A Verified Compiler for an Impure Functional Language

Adam Chlipala
Harvard University, Cambridge, MA, USA
adamc@cs.harvard.edu

Abstract

We present a verified compiler to an idealized assembly language from a small, untyped functional language with mutable references and exceptions. The compiler is programmed in the Coq proof assistant and has a proof of total correctness with respect to big-step operational semantics for the source and target languages. Compilation is staged and includes standard phases like translation to continuation-passing style and closure conversion, as well as a common subexpression elimination optimization. In this work, our focus has been on discovering and using techniques that make our proofs easy to engineer and maintain. While most programming language work with proof assistants uses very manual proof styles, all of our proofs are implemented as adaptive programs in Coq's tactic language, making it possible to reuse proofs unchanged as new language features are added.

In this paper, we focus especially on phases of compilation that rearrange the structure of syntax with nested variable binders. That aspect has been a key challenge area in past compiler verification projects, with much more effort expended in the statement and proof of binder-related lemmas than is found in standard pencil-and-paper proofs. We show how to exploit the representation technique of *parametric higher-order abstract syntax* to avoid the need to prove any of the usual lemmas about binder manipulation, often leading to proofs that are actually shorter than their pencil-and-paper analogues. Our strategy is based on a new approach to encoding operational semantics which delegates all concerns about substitution to the meta language, without using features incompatible with general-purpose type theories like Coq's logic.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Mechanical verification; D.2.4 [*Software Engineering*]: Correctness proofs, formal methods; D.3.4 [*Programming Languages*]: Compilers

General Terms Languages, Verification

Keywords compiler verification, interactive proof assistants

1. Introduction

Mechanized proof about programming languages is rather new as an engineering discipline. Only a handful of "real world" projects have been undertaken with users beyond computer science and mathematics researchers. Still, projected practical applications underlie most recent work. For example, compiler verification holds the promise of dramatically reducing the costs of quality assurance in the development, evolution, and maintenance of compilers. Unfortunately, this sort of verification seems today to require epic investments of time and cleverness by experts in semantics and theorem-proving. The main message of this paper is that, as in more familiar software development, compiler verification admits design patterns that cut down dramatically on the required grunt work, to the point where it seems plausible that the use of verification can actually reduce the overall effort required to build a correct compiler, even when the baseline we compare against is almost-correct compilers where copious testing has found all but the most obscure bugs.

There has been much important research on verifying compilers for relatively low-level languages like C, including in the verified language stack project by Moore (1989) and the more recent CompCert project (Leroy 2006). In that domain, languages researchers generally start projects and discover that, when they pick the wrong abstractions and proof structuring principles, verification requires much more work than they expected. In contrast, when picking the wrong abstractions in mechanized proofs about languages with nested variable binders (such as most functional languages), the same researchers often find themselves buried in details that they thought of as irrelevant. We have heard many stories of knowledgeable semanticists outright giving up on these kinds of proofs.

Recently, there has been much progress in research on the representation techniques that minimize the chances of such defeats. The use of higher-order abstract syntax (HOAS) in Twelf (Pfenning and Schürmann 1999) remains a popular choice, though it seems that a majority of languages researchers prefer interactive proof assistants that, unlike Twelf, can automate large parts of proofs. The de Bruijn index representation (de Bruijn 1972) is another old standard that sees wide use today. Perhaps the most popular methodologies now center around the nominal logic package for Isabelle/HOL (Urban and Tasson 2005) and the Penn approach to locally nameless binding in Coq (Aydemir et al. 2008). With these techniques, many facts about variable freshness and term well-formedness remain present explicitly in proofs, but there are standard recipes for figuring out the right lemmas to prove and when to apply them.

These recipes make it likely that a user with enough perseverance will manage to prove his theorem. This is a great improvement over the situation of just a few years ago, but, in this paper, we argue that we should be striving for more. In software engineering, we focus on maximizing programmer productivity, and we believe that mechanized proof engineering could stand to see a similar focus.

Most parts of most proofs about practical programming languages are exercises in stepping through many cases that are proved in unenlightening ways, with a handful of cases representing the core insights of a proof. Unfortunately, most mechanized proofs still spend significant amounts of code on the uninteresting cases, with significant effort expended to write that code. When it comes time to change a theorem statement, say because a language has been extended with a new construct, the user needs to go back over all of his very manual proofs, editing and adding cases. This is especially tedious with traditional manual proofs in Coq, where proofs are completely unstructured series of commands that modify proof states. Declarative proof languages like Isabelle's Isar (Wenzel 1999) help alleviate much of this complication, but they do so arguably at the expense of greater verbosity of proofs and greater expenditure in building the first version of a formal development. Is there an even more effective means of structuring proof scripts, such that we can realize evolvability benefits similar to those that software engineers have come to expect?

In the course of this paper, we hope to convince the reader that the answer is "yes." We will describe our experience building a verified compiler for an impure functional language in Coq. Three main contributions underly the implementation.

- We apply the *parametric higher-order abstract syntax* technique (Chlipala 2008) on a larger, more realistic compiler verification case study than in previous work. This encoding lets us avoid any code dealing with name freshness or index rearrangement in our compiler pass implementations, and that simplicity makes it easier to write correctness proofs.
- To avoid the usual deluge of lemmas about substitution, we use a new approach to encoding operational semantics. Substitution does not appear explicitly and is instead delegated to the meta language, as in the classical HOAS approach, but in a way compatible with general-purpose type theories like Coq's.
- Each of our Coq proof scripts is a program that is able to adapt to changes to language definitions and theorem statements. Such proof scripts express what are, in our opinion, the real essences of why theorems are true, the insights that could stump someone coming at the proofs from scratch.

Our approach is a synergistic combination of lightweight representations and aggressive theorem-specific automation. We implemented a first version of our compiler for a source language missing several of the features from the final version: let expressions, constants, equality testing, and recursive functions. We were able to add these features after-the-fact with minimal alterations of and additions to our proof scripts; the extended proofs do not even mention the new syntactic constructs or the operational semantics rules that govern them.

Some of the ideas we present here can be mapped back into alternative ways of doing things in pencil-and-paper semantics, but, at the level of detail that is traditional in venues like POPL, our techniques would probably only increase proof length and complexity. Instead, this paper is focused on how to engineer a verified compiler for a functional language. The trickiest parts of doing this with a proof assistant turn out to have little relation to the trickiest parts of doing it on paper. Representations matter a lot, and proof structuring techniques have a serious impact on how easy it is to evolve a verified compiler over time.

Past projects have considered verifying compilers for pure functional languages. As far as we are aware, ours is the first to consider a functional source language with either of mutable references or exceptions. On paper, these features seem straightforward to add to a compiler proof. In a proof assistant, when using the most straightforward proof techniques, impurity infects all of the main theorems and lemmas. It seems a shame to pass up the opportunity to automate the flow of these details through our proofs, and we do our best to take advantage of the possibility.

1.1 The Case Study

Our compiler operates on programs in a kind of untyped Mini-ML, as shown in Figure 1. We have constants from some unspecified base types, comparable with a primitive equality operation; recursive functions; let-binding; unit values; products; sums; mutable references; and exceptions. This language is meant to capture the key features of core ML's dynamic semantics, omitting essential features only when they involve variable numbers of arguments or variable binding structure. In particular, we do not model variable-arity products and sums, mutually-recursive functions, or compound pattern matching.

$$\begin{array}{rcll}
\text{Constants} & c & & \\
\text{Variables} & x, f & & \\
\text{Expressions} & e & ::= & c \mid e = e \mid x \mid e\ e \mid \mathsf{fix}\ f(x).\ e \\
& & & \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\
& & & \mid () \mid \langle e, e \rangle \mid \mathsf{fst}(e) \mid \mathsf{snd}(e) \\
& & & \mid \mathsf{inl}(e) \mid \mathsf{inr}(e) \\
& & & \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow e \mid \mathsf{inr}(x) \Rightarrow e \\
& & & \mid \mathsf{ref}(e) \mid !e \mid e := e \\
& & & \mid \mathsf{raise}(e) \mid e\ \mathsf{handle}\ x \Rightarrow e
\end{array}$$

Figure 1. Source language syntax

$$\begin{array}{rcll}
\text{Registers} & r & ::= & r_0 \mid \dots \mid r_{N-1} \\
\text{Constants} & n & \in & \mathbb{N} \\
\text{Lvalues} & L & ::= & r \mid [r+n] \mid [n] \\
\text{Rvalues} & R & ::= & n \mid r \mid [r+n] \mid [n] \\
\text{Instructions} & I & ::= & L := R \mid r \overset{\pm}{=} n \\
& & & \mid L := R \overset{?}{=} R \mid \mathsf{jnz}\ R, n \\
\text{Control-flow instructions} & J & ::= & \mathsf{halt}\ R \mid \mathsf{fail}\ R \mid \mathsf{jmp}\ R \\
\text{Basic blocks} & B & ::= & (I^*, J) \\
\text{Programs} & P & ::= & (B^*, B)
\end{array}$$

Figure 2. Target assembly language syntax

Our target language is the idealized assembly language shown in Figure 2. It differs from a real assembly language in representing words with natural numbers and in supporting an infinite memory bank of words. There is still a finite supply of N registers. Our particular compiler works for any $N \geq 3$, allocating some variables to the additional registers when possible. An assembly program consists of a list of basic blocks with one distinguished basic block where execution begins. A basic block is a sequence of instructions terminated by a control-flow instruction. The supported varieties of instruction are assignment using different addressing modes (where $[\cdot]$ operands denote memory accesses), increment of a register by a constant, equality comparison, and conditional jump based on whether a value is nonzero. Control-flow instructions include halt, for normal program termination; fail, for termination on an uncaught exception; and jmp, the standard "computed goto." Each halt or fail instruction takes an additional program result code as an argument. The destination operands to jnz and jmp are interpreted as indices into the program's list of basic blocks.

The compiler is idealized in another important way. Unlike in our past work on a compiler for basic lambda calculus (Chlipala 2007), there is no interface with a garbage collector. The output assembly programs allocate new memory but never free any memory. As our present focus is on reasoning about nested binders, we leave the low-level treatment of memory management to future work.

We give the source language a standard big-step operational semantics. Figure 3 shows a sampling of the rules. We define a separate syntactic class of *values* (associated with the metavariable v) in the usual way, taking a restriction of the syntax of expressions and adding a form $\mathsf{ref}(n)$, standing for an allocated reference cell

$$\frac{}{(h, \mathsf{fix}\ f(x).\ e) \Downarrow (h, \mathsf{Ans}(\mathsf{fix}\ f(x).\ e))}$$

$$\frac{(h_1, e_1) \Downarrow (h_2, \mathsf{Ans}(\mathsf{fix}\ f(x).\ e)) \quad (h_2, e_2) \Downarrow (h_3, \mathsf{Ans}(e')) \\ (h_3, e[f \mapsto \mathsf{fix}\ f(x).\ e][x \mapsto e']) \Downarrow (h_4, r)}{(h_1, e_1\ e_2) \Downarrow (h_4, r)}$$

$$\frac{(h_1, e_1) \Downarrow (h_2, \mathsf{Ex}(v))}{(h_1, e_1\ e_2) \Downarrow (h_2, \mathsf{Ex}(v))}$$

$$\frac{(h_1, e_1) \Downarrow (h_2, \mathsf{Ans}(\mathsf{fix}\ f(x).\ e)) \quad (h_2, e_2) \Downarrow (h_3, \mathsf{Ex}(v))}{(h_1, e_1\ e_2) \Downarrow (h_3, \mathsf{Ex}(v))}$$

$$\frac{(h_1, e) \Downarrow (h_2, \mathsf{Ans}(v))}{(h_1, \mathsf{ref}(e)) \Downarrow (v :: h_2, \mathsf{Ans}(\mathsf{ref}(|h_2|)))}$$

$$\frac{(h_1, e) \Downarrow (h_2, \mathsf{Ans}(\mathsf{ref}(n))) \quad h_2.n = v}{(h_1, !e) \Downarrow (h_2, \mathsf{Ans}(v))}$$

$$\frac{(h_1, e) \Downarrow (h_2, \mathsf{Ans}(v))}{(h_1, \mathsf{raise}(e)) \Downarrow (h_2, \mathsf{Ex}(v))}$$

$$\frac{(h_1, e_1) \Downarrow (h_2, \mathsf{Ex}(v)) \quad (h_2, e_2[x \mapsto v]) \Downarrow (h_3, r)}{(h_1, e_1\ \mathsf{handle}\ x \Rightarrow e_2) \Downarrow (h_3, r)}$$

Figure 3. Sample rules from source language semantics

at numerical address n. The basic judgment is $(h_1, e) \Downarrow (h_2, r)$, where h_1 and h_2 are reference cell heaps represented as lists of values, e is the expression to evaluate, and r is the result, which is either $\mathsf{Ans}(v)$ for normal termination with expression result v or $\mathsf{Ex}(v)$ when v was raised as an exception and not caught. There are 38 rules in total, covering all of the points in an expression's evaluation where an exception might be raised. We write $|h|$ for the length of a list h.

Our assembly language also has a big-step operational semantics. There are many equivalent ways of formalizing such a semantics; the specifics that we chose will not matter in what follows. The overall judgment is of the simple form $P \Downarrow h, r$, where h is the final heap and r is either $\mathsf{halt}(n)$ or $\mathsf{fail}(n)$.

Our final theorem says that compiled programs have the same observable behavior as their corresponding source programs. With our limited languages, a program can only exhibit one of three kinds of observable behavior: halting, failing, or diverging. The theorem we prove in this case study ignores non-termination. Our theorem is enough to translate facts about terminating source programs into facts about assembly programs, which is the main kind of verification of interest for a deterministic source language without I/O. We plan eventually to strengthen the final theorem by applying co-inductive big-step operational semantics (Leroy and Grall 2009) to prove that divergent source programs are also mapped to divergent assembly programs.

We write $\lfloor e \rfloor$ for the compilation of expression e. Stating the final theorem requires formalizing the "contract" between the compiler and the programmer. The compiler writer agrees to follow certain data layout conventions, but it is useful to leave some aspects of representation unspecified, to avoid unnecessary restrictions on

$$\frac{}{h \vdash \mathsf{fix}\ f(x).\ e \cong n} \quad \frac{}{h \vdash () \cong n} \quad \frac{}{h \vdash \mathsf{ref}(n) \cong n'}$$

$$\frac{}{h \vdash c \cong \lfloor c \rfloor} \quad \frac{h \vdash v_1 \cong h[n] \quad h \vdash v_2 \cong h[n+1]}{h \vdash (v_1, v_2) \cong n}$$

$$\frac{h[n] = 0 \quad h \vdash v \cong h[n+1]}{h \vdash \mathsf{inl}(v) \cong n} \quad \frac{h[n] = 1 \quad h \vdash v \cong h[n+1]}{h \vdash \mathsf{inr}(v) \cong n}$$

Figure 4. The compiler's data layout contract

optimization. We chose to give a full specification for all kinds of data besides functions and references. Other decisions are possible, with minimal impact on the proofs. A much more specific relation underlies our main inductive proofs, and we arrive at the visible contract simply by forgetting some details of the "real" relation.

The data layout contract is specified as a relation $h \vdash v \cong n$, parameterized by the final assembly-level heap h and relating source-level value v to assembly-level word n. Figure 4 gives the details. We overload the notation $\lfloor c \rfloor$ to denote the compilation of a source-level constant into a word. Our development is parameterized over an arbitrary injective function of this kind.

We lift this relation in the natural way to apply to program results.

$$\frac{h \vdash v \cong n}{h \vdash \mathsf{Ans}(v) \cong \mathsf{halt}(n)} \quad \frac{h \vdash v \cong n}{h \vdash \mathsf{Ex}(v) \cong \mathsf{fail}(n)}$$

Now our main theorem can be stated succinctly.

THEOREM 1 (Semantic correctness of compilation). *For any source program e, heap h, and result r, if $(\cdot, e) \Downarrow (h, r)$, then there exist h' and r' such that $\lfloor e \rfloor \Downarrow h', r'$ and $h' \vdash r \cong r'$.*

1.2 Outline

In the next section, we review the higher-order syntax representation scheme that we introduced in prior work. In Section 3, we present a new substitution-free approach to encoding operational semantics. Section 4 describes, with discussion of their correctness proofs, the main phases of our compiler: conversion from first-order to higher-order syntax, CPS translation, closure conversion, translation to three-address code, and assembly code generation. Section 5 discusses our verified optimizations: common subexpression elimination, dead code elimination, and register allocation. Section 6 gives some statistics about our implementation, Section 7 compares with related work, and we conclude in Section 8.

The case study that we describe in this paper is included in the directory `examples/Untyped` of the latest release of our Lambda Tamer library for compiler verification in Coq, available at

`http://ltamer.sourceforge.net/`

2. Parametric Higher-Order Abstract Syntax

In this section, we summarize key elements of our past work on representing programming language syntax. The following section presents new material that is crucial for scaling up to more realistic languages.

To formalize reasoning about languages with nested variable binders, one needs to settle on a binding representation. Such detail is often swept under the rug in pencil-and-paper proofs. Taking many informal presentations literally, we arrive at concrete representations like the one embodied in this Coq datatype definition for the abstract syntax of untyped lambda calculus.

```
Inductive exp : Type :=
  | Var : string -> exp
  | App : exp -> exp -> exp
  | Abs : string -> exp -> exp.
```

A type definition like this one does not implement usual conventions on its own. At a minimum, we need some well-formedness judgment characterizing when an expression is free of dangling variable references. This means that each of our proofs must include extra premises characterizing which expressions are well-formed with respect to which variable environments. If our formalization requires a notion of capture-avoiding substitution, we need to define one manually. We must also prove a fair number of extra lemmas about well-formedness and substitution. These lemmas must have different proofs for different object languages.

There are other so-called *first-order* representation schemes that alleviate this burden somewhat, including the de Bruijn index, nominal, and locally nameless styles that we mentioned earlier. Significant extra reasoning about freshness and/or well-formedness remains. An alternative is to use *higher-order abstract syntax (HOAS)* (Pfenning and Elliot 1988), which represents object language binders using meta language binders. This pseudo-Coq definition captures the way we revise concrete syntax to arrive at HOAS.

```
Inductive exp : Type :=
  | App : exp -> exp -> exp
  | Abs : (exp -> exp) -> exp.
```

For instance, we represent an application of the identity function to itself with `App (Abs (fun x => x)) (Abs (fun x => x))`. We encode the matching-up of binders with their uses by borrowing our meta language's facility for that kind of matching with anonymous functions.

We called this definition "pseudo-Coq" because Coq will not accept it. An inductive type is not allowed to be defined with a constructor that takes as input a function over the same type. Allowing this would be problematic because it would allow the coding of non-terminating programs, even without use of explicit recursive definitions, by taking advantage of the opportunity to write "exotic terms" that do not correspond to real lambda terms. Since Coq follows the Curry-Howard Isomorphism in identifying proofs with functional programs, non-termination corresponds to logical inconsistency, where any theorem can be "proved" spuriously with an infinite loop. Systems like Twelf avoid this problem by using weaker meta languages like LF (Harper et al. 1993) that crucially omit features like pattern-matching and recursion.

Even in general-purpose functional programming languages, HOAS terms are difficult to deconstruct manually. It is not generally possible to "go under a binder," since languages like ML and Haskell provide no way to introspect into closures at runtime. Washburn and Weirich (2008) proposed a technique for fixing some of these deficiencies by taking advantage of parametric polymorphism. Guillemette and Monnier (2008) showed how the technique can be combined with generalized algebraic datatypes to do static verification of compiler type preservation in GHC Haskell.

Washburn and Weirich's encoding still is not accepted literally by Coq, but a small variation achieves similar benefits. In past work (Chlipala 2008), we showed how to use parametric higher-order abstract syntax (henceforth abbreviated "PHOAS") to formalize the syntax and semantics of programming languages. We were able to construct very simple, highly-automated proofs of the correctness of some transformations on functional programs.

Here is the syntax of lambda calculus reformulated in PHOAS.

```
Section var.
  Variable var : Type.

  Inductive exp : Type :=
    | Var : var -> exp
    | App : exp -> exp -> exp
    | Abs : (var -> exp) -> exp.
End var.

Definition Exp : Type := forall var : Type, exp var.
```

We use Coq's section mechanism to scope a local variable over a definition. Outside of the section, the `exp` type becomes a type family parameterized by a choice of `var` type. This definition *is* accepted by Coq, and the crucial difference from HOAS is that a binder is represented as a function over *variables*, rather than over expressions. This satisfies Coq's positivity constraint for inductive definitions. Now the identity function can be written as `Abs (fun x => Var x)`, with some choice of `var` fixed globally.

Considering just the part of the above code inside the section, we are using the encoding known as weak HOAS (Honsell et al. 2001). If we treat the type `var` as an unknown, Coq's type system ensures that every `exp` corresponds to a real lambda term, since it is not possible for a function over variables to do anything interesting based on its input values, which in effect come from an abstract type. The parametric part of PHOAS comes in treating `var` as more than just a global unknown. We define our final expression representation type `Exp` such that an expression is a first-class polymorphic function from a choice of `var` to an `exp` that uses that choice. For instance, the final PHOAS form of the identity function is `fun var => Abs var (fun x : var => Var var x)`. The parametricity of the meta language makes this scheme equivalent to treating `var` as a global constant, but we gain the ability to instantiate `Exp`s with particular `var` choices to help us write particular functions.

For instance, we can implement capture-avoiding substitution like this:

```
Section flatten.
  Variable var : Type.

  Fixpoint flatten (e : exp (exp var)) : exp var :=
    match e with
      | Var e' => e'
      | App e1 e2 => App (flatten e1) (flatten e2)
      | Abs e1 => Abs (fun x => flatten (e1 (Var x)))
    end.
End flatten.

Definition Exp1 := forall var : Type, var -> exp var.
Definition Subst (E : Exp1) (E' : Exp) : Exp :=
  fun var => flatten (E (exp var) (E' var)).
```

First, we write a function `flatten` that "flattens" an expression where variables are themselves represented as expressions. Every variable is replaced by the expression that it holds. We recurse inside an `Abs` constructor by building a new argument for `Abs` that itself calls `flatten`. The recursive call is on the original function body applied to a particular locally-bound variable.

We can use `flatten` to implement substitution easily. We define `Exp1`, the PHOAS type of an expression with one free variable. Substitution of `E'` in `E` is implemented with an anonymous polymorphic function over a `var` choice. For a particular `var`, we instantiate the one-hole expression `E` to represent variables as expressions and the substitutand `E'` to represent variables with `var`. Applying the former to the latter, we arrive at an expression whose flattening is the proper result of substitution.

In past work (Chlipala 2008), we showed how to use PHOAS to give denotational semantics to statically-typed, strongly-normalizing functional languages. The basic trick is to parameterize variables by types and define a type denotation function that can be used as a variable representation in implementing the expression denotation function. Using this encoding, we implemented and proved totally correct a number of common functional language compiler passes. These proofs usually rely on the fact that values of types like `Exp` are "really parametric." We formalized this property in terms of a judgment that axiomatizes equivalence between expressions that use different variable representations. This judgment for untyped lambda calculus is $\Gamma \vdash e_1 \sim e_2$ as defined below. Where e_i represents variables in $\mathtt{var}_i$, Γ is a context of pairs in $\mathtt{var}_1 \times \mathtt{var}_2$. We write $\#x$ for the `Var` constructor applied to x and λf for `Abs` applied to f.

$$\frac{(x_1, x_2) \in \Gamma}{\Gamma \vdash \#x_1 \sim \#x_2} \quad \frac{\Gamma \vdash e_1 \sim e_1' \quad \Gamma \vdash e_2 \sim e_2'}{\Gamma \vdash e_1\ e_2 \sim e_1'\ e_2'}$$

$$\frac{\forall x_1, x_2.\ \Gamma, (x_1, x_2) \vdash f_1(x_1) \sim f_2(x_2)}{\Gamma \vdash \lambda f_1 \sim \lambda f_2}$$

A parametric expression E is well-formed if, for any $\mathtt{var}_1$ and $\mathtt{var}_2$, we have $\cdot \vdash E(\mathtt{var}_1) \sim E(\mathtt{var}_2)$. We conjecture that this statement holds true for any E of the proper type, and we asserted that fact as an axiom in our past work. In the case study of this paper, we instead prove that expressions are well-formed as needed. Future theoretical work that proved the consistency of this family of axioms would remove the need for specialized well-formedness proofs.

In proving the correctness of a program transformation, it is generally the case that we use one variable choice in evaluating a source program and a different variable choice in translating the program. We use the well-formedness of the expression to derive that the two instantiated expressions are equivalent. Proofs then tend to proceed by induction or inversion on these concrete well-formedness derivations.

At this point, the reader may want to accuse us of misleading advertising, since earlier we complained that first-order representations require too much bookkeeping about well-formedness. The key difference with PHOAS is that (we conjecture) every expression is well-formed by construction. We materialize well-formedness proofs only as needed, and we never need to prove PHOAS analogues of common lemma schemas like substitution, weakening, and permutation. In this case study, we proved well-formedness manually where needed as a kind of due diligence, but we anticipate that the theory will eventually be in place to rest easily assuming axioms of universal well-formedness. In any case, PHOAS avoids the need to generate fresh names or rearrange existing names in implementing a wide variety of transformations. These administrative operations are the bane of programming and proving with first-order representations.

3. Substitution-Free Operational Semantics

Our past work gives programs semantics by interpretation into Coq's strongly-normalizing logic CIC; thus, that work cannot be applied directly to Turing-complete programming languages like our source and target languages here. The main representational innovation of our new work is an effective way of writing operational semantics over PHOAS terms. Operational semantics has proved its worth in the formalization of a wide variety of languages, so our new encoding expands the effective range of PHOAS dramatically.

It is tempting to write operational semantics directly over parametric terms (e.g., in `Exp` from the last section). Doing so is actually fairly straightforward, with the trick for implementing substitution as the one surprise. Unfortunately, inductive proofs over parametric terms tend to involve just as much administrative overhead as we find with first-order representations. Dealing with instantiated terms (e.g., in `exp`) frees us to leave variables deep within syntax trees annotated with arbitrary meta language values. If we work with parametric terms, we must instead represent and apply contexts explicitly in our induction hypotheses, since it is impossible to "go under a binder" without first fixing a `var` choice.

Moreover, working with substitution explicitly brings back the same family of lemmas about the interaction of substitution and other functions. Generally we must prove at least one lemma about substitution for each program transformation function that we write. The details of such lemmas are almost always elided in pencil-and-paper proofs, but we must prove them in full detail to satisfy a proof assistant.

The alternative that we use here is to define an operational semantics over instantiated terms that avoids mentioning substitution explicitly. Our encoding has the flavor of a hybrid between a high-level semantics and an abstract machine, where we track closure allocation explicitly. We want to write semantics equivalent to examples like the one in Figure 3. For basic lambda calculus, it is tempting to start out by defining a type `val` of values like the following, such that we can instantiate `var` as `val` in our semantics.

```
Inductive val : Type :=
  | VAbs : (val -> exp val) -> val.
```

This definition suffers from the same problem as our earlier HOAS pseudo-definition: we try to define `val` in terms of functions over itself. Coq rejects the definition as ill-formed, which is a good thing, because otherwise we would be able to implement a lambda calculus interpreter in Coq, which gives us a trivial way of coding an infinite loop and thus breaking logical consistency.

Our solution is to represent values as natural numbers that index into a heap of "closures," or meta language functions from values to expressions. The technique bears a resemblance to approaches to making allocation explicit in operational semantics, e.g., as in Morrisett et al. (1995). That line of work aims to capture how high-level programs execute on real machines, while keeping at the right level of abstraction. In contrast, our use of explicit allocation is aimed at removing the need to reason about explicit substitution. What we have here is really just an instance of a common pattern in semantics of moving to more explicitly syntactic techniques to circumvent circularities in type definitions.

```
Definition val : Type := nat.
Definition closure : Type := val -> exp val.
Definition closures : Type := list closure.
```

Here is a big-step semantics in this style for basic lambda calculus. Its Coq type signature could be `closures -> exp val -> closures -> val -> Prop`.

$$\frac{}{(H, \#v) \Downarrow (H, v)} \quad \frac{}{(H, \lambda f) \Downarrow (f :: H, |H|)}$$

$$\frac{(H_1, e_1) \Downarrow (H_2, n) \quad (H_2, e_2) \Downarrow (H_3, v) \quad H_3.n = f \quad (H_3, f(v)) \Downarrow (H_4, v')}{(H_1, e_1\ e_2) \Downarrow (H_4, v')}$$

As with HOAS and its relatives in general, we manage to delegate the object-language-specific handling of substitution to the meta language. This delegation happens in the occurrence of $f(v)$ in the application rule. The rule for variables is also more interesting than it may appear at first. By evaluating a variable node $\#v$ to its content v, we effectively push the operation of last section's `flatten` function into our semantics.

The trickiness of usual substitution stems from the need to reason about nested binder scopes. We have replaced that kind of reasoning with global reasoning about a closure heap. We can prove a relatively small set of lemmas about lists and reuse it to handle closure heaps in all developments that use our encoding. There is no need to prove even a single lemma about substitution upon starting with a new object language or transformation.

Our technique generalizes to the full source language from Figure 1. We revise our `val` definition like this, using the illustrative type synonym `label` for `nat` from our library:

```
Inductive val : Type :=
  | VFunc : label -> val
  | VUnit : val
  | VPair : val -> val -> val
  | VInl : val -> val
  | VInr : val -> val
  | VRef : label -> val.
```

The main PHOAS semantics for the source language tracks input and output versions of both a closure heap and a reference heap. We reuse our library of list lemmas to reason about both kinds of heaps.

The definitions above are about expressions specialized to represent variables as values, but it is now easy to define an evaluation relation for parametric terms.

$$\frac{(\cdot, E(\texttt{val})) \Downarrow (H', v)}{E \Downarrow v}$$

We have modified standard operational semantics by adding a level of indirection. In a traditional paper proof at the traditional level of detail, our change would only add bureaucratic hassle. Counterintuitively, the change *reduces* hassle in mechanized proofs, since it helps us delegate to the meta language some details of processing the object language.

The substitution-free encoding is more than just a trick to place PHOAS on a level playing field with first-order representations. PHOAS with our new semantic encoding compares very favorably with other known combinations of syntactic and semantic encodings. As far as we are aware, every competing technique is either invalid in a general-purpose proof system or leads to proof overhead significantly above that in our implementation. In verifying all of our translations that represent syntax solely with PHOAS, there is not a single lemma establishing one of the standard object-language-specific syntactic properties. We only once use proof by induction over the structure of programs, and that occurs for an auxiliary lemma relevant to closure conversion, where explicit reasoning about variables is hard to avoid.

Substitution-free operational semantics has much in common with environment semantics, where an evaluation judgment takes an additional input which assigns a value to each free variable of the expression to evaluate. Both approaches involve explicit first-order treatment of an aspect of evaluation that is implicit in the more common varieties of natural semantics. Where environment semantics treats every variable in a first-order way, substitution-free semantics does the same with closures. We have found the latter to have serious advantages for proof engineering. None of our translations includes more than one case that has any interesting effect on closure allocation sequence. As a result, none of our proofs includes more than one case that must compensate for the effect of such a change on semantics. In contrast, with environment semantics, almost every case of each of our translations would need some accounting for a rearrangement of variable binding structure.

Theorem-specific proof automation is one of the core techniques in our approach to compiler verification, and we will have more to say later on the details of that automation. While first-order encodings of operational semantics usually mention substitution explicitly, the standard lemmas about substitution tend to admit simple automated proof strategies. Nonetheless, we feel it is still a significant burden to have to state and apply all of these lemmas explicitly. Perhaps an automation package could go further and apply substitution properties automatically as needed. In the project described in this paper, we have avoided any automatic application of induction, which rules out proofs of the usual syntactic lemmas. We view our more elementary automation style as a mark in favor of our proposal.

More standard operational semantics with explicit substitution is easier to understand and believe, so we chose to state the final theorem independently of the new technique. The first part of the next section shows how we convert from first-order to higher-order syntax and semantics, along with how we justify the soundness of the conversion.

4. Main Compiler Phases

In this section, we walk through our compiler intermediate languages and the phases that translate into them. We will overload notation by writing $\lfloor \cdot \rfloor$ for the compilation function being discussed in each subsection.

For lack of space, we discuss our proof automation techniques only in the context of CPS translation, in Section 4.2.2. The design patterns introduced there apply equally well to the other translations.

4.1 PHOASification

Our final theorem is stated entirely in terms of standard encodings of syntax and semantics, with no mention of PHOAS. We achieve this by beginning our compiler with a de Bruijn index (de Bruijn 1972) implementation of the syntax from Figure 1. The first compiler phase translates these programs into PHOAS equivalents, where the PHOAS syntax is a different encoding of Figure 1.

It is not generally possible to "cheat" in implementing a translation into PHOAS, in the sense that there is no default value to use for variables when it turns out that the input program is ill-formed somehow. Therefore, we use dependent types to enforce that every source program is closed by construction. This is a standard technique in the dependent types world, where a type `exp` is indexed by a natural number expressing how many free variables are available. Variables in ASTs are represented in types `fin n`, which are isomorphic to sets of natural numbers below `n`.

Our implementation of PHOASification uses another standard dependent type family, which we call `ilist` in our library. For a type `T` and a natural number `n`, an `ilist T n` value is a length-`n` list of values in `T`.

The type of the main translation is `forall (var : Type) (n : nat), Source.exp n -> ilist var n -> Core.exp var`. Besides the expression to translate, $\lfloor \cdot \rfloor$ takes in a list representing a mapping from de Bruijn indices to PHOAS variables. Here are some representative cases from the function's definition, where we write $\sigma.f$ for the projection from the `ilist` σ of the value at the position indicated by `fin` value f. We write $\hat{\lambda}$ for the meta language's function abstraction.

$$\begin{array}{rcl}
\lfloor \#x \rfloor \, \sigma & = & \#(\sigma.x) \\
\lfloor e_1 \; e_2 \rfloor \, \sigma & = & \lfloor e_1 \rfloor \, \sigma \; \lfloor e_2 \rfloor \, \sigma \\
\lfloor \mathsf{fix} \; f(x). \; e_1 \rfloor \, \sigma & = & \mathsf{fix}(\hat{\lambda} f. \hat{\lambda} x. \; \lfloor e_1 \rfloor \, (x :: f :: \sigma))
\end{array}$$

In the source language definition, we simplify the definition of substitution by defining a type of values and including an `exp` constructor that allows the injection of any value into any type `exp n`. Thus, the closed nature of a value is apparent from its type, so we avoid needing to lift de Bruijn indices in the process of substituting a value in an open term.

4.1.1 Correctness Proof

We define two mutually-inductive relations, for characterizing the compatibility of source and PHOAS expressions and values. Both relations are parameterized by PHOAS-level closure heaps, and the expression relation is also parameterized by an `ilist`, like in the definition of the translation. Here are a few representative cases. We write \$ for the constructor that injects source values into source expressions, and we write $H \rightsquigarrow H'$ for the fact that H is a suffix of H'. We use the notation $\mathsf{fix}\ F$ for PHOAS-level application of the AST constructor for recursive functions.

$$\frac{H \vdash \sigma.f \simeq v}{H, \sigma \vdash \#f \simeq \#v} \qquad \frac{H, \sigma \vdash e_1 \simeq e_1' \quad H, \sigma \vdash e_2 \simeq e_2'}{H, \sigma \vdash e_1\ e_2 \simeq e_1'\ e_2'}$$

$$\frac{\forall f, x.\ H, x :: f :: \sigma \vdash e \simeq F(f)(x)}{H, \sigma \vdash \mathsf{fix}\ f(x).\ e \simeq \mathsf{fix}\ F} \qquad \frac{H \vdash v \simeq v'}{H, \sigma \vdash \$v \simeq \#v'}$$

$$\frac{\begin{array}{c} H.n = f \\ \forall x_1, x_2, H'.\ H \rightsquigarrow H' \Rightarrow H', x_2 :: x_1 :: \cdot \vdash e \simeq f(x_1)(x_2) \end{array}}{H \vdash \mathsf{Fix}(e) \simeq \mathsf{Fix}(n)}$$

$$\frac{H \vdash v_1 \simeq v_1' \quad H \vdash v_2 \simeq v_2'}{H \vdash \mathsf{Pair}(v_1, v_2) \simeq \mathsf{Pair}(v_1', v_2')}$$

We prove that both relations are monotone, with respect to replacing a heap H with another heap H' such that $H \rightsquigarrow H'$. We also lift the value relation to apply over results $\mathsf{Ans}(\cdot)$ and $\mathsf{Ex}(\cdot)$ in the natural way.

There are two main lemmas behind the correctness theorem for this phase. First, we prove that the compilation function respects expression compatibility.

LEMMA 1. *For any e and σ with compatible type indices, if e contains no uses of \$, then $\cdot, \sigma \vdash e \simeq \lfloor e \rfloor\ \sigma$.*

From this starting point, we track the parallel execution of e and its compilation. We use a notion of reference heap compatibility $H \vdash h \simeq h'$, which says that h and h' have the same length and that their values belong pairwise to $H \vdash \cdot \simeq \cdot$.

LEMMA 2. ***If:***

- *$(h_1, e) \Downarrow (h_2, r)$ at the source level,*
- ***And** $H, \sigma \vdash e \simeq e'$,*
- ***And** $H \vdash h_1 \simeq h_1'$,*

***Then** there exist H', h_2', and r' such that:*

- *$(H, h_1', e') \Downarrow (H', h_2', r')$,*
- ***And** $H' \vdash r \simeq r'$,*
- ***And** $H' \vdash h_2 \simeq h_2'$.*

Lemma 2 appeals to an auxiliary lemma about substitution. We note that this is the only place in our development where a substitution theorem is proved explicitly.

These lemmas together yield the final theorem directly.

THEOREM 2. *If $(\cdot, e) \Downarrow (h, r)$ and e is closed and does not use \$, then there exist H, h', and r' such that $(\cdot, \cdot, \lfloor e \rfloor \cdot) \Downarrow (H, h', r')$ and $H \vdash r \simeq r'$.*

4.2 Conversion to Continuation-Passing Style

The first main compiler phase translates programs into continuation-passing style. Functions no longer return, explicit exception handling constructs are eliminated, and expression evaluation is broken up with sequences of let bindings of the results of primitive operations on variables. Figure 5 shows the syntax of the translation's target language.

$$\begin{array}{llll}
\text{Primops} & p & ::= & c \mid x = x \mid \mathsf{fix}\ f(x).\ e \\
& & & \mid () \mid \langle x, x \rangle \mid \mathsf{fst}(x) \mid \mathsf{snd}(x) \\
& & & \mid \mathsf{inl}(x) \mid \mathsf{inr}(x) \mid \mathsf{ref}(x) \mid !x \mid x := x \\
\text{Expressions} & e & ::= & \mathsf{halt}(x) \mid \mathsf{fail}(x) \mid x\ x \mid \mathsf{let}\ x = p\ \mathsf{in}\ e \\
& & & \mid \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}(x) \Rightarrow e \mid \mathsf{inr}(x) \Rightarrow e
\end{array}$$

Figure 5. CPS language syntax

$$\begin{array}{rcl}
\lfloor \#x \rfloor\ k_S k_E & = & k_S(x) \\
\lfloor \mathsf{raise}(e) \rfloor\ k_S k_E & = & x \overset{k_E}{\leftarrow} e; k_E(x) \\
\lfloor \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rfloor\ k_S k_E & = & x \overset{k_E}{\leftarrow} e_1; \lfloor e_2 \rfloor\ k_S k_E \\
\lfloor e_1\ e_2 \rfloor\ k_S k_E & = & f \overset{k_E}{\leftarrow} e_1; x \overset{k_E}{\leftarrow} e_2; \\
& & \mathsf{let}\ k_S' = \lambda r.\ k_S(r)\ \mathsf{in} \\
& & \mathsf{let}\ k_E' = \lambda r.\ k_E(r)\ \mathsf{in} \\
& & \mathsf{let}\ p' = \langle k_S', k_E' \rangle\ \mathsf{in} \\
& & \mathsf{let}\ p = \langle x, p' \rangle\ \mathsf{in}\ f\ p
\end{array}$$

Figure 6. CPS translation

We use a higher-order one-pass CPS translation, in the style of Danvy and Filinski (1992). The type of the translation is `forall var, Core.exp var -> (var -> CPS.exp var) -> (var -> CPS.exp var) -> CPS.exp var`. Beyond the input expression, the extra arguments are the current success continuation and the current exception handler, represented as meta language functions over result variables. Figure 6 shows some representative cases of the definition. We write $\lambda x.\ e$ as shorthand for $\mathsf{fix}\ f(x).\ e$ when f does not occur free in e. We write $x \overset{kE}{\leftarrow} e_1; e_2$ as shorthand for $\lfloor e_1 \rfloor\ (\hat{\lambda} x.\ e_2)(k_E)$. The application case demonstrates how the effective domain of each core function is expanded to 3-tuples of a main argument, a success continuation, and an exception handler.

This compilation function takes as an argument a choice of `var` representation. In compiling a parametric expression E, we return an abstraction over `var`, within which we call the concrete compilation function with `var` and $E($`var`$)$ as arguments. We pass always-halt and always-fail functions as the success continuation and exception handler, respectively.

4.2.1 Correctness Proof

As for the last phase, this correctness proof is based around a relation between core and CPS values. Since both languages use PHOAS, the relation is parameterized by a closure heap for each. Here are some representative rules. For a meta language function f representing a function abstraction body, we write $\lfloor f \rfloor$ for the way that the main compilation translates that body. The relation definition below depends on a version of the $\sim$ relation from Section 2, extended to apply to the input language.

$$\frac{\begin{array}{c} H.n = f \quad H'.n' = \lfloor f' \rfloor \\ (\forall x_1, x_1', x_2, x_2'.\quad \Gamma, (x_1, x_1'), (x_2, x_2') \\ \qquad\qquad \vdash f(x_1)(x_2) \sim f'(x_1')(x_2')) \\ (\forall v, v'.\ (v, v') \in \Gamma \Rightarrow H, H' \vdash v \simeq v') \end{array}}{H, H' \vdash \mathsf{Fix}(n) \simeq \mathsf{Fix}(n')}$$

$$\frac{H, H' \vdash v_1 \simeq v_1' \quad H, H' \vdash v_2 \simeq v_2'}{H, H' \vdash \mathsf{Pair}(v_1, v_2) \simeq \mathsf{Pair}(v_1', v_2')} \qquad \frac{}{H, H' \vdash \mathsf{Ref}(n) \simeq \mathsf{Ref}(n)}$$

As in the last subsection, we lift this relation in the natural way to pairs of reference heaps and pairs of results.

Our main theorem is with respect to a substitution-free big-step semantics for CPS programs. The signature is the same as for Core but with final heaps no longer specified, since the final result is all that we care about.

THEOREM 3. ***If:***

- $(H_1, h_1, e) \Downarrow (H_2, h_2, r)$ *at the source level,*
- ***And*** $\Gamma \vdash e \sim e'$,
- ***And*** $H_1, H_1' \vdash h_1 \simeq h_1'$,
- ***And****, for every* $(v, v') \in \Gamma$, $H_1, H_1' \vdash v \simeq v'$,

Then *for every pair of continuations* k_S *and* k_E, ***there exist*** H_2', h_2', *and* r' *such that:*

- ***If*** $r' = \mathsf{Ans}(v)$ *and* $(H_2', h_2', k_S(v)) \Downarrow r''$,
  - ***then*** $(H_1', h_1', \lfloor e' \rfloor\, k_S k_E) \Downarrow r''$,
- ***And, if*** $r' = \mathsf{Ex}(v)$ *and* $(H_2', h_2', k_E(v)) \Downarrow r''$,
  - ***then*** $(H_1', h_1', \lfloor e' \rfloor\, k_S k_E) \Downarrow r''$,
- ***And*** $H_1' \rightsquigarrow H_2'$,
- ***And*** $H_2, H_2' \vdash r \simeq r'$,
- ***And*** $H_2, H_2' \vdash h_2 \simeq h_2'$.

This theorem about expressions specialized to values-as-variables makes it easy to derive the theorem about parametric expressions when we substitute the initial success and exception continuations for k_S and k_E.

4.2.2 Automating the Proofs

We begin by proving that each of our compatibility relations is monotone with respect to extension of closure heaps, which follows by induction on derivations. We also give one-liner proofs for five more lemmas that massage "obvious" facts into forms that Coq's automated resolution prover will be able to use. At that point, we are ready to tackle the proof of Theorem 3, which proceeds by induction on core evaluation judgments.

Figure 7 gives the complete proof script for this theorem, implementing in Coq's domain-specific tactic language Ltac (Delahaye 2000). We will step through the different elements, remarking as appropriate on the design patterns they embody.

The script begins with hints, which extend Coq's resolution prover, supporting higher-order logic programming in the tradition of Prolog. The first hint suggests that proof search should try each of the rules of the evaluation judgment for CPS-level primops. In this way, we avoid having to mention the rules explicitly, which makes it possible for the proof to keep working even after we add new kinds of primops.

```
Hint Constructors CPS.evalP.
```

Next, we use the `Hint Resolve` command to suggest some other rules and lemmas to be applied automatically during resolution.

```
Hint Resolve answer_Ans answer_Ex ....
```

We use a `Hint Extern` command to specify a free-form proof search step. We give `1` as an estimate of the cost of this rule, which effects the order in which rules are attempted. After that, we write a pattern to match against a goal. When the goal matches the pattern, we suggest running the proof script to the right of the arrow. In this case, our script suggests unfolding some definitions, so that we expose the syntactic structure of an expression to evaluate, making it clear which operational semantics rules apply.

```
Hint Extern 1 (CPS.eval _ _ (cpsFunc _ _) _) =>
  unfold cpsFunc, cpsFunc'.
```

Now we are ready for the main body of the proof. We proceed by induction on the first hypothesis $(H_1, h_1, e) \Downarrow (H_2, h_2, r)$, and we chain onto our use of induction a script to apply to every inductive case. The semicolon operator accomplishes this chaining, and we wrap the per-case script with `abstract` to prove every case as a separate lemma, which saves memory by freeing some temporary data structures after each lemma.

```
induction 1; abstract (...
```

We begin every case with an inversion on the expression equivalence judgment $\Gamma \vdash e \sim e'$, which is the first remaining hypothesis.

```
inversion 1;
```

Next, we call a generic simplification tactic from our Lambda Tamer library. This tactic knows nothing about any particular object language. It relies on a number of built-in Coq automation tactics and adds some new strategies, combining propositional simplification, partial evaluation, resolution proving, rewriting, and common rules for simplifying sets of hypotheses dealing with standard datatypes like natural numbers, lists, and optional values.

```
simpler;
```

At this point, the top-level structure of the expressions appearing in a case is known, and we have gotten as far as we can with generic simplification. The next step is to begin a loop over some theorem-specific simplification strategies. Like other tactic-based proof assistants, Coq supports a number of *tacticals*, a kind of higher-order combinators for assembling new proof strategies. We use the `repeat` tactical to structure our loop. The argument to `repeat` is a tactic to attempt repeatedly until it no longer applies. Our argument here uses a `match` tactic expression, which generalizes normal pattern matching in the tradition of ML and Haskell. A `match` tactic matches on the form of a proof goal, including both hypotheses and conclusion.

```
repeat (match goal with
```

Our heuristics are pattern-matching rules, where each pattern has the form `HYPS |- CONC`. The `HYPS` section describes conditions on hypotheses and `CONC` gives a pattern to match against the conclusion to be proved. The former section is a comma-separated list of zero or more entries of the form `H : p`, asserting that there must exist some hypothesis matching pattern `p` and to whose name the local variable `H` should be bound.

The first heuristic looks for a hypothesis asserting some fact $H, H' \vdash v_1 \simeq v_2$. In our implementation, such a fact is written as `H & H' |-- v1 ~~ v2`, using an ASCII notation that we register as a Coq syntax extension or "macro." Thus, the first pattern matches any goal with a hypothesis over this judgment. For each such hypothesis, we apply the tactic `invert_1_2` from our library. This tactic performs inversion if and only if it is possible to deduce from the form of the hypothesis that at most two distinct rules of the underlying judgment could apply. If more than two rules are possible, the tactic invocation fails, triggering backtracking to try a different choice of `H` or, if that fails, the next rule in our `match` expression. By using `invert_1_2`, we avoid having to specialize this heuristic to the details of our object language, for instance by writing one heuristic per case where we can deduce that a particular rule of $\cdot, \cdot \vdash \cdot \simeq \cdot$ must have been used to conclude `H`.

```
| [ H : _ & _ |-- _ ~~ _ |- _ ] => invert_1_2 H
```

The next heuristic follows the logic of the previous one, but for the judgment for result compatibility instead of value compatibility, which has a different notation.

```
| [ H : _ & _ |--- _ ~~ _ |- _ ] => invert_1 H
```

```
Hint Constructors CPS.evalP.
Hint Resolve answer_Ans answer_Ex CPS.EvalCaseL CPS.EvalCaseR EquivRef'.
Hint Extern 1 (CPS.eval _ _ (cpsFunc _ _) _) => unfold cpsFunc, cpsFunc'.

induction 1; abstract (inversion 1; simpler;
  repeat (match goal with
            | [ H : _ & _ |-- _ ~~ _ |- _ ] => invert_1_2 H
            | [ H : _ & _ |--- _ ~~ _ |- _ ] => invert_1 H
            | [ H : forall G e2, Core.exp_equiv G ?E e2 -> _ |- _ ] =>
              match goal with
                | [ _ : Core.eval ?S _ E _ _ _,
                    _ : Core.eval _ _ ?E' ?S _ _,
                    _ : forall G e2, Core.exp_equiv G ?E' e2 -> _ |- _ ] => fail 1
                | _ => match goal with
                         | [ k : val -> expV,
                             ke : val -> exp val,
                             _ : _ & ?s |-- _ ~~ _,
                             _ : context[VCont] |- _ ] =>
                           guessWith ((fun (_ : val) x => ke x) :: (fun (_ : val) x => k x) :: s) H
                         | _ => guess H
                       end
              end
          end; simpler);
  try (match goal with
         | [ H1 : _, H2 : _ |- _ ] => generalize (sall_grab H1 H2)
       end; simpler);
  splitter; eauto 9 with cps_eval; intros;
    try match goal with
          | [ H : _ & _ |--- _ ~~ ?r |- answer ?r _ _ ] => inverter H; simpler; eauto 9 with cps_eval
        end).
```

Figure 7. Complete proof script for Theorem 3

The next and final heuristic from our main loop chooses when and how to apply induction hypotheses (IHes). The first step in that direction is to identify some hypothesis H that has the right syntactic structure to be an IH. Our Coq development uses the predicate Core.exp_equiv for the judgment we write as $\Gamma \vdash e \sim e'$ in the statement of Theorem 3, and the code ?E denotes a pattern variable.

```
| [ H : forall G e2,
        Core.exp_equiv G ?E e2 -> _ |- _ ] =>
```

It would not be effective to apply the IHes in an arbitrary order. Because of the form of Theorem 3, each successful application yields an existentially-quantified conclusion, and eliminating those quantifiers gives us new variables to work with. Those new variables might be needed to instantiate the *universal* quantifiers of a different IH. It turns out that a simple heuristic lets us choose the right IH order in every case: as each IH is associated with an expression, follow the order in which those expressions were evaluated in the original program.

We can track "evaluation order" by inspecting the closure heaps that are threaded through evaluation. One evaluation "comes after" another if the former's starting closure heap equals the latter's ending heap. By clearing each IH as we use it, we make it possible to use the following pattern match to identify an IH that is "not ready yet." In particular, where the current H is for some expression E, there must exist another IH about some expression E', such that evaluation of E begins where evaluation of E' leaves off, in terms of the flow of a closure heap S. Thus, since E comes after E', and since we have not yet applied the IH for E', we are not yet ready to apply the IH for E. We use the fail tactic to backtrack to making a different choice of H.

```
match goal with
  | [ _ : Core.eval ?S _ E _ _ _,
      _ : Core.eval _ _ ?E' ?S _ _,
      _ : forall G e2, Core.exp_equiv G ?E' e2 -> _
      |- _ ] => fail 1
```

If the last rule finds no matches, then we know that H is the appropriate IH to apply now. We perform a further pattern-match to determine whether we need to apply an instantiation strategy specific to the case of function application, one of the few interesting cases of the translation. Since the general case is simpler, we will discuss it first. We simply apply the tactic guess, which comes from our Lambda Tamer library, to our IH. This generates a new *unification variable* for every universal quantifier in the statement of H. Additionally, we apply automatic resolution proving to discharge each hypothesis of H, in the process learning the values of most of the unification variables that we just introduced. For this theorem, unification variables remain for the continuation variables k_S and k_E, but the other unification variables are determined immediately from context. By relying on the versatile guess tactic, we avoid almost all object-language-specific application of IHes. This is one of the key techniques supporting proof reuse.

```
  | _ => guess H
```

In most proofs, guess can handle the "uninteresting" cases that would not be written out in detail in a pencil-and-paper proof. Often a bit more help from the human prover is needed for the cases that lie at the heart of a transformation's purpose. For CPS translation, the only such case is function application, and we use pattern matching to identify that case and treat it specially. We use the pattern context[VCont] to require that some hypothesis mentions a continuation value, which turns out to be enough to isolate the case of interest. We also bind local names for the success continuation `k` and the exception handler `ke`. Finally, we pattern-match out the only closure heap `s` that is mentioned in a value compatibility hypothesis.

From these variables, we can construct our piece of advice to `guess`. More specifically, we use the variant `guessWith`, which lets us suggest a value to be used to instantiate any universal quantifier of proper type, such that the remaining quantifiers are still instantiated with fresh unification variables. We know that each function call allocates a new continuation each for the success continuation and exception handler. The argument we pass to `guessWith` reflects that knowledge, suggesting a closure heap that has the two new continuations pushed on.

```
match goal with
  | [ k : val -> expV,
      ke : val -> exp val,
      _ : _ & ?s |-- _ ~~ _,
      _ : context[VCont] |- _ ] =>
      guessWith ((fun (_ : val) x => ke x)
        :: (fun (_ : val) x => k x) :: s) H
```

Each iteration of the main loop ends with a call to `simpler`, which will take the existentially-quantified conjunctions produced by `guess` and replace them with individual hypotheses that use fresh top-level variables.

After we finish this main loop of heuristics, most of the work in the proof is done. Simple resolution proving can handle most of the remaining goals. We use one additional pattern match to catch a case where it would be useful to add a new hypothesis justified by the library theorem `sall_grab` about heap well-formedness.

```
try (match goal with
       | [ H1 : _, H2 : _ |- _ ] =>
         generalize (sall_grab H1 H2)
     end; simpler);
```

After that, we call the tactic `splitter` to turn a goal like $\exists x_1, \ldots, x_n.\ \phi_1 \wedge \ldots \wedge \phi_m$ into separate goals $\phi_1, \ldots, \phi_m$, with each x_i replaced by a fresh unification variable. We solve most of these goals with a call to the resolution prover `eauto`, specifying a proof tree depth of 9 and an additional hint database `cps_eval`, which includes a rule to apply as many CPS operational semantics rules as possible, counted as a single proof step.

```
splitter; eauto 9 with cps_eval;
```

Each remaining goal is solved by case analysis on whether an unknown evaluation result `r` is normal or represents an uncaught exception. More specifically, we find a hypothesis stating a result compatibility fact, we perform inversion on that hypothesis, and we finish off the resulting cases with standard tactics.

```
try match goal with
      | [ H : _ & _ |--- _ ~~ ?r
          |- answer ?r _ _ ] =>
        inverter H; simpler; eauto 9 with cps_eval
    end).
```

Our inductive proof has 38 cases to consider, with one for each semantic rule. Many of these cases need this kind of further split on results of sub-evaluations. By using automation to structure our proof script, we shield the human proof architect from the need to consider these many cases individually.

4.3 Closure Conversion

The next compiler phase combines the traditional transformations of closure conversion, which changes all functions to take their free variables as explicit arguments; and hoisting, which moves all function definitions to the top level of a program. Since we are using PHOAS, it is easiest to combine these phases into one, such that the closed nature of function definitions can be apparent syntactically from the fact that they only appear at the top level of a program. Figure 8 shows the syntax of this translation's target language.

Primops	p		...as in last language, minus fix...
Expressions	e		...as in last language...
Programs	P	$::=$	$e \mid \mathsf{let}\ f = \mathsf{fix}\ f(x).\ e\ \mathsf{in}\ P$

Figure 8. Closed language syntax

As in our past work on closure conversion with PHOAS (Chlipala 2008), this phase is interesting because we implement it by converting higher-order syntax to first-order syntax, which is passed to a translation that again produces higher-order syntax. Given a parametric expression E, we instantiate it like $E(\mathtt{nat})$, choosing to represent variables as natural numbers. We also follow a specific convention in how we use such a term, which has type `CPS.exp nat`. Our convention is isomorphic to the technique of de Bruijn levels, where bound variables that are not inside nested scopes have level 0, the next binders inside these have level 1, and so on. Compared to the more common de Bruijn indices, this technique has the advantage that all occurrences of a given binder's variable use the same level. Therefore, since PHOAS binders are represented as functions, we can descend into a binder simply by calling its function with the appropriate number.

We formalize this convention with a well-formedness judgment over `CPS.exp nat`. Here are the key rules in a restriction to untyped lambda calculus.

$$\frac{f < n \quad x < n}{n \vdash f\ x\ \mathsf{wf}} \qquad \frac{n+1 \vdash f(n)\ \mathsf{wf}}{n \vdash \lambda f\ \mathsf{wf}}$$

Our closure conversion is dependently-typed, such that it takes a well-formedness proof as input. We prove a theorem saying that, for any well-formed E, we have $0 \vdash E(\mathtt{nat})\ \mathtt{wf}$; and we pass an invocation of this theorem in the initial call to the translation function.

Here is the type of the main translation.

```
forall (var : Type) (n : nat) (e : exp nat), wf n e
  -> (((env var (freeVars n e) -> Closed.exp var)
      -> Closed.prog var)
  -> Closed.prog var
```

The function `freeVars` calculates the free variable set of an expression. When called like `freeVars n e`, it returns a length-`n` list of booleans, indicating which variables up to `n` appear free in `e`. The type family `env` is parameterized by such a list. An `env var fvs` is a tuple of one `var` variable for each entry of `fvs` that is `true`.

The main complexity in the translation type comes from a continuation argument. In translating an expression, the form of that expression implies some top-level function definitions that we should add. It is critical that none of these definitions mentions any local variables, or else we would have an ill-formed program. Thus, in translating an expression, we bind its functions and then call a continuation which may bind additional functions. That continuation then, inside its new definitions, calls a sub-continuation with an environment giving values to all free variables.

We hope that a few representative examples, as shown in Figure 9, make the protocol clear. We write `wf` arguments as ϕ. We rely

$$\begin{aligned}
\lfloor \mathsf{halt}(x) \rfloor\, n\phi k &= k(\hat{\lambda}\sigma.\mathsf{halt}(\mathsf{get}\ x\ \sigma\ \phi)) \\
\lfloor f\ x \rfloor\, n\phi k &= k(\hat{\lambda}\sigma.\mathsf{let}\ \sigma' = \mathsf{fst}(\mathsf{get}\ f\ (\Pi_1(\sigma))\ (\pi_1(\phi)))\ \mathsf{in} \\
&\quad \mathsf{let}\ f' = \mathsf{snd}(\mathsf{get}\ f\ (\Pi_1(\sigma))\ (\pi_1(\phi)))\ \mathsf{in} \\
&\quad \mathsf{let}\ p = \langle \sigma', \mathsf{get}\ x\ (\Pi_2(\sigma))\ (\pi_2(\phi)) \rangle\ \mathsf{in} \\
&\quad f'\ p) \\
\lfloor \mathsf{let}\ p\ \mathsf{in}\ f \rfloor\, n\phi k &= \lfloor p \rfloor\, n(\pi_1(\phi))(\hat{\lambda}k_P. \\
&\quad \lfloor f(n) \rfloor\,(n+1)(\pi_2(\phi))(\hat{\lambda}k_E.k(\hat{\lambda}\sigma. \\
&\quad k_P\ (\Pi_1(\sigma))\ (\hat{\lambda}x. \\
&\quad k_E\ (x :: \Pi_2(\sigma))))))
\end{aligned}$$

Figure 9. Representative cases of closure conversion

on a similar primop translation function whose exact type and definition we will not discuss further here.

There are a few unusual things going on in Figure 9. First, we manipulate well-formedness proofs ϕ as data. When we know from the structure of e that a proof ϕ must be deduced from a rule with two premises, we write $\pi_i(\phi)$ for the extraction of the ith premise's proof. For a function call $f\ x$, the two premises say that the two variables f and x are both less than the current de Bruijn level n. Such less-than proofs may be passed to the get function to enable extracting variables from an environment σ.

We must also do similar splitting of environments. Several cases of the definition of `freeVars` are implemented by joining sets of free variables. When an environment σ has a type based on the union of two sets, then the projections Π_1 and Π_2 translate into environments for those sets. This is always possible to do, since a union of two sets contains any binding required by either set alone.

The case for fix $f(x).\ e$, omitted above, is where new function bindings are created. It relies on two auxiliary functions for packing environments into tuples at closure formation sites and unpacking those tuples in function prologues. Free variable information is used to choose which variables to pack.

4.3.1 Correctness Proof

Reasoning about layers of nested continuations is tricky, so we prove the correctness of our translation by defining an alternate translation that does not use continuations. The alternate translation is specialized to the operational semantics of closed programs, where, as in the definition of `val` from Section 3, we represent function values with natural numbers pointing into a closure heap. By specializing the translation to the semantics, we can refer directly to closures and closure heaps, since function addresses are generated in a predictable way.

Here is the type of the alternate translation:

```
forall (n : nat) (e : CPS.exp nat), Closed.closures
  -> wf n e -> Closed.closures
    * (env Closed.val (freeVars n e)
      -> Closed.exp Closed.val)
```

A call to this function takes a current closure heap as input, and return values are pairs of extended closure heaps and functions from local variable environments to expressions. We did not just use this version as our initial translation because it works with a program representation where types alone do not guarantee well-formedness; any variable might include an "out-of-bounds" function reference.

It is fairly straightforward to prove a correctness theorem for the alternate translation. We need to prove about two dozen lemmas about operations on free variables, environments, and closure heaps. As for the last two phases, we define compatibility relations over the values and reference heaps of the source and target languages. With these pieces in place, we can prove the overall correctness theorem with the usual induction on source evaluation derivations.

A final lemma connects the two translations, proved by mutual structural induction over CPS expressions and primops.

4.4 Flattening

After closure conversion, programs are already almost in the form of three-address code, the family of traditional low-level compiler intermediate languages. Beyond the use of structured control flow with case expressions, the only serious difference is that closed programs use let-binding of immutable variables instead of manipulation of mutable temporaries, of which each procedure in three-address code has a fixed set.

We make this connection precise by defining a flat language and a translation into it. Every let-bound variable in an input function is assigned a distinct temporary in the function's translation, and the we replace the recursive type of programs P with a simpler type of pairs, where each pair consists of a list of flat functions and a flat main program expression. Every variable reference in the source program becomes either a temporary or a numeric index into the global list of functions.

Flattening works like closure conversion in instantiating PHOAS terms for use with de Bruijn levels. As this translation returns us to first-order languages, its correctness proof requires more lemmas about lists and maps, though most are independent of the set of language constructs. The main inductive theorems are comparable in complexity and proof organization to those for closure conversion.

4.5 Code Generation

The final compiler pass translates flat programs into assembly language. At this stage, neither source nor target language is novel. We still prove every theorem with an adaptive tactic program, but the basic organization of theorems is as one would expect from related projects. Our translation uses one register as the heap limit pointer, incrementing it as products, sums, and references are allocated. Another register doubles as the function argument register and as a place to stash short-lived values during double memory indirections. Finally, we reserve a register to store a recursive function's "self" pointer. The remaining $N - 3$ registers are used to store the first $N - 3$ temporaries of each procedure. The remaining temporaries are stored in a global area at the beginning of memory. Since we deal only with compiling whole programs, it is easy to choose a size for this region by finding the highest numbered temporary used in a program.

The correctness proof is comparable in complexity to those of previous phases, but with significantly more supporting lemmas.

5. Optimizations

To better gauge how our approach scales to the algorithms used by real compilers, we also implemented and verified two common optimization passes.

5.1 Common Subexpression Elimination

Between closure conversion and flattening, we perform intraprocedural common subexpression elimination (CSE) on closed programs. Since our languages have no intraprocedural iteration constructs, there is no need to perform dataflow analysis. Instead, a single recursive traversal of a program suffices. The optimization still simplifies cases like application of a known function, where it is possible to avoid building a closure.

As we descend into a program's structure, we maintain a mapping from variables to symbolic values, as defined below.

$$\text{Symbolic values} \quad s \quad ::= \quad \#n \mid c \mid () \mid \langle s, s \rangle \mid \mathsf{inl}(s) \mid \mathsf{inr}(s)$$

Values not built from the basic constant, unit, product, and sum constructors are represented with symbolic variables $\#n$, where a fresh n is generated for each new input-program variable that cannot be determined to have more specific structure.

The purpose of CSE is to remove some redundant bindings and case analyses. This transformation may sound complicated enough to require conversion of input programs to first-order form to analyze them. However, it is possible to implement CSE in an elegant higher-order way. In translating a parametric program P, we must produce a CSE'd version of it for each possible variable representation `var`. Our solution is to do so by instantiating P at variable type `var * sval`, where `sval` is the type of symbolic values s.

Thus, each variable is tagged with a symbolic representation, and this representation may be accessed directly at use sites. The main translation maintains a mapping from symbolic values to variables. We use this mapping to simplify case expressions with discriminees that we see statically are either inl or inr. When proceeding under a let binder, the translation evaluates the bound expression symbolically. If the result is in the map, we avoid creating a new binder in the translation. Instead, we apply the binder body, which is a function over variable/value pairs, to the variable that our map associates with the appropriate symbolic value, paired with that value. If the value we are binding is not found in the map, we do create a new binder, and, in the recursive call inside the binder's scope, we add the new variable to the symbolic map.

The main correctness theorem for this translation is proved very similarly to the main theorem for CPS conversion. The proof can be a bit simpler because we need no value compatibility relation; CSE has no effect on the values that appear during program evaluation. We prove the main theorem with about 20 lines of tactic code for performing appropriate case analyses, applying IHes and a lemma about primops, and materializing known facts about variables mentioned in expression equivalence derivations.

5.2 Combined Register Allocation and Dead Code Elimination

In a single pass performed between flattening and code generation, we combine register allocation and dead code elimination. Code generation automatically assigns the lowest-numbered temporaries to registers. Thus, the task of "register allocation" is simply to minimize the number of temporaries that each procedure uses, by finding opportunities to combine several mutually non-interfering temporaries into one. We use liveness information to calculate interference graphs. As with CSE, the lack of intraprocedural iteration makes it possible to compute an interference graph in a single traversal of procedure syntax. We use the same liveness information to eliminate useless assignments to temporaries.

Our implementation makes use of the finite set and map support in Coq's standard library. We represent liveness information with sets of temporaries, interference graphs with sets of unordered pairs of temporaries, and temporary reassignments with maps from temporaries to temporaries. Coq's library contains functors that build such structures from modules describing keys, and each functor output contains a set of standard theorems about its data structure. On top of this, we also implement and use an abstract data structure for temporary sets whose complements are finite, for use in choosing new names for temporaries.

As with code generation and many other kinds of low-level reasoning, this correctness proof is built from many unsurprising lemmas. By relying on the standard theorems about sets and maps, we manage to avoid most proving that is not specific to our transformation.

Component	**Total**	**Proofs**
Source language	228	0
Core PHOAS language	266	2
PHOASification	28	0
Correctness	390	138
Well-formedness	40	17
CPS language	279	18
CPS translation	94	0
Correctness	221	60
Well-formedness	39	12
Closed language	311	21
Closure conversion	303	13
Correctness	652	238
Well-formedness	261	119
CSE	87	1
Correctness	228	80
Well-formedness	177	70
Flat language	108	0
Flattening	63	0
Correctness	524	156
Register allocation	201	49
Correctness	642	310
Assembly language	105	0
Code generation	153	0
Correctness	1156	491
Overall compiler	13	0
Correctness	89	12
Total	6658	1807

Figure 10. Lines of code in different components

6. Statistics

Figure 10 breaks down our development by the number of lines of code in each component. We include how many lines of code in each component come from proofs, which counts literal proof scripts, resolution hints, and definitions of tactic functions. More or less all of the remaining lines come from definitions of syntax and semantics, in the files corresponding to languages; from compiler phase implementations, in their files; or from theorem statements and auxiliary definitions, in "Correctness" and "Well-formedness" files.

Our development depends upon our Lambda Tamer Coq library, which contains about 1500 lines of object-language-agnostic theorems and tactics. We assert two axioms from the Coq standard library: functional extensionality, which says that two functions are equal if they map equal inputs to equal outputs; and proof irrelevance for equality proofs, which says that no equality proposition has more than one distinct proof. This pair of axioms has been proved on paper to be consistent with CIC.

The "Well-formedness" components in Figure 10 deal with proofs that transformations produce well-formed PHOAS terms from well-formed inputs. We conjecture that there are CIC-consistent axioms stating that all PHOAS terms are well-formed. If this were proved metatheoretically, then we could safely omit the well-formedness proofs.

Our first finished compiler was for our final source language minus let expressions, constants, equality testing, and recursive functions. We also proved a simpler version of the final correctness theorem, stating only that a compiled program exhibits the same binary success-or-failure result as the original, ignoring details of returned values and thrown exceptions. As we added the enhancements needed to reach the final version, we measured how much effort was required.

6.1 Strengthening the Main Theorem

Since our source language is Turing-complete, the interesting aspects of compiler verification must be tackled even if the final theorem only distinguishes between two distinct classes of program outcome. Other distinctions may be modeled using tests within the object language. Thus, it was pragmatic for us to develop our initial proofs using this simplification. Later, we went back and adapted the proofs to allow us to prove Theorem 1 as we stated it earlier in the paper.

This required updating the different object languages so that halt and fail operations take parameters. We added or modified about 100 lines of syntax and semantics. We also had to introduce the "compiler data layout contract" relation and its relatives for the different translation phases, whose definitions added about 150 lines of unsurprising code. Additionally, we added about 100 lines of theorem statements, one-liner automated proofs, and resolution hints for some new theorems about the contract relations.

Beyond that, we modified or added about 80 lines of theorem statements and proof script, with most added to deal with new existential quantifications over program result values. Many of these changes were improvements that occurred to us as we worked on the upgrade, such that we would say that the changes belonged in the original compiler. Notably, the correctness proofs of the optimizations required no changes. We worked on this upgrade over the course of two days, with performance of the Coq proof assistant being the primary limiting factor. Automation of large proofs frequently leads Coq to run out of memory or run for excessively long, and we spent more time than we would like tuning our scripts to skirt these limitations on an old computer with modest resources. We believe that relatively straightforward improvements to Coq's proof engine would make this kind of upgrade quite reasonable to complete in well under a day of work. Even with the current state of the tools, the upgrade did not require us to add any proof code specific to a particular case of any of our inductive proofs.

6.2 Adding let Expressions

Adding let expressions was relatively simple, since general let only appears until CPS translation, and since let does not add a new category of runtime values. Thus, we extended the first two translations only and the syntax and semantics of the first two languages. We also added a let case to the expression compatibility relation in the PHOASification correctness proof. These changes amounted to about 30 new lines of code, with no old lines modified. All of our proofs continued working unchanged. We did not need to update a single theorem statement or proof script. The whole process took under half an hour.

6.3 Adding Constants and Equality Testing

Next, we added constants and equality comparison of constants, which impacts much more of the compiler and its proof. We added or changed about 100 lines defining syntax, semantics, and translations, and we added about 50 lines defining new rules for inductive relations used in the proofs. To adapt the old proofs, we had to change some patterns to mention new constructors of datatypes, to placate Coq's limited support for inferring which datatype is being pattern-matched on. We proved one lemma about the encoding of constants, giving it a one-line proof and adding it as a hint, and we added one additional one-line hint that detects when constants are being compared for equality and then performs case analysis on whether they are equal. These proof changes amount to about 10 lines in total, and we also added 5 more lines to improve performance in a way not related to constants. We spent about half a day on this extension, again with most of that time spent waiting for slow proof search to finish.

6.4 Adding Recursive Functions

Replacing non-recursive anonymous functions $\lambda x.\ e$ with recursive anonymous functions $\mathsf{fix}\ f(x).\ e$ turned out to be the most intensive change. We added or modified about 50 lines to account for additions to syntax, semantics, and translations, and we modified about 20 lines that define function abstraction rules for further inductive relations.

We added or modified about 350 lines of theorem statements and proofs. In each of the earlier phases of the compiler, we modified at most one line of proof in a way that is really specific to recursive functions. The remaining extra lines come from the fact that fix is our only construct that binds more than one variable at once. We had already proved a host of lemmas specialized to the case of one variable at a time, and we needed to duplicate these lemmas, reusing their automated proofs without changes. Other alterations to theorem statements and tactics reflected only the need to handle a new binding pattern. The exception to this trend was code generation, where a change to the calling convention triggered a fair amount of churn in theorem statements. The full upgrade to fix support took us approximately one day.

7. Related Work

Compiler verification for first-order languages has a considerable history, but we will focus on reviewing related work in compiling functional languages. See the bibliography by Dave (2003) for pointers into the traditional literature on first-order compilers. There have been a variety of investigations into verifying compilers for pure functional languages, drawing on several very different representation and proof techniques.

Flatau (1992) described a partial verification of a compiler from a subset of the Nqthm logic to the first-order imperative language Piton, performed with the Nqthm prover. The proof was highly automated, in the usual style of Nqthm and ACL2. Since the compiler worked in a single pass and targeted a first-order language, it avoided most of the issues inherent in reasoning about rearranging variable binders.

Minamide and Okuma (2003) used Isabelle/HOL and Isar to build proofs of correctness for CPS translations for bare-bones untyped lambda calculus, using concrete encoding of binders with variable names. The simplest translation that they verified (that of Plotkin) had a correctness proof of about 250 lines. Their proof for Danvy and Nielsen's translation took about 400 lines, and this figure grew to about 600 when they added let binding to the object language.

Tian (2006) used Twelf with HOAS to prove CPS translation correctness for an untyped, pure mini-ML language without recursion, which is subsumed within our case study source language. His comparable CPS translation theorem took about 50 lines, in the usual, fully manual Twelf style. Tian's development uses a target language more specialized to his particular translation, featuring hardcoded binding of a second-class continuation with any function definition, rather than building this functionality on top of product types and first-class continuations. We expect that the cost of writing such proofs in Twelf becomes more apparent when source and target languages are less tightly coupled, so that some inductive proof cases must build proof trees of non-trivial depth.

In past work (Chlipala 2007), we verified a compiler from basic simply-typed lambda calculus to a target language similar to three-address code. We used the dependent de Bruijn representation throughout the early stages of the compiler, and we relied on a metaprogramming component to prove some of the standard binder lemmas for us. By switching to PHOAS, we have avoided the need to state those lemmas explicitly. Our present compiler extends our past PHOAS-based work (Chlipala 2008) by treating impurity, recursion, and the rest of the compilation pipeline beyond CPS translation and closure conversion.

Dargaye and Leroy (2007) used Coq to verify CPS translations for another mini-ML language encoded with de Bruijn indices. We believe that our CPS translation is comparable in functionality to their optimized translation, with the exception that ours does not do tail call optimization. They give code size statistics for their CPS translation, broken into "specifications" and "proofs." The "proofs" size for the dependencies of the optimized transform correctness proof is 4287 lines, an order of magnitude more than the total size of our CPS correctness file. The complexities of the languages treated by the two projects are not directly comparable; Dargaye and Leroy include variable-arity recursive functions and datatype constructors, but they omit impure features.

Benton and Hur (2009) take a very different approach to compiler verification in Coq, starting with a typed functional language (represented with de Bruijn indices) and using step-indexed logical relations over types to establish correctness. Their compiler works in one pass, so the theorems involved are very different from those in the early phases of our compiler. Their development uses the usual Coq manual proof style and runs to about 4000 lines. Though their source language (basic lambda calculus with recursive functions) is less featureful than the source language of this paper, their final correctness theorem is more general than in any related work, as it facilitates sound linking with code produced by different compilers or written by hand.

8. Conclusion

Mostly-manual interactive proving of theorems about programming languages and compilers can be a very engaging challenge, and it has been a crucial tool in our efforts to figure out the right abstractions for the job. Nonetheless, we do not believe that this style scales to real-world implementations of high-level languages. We think it is not at all too early to be thinking about the techniques that may some day be applied in such a setting. Our case study in this paper is a verified compiler whose mechanized correctness proofs are tactic programs that can adapt automatically to specification changes. We believe strongly that, if compiler verification ever goes mainstream, it will be done more like we propose here than like the styles more commonly employed by languages researchers.

To avoid getting bogged down in administrative lemmas about binding, we exploit the parametric higher-order abstract syntax encoding, along with a new way of using it to encode substitution-free operational semantics. As a result, some of our compiler phase correctness proofs are shorter even than what a diligent semanticist would write on paper. Despite our use of novel representations, our final theorem is stated only in terms of established encodings and relies on no nonstandard axioms.

We hope to expand our case study into a verified compiler for core Standard ML. This requires adding a few more features to the source language and implementing a (hopefully straightforward) elaboration from the official abstract syntax of Standard ML. We would also like to define typing judgments for each language and prove a type preservation theorem for the final compiler, which could output Typed Assembly Language. We conjecture that this extension would require little change to the semantic preservation theorems and proofs. It would also be interesting to try to complete the end-to-end compiler picture with a verified parser, type inference engine, and assembler.

Acknowledgments

We would like to thank Greg Morrisett, Ryan Wisnesky, and the anonymous referees for helpful feedback on earlier versions of this paper. This work was supported by a gift from Microsoft Research.

References

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proc. POPL*, pages 3–15, 2008.

Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. ICFP*, pages 97–108, 2009.

Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. PLDI*, pages 54–65, 2007.

Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proc. ICFP*, pages 143–156, 2008.

Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2 (4):361–391, 1992.

Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *Proc. LPAR*, pages 211–225, 2007.

Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formal manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

David Delahaye. A tactic language for the system Coq. In *Proc. LPAR*, pages 85–95, 2000.

Arthur D. Flatau. *A Verified Implementation of an Applicative Language with Dynamic Storage Allocation*. PhD thesis, University of Texas at Austin, November 1992.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In *Proc. ICFP*, pages 75–86, 2008.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. of the ACM*, 40(1):143–184, 1993.

Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *Proc. ICALP*, pages 963–978, 2001.

Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54, 2006.

Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.

Yasuhiko Minamide and Koji Okuma. Verifying CPS transformations in Isabelle/HOL. In *Proc. MERLIN*, pages 1–8, 2003.

J Strother Moore. A mechanically verified language implementation. *J. Automated Reasoning*, 5(4):461–492, 1989.

G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. FPCA*, pages 66–77, 1995.

F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. PLDI*, pages 199–208, 1988.

Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. CADE*, pages 202–206, 1999.

Ye Henry Tian. Mechanically verifying correctness of CPS compilation. In *Proc. CATS*, pages 41–51, 2006.

C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In *Proc. CADE*, pages 38–53, 2005.

Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *J. Funct. Program.*, 18(1):87–140, 2008.

Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Proc. TPHOLs*, pages 167–184, 1999.