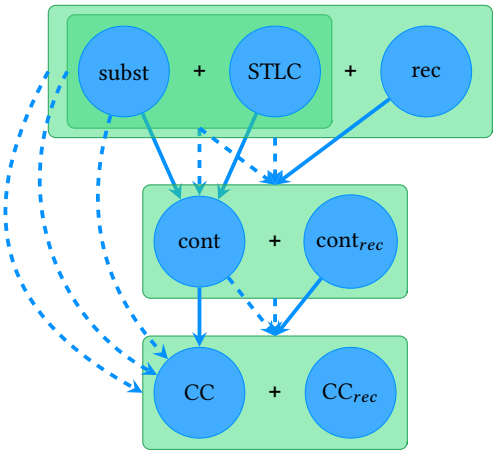

Fig. 9. Compiler extension

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 we 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 exception 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 5.2, is adequate to rule out this case.

5.7 Axiomatic Equivalence and Operational Semantics

For a language that already has a canonical operational semantics, we 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 $e \longrightarrow^* v$ in the source language, then $[\![e]\!] = [\![v]\!]$ in the target where $[\![e]\!]$ 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. We can then use the following theorem to link the two:

Theorem 5.11. *Let $[\![-]\!]$ be a semantics-preserving compiler from S to T. Let $\longrightarrow_S$ and $\longrightarrow_T$ 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 $\vdash_T [\![v_S]\!] = v_T$ and $v_S \sim v_T$.

Then, if $\vdash_S e : t$, $e \to_S^* v$, and $[\![e]\!] \to_T^* v'$ where v and v' are observable values, then $v \sim v'$.

While the statement of Theorem 5.11 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 5.2, this latter property can often be validated by simply running the compiler on the appropriate inputs.

6 Optimization

In our formulation, translation passes are designed for cross-language transformations rather than optimization. We 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 [6] 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, we 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 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 [28, 47, 52]. Due to their rewrite-based interfaces, we expect that such tools can be adapted to our domain of equational theory-based language extensibility.

As an initial proof of concept, we implement a language-agnostic partial-evaluation pass that shares its implementation with some of our 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, we 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 our partial evaluator is fairly naïve, we believe that it shows Pyrosome's potential compatibility with more advanced rewrite-based optimizations.

7 Global State and Evaluation Contexts

One of our more interesting extensions adds a global mutable heap with set and get expressions, the rules for which we provide in Figure 10. 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 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.

$$\frac{\Gamma : \mathrm{ctx} \qquad \vdash A}{\mathrm{config}(\Gamma, A)\ \mathit{sort}} \qquad \frac{H : \mathrm{heap} \qquad \Gamma \vdash e : A}{\langle H, e\rangle : \mathrm{config}(\Gamma, A)} \qquad \frac{\Gamma \vdash e : \mathrm{nat}}{\Gamma \vdash \mathrm{get}\ e : \mathrm{nat}}$$

$$\frac{\Gamma \vdash E : \mathrm{nat} \rightsquigarrow A \qquad n : \mathbb{N}}{\langle H, E[\mathrm{get}\ (\mathrm{ret}\ (\mathrm{nv}\ n))]\rangle = \langle H, E[H(n)]\rangle : \mathrm{config}(\Gamma, A)} \qquad \frac{\Gamma \vdash e : \mathrm{nat} \qquad \Gamma \vdash e' : \mathrm{nat}}{\Gamma \vdash \mathrm{set}\ e := e' : \mathrm{unit}}$$

$$\frac{\Gamma \vdash E : \mathrm{unit} \rightsquigarrow A \qquad n : \mathbb{N} \qquad m : \mathbb{N}}{\langle H, E[\mathrm{set}\ (\mathrm{ret}\ (\mathrm{nv}\ n)) := \mathrm{ret}\ (\mathrm{nv}\ m)]\rangle = \langle H[n \mapsto m], E[\langle\rangle]\rangle : \mathrm{config}(\Gamma, A)}$$

Fig. 10. STLC state extension

It would be more idiomatic in a GAT to formulate such operations with algebraic effects [38]. We 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.

Figure 11 shows the core constructs of our evaluation-context extension on the first line, as well as the contexts for STLC as an example. Plug, which we write $E[e]$, and the empty context [ ] form the core extension. We 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. We can then use equational reasoning in our proofs to decompose terms.

As an incidental benefit of our formal, deeply embedded representation of language rules, we 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 11 are generated from the following description of the evaluation contexts for STLC:

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

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

Since our CPS translation uses bind $x := e;\ e'$ to sequence computations, we compile away evaluation contexts in our first pass. The bottom of Figure 11 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 $\Gamma, k : \neg B, x_h : A \vdash [\![E]\!]$, 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 our heap operations is minimal. We can define the CPS pass for the set and get operations as follows:

$$\begin{aligned} [\![\mathrm{set}\ e := e']\!] &\triangleq \mathrm{bind}\ x := e; \mathrm{bind}\ y := e'; \mathrm{set}\ x := y\ \mathrm{in}\ k\ \langle\rangle \\ [\![\mathrm{get}\ e]\!] &\triangleq \mathrm{bind}\ x := e; \mathrm{get}\ x\ \mathrm{as}\ y\ \mathrm{in}\ k\ y \end{aligned}$$

To match the pattern of the CPS calculus, each heap operation now takes its continuation as a subterm. This inversion is why 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

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

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

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

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

$$\begin{array}{rcl}
[\![[\,]]\!] & \triangleq & k\ x_h \\
[\![E[e]]\!] & \triangleq & \text{bind } x := [\![e]\!];\ [\![E]\!] \\
[\![E\ e]\!] & \triangleq & \text{bind } x := [\![E]\!];\ \text{bind } y := [\![e]\!];\ x\ \langle y, k\rangle \\
[\![v\ E]\!] & \triangleq & \text{bind } y := [\![E]\!];\ [\![v]\!]\ \langle y, k\rangle \\
 & & \text{bind } x := e;\ e' \triangleq e[\lambda(x : B).\ e'/k] \text{ given } \Gamma, k : \neg B \vdash e
\end{array}$$

Fig. 11. Evaluation-context extension with STLC

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 get x as v in e:

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

8 Polymorphism and Advanced Metaprogramming

Unlike other case-study extensions, polymorphism necessitates modifying the rules of our simply typed extensions to add the type environment. To do so, we 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 we also use it to add type information to our base substitution calculus, which allows the type substitutions and term substitutions to share a common set of underlying definitions. We prove that this parameterization function is semantics-preserving on sorts, terms, contexts, and languages. Using these facts, we further show that it preserves the Preserving predicate on compilers, allowing the output compilers to participate in all of our 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. Our 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 we do not parameterize. Fortunately, we reduce them to sound, decidable checks that can guarantee a valid parameterization without inducing additional proof obligations. The result is Theorem 8.1.

$$\frac{\Delta, X \vdash A}{\Delta \vdash \forall X.A} \qquad \frac{\Delta, X \vdash A}{\Delta \vdash \exists X.A} \qquad \frac{\Delta; \Gamma \vdash e : \forall X.A \qquad \Delta \vdash B}{\Delta; \Gamma \vdash e\ [B] : A[B/X]} \qquad \frac{\Delta \vdash \Gamma \qquad \Delta, X; \Gamma \vdash e : A}{\Delta; \Gamma \vdash_v \Lambda X.e : \forall X.A}$$

$$\frac{\Delta \vdash \Gamma \qquad \Delta, X; \Gamma \vdash e : A \qquad \Delta \vdash B}{\Delta; \Gamma \vdash (\text{ret } \Lambda X.e)\ [B] = e[B/X]} \qquad \frac{\Delta; \Gamma \vdash_v v : \forall X.A}{\Delta; \Gamma \vdash_v \Lambda X.v\ [X] = v : \forall X.A}$$

$$\frac{\Delta; \Gamma \vdash e : \exists X.A \qquad \Delta \vdash C \qquad \Delta, X; \Gamma, (x : A) \vdash e' : C}{\Delta; \Gamma \vdash \text{let } \langle X, x \rangle := e \text{ in } e' : C} \qquad \frac{\Delta \vdash B \qquad \Delta; \Gamma \vdash e : A[B/X]}{\Delta; \Gamma \vdash \langle B, e \rangle : \exists X.A}$$

$$\begin{array}{rcl}
[\![\forall X.A]\!] & \triangleq & \neg\exists X.\neg[\![A]\!] \\
[\![\Lambda X.e]\!] & \triangleq & \lambda k.\text{let } \langle X, k \rangle := k \text{ in } [\![e]\!] \\
[\![e\ [A]]\!] & \triangleq & \text{bind } x := [\![e]\!];\ x\ \langle [\![A]\!], k \rangle \\
[\![\exists X.A]\!] & \triangleq & \exists X.[\![A]\!] \\
[\![\langle A, e \rangle]\!] & \triangleq & \text{bind } x := [\![e]\!];\ \langle [\![A]\!], x \rangle \\
[\![\text{let } \langle X, x \rangle := e \text{ in } e']\!] & \triangleq & \text{bind } y := [\![e]\!];\ \text{let } \langle X, x \rangle := y \text{ in } [\![e']\!] \\
 & & \text{bind } x := e;\ e' \triangleq e[\lambda(x : B).\ e'/k] \text{ given } \Gamma, k : \neg B \vdash e
\end{array}$$

Fig. 12. Polymorphic calculus extensions (selected rules)

Theorem 8.1 (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 satisfying Preserving(L_t, c, L_s). Then, if all syntactic checks hold on the above components, we have*

$$Preserving_{id_{L_\Delta}}(P_L(\Delta, s_\Delta, spec_t, L_t), P_c(\Delta, spec_t, spec_s, c), P_L(\Delta, s_\Delta, spec_s, L_s))$$

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 we have applied the parameterization procedure, instantiated with the information specific to the type-environment parameter, to our running case study, we can develop the type-abstraction and existential-types extensions to validate that the parameterization process had the intended effect. We display a portion of the rules for the source-language extension, as well as a selection of CPS-pass cases, in Figure 12. In the target, the parameterized continuation calculus, we choose to add only existentials, as type abstractions can be compiled to existential continuations via CPS. Specifically, we translate the type $\forall X.A$ to $\neg\exists X.\neg[\![A]\!]$. 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.

9 Linear Lambda Calculus

Our primary case study builds entirely on a standard, intuitionistic substitution calculus. To demonstrate Pyrosome's expressiveness, we also verify a CPS pass analogous to our simply typed one from a separate, linear lambda calculus to a similarly linear continuation calculus. The linear substitution calculus is similar to the intuitionistic one, including retaining all rules in Figure 3, but it restricts substitutions and variables to prohibit weakening. We show the rules for the linear functions extension in Figure 13. These rules are similar to those of STLC, but in the rules 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

$$\frac{\vdash A \qquad \vdash B}{\vdash A \multimap B} \qquad \frac{\Gamma \vdash e : A \multimap B \qquad \Gamma' \vdash e' : A}{\Gamma, \Gamma' \vdash e\ e' : B} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash_v \lambda(x : A).\ e : A \multimap B}$$

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

$$\frac{\Gamma \vdash e : A \multimap B \qquad \Delta \vdash e' : A \qquad \gamma : \Gamma' \Rightarrow \Gamma \qquad \delta : \Delta' \Rightarrow \Delta}{\Gamma', \Delta' \vdash (e\ e')[\gamma, \delta] = e[\gamma]\ e'[\delta] : B}$$

$$\frac{\Gamma, x : A \vdash e : B \qquad \gamma : \Gamma' \Rightarrow \Gamma}{\Gamma' \vdash_v (\lambda(x : A).\ e)[\gamma] = \lambda(x : A).\ e[\gamma, x/x] : A \multimap B}$$

Fig. 13. Linear Lambda Calculus

break into two parts, one for e and one for e'. Just like with the intuitionistic substitution calculus, we 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. This case study currently has more manual proofs since our automation is not yet sufficient to handle environment splitting without user input, but it still benefits from the metatheory and principled subgoal management Pyrosome provides.

10 IMP

To show that our framework is not restricted to functional languages, we also compile an imperative calculus, the grammar of which is shown in Figure 14, similar to the one in Chaper 2 of Winskel [48] to the same target as our functional code. We reuse the heap definition from our 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, we 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 our main case study, we obtain some guarantees about the interoperation of imperative and functional code. Specifically, they can be linked, directly or with target-level glue code, so long as the target types line up, and we can prove that linking a compiled imperative program $[\![S]\!]$ with a compiled functional program $[\![f]\!]$ by either method produces an equivalent program to linking some $[\![S']\!]$ and $[\![f']\!]$ the same way if $S = S'$ and $f = f'$ in their respective source languages.

11 Inference and Automation

We rely on automation for type inference and to do most of the heavy lifting in our proofs. Formally, GATs require the user to specify values for every metavariable when using a term former. For

$$
\begin{array}{rcl}
a, a' & ::= & n \mid x \mid a + a' \mid a - a' \\
s, s' & ::= & \mathsf{skip} \mid \mathsf{assign}\ x := a \mid s; s' \mid \mathsf{if}\ a\ \mathsf{then}\ s\ \mathsf{else}\ s' \mid \mathsf{while}\ a\ \{s\} \\
n, x & \in & \mathbb{N}
\end{array}
$$

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

Fig. 14. IMP (excerpt)

example, an application node must include the input and output types, as well as the context, as additional arguments. To make writing GATs in our mechanization practical, we use the term below the line to determine which arguments must be written and which should be inferred. To return to application, since we write it $e\ e'$, the arguments e and e' are explicit while Γ, A, and B are implicit.

To support inferring implicit arguments, we 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 pre-elaboration syntax, and ensure that the term, language, or compiler under scrutiny lines up with it. We then derive the full term, language, or compiler with tactics that generate correct-by-construction elaborations. The tactics we developed are fairly general and bear no ties to any of the languages in our case study, including to the substitution calculus, as shown by the fact that the same inference tactics also work on the linear and imperative examples.

We largely automate proofs of equivalence preservation using a tactic that normalizes both sides of the goal, the structure of which is shown in Figure 15. To do so, we rely on the convention that the equations specified in each language can be read left-to-right as reduction rules. This convention is purely a heuristic convenience, but on the examples we have worked through it has been highly successful. Since both our terms and language definitions are represented as deeply embedded syntax, we define a Gallina function, the unverified evaluator in Figure 15, that takes as input a term and the language it belongs to and rewrites it from left to right along each applicable rule. To be useful in our proofs, this evaluator generates a reified proof term representing the equivalence between its input and output. We additionally define and verify a proof-checking function that computes whether such a proof is valid. Our tactic then runs the evaluator on the terms in the goal and inserts a call to the checker with the result in a proof-by-reflection style. Structuring the tactic this way means that we do not have to verify the evaluator code, which is notably more complex than the proof checker. As an added benefit, since the evaluator does not appear in the proof term, only the checking function must be rerun at the end of the proof, which improves performance.

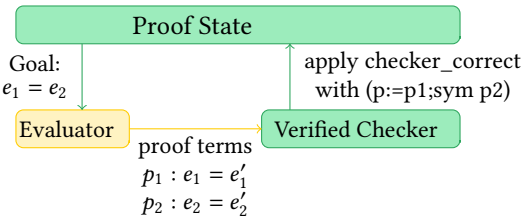

Fig. 15. Architecture of equation-solving tactic

This approach turned out to be effective, solving almost all of the equivalence-preservation goals in our case study. The obligations that required manual guiding generally fit at least one of two categories: either they required the use of an η-law, or they interacted with the heap. Our tactics featured reduced efficacy on heap rules related to the encoding of pointer comparisons in our rule definitions.

12 Related Work

12.1 Alternative Generic Frameworks

The principle benefit of GATs [9] over most generic frameworks for programming-language metatheory [13, 14, 18] is its presentation via equational theories rather than operational semantics. Felleisen [18] gives meaning to its programs by an arbitrary termination predicate and a contextual-equivalence relation defined in terms of that predicate. As discussed in subsection 4.1, contextual equivalence is too strong for extensible reasoning. We, 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 Coq that covered components like language interpreters, including modular soundness properties [13, 14], in the style of Swierstra [45]. Allais et al. [4] 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 [20], 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 this is attractive for ease of use since it lifts substitution into the metatheory, it restricts the generality of the framework in ways that we wish to avoid in Pyrosome.

The K Framework [40] has built an extensive ecosystem of generic language-specification tooling, and their logic is powerful enough to express a variety of binders internally [11]. The project has also recently expanded to cover certified proofs of the behavior of individual programs [10]. However, they do not address the higher-order concerns of verified compilation and compiler extensibility that we cover in this work.

The recent GATlab project [29] is based on a very similar system of GATs and GAT morphisms, implemented as an embedded domain-specific language in Juila 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 we do.

12.2 Multilanguage Semantics

Existing literature documents some of the uses of multilanguage semantics, including in compiler verification [2, 31, 37]. 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 we take advantage of. Furthermore, since they use contextual equivalence in their multilanguages in their specifications of compiler correctness [37], 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 [8] 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 we do not consider, we 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 [36]. This 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 our primary concern of compiler correctness. In contrast, we define a source-level equational theory, which allows us 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.

12.3 Existing Verified Compilers

Since Pyrosome currently is a foundational study rather than a production tool, we cannot compare it directly to the tooling built up for more mature projects like CakeML [25] and CompCert [24, 26]. However, we can discuss what improvements or extensions to our theoretical results might be necessary before compilers with similar feature sets can be implemented in Pyrosome. One major hurdle is developing our story for optimization as discussed in section 6. While prior work shows that real optimizations can be verified against equational theories, it remains to properly explore the design space regarding how best to make them extensible.

The other significant gap between what we theoretically support and existing verified-compiler ecosystems is our 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. We 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. [5]. CompCert also features graph-shaped IRs, which we have yet to investigate.

12.4 DimSum

The recent DimSum framework [41] gives a convincing accounting of language-agnostic compiler verification and cross-language linking for low-level, event-based systems. However, we 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 us 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, we enable users of Pyrosome to reason about programs, languages, and compilers in a way that is preserved by language extension as well. Our syntactic core also supports both programmatic manipulation of language definitions and our extensive automation of proof goals.

In reference to prior work on multilanguage semantics [2, 31, 37], Sammler et al. [41] 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." We largely agree with this analysis. As we discuss above in 12.2 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 we have addressed both of these concerns in Pyrosome, we 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.

13 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. We 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 we currently only target Pyrosome languages with our case studies, our compilers and metatheorems support any target model that validates core well-formedness and equivalence properties. We hope in the future to target established projects like CompCert [26] or Bedrock [17] 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 [49]. 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 [9, 44] in language definitions. For example, we have mechanized a dependent variant of our substitution language, in other words Martin-Löf Type Theory without connectives, following the descriptions in prior work [16]. We expect a sufficiently motivated user could extend the formalization of this calculus with the usual dependent connectives, such as dependent products and sums, or even a hierarchy of universes [44]. Such extensions would rely on our monotonicity and compiler-extension theorems just like our case study. However, in this paper we chose to limit ourselves to features that fit within our current proof automation. At present many dependent connectives would require more manual proofs due to our current automation's reliance on syntactic type equality during type inference. This restriction is not related to our theorems but is rather an issue of tactic-engineering complexity, and we hope to extend our automation to support such features better in the future.

14 Data-Availibility Statement

This paper is accompanied by the associated Coq development as its artifact [21]. The development contains proofs of all mentioned theorems, sometimes in slightly more generality than described in

the paper. All theorems relevant to the paper should be closed under the global context, with no assumed axioms.

Acknowledgments

We thank Amal Ahmed for early guidance on this project and the anonymous reviewers for extensive constructive feedback. This research was supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. 2141064. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. 1989. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 31–46.

[2] Amal Ahmed. 2015. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 15–31. https://doi.org/10.4230/LIPIcs.SNAPL.2015.15

[3] Amal Ahmed and Matthias Blume. 2011. An equivalence-preserving CPS translation via multi-language semantics. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. 431–444.

[4] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A Type and Scope Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Proc. ACM Program. Lang.* 2, ICFP, Article 90 (July 2018), 30 pages. https://doi.org/10.1145/3236785

[5] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. 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), 789–817.

[6] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. *ACM SIGPLAN Notices* 39, 1 (2004), 14–25.

[7] Jasmin Christian Blanchette, Lorenzo Gheri, Andrei Popescu, and Dmitriy Traytel. 2019. Bindings As Bounded Natural Functors. *Proc. ACM Program. Lang.* 3, POPL, Article 22 (Jan. 2019), 34 pages. https://doi.org/10.1145/3290335

[8] William J Bowman. 2021. Compilation as Multi-Language Semantics. In *Workshop on Principles of Secure Compilation*. ACM, New York, NY, USA.

[9] John Cartmell. 1986. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic* 32 (1986), 209–243. https://doi.org/10.1016/0168-0072(86)90053-9

[10] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. 2021. Towards a Trustworthy Semantics-Based Language Framework via Proof Generation. In *Proceedings of the 33rd International Conference on Computer-Aided Verification*. ACM.

[11] Xiaohong Chen and Grigore Roşu. 2020. *A General Approach to Define Binders Using Matching Logic*. Technical Report http://hdl.handle.net/2142/106608. University of Illinois at Urbana-Champaign.

[12] Adam Chlipala. 2010. 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). http://adam.chlipala.net/papers/ImpurePOPL10/

[13] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à La Carte. In *POPL (2013)* (Rome, Italy). ACM, 207–218.

[14] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. 2013. Modular Monadic Meta-theory. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM, New York, NY, USA, 319–330. https://doi.org/10.1145/2500365.2500587

[15] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 164–177.

[16] Peter Dybjer. 1995. Internal type theory. In *International Workshop on Types for Proofs and Programs*. Springer, 120–134.

[17] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification Across Software and Hardware for a Simple Embedded System. In *PLDI*, Vol. 21. 2021.

[18] Matthias Felleisen. 1991. On the expressive power of programming languages. *Science of computer programming* 17, 1-3 (1991), 35–75.

[19] Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.* 6, POPL, Article 53 (Jan. 2022), 29 pages. https://doi.org/10.1145/3498715

[20] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *Journal of the ACM (JACM)* 40, 1 (1993), 143–184.

[21] Dustin Jamner, Gabriel Kammer, and Ritam Nag. 2025. *Pyrosome.* https://doi.org/10.5281/zenodo.15762503

[22] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 178–190. https://doi.org/10.1145/2837614.2837642

[23] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.* 3, POPL, Article 2 (Jan. 2019), 24 pages. https://doi.org/10.1145/3290315

[24] Jérémie Koenig and Zhong Shao. 2021. CompCertO: compiling certified open C components. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1095–1109. https://doi.org/10.1145/3453483.3454097

[25] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL 2014* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 13 pages.

[26] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – A Formally Verified Optimizing Compiler. In *ERTS 2016*. SEE.

[27] PaulBlain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and computation* 185, 2 (2003), 182–210.

[28] John M. Li and Andrew W. Appel. 2021. Deriving Efficient Program Transformations from Rewrite Rules. *Proc. ACM Program. Lang.* 5, ICFP, Article 74 (aug 2021), 29 pages. https://doi.org/10.1145/3473579

[29] Owen Lynch, Kris Brown, James Fairbanks, and Evan Patterson. 2024. 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). https://doi.org/10.46298/entics.14666

[30] Phillip Mates, Jamie Perconti, and Amal Ahmed. 2019. Under control: Compositionally correct closure conversion with mutable state. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. 1–15.

[31] Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-language Programs. *ACM Trans. Program. Lang. Syst.* 31, 3, Article 12 (April 2009), 44 pages. https://doi.org/10.1145/1498926.1498930

[32] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. 1996. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 271–283.

[33] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568.

[34] Max S New, William J Bowman, and Amal Ahmed. 2016. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 103–116.

[35] Daniel Patterson and Amal Ahmed. 2019. The next 700 compiler correctness theorems (functional pearl). *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.

[36] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic Soundness for Language Interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 609–624. https://doi.org/10.1145/3519939.3523703

[37] Jamie Perconti and Amal Ahmed. 2014. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming Languages and Systems*. Springer, 128–148.

[38] Gordon Plotkin and John Power. 2002. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 342–356.

[39] Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theoretical computer science* 5, 3 (1977), 223–255.

[40] Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397–434.

[41] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 27 (jan 2023), 31 pages. https://doi.org/10.1145/3571220

[42] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. 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, 359–374.

[43] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. 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*. 166–180.

[44] Jonathan Sterling. 2019. Algebraic Type Theory and Universe Hierarchies. (2019). `https://arxiv.org/pdf/1902.08848`.

[45] Wouter Swierstra. 2008. Data Types à La Carte. *J. Funct. Program.* 18, 4 (July 2008), 423–436. https://doi.org/10.1017/S0956796808006758

[46] Peng Wang, Santiago Cuellar, and Adam Chlipala. 2014. 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). http://adam.chlipala.net/papers/CitoOOPSLA14/

[47] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[48] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press. https://doi.org/10.7551/mitpress/3054.001.0001

[49] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

[50] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI 2011* (San Jose, California, USA). ACM, 12 pages.

[51] Ling Zhang, Yuting Wang, Jinhua Wu, Jérémie Koenig, and Zhong Shao. 2024. Fully Composable and Adequate Verified Compilation with Direct Refinements between Open Modules. *Proc. ACM Program. Lang.* 8, POPL, Article 72 (Jan. 2024), 31 pages. https://doi.org/10.1145/3632914

[52] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better together: Unifying datalog and equality saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 468–492.

Received 2024-10-16; accepted 2025-08-12