Parametric Higher-Order Abstract Syntax for Mechanized Semantics

Adam Chlipala

Harvard University, Cambridge, MA, USA

adamc@cs.harvard.edu

Abstract

We present *parametric higher-order abstract syntax (PHOAS)*, a new approach to formalizing the syntax of programming languages in computer proof assistants based on type theory. Like higher-order abstract syntax (HOAS), PHOAS uses the meta language's binding constructs to represent the object language's binding constructs. Unlike HOAS, PHOAS types are definable in general-purpose type theories that support traditional functional programming, like Coq's Calculus of Inductive Constructions. We walk through how Coq can be used to develop certified, executable program transformations over several statically-typed functional programming languages formalized with PHOAS; that is, each transformation has a machine-checked proof of type preservation and semantic preservation. Our examples include CPS translation and closure conversion for simply-typed lambda calculus, CPS translation for System F, and translation from a language with ML-style pattern matching to a simpler language with no variable-arity binding constructs. By avoiding the syntactic hassle associated with first-order representation techniques, we achieve a very high degree of proof automation.

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

General Terms Languages, Verification

Keywords compiler verification, interactive proof assistants, dependent types, type-theoretic semantics

1. Introduction

Compiler verification is one of the classic problems of formal methods. Most all computer scientists understand the importance of the problem, as compiler bugs can negate the benefits of any techniques for program correctness assurance, formal or informal. Unlike most potential subjects for program verification, we have clear correctness specifications for compilers, based on the formal semantics of programming languages. Even better, the researchers interested in program verification tend already to be quite familiar with compilers.

None of these points in favor of studying the problem are new; they have all been in force for decades, at least. Pondering the future 20 years ago, then, it might have seemed reasonable to hope that at least the most widely-used production compilers would have machine-checked soundness proofs today. Of course, this is far from the case; we are not aware of any certified compilers in production use today.

There have been some notably successful research projects. Moore (1989) used the Boyer-Moore theorem prover to certify the correctness of a language implementation stack for the Piton language. However, the Boyer-Moore prover has an important disadvantage here: it is implemented as a monolithic set of decision procedures, all of which we must trust are implemented correctly to believe the result of the verification. We can be more confident in results produced with provers that, like Isabelle, follow the LCF tradition; or that, like Coq, are based on foundational type theories. Both of these implementation strategies allow the use of small proof-checking kernels, which are all we need to trust to believe the final results.

More recently, Leroy (2006) has implemented a certified compiler for a subset of C in Coq in the CompCert project. The final proof can be checked with a relatively small proof checker which amounts to a type-checker for a dependently-typed lambda calculus. However, the path to this result is a bumpy one. In addition to the compiler implementation proper, the implementation includes about 17,000 lines of proof. With this as the state-of-the-art, it does not seem surprising that most compiler implementers would decline to verify their compilers, leaving that task to researchers who specialize in it.

Can we get the best of both worlds? Can we verify compilers using cooperating decision procedures *and* produce easily-checkable proof witnesses in the end? In this paper, we suggest a new technique that removes one barrier in the way of that goal.

Nearly all compiler verification research being done until very recently, including the two projects that we have already mentioned, focuses on first-order, imperative programming languages. However, HOT (higher-order typed) languages like ML and Haskell can benefit just as much from certified implementations, and, of course, they are near and dear to the hearts of many people working in program verification. Unfortunately, the pantheon of challenges associated with compiler verification only grows with the addition of higher-order features and fancy type systems.

Central among these challenges is the question of how to deal with nested variable binding constructs in object languages. The POPLmark Challenge (Aydemir et al. 2005) helped draw attention to these issues recently, issuing a challenge problem in mechanized language metatheory that drew many solutions employing many different binder representation strategies. The solutions were largely split between implementations in Coq (Bertot and Castéran

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

2004) and Isabelle (Paulson 1994) using first-order variable representations, and solutions in Twelf (Pfenning and Schürmann 1999) using higher-order abstract syntax (Pfenning and Elliot 1988). The first-order proofs required much more verbosity surrounding bookkeeping about variables, but the Twelf implementations involved more tedious proving for the lemmas that would actually appear in a paper proof, as Twelf lacks any production-quality proof automation. This failing of the first-order solutions seems like an unavoidable drawback. The failing of the higher-order solutions seems less fundamental, though certainly much more is known about proof assistant support for the logics at the cores of Isabelle and Coq than about such support for Twelf's meta-logic.

In past work (Chlipala 2007), we tackled these representation issues in the context of compiler verification. Language metatheory problems are popular as benchmarks because they can admit relatively straightforward pencil-and-paper solutions and because most programming languages researchers are already familiar with them. At the same time, hardly anyone working outside of programming language research, including other computer scientists, recognizes their importance. It is also true that more *denotational* approaches to language semantics remove the need for traditional syntactic metatheory proofs. If you accept a denotational semantics mapping terms of an object language into a foundational type theory as *the* specification of program meanings, then standard theorems like type safety and strong normalization can be "borrowed" from the meta language "for free."

We presented a certified type-preserving compiler from simply-typed lambda calculus to an idealized assembly language used with a garbage-collecting runtime system. We proved type preservation and semantic preservation for each phase of the compiler, automating the proofs to a large extent using several decision procedures, including support from a new Coq plug-in for automatic generation of lemmas about operations that rearrange variables in object terms. We formalized language syntax and typing rules using *dependently-typed abstract syntax trees*, and we used *foundational type-theoretic semantics* to assign dynamic semantics via computable translations to Coq's Calculus of Inductive Constructions. Unfortunately, the hassles of first-order binder representations, such as the de Bruijn representation we chose, are only exacerbated by adding dependent types.

It is not obvious how to lessen this burden. Higher-order abstract syntax, as it is usually implemented, allows for writing nonterminating programs if standard pattern matching facilities are also provided. The logic/programming language underlying Coq is designed to forbid non-termination, which would correspond to logical unsoundness.

In this paper, we present a retooling of our previous work based on a new representation technique, *parametric higher-order abstract syntax (PHOAS)*. PHOAS retains most of the primary benefits of HOAS and first-order encodings. That is, we use the meta language to do almost all of our variable book-keeping for us, but we are still able to write many useful (provably total) recursive, pattern-matching functions over syntax in a very direct way. These functions are definable in the Calculus of Inductive Constructions (CIC), and we can use Coq to prove non-trivial theorems about them, automating almost all of the work with reusable tactics.

In the next section, we introduce PHOAS and use it to implement a number of translations between statically-typed lambda calculi in a way that gives us static proof of type preservation. In Section 3, we sketch how we used the Coq proof assistant to build machine-checked proofs of semantic preservation for the translations from Section 2. Section 4 provides some statistics on the complexity of our Coq developments, including measurements of how much work is needed to extend a CPS translation with new types. We wrap up by surveying related work.

We have added PHOAS support to our Coq language formalization library called Lambda Tamer. The source distribution, which includes this paper's main case studies as examples, can be obtained from:

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

One final note is in order before we proceed with the main presentation. While we start off with a few small examples that are useful in operational treatments of language semantics, our focus is on more denotational methods. Indeed, we make no particular claims about the overall benefits of PHOAS in settings where object languages are not given meaning by executable translations into a meta language. Many effectful programming language features are hard to represent in the traditional setting of denotational semantics with domain theory, but we can mix type-theoretic and operational reasoning to encode these features without using non-trivial domain constructions. The features that we handle type-theoretically are much easier to reason about, and we can restrict our operational reasoning to a "universal" core calculus with good support for effects. All this is to justify that the type-theoretic semantics approach is worth using as the primary source of examples for PHOAS. In this paper, our example object languages will be pure with easy pure type-theoretic treatments, and we leave the orthogonal issue of encoding impure languages for a future paper.

2. Programming with PHOAS

We will begin by demonstrating how to write a variety of useful programs that manipulate syntax implemented with PHOAS. We use a non-ASCII notation that is a combination of Coq, Haskell, and ML conventions, designed for clarity for readers familiar with statically-typed functional programming.

2.1 Untyped Lambda Calculus

We begin with the syntax of untyped lambda calculus by defining a recursive type term that follows the usual HOAS convention.

$$\begin{array}{rcl} \mathsf{term} & : & \star \\ \mathsf{App} & : & \mathsf{term} \to \mathsf{term} \to \mathsf{term} \\ \mathsf{Abs} & : & (\mathsf{term} \to \mathsf{term}) \to \mathsf{term} \end{array}$$

The fact that term is a type is indicated by assigning it type $\star$, the type of types. While Coq has an infinite hierarchy of universes for describing types in a way that disallows certain soundness-breaking paradoxes, we will collapse that hierarchy in this paper in the interests of clarity. The dubious reader can consult our implementation to see the corresponding versions that do not take this shortcut.

App and Abs are the two constructors of terms, corresponding to function application and abstraction. The type of Abs does not mention any specification of which variable is being bound, as we can use the function space of CIC to encode binding structure. For instance, the identity function $\lambda x.\ x$ can be encoded as:

$$\mathsf{id} \quad = \quad \mathsf{Abs}\ (\lambda x.\ x)$$

The λ on the righthand side of the equation marks a CIC function abstraction, not the object language abstraction that we are trying to formalize. This first example looks almost like cheating, since it includes the original term in it, but we can encode any lambda calculus term in this way, including the famous infinite-looping term $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$:

$$\mathsf{diverge} \quad = \quad \mathsf{App}\ (\mathsf{Abs}\ (\lambda x.\ \mathsf{App}\ x\ x))\ (\mathsf{Abs}\ (\lambda x.\ \mathsf{App}\ x\ x))$$

$$\begin{aligned}\mathsf{selfApply} &= \lambda x : \mathsf{term}.\ \mathsf{match}\ x\ \mathsf{with} \\ &\qquad \mid \mathsf{App}\ x\ y \Rightarrow \mathsf{App}\ x\ y \\ &\qquad \mid \mathsf{Abs}\ f \Rightarrow f\ (\mathsf{Abs}\ f) \\ \mathsf{bad} &= \mathsf{selfApply}\ (\mathsf{Abs}\ \mathsf{selfApply})\end{aligned}$$

Figure 1. An example divergent term

Coq encodes programs and proofs in the same syntactic class, following the Curry-Howard Isomorphism. If we allow the existence of terms in this general class that do not terminate when run as programs, then we can prove any theorem with an infinite loop. Splitting programs and proofs into separate classes would be possible, but it would complicate the metatheory and implementation. Moreover, programs that must terminate are simply easier to reason about, and this will be very important when we want to prove correctness theorems for program transformations.

Unfortunately, if Coq allowed the definitions we have developed so far, we could write non-terminating programs and compromise soundness. We do not even need any facility for recursive function definitions; simple pattern-matching is enough. Consider the term bad defined in Figure 1, following the same basic trick as the last example. No matter how many β-reductions and simplifications of pattern matches we apply, bad resists reducing to a normal form.

The root of the trouble here is that we can write terms that do not correspond to terms of the lambda calculus. Counterexamples like bad are called *exotic terms*. There are a number of tricks for building HOAS encodings that rule out exotic terms, including meta language enhancements based on new type systems (Fegaras and Sheard 1996; Schürmann et al. 2001). The technique that we will use, PHOAS, does not require such enhancements. It is essentially a melding of weak HOAS (Despeyroux et al. 1995; Honsell et al. 2001) and the "boxes go bananas" (BGB) (Washburn and Weirich 2008) HOAS technique.

We can illustrate the central ideas by modifying our example type definition for this section so that Coq will accept it as valid. Coq uses simple syntactic criteria to rule out inductive type definitions that could lead to non-termination. The particular restriction that rules out the standard HOAS encoding is the *positivity restriction*, which says (roughly) that an inductive type may not be used in the domain of a nested function type within its own definition.

A small variation on the original definition sidesteps this rule. Instead of defining a type term, we will define an inductive type family $\mathsf{term}(\mathcal{V})$. Here we see the source of the name "parametric higher-order abstract syntax," as the family term is *parametric* in an arbitrary type $\mathcal{V}$ of *variables*.

$$\begin{aligned}\mathsf{term}(\mathcal{V}) &: \star \\ \mathsf{Var} &: \mathcal{V} \to \mathsf{term}(\mathcal{V}) \\ \mathsf{App} &: \mathsf{term}(\mathcal{V}) \to \mathsf{term}(\mathcal{V}) \to \mathsf{term}(\mathcal{V}) \\ \mathsf{Abs} &: (\mathcal{V} \to \mathsf{term}(\mathcal{V})) \to \mathsf{term}(\mathcal{V})\end{aligned}$$

Var, App, and Abs are the three constructors of terms. The new representation strategy differs from the old HOAS strategy only in that binders bind *variables* instead of terms, and those variables are "injected" explicitly via the Var constructor. For example, we now represent the identity function as:

$$\mathsf{id} \quad = \quad \mathsf{Abs}\ (\lambda x.\ \mathsf{Var}\ x)$$

If we fix a type $\mathcal{V}$ at the top level of a logical development and assert some axioms characterizing the properties of variables, we can arrive at weak HOAS with the Theory of Contexts (Honsell et al. 2001). This would be enough to allow us to build relational versions of all of the syntactic operations we care about, but it does not support a natural style of functional programming with syntax. For instance, it is unclear how to write a recursive function that counts the number of variable occurrences in a term.

$$\begin{aligned}\mathsf{numVars} &: \mathsf{term}(\mathsf{unit}) \to \mathbb{N} \\ \mathsf{numVars}(\mathsf{Var}\ \_) &= 1 \\ \mathsf{numVars}(\mathsf{App}\ e_1\ e_2) &= \mathsf{numVars}\ e_1 + \mathsf{numVars}\ e_2 \\ \mathsf{numVars}(\mathsf{Abs}\ e') &= \mathsf{numVars}\ (e'\ ()) \\ \mathsf{NumVars} &: \mathsf{Term} \to \mathbb{N} \\ \mathsf{NumVars}(E) &= \mathsf{numVars}\ (E\ \mathsf{unit})\end{aligned}$$

Figure 2. A function for counting variable uses in a term

The BGB trick is to take advantage of the meta language's parametricity properties to rule out exotic terms in a way that lets us "stash data inside of variables" when we later decide to analyze terms in particular ways. In our setting, we accomplish this by defining the final type of terms like this:

$$\mathsf{Term} \quad = \quad \forall \mathcal{V} : \star.\ \mathsf{term}(\mathcal{V})$$

That is, we define a polymorphic type, where the universally quantified variable $\mathcal{V}$ may be instantiated to any type we like. Parametricity is the "theorems for free" property that lets us draw conclusions like that the type $\forall \tau : \star.\ \tau \to \tau$ is inhabited only by the polymorphic identity function. For our running example, we can rely on parametricity to guarantee that Terms only "push variables around," never examining or producing them in any other way. We are not aware of a formal proof of parametricity for CIC, but we can work around this meta-theoretic gap by strengthening our theorem statements to require parametricity-like properties to hold of particular concrete terms; we consider this issue more in Section 3.

Now we can quite easily define a function that counts the number of variable occurrences in a term, as in Figure 2. The function NumVars is passed a term E that can be instantiated to any concrete choice of a variable type. As the only thing we need to know about variables is where they are, we can choose to make $\mathcal{V}$ the singleton type unit. Now we can use a straightforward recursive traversal of the term(unit) that results. The most interesting part of the definition is where, for the case of numVars(Abs e'), we call the meta language function e' on (), the value of type unit, to specify "which variable we are binding" or, alternatively, which data we want to associate with that variable.

Coq imposes strict syntactic termination conditions on recursive function definitions, but the definition here of numVars satisfies them, because every recursive call is on a direct syntactic subterm of the original function argument. The notion of syntactic subterms includes arbitrary calls to functions that are arguments of constructors.

For an example of a non-trivial choice of $\mathcal{V}$, consider the function in Figure 3, which checks if its term argument is a candidate for a top-level η-reduction. To perform this check, we instantiate $\mathcal{V}$ as bool and pattern match on the resulting specialized term. We can return false immediately if the term is not an abstraction. Otherwise, we *apply the body to* false, effectively substituting false for the abstraction's argument variable. The term we get by applying to false had better be an application of some unknown term e_1 to a variable that has been tagged false (that is, the argument variable). If so, we traverse e_1, substituting true for each new bound variable we encounter and checking that every free variable is tagged

$$\begin{array}{rcl}
\mathsf{canEta}' & : & \mathsf{term}(\mathsf{bool}) \to \mathsf{bool}\\
\mathsf{canEta}'(\mathsf{Var}\ b) & = & b\\
\mathsf{canEta}'(\mathsf{App}\ e_1\ e_2) & = & \mathsf{canEta}'(e_1)\ \&\&\ \mathsf{canEta}'(e_2)\\
\mathsf{canEta}'(\mathsf{Abs}\ e') & = & \mathsf{canEta}'(e'\ \mathsf{true})\\
\mathsf{canEta} & : & \mathsf{term}(\mathsf{bool}) \to \mathsf{bool}\\
\mathsf{canEta}(\mathsf{Abs}\ e') & = & \mathsf{match}\ e'\ \mathsf{false}\ \mathsf{with}\\
 & & \quad \mid \mathsf{App}\ e_1\ (\mathsf{Var}\ \mathsf{false}) \Rightarrow\\
 & & \qquad \mathsf{canEta}'(e_1)\\
 & & \quad \mid \_ \Rightarrow \mathsf{false}\\
\mathsf{canEta}(\_) & = & \mathsf{false}\\
\mathsf{CanEta} & : & \mathsf{Term} \to \mathsf{bool}\\
\mathsf{CanEta}(E) & = & \mathsf{canEta}(E\ \mathsf{bool})
\end{array}$$

Figure 3. Testing for η-reducibility

true. If the traversal's test passes, then we know that the original variable never appears in e_1, so the original term is a candidate for η-reduction.

We can also write recursive functions that construct terms. We consider capture-avoiding substitution as an example. Since we are using a higher-order binding encoding, we define the type of terms with one free variable like this:

$$\mathsf{Term1} \quad = \quad \forall \mathcal{V} : \star.\ \mathcal{V} \to \mathsf{term}(\mathcal{V})$$

Now we can implement substitution in a way where, when we are trying to build a term for a particular $\mathcal{V}$ type, we do our intermediate work with the variable type $\mathsf{term}(\mathcal{V})$. That is, we tag each variable with the term we want to substitute for it.

$$\begin{array}{rcl}
\mathsf{subst} & : & \forall \mathcal{V} : \star.\ \mathsf{term}(\mathsf{term}(\mathcal{V})) \to \mathsf{term}(\mathcal{V})\\
\mathsf{subst}(\mathsf{Var}\ e) & = & e\\
\mathsf{subst}(\mathsf{App}\ e_1\ e_2) & = & \mathsf{App}\ (\mathsf{subst}(e_1))\ (\mathsf{subst}(e_2))\\
\mathsf{subst}(\mathsf{Abs}\ e') & = & \mathsf{Abs}\ (\lambda x.\ \mathsf{subst}(e'\ (\mathsf{Var}\ x)))\\
\mathsf{Subst} & : & \mathsf{Term1} \to \mathsf{Term} \to \mathsf{Term}\\
\mathsf{Subst}\ E_1\ E_2 & = & \lambda \mathcal{V} : \star.\ \mathsf{subst}(E_1\ (\mathsf{term}(\mathcal{V}))\ (E_2\ \mathcal{V}))
\end{array}$$

The $\mathsf{subst}(\mathsf{Abs}\ e')$ case is trickiest from a termination checking perspective, but the same syntactic subterm rule applies. Any call to a function that was an argument to the constructor we are pattern matching on is allowed, even if the call is inside a meta language binder and uses the bound variable.

We hope that the examples in this section have provided a good sense for how PHOAS supports relatively direct functional definitions of syntactic operations. The choice of different variable types for different functions provokes some cleverness from the programmer, on the order of the effort needed in selecting helper functions in traditional functional programming. Nonetheless, the convenience advantage of PHOAS over first-order techniques becomes clear when we move to formal proofs about the functions we define, letting us proceed without proving any auxiliary lemmas about variable manipulation.

While examples of mechanized language formalization have almost always been drawn from the array of syntactic metatheory properties of languages with operational semantics, in the rest of this paper we are concerned instead with proving that code translations on languages with type-theoretic semantics preserve program meaning. In the rest of Section 2, we will show how to define several translations on typed lambda calculi, using dependent types to prove type preservation simultaneously with defining the translations themselves. Section 3 expands on our results to prove semantic preservation for the same translations.

$$\begin{array}{rlcl}
\text{Types} & \tau & ::= & \mathsf{bool} \mid \tau \to \tau\\
\text{Variables} & x & & \\
\text{Terms} & e & ::= & |x| \mid \mathsf{true} \mid \mathsf{false} \mid e\ e \mid \lambda f\\
\text{Term functions} & f & &
\end{array}$$

Figure 4. Syntax for STLC that makes PHOAS explicit

2.2 CPS Translation for Simply-Typed Lambda Calculus

We will start by writing a translation from direct-style simply-typed lambda calculus (STLC) into continuation-passing style. As our single base type, we choose bool, so that every type is inhabited by multiple distinct values, making our final correctness theorem in Section 3 non-trivial.

$$\begin{array}{rlcl}
\text{Types} & \tau & ::= & \mathsf{bool} \mid \tau \to \tau\\
\text{Variables} & x & & \\
\text{Terms} & e & ::= & x \mid \mathsf{true} \mid \mathsf{false} \mid e\ e \mid \lambda x.\ e
\end{array}$$

We assume the standard typing rules and omit them here, though they are implied by the CIC representation that we choose for terms. We have a straightforward algebraic datatype definition of types:

$$\begin{array}{rcl}
\mathsf{type} & : & \star\\
\mathsf{Bool} & : & \mathsf{type}\\
\mathsf{Arrow} & : & \mathsf{type} \to \mathsf{type} \to \mathsf{type}
\end{array}$$

We represent terms with a type family $\mathsf{term}(\mathcal{V})$, as before. The difference is that now choices of $\mathcal{V}$ have type $\mathsf{type} \to \star$ instead of type $\star$. That is, we have a different type of variables for each object language type.

$$\begin{array}{rcl}
\mathsf{term}(\mathcal{V}) & : & \mathsf{type} \to \star\\
\mathsf{Var} & : & \forall \tau : \mathsf{type}.\ \mathcal{V}(\tau) \to \mathsf{term}(\mathcal{V})\ \tau\\
\mathsf{Tru} & : & \mathsf{term}(\mathcal{V})\ \mathsf{Bool}\\
\mathsf{Fals} & : & \mathsf{term}(\mathcal{V})\ \mathsf{Bool}\\
\mathsf{App} & : & \forall \tau_1, \tau_2 : \mathsf{type}.\ \mathsf{term}(\mathcal{V})\ (\mathsf{Arrow}\ \tau_1\ \tau_2)\\
 & & \quad \to \mathsf{term}(\mathcal{V})\ \tau_1 \to \mathsf{term}(\mathcal{V})\ \tau_2\\
\mathsf{Abs} & : & \forall \tau_1, \tau_2 : \mathsf{type}.\ (\mathcal{V}(\tau_1) \to \mathsf{term}(\mathcal{V})\ \tau_2)\\
 & & \quad \to \mathsf{term}(\mathcal{V})\ (\mathsf{Arrow}\ \tau_1\ \tau_2)\\
\mathsf{Term} & = & \lambda \tau : \mathsf{type}.\ \forall \mathcal{V} : \mathsf{type} \to \star.\ \mathsf{term}(\mathcal{V})\ \tau
\end{array}$$

This follows the general idea of abstract syntax tree types implemented using generalized algebraic datatypes in, for instance, GHC Haskell. The main difference is that, in Haskell, type indices must be meta language types, so we might use the Haskell boolean type in place of Bool and the Haskell function type constructor in place of Arrow. Thus, to write recursive functions over those indices in Haskell requires something like type classes with functional dependencies (or the more experimental type operators), rather than the direct pattern-matching definitions that are possible with our Coq encoding.

Defining the syntax of every object language with the same simple, mostly textual inductive type definition mechanism is convenient from a foundational perspective, but it is generally clearer to work mostly with syntactic abbreviations closer to those used in pencil-and-paper formalisms. Coq even supports the registration of arbitrary user-specified recursive descent parsing rules, so we work with the same simplification in our implementation, modulo a restriction to the ASCII character set. The syntax that we will use for

```
Section vars.
  Variable var : type -> Type.

  Inductive term : type -> Type :=
  | EVar : forall t,
    var t
    -> term t
  | ETrue : term TBool
  | EFalse : term TBool
  | EApp : forall t1 t2,
    term (TArrow t1 t2)
    -> term t1
    -> term t2
  | EAbs : forall t1 t2,
    (var t1 -> term t2)
    -> term (TArrow t1 t2).
End vars.
```

Figure 5. Coq code to define STLC syntax

STLC, shown in Figure 4, is a slight modification that exposes the relevant parts of variable representation.

We explicitly inject a variable x into the term type as $|x|$, and abstractions λf explicitly involve functions f from variables to terms. From this point on, we will distinguish between meta language and object language lambdas by writing the former as $\widehat{\lambda}$ and the latter as the usual λ. We will write $\lambda x.\ e$ as shorthand for $\lambda(\widehat{\lambda} x.\ e)$.

Figure 5 shows Coq code for this syntax formalization scheme. We use Coq's "sections" facility to parameterize the `term` type with a type family `var` without needing to mention `var` repeatedly within the definition.

The target language for our CPS translation is a linearized form of the source language, where functions never return, indicated by continuation types of the form $\tau \rightarrow 0$; and programs are broken up into sequences of primitive operations. Since we will give the language semantics type-theoretically, we do not bother to include a syntactic class of "values," and we put constants like true and false directly in the class of primops.

$$
\begin{array}{rcrl}
\text{Types} & \tau & ::= & \mathsf{bool} \mid \tau \rightarrow 0 \mid \tau \times \tau \\
\text{Variables} & x & & \\
\text{Terms} & e & ::= & \mathsf{halt}(x) \mid x\ x \mid \mathsf{let}\ p\ \mathsf{in}\ f \\
\text{Primops} & p & ::= & |x| \mid \mathsf{true} \mid \mathsf{false} \mid \lambda f \\
 & & & \mid \langle x, x\rangle \mid \pi_1 x \mid \pi_2 x \\
\text{Term functions} & f & &
\end{array}
$$

We will use $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ as shorthand for $\mathsf{let}\ e_1\ \mathsf{in}\ \widehat{\lambda} x.\ e_2$.

In the interests of space, we will omit here the Coq definition of the mutually inductive PHOAS types for terms and primops. The main complication of the CPS language over the source language is that terms are represented with a type family $\mathsf{cpsTerm}(\mathcal{V}, \tau)$, with $\tau : \mathsf{cpsType}$. While terms do not return directly, the meta language type of a term includes the parameter τ to determine which type of argument "the top-level continuation" is expecting. A primop whose value is of type τ_2 and whose top-level continuation expects type τ_1 has type $\mathsf{cpsPrimop}(\mathcal{V}, \tau_1)\ \tau_2$.

We can now give a straightforward definition of a CPS translation that translates almost literally into Coq code. Coq has a notion of notation scopes to support overloaded parsing rules, and we will take advantage of similar conventions here to shorten the definition. For instance, the text bool can mean either the source or CPS boolean type, depending on context, and we will use the notation $\lfloor \cdot \rfloor$ to indicate both the translation $\lfloor \tau \rfloor$ of a type and the translation $\lfloor e \rfloor$ of a term. Our type translation is:

$$
\begin{array}{rcl}
\mathsf{letTerm} & : & \forall \mathcal{V} : \mathsf{cpsType} \rightarrow \star. \\
 & & \forall \tau_1, \tau_2 : \mathsf{cpsType}. \\
 & & \quad \mathsf{cpsTerm}(\mathcal{V}, \tau_1) \\
 & & \quad \rightarrow (\mathcal{V}(\tau_1) \rightarrow \mathsf{cpsTerm}(\mathcal{V}, \tau_2)) \\
 & & \quad \rightarrow \mathsf{cpsTerm}(\mathcal{V}, \tau_2) \\
\mathsf{letTerm}\ (\mathsf{halt}(x))\ e' & = & e'\ x \\
\mathsf{letTerm}\ (x_1\ x_2)\ e' & = & x_1\ x_2 \\
\mathsf{letTerm}\ (\mathsf{let}\ p\ \mathsf{in}\ e)\ e' & = & \mathsf{let}\ x = \mathsf{letPrim}\ p\ e' \\
 & & \mathsf{in}\ \mathsf{letTerm}\ (e\ x)\ e' \\
\mathsf{letPrim} & : & \forall \mathcal{V} : \mathsf{cpsType} \rightarrow \star. \\
 & & \forall \tau_1, \tau_2, \tau : \mathsf{cpsType}. \\
 & & \quad \mathsf{cpsPrimop}(\mathcal{V}, \tau_1)\ \tau \\
 & & \quad \rightarrow (\mathcal{V}(\tau_1) \rightarrow \mathsf{cpsTerm}(\mathcal{V}, \tau_2)) \\
 & & \quad \rightarrow \mathsf{cpsPrimop}(\mathcal{V}, \tau_2)\ \tau \\
\mathsf{letPrim}\ (\lambda f)\ e' & = & \lambda x.\ \mathsf{letTerm}\ (f\ x)\ e' \\
\mathsf{letPrim}\ p\ e' & = & p
\end{array}
$$

Figure 6. Term splicing

$$
\begin{array}{rcl}
\lfloor \cdot \rfloor & : & \mathsf{type} \rightarrow \mathsf{cpsType} \\
\lfloor \mathsf{bool} \rfloor & = & \mathsf{bool} \\
\lfloor \tau_1 \rightarrow \tau_2 \rfloor & = & (\lfloor \tau_1 \rfloor \times (\lfloor \tau_2 \rfloor \rightarrow 0)) \rightarrow 0
\end{array}
$$

We traverse type structure, changing all function types into continuation types where a return continuation has been added as an extra argument.

The let term form only allows us to bind a primop in a term. To define the main term translation, we want a derived let for binding terms in terms, and we can define it as the function letTerm (which is defined by mutual recursion with letPrim) in Figure 6.

Finally, we give the overall term translation in Figure 7.

The definition is deceptively easy; Coq accepts it with no further fuss, which implies that a proof of type preservation for the translation is implicit in the translation's definition. The trick to making this work lies in taking advantage of our freedom to pick smart instantiations for the variable type family $\mathcal{V}$. In particular, we see an odd variable type choice in the type of the term translation.

For a given $\mathcal{V}$ that we want to use for the resulting CPS term, we choose to type source variables with the function $\mathcal{V} \circ \lfloor \cdot \rfloor$, the composition of $\mathcal{V}$ with the type translation function. That is, while the translation takes source terms as input, we interpret their variables in a CPS-specific way. The source term is handed to us in parametric form, so it is no problem to choose its $\mathcal{V}$ to be the correct function. In doing so, we find that each variable in the term we produce has exactly the right type to use as a CPS variable in the translation result.

Our translation is a term of CIC, and CIC's strong normalization theorem implies that any application of the translation to a concrete, well-typed source term can be normalized to a concrete, well-typed CPS term. Coq will perform such normalizations for us automatically, during proofs and in independent queries. Coq will extract our translation to executable OCaml or Haskell code automatically.

Figure 8 gives the Coq code for the main translation. We again make use of sections, where, conceptually, we fix for the whole section the $\mathcal{V}$ choice `var` that we are compiling into, and, when we close the section, the function `cpsTerm` is extended to take `var` as

$$
\begin{array}{rcl}
\lfloor\cdot\rfloor & : & \forall\mathcal{V} : \mathsf{cpsType} \to \star.\ \forall\tau : \mathsf{type}. \\
 & & \quad \mathsf{term}(\mathcal{V} \circ \lfloor\cdot\rfloor)\ \tau \to \mathsf{cpsTerm}(\mathcal{V})\ \lfloor\tau\rfloor \\
\lfloor|x|\rfloor & = & \mathsf{halt}(x) \\
\lfloor\mathsf{true}\rfloor & = & \mathsf{let}\ x = \mathsf{true}\ \mathsf{in}\ \mathsf{halt}(x) \\
\lfloor\mathsf{false}\rfloor & = & \mathsf{let}\ x = \mathsf{false}\ \mathsf{in}\ \mathsf{halt}(x) \\
\lfloor e_1\ e_2\rfloor & = & \mathsf{letTerm}\ \lfloor e_1\rfloor\ (\widehat{\lambda} f. \\
 & & \quad \mathsf{letTerm}\ \lfloor e_2\rfloor\ (\widehat{\lambda} x. \\
 & & \quad \mathsf{let}\ k = \lambda r.\ \mathsf{halt}(r)\ \mathsf{in} \\
 & & \quad \mathsf{let}\ p = \langle x, k\rangle\ \mathsf{in} \\
 & & \quad f\ p)) \\
\lfloor\lambda f\rfloor & = & \mathsf{let}\ f = \lambda p. \\
 & & \quad \mathsf{let}\ x = \pi_1 p\ \mathsf{in} \\
 & & \quad \mathsf{let}\ k = \pi_2 p\ \mathsf{in} \\
 & & \quad \mathsf{letTerm}\ \lfloor f\ x\rfloor\ (\widehat{\lambda} r. \\
 & & \quad k\ r) \\
 & & \mathsf{in}\ \mathsf{halt}(f) \\
\lfloor\cdot\rfloor & : & \forall\tau : \mathsf{type}.\mathsf{Term}\ \tau \to \mathsf{CpsTerm}\ \lfloor\tau\rfloor \\
\lfloor E\rfloor & = & \widehat{\lambda}\mathcal{V} : \mathsf{cpsType} \to \star.\ \lfloor E\ (\mathcal{V} \circ \lfloor\cdot\rfloor)\rfloor
\end{array}
$$

Figure 7. CPS translation for STLC

```
Section cpsTerm.
  Variable var : ptype -> Type.

  Fixpoint cpsTerm t
      (e : term (fun t => var (cpsType t)) t)
      {struct e} : pterm var (cpsType t) :=
    match e in (term _ t)
        return (pterm var (cpsType t)) with
      | EVar _ v => PHalt (var := var) v
      | ETrue => x <- Tru; Halt x
      | EFalse => x <- Fals; Halt x
      | EApp _ _ e1 e2 =>
        f <-- cpsTerm e1;
        x <-- cpsTerm e2;
        k <- \r, PHalt (var := var) r;
        p <- [x, k];
        f @@ p
      | EAbs _ _ e' =>
        f <- PAbs (var := var) (fun p =>
          x <- #1 p;
          k <- #2 p;
          r <-- cpsTerm (e' x);
          k @@ r);
        Halt f
    end.
End cpsTerm.
```

Figure 8. Coq code for the STLC CPS translation

$$
\frac{}{(\widehat{\lambda}\_.\ \mathsf{bool})[\tau] \mapsto \mathsf{bool}}
\qquad
\frac{}{(\widehat{\lambda}\alpha.\ |\alpha|)[\tau] \mapsto \tau}
\qquad
\frac{}{(\widehat{\lambda}\_.\ |\alpha|)[\tau] \mapsto |\alpha|}
$$

$$
\frac{\tau_1[\tau] \mapsto \tau_1' \quad \tau_2[\tau] \mapsto \tau_2'}{(\widehat{\lambda}\alpha.\ \tau_1(\alpha) \to \tau_2(\alpha))[\tau] \mapsto \tau_1' \to \tau_2'}
$$

$$
\frac{\forall\alpha.\ (\widehat{\lambda}\alpha'.\tau_1(\alpha')(\alpha))[\tau] \mapsto \tau_1'(\alpha)}{(\widehat{\lambda}\alpha.\ \forall\tau_1(\alpha))[\tau] \mapsto \forall\tau_1'}
$$

Figure 9. Relational type variable substitution judgment

an extra argument. We use syntactic sugar for CPS terms that is defined elsewhere. Sometimes we need to drop down to using the raw constructors of the CPS language to help type inference. For instance, the snippet `PHalt (var := var) v` gives an explicit value for the implicit parameter `var` of the halt term constructor.

2.3 CPS Translation for System F

We can extend the development from the last subsection to arrive at a CPS translation for System F. The first wrinkle is that now the definition of the type languages becomes nontrivial, thanks to the presence of type variables. Fortunately PHOAS adapts to this change quite naturally, and we can produce the following revised version of our source type language definition, where the parameter $\mathcal{T}$ has type $\star$. From this point on in the paper, we will avoid textual names for constructors of inductive types, instead introducing constructors with their syntactic shorthands.

$$
\begin{array}{rcl}
\mathsf{type}(\mathcal{T}) & : & \star \\
|\cdot| & : & \mathcal{T} \to \mathsf{type}(\mathcal{T}) \\
\mathsf{bool} & : & \mathsf{type}(\mathcal{T}) \\
\cdot \to \cdot & : & \mathsf{type}(\mathcal{T}) \to \mathsf{type}(\mathcal{T}) \to \mathsf{type}(\mathcal{T}) \\
\forall\,\cdot & : & (\mathcal{T} \to \mathsf{type}(\mathcal{T})) \to \mathsf{type}(\mathcal{T})
\end{array}
$$

We have a variable injection form $|\alpha|$ as we had before at the term level, and we have a universal type constructor $\forall f$. We will write $\forall\alpha.\ \tau$ as shorthand for $\forall(\widehat{\lambda}\alpha.\ \tau)$.

Now we would like to define the syntax and typing of terms. To do this, we need to implement substitution of types for type variables in types. We cannot use the substitution function strategy from Section 2. The situation is roughly that, once we fix a particular variable type, we cannot implement as many syntactic functions, and we must fix a variable type before we can deconstruct syntax recursively. We can, however, implement some of these syntactic operations *relationally* after fixing a variable type. We will define substitution relationally with inference rules, following the approach used by Despeyroux et al. (1995).

In Figure 9, we define a judgment $\tau_1[\tau_2] \mapsto \tau_3$, where τ_1 is a function from type variables to types, and τ_2 and τ_3 are types. The meaning is that substituting τ_2 for the type variable that τ_1 abstracts over leads to τ_3. The functions that we give for τ_1 are interpreted as functions of the meta language, with the usual variable convention, so that, e.g., the functions $\widehat{\lambda}\alpha.\ |\alpha|$ and $\widehat{\lambda}\_.\ |\alpha|$ are provably distinct, as long as our domain of type variables has at least two elements.

Now we can define the syntax of terms, which are parameterized on two different variable types, in Figure 10. We have type abstractions $\Lambda\ f$ for functions f from type variables to types, and we have type application $e[\tau]$ for term e with a $\forall$ type. The type of the type application constructor includes τ', the type that results from substituting the type argument in the body of e's $\forall$ type. Not only that, but it is necessary to pass *a first-class substitution proof* to the type application constructor. The type $(\tau_1[\tau_2] \mapsto \tau')$ stands

$$
\begin{array}{rcl}
\mathsf{term}(\mathcal{T},\mathcal{V}) & : & \mathsf{type}(\mathcal{T}) \to \star \\
|\cdot| & : & \forall\tau : \mathsf{type}(\mathcal{T}).\ \mathcal{V}(\tau) \to \mathsf{term}(\mathcal{T},\mathcal{V})\ \tau \\
\mathsf{true} & : & \mathsf{term}(\mathcal{T},\mathcal{V})\ \mathsf{bool} \\
\mathsf{false} & : & \mathsf{term}(\mathcal{T},\mathcal{V})\ \mathsf{bool} \\
\cdot\ \cdot & : & \forall\tau_1,\tau_2 : \mathsf{type}(\mathcal{T}).\ \mathsf{term}(\mathcal{T},\mathcal{V})\ (\tau_1 \to \tau_2) \\
 & & \quad \to \mathsf{term}(\mathcal{T},\mathcal{V})\ \tau_1 \to \mathsf{term}(\mathcal{T},\mathcal{V})\ \tau_2 \\
\lambda\ \cdot & : & \forall\tau_1,\tau_2 : \mathsf{type}(\mathcal{T}).\ (\mathcal{V}(\tau_1) \to \mathsf{term}(\mathcal{T},\mathcal{V})\ \tau_2) \\
 & & \quad \to \mathsf{term}(\mathcal{T},\mathcal{V})\ (\tau_1 \to \tau_2) \\
\cdot[\cdot] & : & \forall\tau_1 : \mathcal{T} \to \mathsf{type}(\mathcal{T}).\ \forall\tau_2,\tau' : \mathsf{type}(\mathcal{T}). \\
 & & \quad \mathsf{term}(\mathcal{T},\mathcal{V})\ (\forall\tau_1) \to (\tau_1[\tau_2] \mapsto \tau') \\
 & & \quad \to \mathsf{term}(\mathcal{T},\mathcal{V})\ \tau' \\
\Lambda\ \cdot & : & \forall\tau : \mathcal{T} \to \mathsf{type}(\mathcal{T}). \\
 & & \quad (\forall\alpha : \mathcal{T}.\ \mathsf{term}(\mathcal{T},\mathcal{V})\ (\tau\ \alpha)) \\
 & & \quad \to \mathsf{term}(\mathcal{T},\mathcal{V})\ (\forall\tau)
\end{array}
$$

Figure 10. PHOAS syntax definitions for System F

for proofs of that particular proposition. To support building values of this type, we can formalize the substitution inference rules quite directly in Coq with an inductive definition. We can then build substitution proofs for concrete terms quite easily by interpreting the judgment definition as a logic program, and we can prove general lemmas about substitutions where the proofs are fully automated using logic programming. Using Coq's extraction mechanism, we can build an executable version of a translation where proof terms have been erased in a sound way. Thus, first-class proofs impose no runtime overhead, in contrast to the situation with, e.g., analogous implementations using GADTs in GHC Haskell.

Finally, we can define the packaged PHOAS versions of types and terms:

$$
\begin{array}{rcl}
\mathsf{Type} & : & \star \\
\mathsf{Type} & = & \forall\mathcal{T} : \star.\ \mathsf{type}(\mathcal{T}) \\
\mathsf{Term} & : & \mathsf{Type} \to \star \\
\mathsf{Term} & = & \widehat{\lambda}T : \mathsf{Type}.\ \forall\mathcal{T} : \star.\ \forall\mathcal{V} : \mathsf{type}(\mathcal{T}) \to \star. \\
 & & \quad \mathsf{term}(\mathcal{T},\mathcal{V})\ (T\ \mathcal{T})
\end{array}
$$

We define a CPS version of System F, following the same conventions as in the last subsection. Here we will only give the grammar for this language:

$$
\begin{array}{rrcl}
\text{Types} & \tau & ::= & \alpha \mid \mathsf{bool} \mid \tau \to 0 \mid \tau \times \tau \mid \forall\alpha.\ \tau \\
\text{Terms} & e & ::= & \mathsf{halt}(x) \mid x\ x \mid \mathsf{let}\ p\ \mathsf{in}\ f \\
\text{Primops} & p & ::= & |x| \mid \mathsf{true} \mid \mathsf{false} \mid \lambda f \\
 & & & \mid \langle x,x\rangle \mid \pi_1 x \mid \pi_2 x \mid x[\tau] \mid \Lambda g \\
\text{Term functions} & f & & \\
\text{Primop functions} & g & &
\end{array}
$$

Now we can adapt the CPS translation from the last subsection very naturally. Here is the new type translation:

$$
\begin{array}{rcl}
\lfloor\cdot\rfloor & : & \forall\mathcal{T} : \star.\ \mathsf{type}(\mathcal{T}) \to \mathsf{cpsType}(\mathcal{T}) \\
\lfloor|\alpha|\rfloor & = & |\alpha| \\
\lfloor\mathsf{bool}\rfloor & = & \mathsf{bool} \\
\lfloor\tau_1 \to \tau_2\rfloor & = & (\lfloor\tau_1\rfloor \times (\lfloor\tau_2\rfloor \to 0)) \to 0 \\
\lfloor\forall\tau\rfloor & = & \forall\alpha.\ (\lfloor\tau(\alpha)\rfloor \to 0) \to 0
\end{array}
$$

We apply a standard double negation transform (Harper and Lillibridge 1993) to ∀ types, which moves the type's body into a position where we can quantify over its free type variable without running afoul of the functions-never-return property of the CPS language.

$$
\begin{array}{rcl}
\lfloor\cdot\rfloor & : & \forall\mathcal{T} : \star.\ \forall\mathcal{V} : \mathsf{cpsType}(\mathcal{T}) \to \star.\ \forall\tau : \mathsf{type}(\mathcal{T}). \\
 & & \quad \mathsf{term}(\mathcal{T},\mathcal{V} \circ \lfloor\cdot\rfloor)\ \tau \to \mathsf{cpsTerm}(\mathcal{T},\mathcal{V})\ \lfloor\tau\rfloor \\
\lfloor e[\tau]\rfloor & = & \mathsf{letTerm}\ \lfloor e\rfloor\ (\widehat{\lambda}f. \\
 & & \quad \mathsf{let}\ f' = f[\lfloor\tau\rfloor]\ \mathsf{in} \\
 & & \quad \mathsf{let}\ k = \lambda r.\ \mathsf{halt}(r)\ \mathsf{in} \\
 & & \quad f'\ k) \\
\lfloor\Lambda\ e\rfloor & = & \mathsf{let}\ f = \Lambda\alpha.\ \lambda k. \\
 & & \quad \mathsf{letTerm}\ \lfloor e\ \alpha\rfloor\ (\widehat{\lambda}v. \\
 & & \quad k\ v) \\
 & & \mathsf{in}\ \mathsf{halt}(f) \\
\lfloor\cdot\rfloor & : & \forall T : \mathsf{Type}. \mathsf{Term}\ T \to \mathsf{CpsTerm}\ \lfloor T\rfloor \\
\lfloor E\rfloor & = & \widehat{\lambda}\mathcal{T} : \star.\ \widehat{\lambda}\mathcal{V} : \mathsf{cpsType}(\mathcal{T}) \to \star.\ \lfloor E\ \mathcal{T}\ (\mathcal{V} \circ \lfloor\cdot\rfloor)\rfloor
\end{array}
$$

Figure 11. Selected cases of CPS translation for System F

The letTerm and letPrim functions of the last subsection are readily adapted to System F, and we use them in the adapted term translation, whose new cases are shown in Figure 11.

Writing the term translation this way elides one detail. Like at the source level, building a CPS type application term requires providing a proof that a particular type variable substitution is valid. We prove the following theorem, where the substitution notation is overloaded for both the source and CPS languages:

THEOREM 1 (CPS translation of substitution proofs). *For all* τ_1, τ_2, *and* τ_3, *if* $\tau_1[\tau_2] \mapsto \tau_3$, *then* $(\widehat{\lambda}\alpha.\ \lfloor\tau_1(\alpha)\rfloor)[\lfloor\tau_2\rfloor] \mapsto \lfloor\tau_3\rfloor$.

A straightforward induction on the derivation of the premise proves Theorem 1. In Coq, the proof is literally just a statement of which induction principle to use, chained onto an invocation of a generic simplification tactic from the Lambda Tamer library. The real term translation references this theorem explicitly to build a CPS substitution proof from a source substitution proof.

2.4 Pattern Match Compilation

To have any hope of handling real programming languages, PHOAS must be able to cope with constructs that bind multiple variables at once in complicated ways. ML-style pattern matching provides a familiar example. In this subsection, we will show how to implement a compilation from pattern matching to a more primitive type theory. Our source language's grammar, in usual informal notation, is:

$$
\begin{array}{rrcl}
\text{Types} & \tau & ::= & \mathsf{unit} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \\
\text{Patterns} & p & ::= & x \mid \langle p,p\rangle \mid \mathsf{inl}\ p \mid \mathsf{inr}\ p \\
\text{Terms} & e & ::= & x \mid () \mid e\ e \mid \lambda x.\ e \mid \langle e,e\rangle \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \\
 & & & \mid (\mathsf{case}\ e\ \mathsf{of}\ \vec{p} \Rightarrow \vec{e} \mid \_ \Rightarrow e)
\end{array}
$$

To avoid dealing with inexhaustive match failures, we force case expressions to include default cases. Also, while the syntax allows a variable to appear multiple times in a pattern, our concrete CIC encoding makes pattern variables unique by construction.

The key issue in formalizing this language is deciding how to represent patterns and their uses to capture binding structure correctly, without making it too hard to write transformations. We start in Figure 12 by defining patterns similarly to terms from Section 2.2, but with an extra type index giving the types of the variables that a pattern binds. This index has type list type, using the Coq list constructor, with "cons" operator :: and concatenation operator ⊕.

$$
\begin{array}{rcl}
\mathsf{pat}(\mathcal{V}) & : & \mathsf{type} \rightarrow \mathsf{list\ type} \rightarrow \star \\
|\cdot| & : & \forall \tau : \mathsf{type}.\ \mathsf{pat}(\mathcal{V})\ \tau\ [\tau] \\
\langle \cdot, \cdot \rangle & : & \forall \tau_1, \tau_2 : \mathsf{type}.\ \forall \vec{\tau_1}, \vec{\tau_2} : \mathsf{list\ type}. \\
 & & \quad \mathsf{pat}(\mathcal{V})\ \tau_1\ \vec{\tau_1} \rightarrow \mathsf{pat}(\mathcal{V})\ \tau_2\ \vec{\tau_2} \\
 & & \quad \rightarrow \mathsf{pat}(\mathcal{V})\ (\tau_1 \times \tau_2)\ (\vec{\tau_1} \oplus \vec{\tau_2}) \\
\mathsf{inl}\ \cdot & : & \forall \tau_1, \tau_2 : \mathsf{type}.\ \forall \vec{\tau} : \mathsf{list\ type}. \\
 & & \quad \mathsf{pat}(\mathcal{V})\ \tau_1\ \vec{\tau} \rightarrow \mathsf{pat}(\mathcal{V})\ (\tau_1 + \tau_2)\ \vec{\tau} \\
\mathsf{inr}\ \cdot & : & \forall \tau_1, \tau_2 : \mathsf{type}.\ \forall \vec{\tau} : \mathsf{list\ type}. \\
 & & \quad \mathsf{pat}(\mathcal{V})\ \tau_2\ \vec{\tau} \rightarrow \mathsf{pat}(\mathcal{V})\ (\tau_1 + \tau_2)\ \vec{\tau}
\end{array}
$$

Figure 12. Pattern syntax

To make use of this binding information in the type we give case expressions, we will need an auxiliary type definition. The indexed heterogeneous list type family tuple is defined in the Lambda Tamer library:

$$
\begin{array}{rcl}
\mathsf{tuple} & : & \forall T : \star.\ (T \rightarrow \star) \rightarrow \mathsf{list}\ T \rightarrow \star \\
\mathsf{tuple}\ f\ [] & = & \mathsf{unit} \\
\mathsf{tuple}\ f\ (h :: t) & = & f h \times \mathsf{tuple}\ f\ t
\end{array}
$$

We can give the case constructor the following type, using tuple to represent groups of variables being bound at once:

$$
\begin{array}{rcl}
\mathsf{case}\ \cdot\ \mathsf{of}\ \cdot \Rightarrow \cdot \mid \_ \Rightarrow \cdot & : & \forall \tau_1, \tau_2 : \mathsf{type}.\ \mathsf{term}(\mathcal{V})\ \tau_1 \\
 & & \rightarrow \mathsf{list}\ (\Sigma \vec{\tau}.\ \mathsf{pat}(\mathcal{V})\ \tau_1\ \vec{\tau} \\
 & & \quad \times\ (\mathsf{tuple}\ \mathcal{V}\ \vec{\tau} \rightarrow \mathsf{term}(\mathcal{V})\ \tau_2)) \\
 & & \rightarrow \mathsf{term}(\mathcal{V})\ \tau_2 \rightarrow \mathsf{term}(\mathcal{V})\ \tau_2
\end{array}
$$

The list argument is the interesting one. We represent the pattern matching branches as a list of pairs of patterns and expressions. We need to use a Σ dependent pair type to enforce the relationship between the types of the variables a pattern binds and the types that the corresponding expression expects. The type family tuple $\mathcal{V}$ translates a list of types into the type of a properly-typed variable for each.

The elaborated version of this language is a small variation on the source language of Section 2.2. We give only the syntax, in standard informal style:

$$
\begin{array}{lrcl}
\text{Types} & \tau & ::= & \mathsf{unit} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \\
\text{Terms} & e & ::= & x \mid () \mid e\ e \mid \lambda x.\ e \\
 & & & \mid \langle e, e \rangle \mid \pi_1 e \mid \pi_2 e \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \\
 & & & \mid (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x \Rightarrow e \mid \mathsf{inr}\ x \Rightarrow e)
\end{array}
$$

We want to translate pattern matching in a way that avoids decomposing the same term twice in the dynamic execution of the translation of the same source case expression. To do this, we will use an intermediate representation of patterns that maps every possible "shape" of the discriminee to the proper expression to evaluate. elabTerm is the type family for terms of the target language.

$$
\begin{array}{rcl}
\mathsf{ctree}(\mathcal{V}) & : & \mathsf{type} \rightarrow \star \rightarrow \star \\
\mathsf{ctree}(\mathcal{V})\ (\tau_1 \times \tau_2)\ T & = & \mathsf{ctree}(\mathcal{V})\ \tau_1\ (\mathsf{ctree}(\mathcal{V})\ \tau_2\ T) \\
\mathsf{ctree}(\mathcal{V})\ (\tau_1 + \tau_2)\ T & = & \mathsf{ctree}(\mathcal{V})\ \tau_1\ T \times \mathsf{ctree}(\mathcal{V})\ \tau_2\ T \\
\mathsf{ctree}(\mathcal{V})\ \tau\ T & = & \mathsf{elabTerm}(\mathcal{V})\ \tau \rightarrow T
\end{array}
$$

ctree expands a type into a possibly exponential number of functions, one for each shape of that type. For instance, for any $T, \mathsf{ctree}(\mathcal{V})\ ((\mathsf{unit} + (\mathsf{unit} \rightarrow \mathsf{unit})) \times \mathsf{unit})\ T$ reduces to the term in Figure 13.

$$
\begin{array}{rcl}
\mathsf{xPat}(\mathcal{V}) & : & \forall \tau : \mathsf{type}.\ \forall \vec{\tau} : \mathsf{list\ type}.\ \forall T : \star. \\
 & & \quad \mathsf{pat}(\mathcal{V})\ \tau\ \vec{\tau} \\
 & & \quad \rightarrow (\mathsf{tuple}\ (\mathsf{elabTerm}(\mathcal{V}))\ \vec{\tau} \rightarrow T) \\
 & & \quad \rightarrow T \rightarrow \mathsf{ctree}(\mathcal{V})\ \tau\ T \\
\mathsf{xPat}(\mathcal{V})\ |x|\ s\ f & = & \mathsf{everywhere}\ (\widehat{\lambda} d.\ s\ (d, ())) \\
\mathsf{xPat}(\mathcal{V})\ \langle p_1, p_2 \rangle\ s\ f & = & \mathsf{xPat}(\mathcal{V})\ p_1 \\
 & & \quad (\widehat{\lambda} \vec{x_1}.\ \mathsf{xPat}(\mathcal{V})\ p_2 \\
 & & \qquad (\widehat{\lambda} \vec{x_2}.\ s\ (\vec{x_1} \oplus \vec{x_2}))\ f) \\
 & & \quad (\mathsf{everywhere}\ (\widehat{\lambda} \_.\ f)) \\
\mathsf{xPat}(\mathcal{V})\ (\mathsf{inl}\ p)\ s\ f & = & (\mathsf{xPat}(\mathcal{V})\ p\ s\ f, \mathsf{everywhere}\ (\widehat{\lambda} \_.\ f)) \\
\mathsf{xPat}(\mathcal{V})\ (\mathsf{inr}\ p)\ s\ f & = & (\mathsf{everywhere}\ (\widehat{\lambda} \_.\ f), \mathsf{xPat}(\mathcal{V})\ p\ s\ f)
\end{array}
$$

Figure 14. Pattern compilation

We can define a translation of pats into ctrees, as in Figure 14. The last two arguments of the function xPat are success and failure continuations. We overload $\oplus$ to denote concatenation of tuples. We use an auxiliary function everywhere, which takes an expression that needs no more free variables provided and builds a ctree that maps every shape to that expression.

The xPat function is the essence of the translation. The remaining pieces are a way of merging ctrees, a way of expanding them into expressions of the target language, and the translations for the kinds of expressions beyond case, which are very straightforward. As usual, all of these pieces have static types that guarantee that they map well-typed syntax to well-typed syntax.

2.5 Closure Conversion for Simply-Typed Lambda Calculus

The last three subsections have demonstrated how PHOAS supports convenient programming with a variety of different kinds of variable binding. In every one of these examples, details of variable identity have been unimportant. However, there *are* important classes of language formalization problems where variable identity is central. The example that we will use in this subsection is closure conversion, which has long been regarded as a tricky challenge problem for HOAS. In Twelf, closure conversion might be formalized using syntactic marker predicates over variables or another approach that leaves binding completely higher-order. In contrast, here we will use the opposite approach. To write particular functions, we can choose the variable type $\mathcal{V}$ to follow any of the standard first-order representation techniques. That is, PHOAS can function as something of a chameleon, passing back and forth between first-order and higher-order representations as is convenient.

To implement closure conversion for STLC, we chose to work with $\mathcal{V}$ as nat, the type of natural numbers, using a particular convention for choosing the number to assign to a variable each time we "go inside a binder." Variable numbers will be interpreted as de Bruijn levels, where a variable's number is calculated by finding its binder and counting how many other binders enclose the original binder. This gives us the convenient property that a variable's number is the same throughout the variable's scope. The encoding enjoys similar properties to de Bruijn indices (de Bruijn 1972), which count binders starting from a variable use and moving towards the AST root rather than in the opposite direction, but de Bruijn levels are more convenient in this case.

Since we are being explicit about variable identity, we can no longer get away with relying only on CIC's parametricity in defining the translation. We need to define a notion of well-formedness

$$(\mathsf{elabTerm}(\mathcal{V})\ \mathsf{unit} \to \mathsf{elabTerm}(\mathcal{V})\ \mathsf{unit} \to T) \times (\mathsf{elabTerm}(\mathcal{V})\ (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{elabTerm}(\mathcal{V})\ \mathsf{unit} \to T)$$

Figure 13. Example normalized ctree type

of de Bruijn level terms. Later we will prove that every parametric term is well-formed when instantiated with $\mathcal{V}$ as nat.

Our actual implementation of closure conversion is meant to come after CPS conversion in a compilation pipeline, but here we will treat the source language of Section 2.2 instead for simplicity. We define well-formedness with a recursive function over terms, parameterized on an explicit type environment. The judgment is also parameterized on a subset of the in-scope variables. The intended meaning is that only free variables included in this subset may be used. It is important that we are able to reason about well-formedness of terms in this way, because we will be choosing subsets of the variables in a term's environment when we pack those terms into closures.

$$\begin{array}{rcl}
\mathsf{isfree} & = & \mathsf{tuple}\ (\widehat{\lambda}\_ : \mathsf{type}.\ \mathsf{bool}) \\
\mathsf{wf} & : & \forall \Gamma : \mathsf{list\ type}.\ \forall \gamma : \mathsf{isfree}\ \Gamma.\ \forall \tau : \mathsf{type}. \\
 & & \quad \mathsf{term}(\mathsf{nat})\ \tau \to \star \\
\mathsf{wf}\ \gamma\ |n| & = & \gamma.n = \tau \text{ (where } \tau \text{ is the type passed in for } |n|) \\
\mathsf{wf}\ \gamma\ \mathsf{true} & = & \mathsf{unit} \\
\mathsf{wf}\ \gamma\ \mathsf{false} & = & \mathsf{unit} \\
\mathsf{wf}\ \gamma\ (e_1\ e_2) & = & \mathsf{wf}\ \gamma\ e_1 \times \mathsf{wf}\ \gamma\ e_2 \\
\mathsf{wf}\ \gamma\ (\lambda e) & = & \mathsf{wf}\ (\mathsf{true}, \gamma)\ (e(\mathsf{length}\ \gamma))
\end{array}$$

The notation $\gamma.n = \tau$ stands for the type of first-class proofs that the nth variable from the end of the associated Γ is assigned type τ and marked as present in γ. Since we number variables from the end of Γ, the λe case passes the term function e the length of γ as the value of its new free variable, while we extend γ with true to indicate that the new variable may be referenced.

Another central definition is that of the function for calculating the set of variables that occur free in a term. We use this function in the translation of every function abstraction, populating the closure we create with only the free variables. We use auxiliary functions isfree_none, which builds an isfree tuple of all false values; isfree_one, which builds an isfree tuple where only one position is marked true, based on the natural number argument passed in; and isfree_merge, which walks two isfrees for the same Γ, applying boolean "or" to the values in each position.

$$\begin{array}{rcl}
\mathsf{fvs} & : & \forall \tau : \mathsf{type}.\ \mathsf{term}(\mathsf{nat})\ \tau \to \forall \Gamma : \mathsf{list\ type}. \\
 & & \quad \mathsf{isfree}\ \Gamma \\
\mathsf{fvs}\ |n|\ \Gamma & = & \mathsf{isfree\_one}\ n \\
\mathsf{fvs\ true}\ \Gamma & = & \mathsf{isfree\_none} \\
\mathsf{fvs\ false}\ \Gamma & = & \mathsf{isfree\_none} \\
\mathsf{fvs}\ (e_1\ e_2)\ \Gamma & = & \mathsf{isfree\_merge}\ (\mathsf{fvs}\ e_1\ \Gamma)\ (\mathsf{fvs}\ e_2\ \Gamma) \\
\mathsf{fvs}\ (\lambda e)\ \Gamma & = & \pi_2\ (\mathsf{fvs}\ (e\ (\mathsf{length}\ \Gamma))\ (\tau :: \Gamma)) \\
 & & \quad \text{(where } \tau \text{ is the abstraction domain type)}
\end{array}$$

We need to prove a critical theorem about the relationship between wf and fvs, even if we just want to get our closure conversion function to type-check:

THEOREM 2 (Minimality of fvs). *For any Γ and τ, any γ for Γ, and any e of type τ, if* wf γ e, *then* wf (fvs e Γ) e.

The proof is based on two main lemmas. The first of them asserts that, when a term e is well-formed for any set of free variables for Γ, that term is also well-formed for every set of free variables containing fvs e Γ. The second lemma asserts that, when a term is well-formed for any set of free variables γ, every variable included in fvs e Γ is also included in γ. Both of these lemmas are proved by induction on the structure of the term, appealing to a few smaller lemmas characterizing the interactions of the isfree_$*$ functions with variable lookup.

The last element we need to present the closure conversion is a type of *explicit environments*. We implement a type family envOf by recursion over an isfree value. The resulting environment type contains variables for exactly those positions marked with true.

$$\begin{array}{rcl}
\mathsf{envOf}(\mathcal{V}) & : & \forall \Gamma : \mathsf{list\ type}.\ \mathsf{isfree}\ \Gamma \to \star \\
\mathsf{envOf}(\mathcal{V})\ []\ () & = & \mathsf{unit} \\
\mathsf{envOf}(\mathcal{V})\ (\tau :: \Gamma)\ (\mathsf{true}, \gamma) & = & \mathcal{V}(\tau) \times \mathsf{envOf}\ \Gamma\ \gamma \\
\mathsf{envOf}(\mathcal{V})\ (\tau :: \Gamma)\ (\mathsf{false}, \gamma) & = & \mathsf{envOf}\ \Gamma\ \gamma
\end{array}$$

The full closure conversion involves a lot of machinery, so we will only present some highlights here. The type of the translation builds on the familiar form from the previous subsections:

$$\begin{array}{rcl}
\mathsf{xTerm} & : & \forall \tau : \mathsf{type}.\ \forall e : \mathsf{term}(\mathsf{nat})\ \tau. \\
 & & \quad \forall \Gamma : \mathsf{list\ type}.\ \forall \gamma : \mathsf{isfree}\ \Gamma. \\
 & & \quad \mathsf{wf}\ e\ \gamma \to \mathsf{envOf}\ \Gamma\ \gamma \to \mathsf{ccTerm}(\mathcal{V})\ \lfloor \tau \rfloor
\end{array}$$

To translate a term, we must provide both a superset of its free variables and a well-formedness proof relative to that set. We can see why we want to require these proofs by looking at the translation case for variables. We use the meta-variables ϕ to stand for wf proofs and σ to stand for envOf explicit environments.

$$\mathsf{xTerm}\ |n|\ \phi\ \sigma \quad = \quad |\sigma(n) \sim \phi|$$

Here we use the notation $e \sim \phi$ to denote the casting of a CIC expression e of type τ to type τ', by way of presenting an explicit proof ϕ that $\tau = \tau'$. This is exactly the kind of proof provided to us by the variable case of wf.

To have these equality proofs available at the leaves of a term, we need to thread them throughout the translation, in cases like that for function application:

$$\begin{array}{rcl}
\mathsf{xTerm}\ (e_1\ e_2)\ \phi\ \sigma & = & (\mathsf{xTerm}\ e_1\ (\pi_1 \phi)\ \sigma) \\
 & & \quad (\mathsf{xTerm}\ e_2\ (\pi_2 \phi)\ \sigma)
\end{array}$$

We used a pair type in the wf definition, applying it Curry-Howard style as the type of proofs of a conjunction of two other wf derivations. Now we can use π_1 and π_2 to project out the two sub-proofs and apply them in the recursive xTerm calls.

Most of the action in closure conversion happens for the function abstraction case. We present a very sketchy picture of this case, since there are many auxiliary functions involved that are not particularly surprising.

$$\begin{array}{rcl}
\mathsf{xTerm}\ (\lambda e)\ \phi\ \sigma & = & \mathsf{let}\ f = \lambda x_{env}.\ \lambda x_{arg}. \\
 & & \quad \mathsf{let}\ \vec{x} = x_{env} \\
 & & \quad \mathsf{in\ xTerm}\ (e\ (\mathsf{length}\ \sigma)) \\
 & & \quad\quad (\mathsf{wfFvs}\ \phi)\ (x_{arg}, \vec{x}) \\
 & & \mathsf{in}\ f\ (\mathsf{makeEnv}\ (\mathsf{wfFvs}'\ \phi)\ \sigma)
\end{array}$$

The basic idea is that each function is modified to take its environment of free variables as a new first argument, which is a tuple of the appropriate size and types. The notation let $\vec{x} = x_{env}$ is for a recursive function that binds all of the constituent free variables into genuine variables by pulling them out of the tuple x_{env}. With those variables bound, we can translate the function body e. We use a function wfFvs for translating the proof of e's well-formedness for an arbitrary γ into a proof for e's real set of free variables, using Theorem 2. The explicit environment we pass to xTerm is formed by adding e's "real" argument x_{arg} to the front of an environment built from the $\vec{x}$. Finally, we unpack the closure we have built, using a function $\mathsf{makeEnv}$ to build an explicit environment by choosing values out of σ. We need another auxiliary proof-translation function wfFvs', again based on Theorem 2.

Our actual closure conversion case study works on the output of Section 2.2's CPS translation, which adds additional complications. We also combine the translations commonly called "closure conversion" and "hoisting" into a single translation, moving all function definitions to the top level at the same time that we change them to take their environments as arguments. If we implemented the phases separately, it would be harder to enforce that functions are really closed, since they would have some "off-limits" variables in their PHOAS scopes. With our implementation, the type of the closure conversion function guarantees not only that well-typed syntax is mapped to well-typed syntax, but also that the output terms contain only closed functions.

We export the final closure conversion as a function over universally-typed packages, so the messy use of de Bruijn levels is hidden completely from the outside. The closure conversion implementation is essentially working with a first-order representation, and more code is needed than if we had fixed a first-order representation from the start, since we need to convert our higher-order terms into their first-order equivalents. Thus, PHOAS would not make sense for an implementation that just did closure conversion. The benefit comes when one part of an implementation needs first-order terms, while other parts are able to take advantage of higher-order terms. PHOAS allows us to *compose* the two kinds of phases seamlessly, where phases need not telegraph through their types which representations they choose.

3. Proving Semantic Preservation

Writing the translations of the last section with dependently-typed abstract syntax has given us all of the benefits of type-preserving compilation, without the need to rely on ad-hoc testing to discover if our translations may sometimes propagate type annotations incorrectly. We would like to go even further and provide the classic deliverable of compiler verification, which is proof of semantic preservation. For some suitable notion of program meaning for each language we manipulate, we want to know that the output of a translation has the same meaning as the input. Following our past approach (Chlipala 2007), we choose a denotational style of meaning assignment that has been called *type-theoretic semantics* (Harper and Stone 2000). That is, we provide definitional compilers from all of the languages we formalize into CIC, and we construct machine-checked proofs using Coq's very good built-in support for reasoning about the terms of CIC, in contrast to working with an explicit operational or denotational semantics for it. Past uses of type-theoretic semantics have tended to use custom-tailored type theories, while we use a small, "universal" type theory for all object languages; hence, we call our approach *foundational type-theoretic semantics*.

We want to automate our proofs as far as possible, to minimize the overhead of adding new features to a language and its certified implementation. Towards this end, we have implemented a number of new tactics using Coq's tactical language (Delahaye 2000). This is a dynamically-typed language whose most important feature is a very general construct for pattern matching on CIC terms and proof sequents, with a novel backtracking semantics for pattern match failure.

$$
\begin{array}{rcl}
\llbracket\cdot\rrbracket & : & \mathsf{type} \longrightarrow \star \\
\llbracket\mathsf{bool}\rrbracket & = & \mathsf{bool} \\
\llbracket\tau_1 \longrightarrow \tau_2\rrbracket & = & \llbracket\tau_1\rrbracket \longrightarrow \llbracket\tau_2\rrbracket \\
\llbracket\cdot\rrbracket & : & \forall\tau : \mathsf{type}.\ \mathsf{term}(\llbracket\cdot\rrbracket)\ \tau \longrightarrow \llbracket\tau\rrbracket \\
\llbracket x \rrbracket & = & x \\
\llbracket\mathsf{true}\rrbracket & = & \mathsf{true} \\
\llbracket\mathsf{false}\rrbracket & = & \mathsf{false} \\
\llbracket e_1\ e_2\rrbracket & = & \llbracket e_1\rrbracket\,\llbracket e_2\rrbracket \\
\llbracket\lambda e\rrbracket & = & \widehat{\lambda} x.\ \llbracket e(x)\rrbracket \\
\llbracket\cdot\rrbracket & : & \forall\tau : \mathsf{type}.\ \mathsf{Term}\ \tau \longrightarrow \llbracket\tau\rrbracket \\
\llbracket E\rrbracket & = & \llbracket E\ \llbracket\cdot\rrbracket\rrbracket
\end{array}
$$

Figure 15. Denotation functions for STLC

The Lambda Tamer library contains about 100 lines of tactic code that we rely on in the proofs that we will sketch in this section. Most proofs are performed by looping through a number of different simplification procedures until no further progress can be made, at which point either the proof is finished or we can report the set of unproved subgoals to the user. The core simplification procedures we use are simplification of propositional structure, application of CIC computational reduction rules, Prolog-style higher-order logic programming, and rewriting with quantified equalities. We add a few tactics that simplify goals that use dependent types in tricky ways.

We also add a quantifier instantiation framework. It can be hard to prove goals that begin with $\exists$ quantifiers or use hypotheses that begin with $\forall$ quantifiers, because we need to pick instantiations for the quantified variables before proceeding. Thus, it is helpful to provide a quantifier instantiation tactic parameterized on a function that chooses an instantiating term given a CIC type. A default strategy of picking any properly-typed term that occurs in the goal works surprisingly well because of the rich dependent types that we use.

3.1 CPS Translation for Simply-Typed Lambda Calculus

The correctness proof for the CPS translation of Section 2.2 is the simplest and involves almost no code that is not a small constant factor away from the complexity of a standard pencil-and-paper solution. We argue that the proof is actually simpler than it would be on paper, because automation takes care of almost all of the details. The human proof architect really only needs to suggest lemmas and the right induction principles to use in proving them.

Before we can prove the correctness of the translation, we need to give dynamic semantics to our source and target languages. We can write a very simple denotation function for the source language, this time overloading the notation $\llbracket\cdot\rrbracket$ for the denotation functions for types and terms, as shown in Figure 15.

There is an interesting development hidden within the superficially trivial definition of the term denotation function. We choose the variable type family $\mathcal{V}$ to be the type denotation function. That is, we work with syntax trees where variables are actually the denotations of terms. This is a perfectly legal choice of variable type, as shown in the final line above, which defines the denotation of a universally packaged term. As a result, the translation of variables is trivial, and the translation of function abstractions can use a CIC binder which passes the bound variable directly to the syntactic abstraction body e.

Figure 16 gives the Coq code corresponding to Figure 15.

```
Fixpoint typeDenote (t : type) : Set :=
  match t with
    | TBool => bool
    | TArrow t1 t2 => typeDenote t1 -> typeDenote t2
  end.

Fixpoint termDenote t (e : term typeDenote t)
    {struct e} : typeDenote t :=
  match e in (term _ t) return (typeDenote t) with
    | EVar _ v => v
    | ETrue => true
    | EFalse => false
    | EApp _ _ e1 e2 =>
      (termDenote e1) (termDenote e2)
    | EAbs _ _ e' => fun x => termDenote (e' x)
  end.

Definition Term t := forall var, term var t.
Definition TermDenote t (E : Term t) :=
  termDenote (E _).
```

Figure 16. Coq code for STLC denotation functions

For space reasons, we omit the details of the semantics for the CPS language. It is in a slightly different form, where the meaning of term e of type τ is written $[\![e]\!]k$ for some continuation k of type $[\![\tau]\!] \to \mathsf{bool}$. The final result of evaluating e is thrown to k, which returns a boolean, the simplest type that we can use to express the results of a variety of possibly-failing tests.

To state the semantic correctness theorem, we define a standard semantic logical relation, by recursion on the structure of syntactic types:

$$\begin{array}{rcl} \simeq_\cdot & : & \forall\tau : \mathsf{type}.\ [\![\tau]\!] \to [\![\lfloor\tau\rfloor]\!] \to \star \\ b_1 \simeq_{\mathsf{bool}} b_2 & = & b_1 = b_2 \\ f_1 \simeq_{\tau_1 \to \tau_2} f_2 & = & \forall x_1.\ \forall x_2.\ x_1 \simeq_{\tau_1} x_2 \to \forall k.\ \exists r. \\ & & \quad f_2\ (x_2, k) = k\ r \wedge f_1\ x_1 \simeq_{\tau_2} r \end{array}$$

Now we can state semantic correctness as:

THEOREM 3 (Semantic correctness). *For every* τ : type *and* E : $\mathsf{Term}\ \tau$, *for any continuation* k : $[\![\lfloor\tau\rfloor]\!] \to \mathsf{bool}$, *there exists* $r : [\![\lfloor\tau\rfloor]\!]$ *such that* $[\![\lfloor E\rfloor]\!]k = k\ r$ *and* $[\![E]\!] \simeq_\tau r$.

When we specialize the theorem to object language type bool, we get that, for any E : $\mathsf{Term\ bool}$, $[\![\lfloor E\rfloor]\!](\widehat{\lambda} b.\ b) = [\![E]\!]$. To convince ourselves that this is the result we wanted, we only need to consider adequacy of the definitions of the syntax and semantics of the source and target languages, assuming that we believe that any compiler errors can be detected by boolean tests. This simplification will be even more welcome in the proofs of more complicated translations, where we do not want to include details of the logical relations we choose in the trusted code base.

Thinking informally, we can prove Theorem 3 by induction on the structure of E, relying on one other inductively-proved lemma about the correctness of $\mathsf{letTerm}$. Unfortunately, Coq has no concept of an induction principle for a function type, and that is how we are representing terms. Twelf's meta logic is all about supporting induction over types involving function spaces. Can we import some of that convenience to Coq? In the rest of this subsection, we focus on how to do that by introducing a kind of explicit well-formedness relation on terms. We can assert as an axiom that every term is well-formed, and indeed we believe that this is a consistent axiom, thanks to parametricity of CIC. Nonetheless, we have no proof of this worked out. It may be possible to adapt the proof for the Theory of Contexts (Bucalo et al. 2006) or apply the techniques suggested by Hofmann (1999) for reasoning about HOAS. Regardless of the form that our confidence in the axiom takes, it might be worthwhile to consider an extension to Coq that makes this axiom provable within the logic, in much the same way that support for inductive types was added to an earlier version of Coq, but we leave that for future work.

$$\frac{(x_1, x_2) \in \Gamma}{\Gamma \vdash |x_1| \equiv |x_2|} \qquad \frac{}{\Gamma \vdash \mathsf{true} \equiv \mathsf{true}} \qquad \frac{}{\Gamma \vdash \mathsf{false} \equiv \mathsf{false}}$$

$$\frac{\Gamma \vdash f_1 \equiv f_2 \quad \Gamma \vdash a_1 \equiv a_2}{\Gamma \vdash f_1\ a_1 \equiv f_2\ a_2}$$

$$\frac{\forall x_1, x_2.\ (x_1, x_2), \Gamma \vdash e_1(x_1) \equiv e_2(x_2)}{\Gamma \vdash \lambda e_1 \equiv \lambda e_2}$$

Figure 17. Term equivalence inference rules

At the same time, asserting the axiom is really only a convenience in our development. We could restate our theorems to require that the "axiom" holds when applied to any terms that are mentioned, and we could prove that our translations preserve well-formedness. This clutters PHOAS developments, but it is not hard to imagine some new support from Coq for automating these changes, letting the proof developer work with notation like what appears in our current development. We would be doing extra work to prove lemmas that seem likely to be instances of more general meta-theorems, but we would at least avoid explicit dependence on axioms. Any ground instance of the axiom is provable easily by a simple logic program.

In any case, it is convenient to have a well-formedness judgment defined for our PHOAS terms. Rather than define a traditional well-formedness judgment directly, we instead formalize *what it means for two terms with different concrete choices for* $\mathcal{V}$ *to be equivalent*. We say that a universally-packaged term is well-formed if and only if any choice of two $\mathcal{V}$ instantiations leads to a pair of equivalent terms. We denote the judgment as $\Gamma \vdash e_1 \equiv e_2$, where Γ is a set of pairs of variables from the $\mathcal{V}$ types of e_1 and e_2. In Figure 17, we give the equivalence judgment for STLC in the usual informal natural deduction style, omitting some complications arising from typing. Now we assert as an axiom that, for any E : $\mathsf{Term}\ \tau$ and any types $\mathcal{V}_1$ and $\mathcal{V}_2$, $\emptyset \vdash E\ \mathcal{V}_1 \equiv E\ \mathcal{V}_2$.

The judgment $\equiv$ suggests a useful proof strategy for Theorem 3. By inducting over $\equiv$ derivations, we can in effect perform *parallel induction* over a term E, where at each stage we have several versions of E available, corresponding to different choices of $\mathcal{V}$ but known to share the same structure. To prove Theorem 3, it is useful to do parallel induction where we choose $\mathcal{V}$ to be both the source-level type denotation function and the target-level denotation function composed with the type translation. In particular, the main action happens in proving this lemma:

LEMMA 1. *For every* τ : type, e_1 : $\mathsf{term}([\![\cdot]\!])\ \tau$, e_2 : $\mathsf{term}([\![\cdot]\!] \circ \lfloor\cdot\rfloor)\ \tau$, *and set of variable pairs* Γ*:*

- ***If*** $\Gamma \vdash e_1 \equiv e_2$
- ***And*** *for every* $(x_1, x_2) \in \Gamma$ *associated with source type* τ', *it follows that* $x_1 \simeq_{\tau'} x_2$,
- ***Then*** *for any continuation* k : $[\![\lfloor\tau\rfloor]\!] \to \mathsf{bool}$, *there exists* $r : [\![\lfloor\tau\rfloor]\!]$ *such that* $[\![\lfloor e_2\rfloor]\!]k = k\ r$ *and* $[\![e_1]\!] \simeq_\tau r$.

The proof script for this lemma involves stating a few proof hints, asking to induct on the $\equiv$ derivation, and calling the generic simplification and quantifier instantiation tactic. We derive Theorem 3 as an easy corollary, producing the initial $\equiv$ proof by using the axiom we asserted.

It is interesting to stop at this point and consider how "higher-order" this proof is. What have we gained over proofs with, for instance, nominal or de Bruijn representations? The main induction principle comes from the rules for the equivalence judgment, which includes a first-order list of variable pairs. Parts of the proof involve sub-proofs of membership in this list, which we can think of as isomorphic to natural numbers. Nonetheless, in crucial contrast to nominal proofs, our proofs require no treatment of reasoning about variable renamings or permutations; and, in contrast to de Bruijn proofs, the contexts of our equivalence judgment are freely reorderable, with no need to mirror reorderings in terms by tweaking variable indices. We could also go even further and parameterize the well-formedness judgment by a predicate on variable pairs, removing the explicit list and just requiring that free variable pairs satisfy the predicate. This solution ends up working a lot like the Twelf facility for defining regular worlds and seems to be "just as higher-order," though as the subject of an axiom it is perhaps harder to believe.

3.2 CPS Translation for System F

We can extend this correctness proof to Section 2.3's CPS translation for System F. The main change is that we choose to do parallel induction with three different choices of the type variable type $\mathcal{T}$; we consider versions of source types where variables are *source-level type denotations*, where variables are *target-level type denotations*, and where variables are *relations between source- and target-level denotation types*. We do this within an analogous parallel induction over terms.

The idea is that, as we recurse through term structure, we stash the appropriate specialization of our logical relation for each free type variable *in that variable* in the third parallel version of the term.

The logical relation from the last subsection is revised to deal properly with type variables and universal types. Instead of taking a single type as its main argument, it instead takes all three parallel versions of the source type. Figure 18 presents the relation.

We have omitted some explicit casts and other details needed to get the dependent typing to work out. The key new lemma that we must prove about this logical relation, compared to the last subsection, is that for any $\forall$ type body t and relation R over arbitrary types, $\simeq$ gives us the same relation when called with $t(R)$ as when called with the result of substituting R for t's free variable using the substitution relation $\cdot[\cdot] \mapsto \cdot$ from Section 2.3. Our proof of that lemma is hundreds of lines, since we do not yet have effective automation support for the uses of dependent typing that arise. With that lemma available, we prove the main theorem in a few dozen lines.

3.3 Pattern Match Compilation

The correctness proof for the pattern match compiler from Section 2.4 is almost trivial once the translation is defined. We do not need a logical relation, because the source and target type systems are identical; the final theorem is stated with simple equality. We need to state a few lemmas and give the right induction principles to use to prove them, but the proof is almost entirely automatic.

3.4 Closure Conversion for Simply-Typed Lambda Calculus

As for pattern match compilation, we state the closure conversion correctness theorem for Section 2.5's translation using equality. Since we have significantly more auxiliary functions than in the other examples, we need to state more lemmas about them, but the overall proof is again largely automated, relying on a few dozen carefully-chosen proof hints. We can also prove that every term is well-formed in the sense of Section 2.5's wf judgment, as a consequence of the standard higher-order well-formedness axiom that we assume.

Component	**Syntax**	**Semantics**	**Eq. Rel.**
STLC source	37	15	20
STLC CPS	72	29	41
STLC closure converted	94	38	-
System F source	73	26	78
System F CPS	108	38	-
Pattern matching source	84	49	-
Pattern matching target	67	13	-

Figure 19. Lines-of-code counts for the object languages in the main case studies

Component	**Translation**	**Correctness proof**
STLC CPS	54	67
STLC closure conversion	554	284
System F CPS	97	413
Match compilation	194	138

Figure 20. Lines-of-code counts for the translations in the main case studies

Feature	unit	$\times$	$+$	$\mathbb{N}$	Lists
Source syntax	5	18	20	16	16
Source semantics	2	4	8	8	4
Equivalence judgment	2	10	11	9	7
CPS syntax	5	18	20	16	16
CPS semantics	2	4	8	7	16
Translation	5	17	17	15	39
Logical relation	1	3	6	1	11
Lemmas	0	0	0	0	26
Proof hints	1	1	1	1	15

Figure 21. Lines-of-code counts for features added to the STLC CPS case study

4. Measuring the Overhead of Formal Proof

Implementations and correctness proofs for the translations of Sections 2.2 through 2.5 are included in our source distribution. In this section, we summarize the amounts of code needed for the different pieces of these components and for some additional experiments.

Figure 19 shows the number of lines of code used to formalize each language from the case studies. For each language, we show how many lines are needed to define its combined syntax and type system, along with how many lines are needed to give its dynamic semantics and how many lines are needed to define its equivalence judgment, if we needed one for that case study. For each translation, Figure 20 give the size of the translation proper along with the size of its correctness proof. The latter counts include both code to state theorems and code to prove them.

We also ran some experiments with extending our STLC CPS conversion with a number of standard types from functional programming, measuring how much we had to change our implementation to add each feature. Figure 21 gives the results. We added unit, with its single term constructor; product types, with a pair formation constructor and two projection operators; sum types,

$$\begin{array}{rcl}
\simeq_{\cdot,\cdot,\cdot} & : & \forall \tau_R : \mathsf{type}(\Sigma T_1, T_2 : \star.\ T_1 \rightarrow \mathcal{T}_2 \rightarrow \star).\ \forall \tau_1, \tau_2 : \mathsf{type}(\star).\ [\![\tau_1]\!] \rightarrow [\![\lfloor \tau_2 \rfloor]\!] \rightarrow \star \\
b_1 \simeq_{\mathsf{bool},\mathsf{bool},\mathsf{bool}} b_2 & = & b_1 = b_2 \\
f_1 \simeq_{\tau_{R1} \rightarrow \tau_{R2}, \tau_{11} \rightarrow \tau_{12}, \tau_{21} \rightarrow \tau_{22}} f_2 & = & \forall x_1.\ \forall x_2.\ x_1 \simeq_{\tau_{R1}, \tau_{11}, \tau_{21}} x_2 \rightarrow \forall k.\ \exists r.\ f_2\ (x_2, k) = k\ r \wedge f_1\ x_1 \simeq_{\tau_{R2}, \tau_{12}, \tau_{22}} r \\
v_1 \simeq_{|R|, |T_1|, |T_2|} v_2 & = & R\ v_1\ v_2 \\
f_1 \simeq_{\forall \tau_R, \forall \tau_1, \forall \tau_2} f_2 & = & \forall T_1, T_2 : \star.\ \forall R : T_1 \rightarrow T_2 \rightarrow \star.\ \forall k.\ \exists r.\ f_2\ T_2\ k = k\ r \wedge f_1\ T_1 \simeq_{\tau_R(R), \tau_1(T_1), \tau_2(T_2)} r \\
\_ \simeq_{\_}\ \_ & = & \mathsf{False}
\end{array}$$

Figure 18. Logical relation for System F CPS translation

with inl and inr injections and case analysis; natural numbers, with "zero" and "successor" constructors and one-level case analysis; and lists, with "nil" and "cons" operators and built-in "fold left" functions.

For each feature, we list how many lines were needed for its syntax/type system and dynamic semantics in the source and target languages, its inference rules for the source-level equivalence judgment, the CPS translation of its types and terms, its case in the logical relation used by the soundness proof, any extra lemmas used in that proof, and the proof hints added to be used by the automation machinery. Only adding list types required proving a new lemma, which has to do with folding over two lists in parallel, maintaining a binary relation between accumulators. The proof hints added are along the lines of "replace any variable of type unit with ()" and "when you see a pattern match on a value of sum type in the goal, try a case analysis on that value."

5. Related Work

PHOAS is weak HOAS (Despeyroux et al. 1995; Honsell et al. 2001) where we replace a global type parameter with a parameter bound locally and instantiated with different values throughout a development. In both settings, we rely on axioms to prove semantic correctness theorems, though we need no axioms for our type preservation theorems with PHOAS. The ability to choose different concrete variable types for different contexts gives PHOAS some additional power in both functional programming and proving. On the other hand, the axioms we assume for PHOAS are more complicated and language-specific than those weak HOAS assumes for the Theory of Contexts, though we could avoid the language specificity by encoding all syntax in a single parameterized universal syntax type.

Trifonov et al. (2000) used parametricity to facilitate an inductive definition of HOAS-style syntax for a language supporting intensional type analysis. They used kind polymorphism to rule out exotic terms in the encoding of type variable binders.

Guillemette and Monnier (2008) used GHC Haskell to implement a compiler with a proof of type preservation but not semantic preservation. The implementations of their transformations are very similar to ours. In one notable exception, they resort to first-order representation of type variables, to make theorems about substitution easier to prove. With Coq's support for automating higher-order proofs, we are able to stick to higher-order variable representation for both type and term variables.

There have been many studies of the classic first-order variable binding representations within proof assistants, including studies using nominal syntax with two classes of variables in LEGO (Mckinna and Pollack 1999), de Bruijn indices in LEGO (Altenkirch 1993), nominal syntax in Isabelle/HOL (Urban and Tasson 2005), and locally nameless syntax in Coq (Aydemir et al. 2008). All of these first-order approaches involve extra syntactic bookkeeping in the definition of functions over syntax and the statement and proofs of theorems about syntax. While this overhead should be compared to our use of term equivalence relations in PHOAS, we only pay that cost in our semantic correctness proofs, and our meta language parametricity lets us manifest proofs of well-formedness judgments where needed, saving us from the standard first-order technique of threading such proofs throughout a development.

Developments using HOAS have most commonly been done in Twelf (Pfenning and Schürmann 1999), which supports logic programming, but not functional programming, over syntax. Traditional HOAS removes the need to implement syntactic helper functions like substitution. We mostly get around the problem in PHOAS by sticking to type-theoretic formalizations that involve few low-level syntactic operations. Twelf's meta-logic includes many features for reasoning about judgment contexts in inductive proofs; with PHOAS, we are reimplementing special cases of those features, with examples like the term equivalence judgments parameterized on variable contexts. There have been several approaches proposed for functional programming over HOAS terms (Schürmann et al. 2001; Pientka 2008), but they all involve creating new type systems rather than working within a general-purpose type theory like CIC, and their implementations are still immature and lacking in the kind of "proof assistant ecosystem" associated with tools like Coq, Isabelle, and Twelf.

Several projects have considered "hybrid" approaches, where syntax is implemented with de Bruijn indices or another first-order technique at the lowest level, but a HOAS interface is built on top, including convenient induction principles. This has been implemented in Isabelle/HOL (Ambler et al. 2002), Nuprl (Barzilay and Allen 2002), Coq (Capretta and Felty 2006), and MetaPRL (Hickey et al. 2006). PHOAS has a close qualitative connection to these approaches, as it also allows switching between first-order and higher-order views of terms, as demonstrated in our closure conversion.

We already mentioned a few projects in compiler verification for first-order languages. The bibliography by Dave (2003) provides extensive pointers to other work.

Minamide and Okuma (2003) verified CPS translations in Isabelle/HOL, using a nominal representation, and Dargaye and Leroy (2007) used Coq to implement a certified CPS translation for a simply-typed lambda calculus with a number of ML-like features. The languages of the latter project are more realistic than those we have treated in our case studies, but the target language has the drawback that it includes two different classes of variables to make the translation easier to verify. Both projects pay the usual bookkeeping costs of first-order methods.

Tian (2006) formalized CPS translation correctness in Twelf. As Twelf contains no production-quality proof automation, the proofs are entirely manual, leading to a much larger development than in our corresponding case study.

6. Conclusion

We have shown how parametric higher-order abstract syntax (PHOAS) can be used to support convenient functional programming with the syntax of languages with nested variable binders. Proof assistants like Coq can be used to produce very compact and highly automated proofs of correctness for program transformations implemented with PHOAS. Translations that need to take variable identity into account take more effort to write, but the encoding allows for relatively direct implementations, and compiler phases that need variable identity can be composed with phases that do not, without sacrificing ease of development and proof for the latter category.

Acknowledgments

We thank Greg Morrisett, Ryan Wisnesky, and the anonymous referees for helpful feedback on drafts of this paper.

References

Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In *Proc. TLCA*, pages 13–28, 1993.

Simon Ambler, Roy L. Crole, and Alberto Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In *Proc. TPHOLs*, pages 13–30, 2002.

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

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMARK challenge. In *Proc. TPHOLs*, pages 50–65, 2005.

Eli Barzilay and Stuart F. Allen. Reflecting higher-order abstract syntax in Nuprl. In *Proc. TPHOLs (Track B)*, pages 23–32, 2002.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

Anna Bucalo, Furio Honsell, Marino Miculan, Ivan Scagnetto, and Martin Hofmann. Consistency of the theory of contexts. *J. Funct. Program.*, 16(3):327–372, 2006.

Venanzio Capretta and Amy P. Felty. Combining de Bruijn indices and higher-order abstract syntax in Coq. In *Proc. TYPES*, pages 63–77, 2006.

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

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.

Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In *Proc. TLCA*, pages 124–138, 1995.

Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proc. POPL*, pages 284–294, 1996.

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

Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Proc. POPL*, pages 206–219, 1993.

Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 341–387, 2000.

Jason Hickey, Aleksey Nogin, Xin Yu, and Alexei Kopylov. Mechanized meta-reasoning using a hybrid HOAS/de Bruijn representation and reflection. In *Proc. ICFP*, pages 172–183, 2006.

Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. LICS*, pages 204–213, 1999.

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.

James Mckinna and Robert Pollack. Some lambda calculus and type theory formalized. *J. Autom. Reason.*, 23(3):373–409, 1999.

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.

L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321, 1994.

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.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proc. POPL*, pages 371–382, 2008.

Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.

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

Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Proc. ICFP*, pages 82–93, 2000.

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.