A Higher-Order Abstract Syntax Approach to Verified Transformations on Functional Programs

Yuting Wang and Gopalan Nadathur(✉)

University of Minnesota, Minneapolis, USA
{yuting,gopalan}@cs.umn.edu

Abstract. We describe an approach to the verified implementation of transformations on functional programs that exploits the higher-order representation of syntax. In this approach, transformations are specified using the logic of hereditary Harrop formulas. On the one hand, these specifications serve directly as implementations, being programs in the language λProlog. On the other hand, they can be used as input to the Abella system which allows us to prove properties about them and thereby about the implementations. We argue that this approach is especially effective in realizing transformations that analyze binding structure. We do this by describing concise encodings in λProlog for transformations like typed closure conversion and code hoisting that are sensitive to such structure and by showing how to prove their correctness using Abella.

1 Introduction

This paper concerns the verification of compilers for functional (programming) languages. The interest in this topic is easily explained. Functional languages support an abstract view of computation that makes it easier to construct programs and the resulting code also has a flexible structure. Moreover, these languages have a strong mathematical basis that simplifies the process of proving programs to be correct. However, there is a proviso to this observation: to derive the mentioned benefit, the reasoning must be done relative to the abstract model underlying the language, whereas programs are typically executed only in their compiled form. To close the gap, it is important also to ensure that the compiler that carries out the translation preserves the meanings of programs.

The key role that compiler verification plays in overall program correctness has been long recognized; e.g. see [22,27] for early work on this topic. With the availability of sophisticated systems such as Coq [8], Isabelle [33] and HOL [15] for mechanizing reasoning, impressive strides have been taken in recent years towards actually verifying compilers for real languages, as seen, for instance, in the CompCert project [21]. Much of this work has focused on compiling imperative languages like C. Features such as higher-order and nested functions that are present in functional languages bring an additional complexity to their

P. Thiemann (Ed.): ESOP 2016, LNCS 9632, pp. 752–779, 2016.
DOI: 10.1007/978-3-662-49498-1_29

implementation. A common approach to treating such features is to apply transformations to programs that render them into a form to which more traditional compilation methods can be applied. These transformations must manipulate binding structure in complex ways, an aspect that requires special consideration at both the implementation and the verification level [3].

Applications such as those above have motivated research towards developing good methods for representing and manipulating binding structure. Two particular approaches that have emerged from this work are those that use the nameless representation of bound variables due to De Bruijn [9] and the nominal logic framework of Pitts [35]. These approaches provide an elegant treatment of aspects such as α-convertibility but do not directly support the analysis of binding structure or the realization of binding-sensitive operations such as substitution. A third approach, commonly known as the *higher-order abstract syntax* or HOAS approach, uses the abstraction operator in a typed λ-calculus to represent binding structure in object-language syntax. When such representations are embedded within a suitable logic, they lead to a succinct and flexible treatment of many binding related operations through β-conversion and unification.

The main thesis of this paper, shared with other work such as [7,16], is that the HOAS approach is in fact well-adapted to the task of implementing and verifying compiler transformations on functional languages. Our specific objective is to demonstrate the usefulness of a particular framework in this task. This framework comprises two parts: the λProlog language [30] that is implemented, for example, in the Teyjus system [36], and the Abella proof assistant [4]. The λProlog language is a realization of the hereditary Harrop formulas or HOHH logic [25]. We show that this logic, which uses the simply typed λ-calculus as a means for representing objects, is a suitable vehicle for specifying transformations on functional programs. Moreover, HOHH specifications have a computational interpretation that makes them *implementations* of compiler transformations. The Abella system is also based on a logic that supports the HOAS approach. This logic, which is called $\mathcal{G}$, incorporates a treatment of fixed-point definitions that can also be interpreted inductively or co-inductively. The Abella system uses these definitions to embed HOHH within $\mathcal{G}$ and thereby to reason directly about the specifications written in HOHH. As we show in this paper, this yields a convenient means for verifying implementations of compiler transformations.

An important property of the framework that we consider, as also of systems like LF [17] and Beluga [34], is that it uses a weak λ-calculus for representing objects. There have been attempts to derive similar benefits from using functional languages or the language underlying systems such as Coq. Some benefits, such as the correct implementation of substitution, can be obtained even in these contexts. However, the equality relation embodied in these systems is very strong and the analysis of λ-terms in them is therefore not limited to examining just their syntactic structure. This is a significant drawback, given that such examination plays a key role in the benefits we describe in this paper. In light of this distinction, we shall use the term λ-*tree syntax* [24] for the more restricted version of HOAS whose use is the focus of our discussions.

The rest of this paper is organized as follows. In Sect. 2 we introduce the reader to the framework mentioned above. We then show in succeeding sections how this framework can be used to implement and to verify transformations on functional programs. We conclude the paper by discussing the relationship of the ideas we describe here to other existing work.[1]

2 The Framework

We describe, in turn, the specification logic and λProlog, the reasoning logic, and the manner in which the Abella system embeds the specification logic.

2.1 The Specification Logic and λProlog

The HOHH logic is an intuitionistic and predicative fragment of Church's Simple Theory of Types [12]. Its types are formed using the function type constructor $\rightarrow$ over user defined primitive types and the distinguished type $\mathbf{o}$ for formulas. Expressions are formed from a user-defined *signature* of typed constants whose argument types do not contain $\mathbf{o}$ and the *logical constants* $\Rightarrow$ and $\&$ of type $\mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o}$ and Π_τ of type $(\tau \rightarrow \mathbf{o}) \rightarrow \mathbf{o}$ for each type τ not containing $\mathbf{o}$. We write $\Rightarrow$ and $\&$, which denote implication and conjunction respectively, in infix form. Further, we write $\Pi_\tau\ \lambda(x : \tau)M$, which represents the universal quantification of x over M, as $\Pi_\tau x\, M$.

The logic is oriented around two sets of formulas called *goal formulas* and *program clauses* that are given by the following syntax rules:

$$\begin{array}{lll} G & ::= & A \mid G \,\&\, G \mid D \Rightarrow G \mid \Pi_\tau x\, G \\ D & ::= & A \mid G \Rightarrow A \mid \Pi_\tau x\, D \end{array}$$

Here, A represents *atomic formulas* that have the form $(p\ t_1\ \ldots\ t_n)$ where p is a (user defined) *predicate constant*, *i.e.* a constant with target type $\mathbf{o}$. Goal formulas of the last two kinds are referred to as hypothetical and universal goals. Using the notation $\Pi_{\bar\tau}\bar x$ to denote a sequence of quantifications, we see that a program clause has the form $\Pi_{\bar\tau}\bar x\, A$ or $\Pi_{\bar\tau}\bar x\, G \Rightarrow A$. We refer to A as the head of such a clause and G as the body; in the first case the body is empty.

A collection of program clauses constitutes a *program*. A program and a signature represent a specification of all the goal formulas that can be derived from them. The derivability of a goal formula G is expressed formally by the judgment $\Sigma; \Theta; \Gamma \vdash G$ in which Σ is a signature, Θ is a collection of program clauses defined by the user and Γ is a collection of dynamically added program clauses. The validity of such a judgment—also called a sequent—is determined by provability in intuitionistic logic but can equivalently be characterized in a goal-directed fashion as follows. If G is conjunctive, it yields sequents for "solving" each of its conjuncts in the obvious way. If it is a hypothetical or a universal goal, then one of the following rules is used:

[1] The actual development of several of the proofs discussed in this paper can be found at the URL http://www-users.cs.umn.edu/~gopalan/papers/compilation/.

$$\frac{\Sigma; \Theta; \Gamma, D \vdash G}{\Sigma; \Theta; \Gamma \vdash D \Rightarrow G} \Rightarrow R \qquad \frac{(c \notin \Sigma) \quad \Sigma, c : \tau; \Theta; \Gamma \vdash G[c/x]}{\Sigma; \Theta; \Gamma \vdash \Pi_\tau x\, G} \Pi R$$

In the ΠR rule, c must be a constant not already in Σ; thus, these rules respectively cause the program and the signature to grow while searching for a derivation. Once G has been simplified to an atomic formula, the sequent is derived by generating an instance of a clause from Θ or Γ whose head is identical to G and by constructing a derivation of the corresponding body of the clause if it is non-empty. This operation is referred to as backchaining on a clause.

In presenting HOHH specifications in this paper we will show programs as a sequence of clauses each terminated by a period. We will leave the outermost universal quantification in these clauses implicit, indicating the variables they bind by using tokens that begin with uppercase letters. We will write program clauses of the form $G \Rightarrow A$ as A `:-` G. We will show goals of the form $G_1 \wedge G_2$ and $\Pi_\tau y\, G$ as G_1 `,`G_2 and **pi** $y : \tau \backslash\, G$, respectively, dropping the type annotation in the latter if it can be filled in uniquely based on the context. Finally, we will write abstractions as $y \backslash M$ instead of $\lambda y\, M$.

Program clauses provide a natural means for encoding rule based specifications. Each rule translates into a clause whose head corresponds to the conclusion and whose body represents the premises of the rule. These clauses embody additional mechanisms that simplify the treatment of binding structure in object languages. They provide λ-terms as a means for representing objects, thereby allowing binding to be reflected into an explicit meta-language abstraction. Moreover, recursion over such structure, that is typically treated via side conditions on rules expressing requirements such as freshness for variables, can be captured precisely through universal and hypothetical goals. This kind of encoding is concise and has logical properties that we can use in reasoning.

We illustrate the above ideas by considering the specification of the typing relation for the simply typed λ-calculus (STLC). Let N be the only atomic type. We use the HOHH type `ty` for representations of object language types that we build using the constants `n : ty` and `arr : ty → ty → ty`. Similarly, we use the HOHH type `tm` for encodings of object language terms that we build using the constants `app : tm → tm → tm` and `abs : ty → (tm → tm) → tm`. The type of the latter constructor follows our chosen approach to encoding binding: for example, we represent the STLC expression $(\lambda(y : N \to N)\, \lambda(x : N)\, (y\ x))$ by the HOHH term `(abs (arr n n) (y\ (abs n (x\ (app y x)))))`. Typing for the STLC is a judgment written as $\Gamma \vdash T$: Ty that expresses a relationship between a context Γ that assigns types to variables, a term T and a type Ty. Such judgments are derived using the following rules:

$$\frac{\Gamma \vdash T_1 : Ty_1 \to Ty_2 \quad \Gamma \vdash T_2 : Ty_1}{\Gamma \vdash T_1\ T_2 : Ty_2} \qquad \frac{\Gamma, y : Ty_1 \vdash T : Ty_2}{\Gamma \vdash \lambda(y : Ty_1)\, T : (Ty_1 \to Ty_2)}$$

The second rule has a proviso: y must be fresh to Γ. In the λ-tree syntax approach, we encode typing as a binary relation between a term and a type, treating the typing context implicitly via dynamically added clauses. Using the predicate `of` to represent this relation, we define it through the following clauses:

```
of (app T1 T2) Ty2 :- of T1 (arr Ty1 Ty2), of T2 Ty1.
of (abs Ty1 T) (arr Ty1 Ty2) :- pi y\ (of y Ty1 ⇒ of (T y) Ty2).
```

The second clause effectively says that `(abs Ty1 T)` has the type `(arr Ty1 Ty2)` if `(T y)` has type `Ty2` in an extended context that assigns `y` the type `Ty1`. Note that the universal goal ensures that `y` is new and, given our encoding of terms, `(T y)` represents the body of the object language abstraction in which the bound variable has been replaced by this new name.

The rules for deriving goal formulas give HOHH specifications a computational interpretation. We may also leave particular parts of a goal unspecified, representing them by "meta-variables," with the intention that values be found for them that make the overall goal derivable. This idea underlies the language λProlog that is implemented, for example, in the Teyjus system [36].

2.2 The Reasoning Logic and Abella

The inference rules that describe a relation are usually meant to be understood in an "if and only if" manner. Only the "if" interpretation is relevant to using rules to effect computations and their encoding in the HOHH logic captures this part adequately. To reason about the *properties* of the resulting computations, however, we must formalize the "only if" interpretation as well. This functionality is realized by the logic $\mathcal{G}$ that is implemented in the Abella system.

The logic $\mathcal{G}$ is also based on an intuitionistic and predicative version of Church's Simple Theory of Types. Its types are like those in HOHH except that the type **prop** replaces **o**. Terms are formed from user-defined constants whose argument types do not include **prop** and the following logical constants: **true** and **false** of type **prop**; $\wedge$, $\vee$ and $\rightarrow$ of type $\mathbf{prop} \rightarrow \mathbf{prop} \rightarrow \mathbf{prop}$ for conjunction, disjunction and implication; and, for every type τ not containing **prop**, the quantifiers $\forall_\tau$ and $\exists_\tau$ of type $(\tau \rightarrow \mathbf{prop}) \rightarrow \mathbf{prop}$ and the equality symbol $=_\tau$ of type $\tau \rightarrow \tau \rightarrow \mathbf{prop}$. The formula $B =_\tau B'$ holds if and only if B and B' are of type τ and equal under $\alpha\beta\eta$ conversion. We will omit the type τ in logical constants when its identity is clear from the context.

A novelty of $\mathcal{G}$ is that it is parameterized by *fixed-point definitions*. Such definitions consist of a collection of *definitional clauses* each of which has the form $\forall \bar{x}, A \triangleq B$ where A is an atomic formula all of whose free variables are bound by $\bar{x}$ and B is a formula whose free variables must occur in A; A is called the head of such a clause and B is called its body.[2] To illustrate definitions, let `olist` represent the type of lists of HOHH formulas and let `nil` and `::`, written in infix form, be constants for building such lists. Then the append relation at the `olist` type is defined in $\mathcal{G}$ by the following clauses:

```
append nil L L;
append (X :: L1) L2 (X :: L3) ≜ append L1 L2 L3.
```

This presentation also illustrates several conventions used in writing definitions: clauses of the form $\forall \bar{x}, A \triangleq \mathbf{true}$ are abbreviated to $\forall \bar{x}, A$, the outermost

[2] To be acceptable, definitions must cumulatively satisfy certain stratification conditions [23] that we adhere to in the paper but do not explicitly discuss.

universal quantifiers in a clause are made implicit by representing the variables they bind by tokens that start with an uppercase letter, and a sequence of clauses is written using semicolon as a separator and period as a terminator.

The proof system underlying $\mathcal{G}$ interprets atomic formulas via the fixed-point definitions. Concretely, this means that definitional clauses can be used in two ways. First, they may be used in a backchaining mode to derive atomic formulas: the formula is matched with the head of a clause and the task is reduced to deriving the corresponding body. Second, they can also be used to do case analysis on an assumption. Here the reasoning structure is that if an atomic formula holds, then it must be because the body of one of the clauses defining it holds. It therefore suffices to show that the conclusion follows from each such possibility.

The clauses defining a particular predicate can further be interpreted inductively or coinductively, leading to corresponding reasoning principles relative to that predicate. As an example of how this works, consider proving

```
∀L1 L2 L3, append L1 L2 L3 → append L1 L2 L3' → L3 = L3'
```

assuming that we have designated `append` as an inductive predicate. An induction on the first occurrence of `append` then allows us to assume that the entire formula holds any time the leftmost atomic formula is replaced by a formula that is obtained by unfolding its definition and that has `append` as its predicate head.

Many arguments concerning binding require the capability of reasoning over structures with free variables where each such variable is treated as being distinct and not further analyzable. To provide this capability, $\mathcal{G}$ includes the special *generic* quantifier ∇_τ, pronounced as "nabla", for each type τ not containing **prop** [26]. In writing this quantifier, we, once again, elide the type τ. The rules for treating ∇ in an assumed formula and a formula in the conclusion are similar: a "goal" with `(∇x M)` in it reduces to one in which this formula has been replaced by `M[c/x]` where `c` is a fresh, unanalyzable constant called a *nominal constant*. Note that ∇ has a meaning that is different from that of $\forall$: for example, `(∇ x y, x = y →` **false**`)` is provable but `(∀ x y, x = y →` **false**`)` is not.

$\mathcal{G}$ allows the ∇ quantifier to be used also in the heads of definitions. The full form for a definitional clause is in fact $\forall \bar{x} \nabla \bar{z}, A \triangleq B$, where the ∇ quantifiers scope only over A. In generating an instance of such a clause, the variables in $\bar{z}$ must be replaced with nominal constants. The quantification order then means that the instantiations of the variables in $\bar{x}$ cannot contain the constants used for $\bar{z}$. This extension makes it possible to encode structural properties of terms in definitions. For example, the clause `(∇ x, name x)` defines `name` to be a recognizer of nominal constants. Similarly, the clause `(∇ x, fresh x B)` defines `fresh` such that `(fresh X B)` holds just in the case that `X` is a nominal constant and `B` is a term that does not contain `X`. As a final example, consider the following clauses in which `of` is the typing predicate from the previous subsection.

```
ctx nil;
∇x, ctx (of x T :: L) ≜ ctx L.
```

These clauses define `ctx` such that `(ctx L)` holds exactly when `L` is a list of type assignments to distinct variables.

2.3 The Two-Level Logic Approach

Our framework allows us to write specifications in HOHH and reason about them using $\mathcal{G}$. Abella supports this *two-level logic approach* by encoding HOHH derivability in a definition and providing a convenient interface to it. The user program and signature for these derivations are obtained from a λProlog program file. The state in a derivation is represented by a judgment of the form $\{\Gamma \vdash G\}$. where Γ is the list of dynamically added clauses; additions to the signature are treated implicitly via nominal constants. If Γ is empty, the judgment is abbreviated to {G}. The theorems that are to be proved mix such judgments with other ones defined directly in Abella. For example, the uniqueness of typing for the STLC based on its encoding in HOHH can be stated as follows:

```
∀L M T T', ctx L → {L ⊢ of M T} → {L ⊢ of M T'} → T = T'.
```

This formula talks about the typing of *open* terms relative to a dynamic collection of clauses that assign unique types to (potentially) free variables.

The ability to mix specifications in HOHH and definitions in Abella provides considerable expressivity to the reasoning process. This expressivity is further enhanced by the fact that both HOHH and $\mathcal{G}$ support the λ-tree syntax approach. We illustrate these observations by considering the explicit treatment of substitutions. We use the type `map` and the constant `map: tm → tm → map` to represent mappings for individual variables (encoded as nominal constants) and a list of such mappings to represent a substitution; for simplicity, we overload the constructors `nil` and `::` at this type. Then the predicate `subst` such that `subst ML M M'` holds exactly when `M'` is the result of applying the substitution `ML` to `M` can be defined by the following clauses:

```
subst nil M M;
∇x, subst ((map x V) :: ML) (R x) M ≜ subst ML (R V) M.
```

Observe how quantifier ordering is used in this definition to create a "hole" where a free variable appears in a term and application is then used to plug the hole with the substitution. This definition makes it extremely easy to prove structural properties of substitutions. For example, the fact that substitution distributes over applications and abstractions can be stated as follows:

```
∀ML M1 M2 M', subst ML (app M1 M2) M' →
   ∃M1' M2', M' = app M1' M2' ∧ subst ML M1 M1' ∧ subst ML M2 M2'.
∀ML R T M', subst ML (abs T R) M' →
   ∃R', M' = abs T R' ∧ ∇x, subst ML (R x) (R' x).
```

An easy induction over the definition of substitution proves these properties.

As another example, we may want to characterize relationships between closed terms and substitutions. For this, we can first define well-formed terms through the following HOHH clauses:

```
tm (app M N) :- tm M, tm N.
tm (abs T R) :- pi x\ tm x ⇒ tm (R x).
```

Then we characterize the context used in `tm` derivations in Abella as follows:

```
tm_ctx nil;
∇x, tm_ctx (tm x :: L) ≜ tm_ctx L.
```

Intuitively, if `tm_ctx L` and `{L ⊢ tm M}` hold, then `M` is a well-formed term whose free variables are given by `L`. Clearly, if `{tm M}` holds, then `M` is closed. Now we can state the fact that a closed term is unaffected by a substitution:

```
∀ML M M', {tm M} → subst ML M M' → M = M'.
```

Again, an easy induction on the definition of substitutions proves this property.

3 Implementing Transformations on Functional Programs

We now turn to the main theme of the paper, that of showing the benefits of our framework in the verified implementation of compilation-oriented program transformations for functional languages. The case we make has the following broad structure. Program transformations are often conveniently described in a syntax-directed and rule-based fashion. Such descriptions can be encoded naturally using the program clauses of the HOHH logic. In transforming functional programs, special attention must be paid to binding structure. The λ-tree syntax approach, which is supported by the HOHH logic, provides a succinct and logically precise means for treating this aspect. The executability of HOHH specifications renders them immediately into implementations. Moreover, the logical character of the specifications is useful in the process of reasoning about their correctness.

This section is devoted to substantiating our claim concerning implementation. We do this by showing how to specify transformations that are used in the compilation of functional languages. An example we consider in detail is that of closure conversion. Our interest in this transformation is twofold. First, it is an important step in the compilation of functional programs: it is, in fact, an enabler for other transformations such as code hoisting. Second, it is a transformation that involves a complex manipulation of binding structure. Thus, the consideration of this transformation helps shine a light on the special features of our framework. The observations we make in the context of closure conversion are actually applicable quite generally to the compilation process. We close the section by highlighting this fact relative to other transformations that are of interest.

3.1 The Closure Conversion Transformation

The closure conversion transformation is designed to replace (possibly nested) functions in a program by *closures* that each consist of a function and an environment. The function part is obtained from the original function by replacing its free variables by projections from a new environment parameter. Complementing this, the environment component encodes the construction of a value for the new parameter in the enclosing context. For example, when this transformation is applied to the following pseudo OCaml code segment

$$
\begin{array}{l}
T ::= \mathbb{N} \mid T_1 \rightarrow T_2 \mid \textbf{unit} \mid T_1 \times T_2 \\
M ::= n \mid x \mid \textbf{pred}\ M \mid M_1 + M_2 \\
\quad \mid \textbf{if}\ M_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3 \\
\quad \mid () \mid (M_1, M_2) \mid \textbf{fst}\ M \mid \textbf{snd}\ M \\
\quad \mid \textbf{let}\ x = M_1\ \textbf{in}\ M_2 \\
\quad \mid \textbf{fix}\ f\, x.M \mid (M_1\ M_2) \\
V ::= n \mid \textbf{fix}\ f\, x.M \mid () \mid (V_1, V_2)
\end{array}
$$

Fig. 1. Source language syntax

$$
\begin{array}{l}
T ::= \mathbb{N} \mid T_1 \rightarrow T_2 \mid T_1 \Rightarrow T_2 \mid \textbf{unit} \mid T_1 \times T_2 \\
M ::= n \mid x \mid \textbf{pred}\ M \mid M_1 + M_2 \\
\quad \mid \textbf{if}\ M_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3 \\
\quad \mid () \mid (M_1, M_2) \mid \textbf{fst}\ M \mid \textbf{snd}\ M \\
\quad \mid \textbf{let}\ x = M_1\ \textbf{in}\ M_2 \mid\ \lambda x.M \mid (M_1\ M_2) \\
\quad \mid \langle M_1, M_2 \rangle \mid \textbf{open}\ \langle x_f, x_e \rangle = M_1\ \textbf{in}\ M_2 \\
V ::= n \mid \lambda x.M \mid () \mid (V_1, V_2) \mid \langle V_1, V_2 \rangle
\end{array}
$$

Fig. 2. Target language syntax

```
let x = 2 in let y = 3 in (fun z. z + x + y)
```

it will yield

```
let x = 2 in let y = 3 in <fun z e. z + e.1 + e.2, (x,y)>
```

We write `<F,E>` here to represent a closure whose function part is `F` and environment part is `E`, and `e.i` to represent the i-th projection applied to an "environment parameter" `e`. This transformation makes the function part independent of the context in which it appears, thereby allowing it to be extracted out to the top-level of the program.

The Source and Target Languages. Figures 1 and 2 present the syntax of the source and target languages that we shall use in this illustration. In these figures, T, M and V stand respectively for the categories of types, terms and the terms recognized as values. $\mathbb{N}$ is the type for natural numbers and n corresponds to constants of this type. Our languages include some arithmetic operators, the conditional and the tuple constructor and destructors; note that **pred** represents the predecessor function on numbers, the behavior of the conditional is based on whether or not the "condition" is zero and **fst** and **snd** are the projection operators on pairs. The source language includes the recursion operator **fix** which abstracts simultaneously over the function and the parameter; the usual abstraction is a degenerate case in which the function parameter does not appear in the body. The target language includes the expressions $\langle M_1, M_2 \rangle$ and (**open** $\langle x_f, x_e \rangle = M_1$ **in** M_2) representing the formation and application of closures. The target language does not have an explicit fixed point constructor. Instead, recursion is realized by parameterizing the function part of a closure with a function component; this treatment should become clear from the rules for typing closures and for evaluating the application of closures that we present below. The usual forms of abstraction and application are included in the target language to simplify the presentation of the transformation. The usual function type is reserved for closures; abstractions are given the type $T_1 \Rightarrow T_2$ in the target language. We abbreviate $(M_1, \ldots, (M_n, ()))$ by $(M_1, \ldots, M_n)$ and **fst** (**snd** $(\ldots(\textbf{snd}\ M)))$ where **snd** is applied $i-1$ times for $i \geq 1$ by $\pi_i(M)$.

$$\frac{}{\rho \triangleright n \rightsquigarrow n}\ \text{cc-nat} \qquad \frac{(x \mapsto M) \in \rho}{\rho \triangleright x \rightsquigarrow M}\ \text{cc-var} \qquad \frac{\rho \triangleright x_1 \rightsquigarrow M_1 \quad \ldots \quad \rho \triangleright x_n \rightsquigarrow M_n}{\rho \triangleright (x_1, \ldots, x_n) \rightsquigarrow_e (M_1, \ldots, M_n)}\ \text{cc-env}$$

$$\frac{\rho \triangleright M \rightsquigarrow M'}{\rho \triangleright \mathbf{pred}\ M \rightsquigarrow \mathbf{pred}\ M'}\ \text{cc-pred} \qquad \frac{\rho \triangleright M_1 \rightsquigarrow M_1' \quad \rho \triangleright M_2 \rightsquigarrow M_2'}{\rho \triangleright M_1 + M_2 \rightsquigarrow M_1' + M_2'}\ \text{cc-plus}$$

$$\frac{\rho \triangleright M \rightsquigarrow M' \quad \rho \triangleright M_1 \rightsquigarrow M_1' \quad \rho \triangleright M_2 \rightsquigarrow M_2'}{\rho \triangleright \mathbf{if}\ M\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \rightsquigarrow \mathbf{if}\ M'\ \mathbf{then}\ M_1'\ \mathbf{else}\ M_2'}\ \text{cc-if} \qquad \frac{}{\rho \triangleright () \rightsquigarrow ()}\ \text{cc-unit}$$

$$\frac{\rho \triangleright M_1 \rightsquigarrow M_1' \quad \rho \triangleright M_2 \rightsquigarrow M_2'}{\rho \triangleright (M_1, M_2) \rightsquigarrow (M_1', M_2')}\ \text{cc-pair} \qquad \frac{\rho \triangleright M \rightsquigarrow M'}{\rho \triangleright \mathbf{fst}\ M \rightsquigarrow \mathbf{fst}\ M'}\ \text{cc-fst} \qquad \frac{\rho \triangleright M \rightsquigarrow M'}{\rho \triangleright \mathbf{snd}\ M \rightsquigarrow \mathbf{snd}\ M'}\ \text{cc-snd}$$

$$\frac{\rho \triangleright M_1 \rightsquigarrow M_1' \quad \rho, x \mapsto y \triangleright M_2 \rightsquigarrow M_2'}{\rho \triangleright \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 \rightsquigarrow \mathbf{let}\ y = M_1'\ \mathbf{in}\ M_2'}\ \text{cc-let} \quad y \text{ must be fresh}$$

$$\frac{\rho \triangleright M_1 \rightsquigarrow M_1' \quad \rho \triangleright M_2 \rightsquigarrow M_2'}{\rho \triangleright M_1\ M_2 \rightsquigarrow \mathbf{let}\ g = M_1'\ \mathbf{in}\ \mathbf{open}\ \langle x_f, x_e \rangle = g\ \mathbf{in}\ x_f\ (g, M_2', x_e)}\ \text{cc-app} \quad g \text{ must be fresh}$$

$$\frac{(x_1, \ldots, x_n) \supseteq \mathbf{fvars}(\mathbf{fix}\ f\ x.M) \quad \rho \triangleright (x_1, \ldots, x_n) \rightsquigarrow_e M_e \quad \rho' \triangleright M \rightsquigarrow M'}{\rho \triangleright \mathbf{fix}\ f\ x.M \rightsquigarrow \langle \lambda p.\mathbf{let}\ g = \pi_1(p)\ \mathbf{in}\ \mathbf{let}\ y = \pi_2(p)\ \mathbf{in}\ \mathbf{let}\ x_e = \pi_3(p)\ \mathbf{in}\ M', M_e \rangle}\ \text{cc-fix}$$

where $\rho' = (x \mapsto y, f \mapsto g, x_1 \mapsto \pi_1(x_e), \ldots, x_n \mapsto \pi_n(x_e))$ and p, g, y, and x_e are fresh variables

Fig. 3. Closure conversion rules

Typing judgments for both the source and target languages are written as $\Gamma \vdash M : T$, where Γ is a list of type assignments for variables. The rules for deriving typing judgments are routine, with the exception of those for introducing and eliminating closures in the target language that are shown below:

$$\frac{\vdash M_1 : ((T_1 \to T_2) \times T_1 \times T_e) \Rightarrow T_2 \quad \Gamma \vdash M_2 : T_e}{\Gamma \vdash \langle M_1, M_2 \rangle : T_1 \to T_2}\ \text{cof-clos}$$

$$\frac{\Gamma \vdash M_1 : T_1 \to T_2 \quad \Gamma, x_f : ((T_1 \to T_2) \times T_1 \times l) \Rightarrow T_2, x_e : l \vdash M_2 : T}{\Gamma \vdash \mathbf{open}\ \langle x_f, x_e \rangle = M_1\ \mathbf{in}\ M_2 : T}\ \text{cof-open}$$

In cof-clos, the function part of a closure must be typable in an empty context. In cof-open, x_f, x_e must be names that are new to Γ. This rule also uses a "type" l whose meaning must be explained. This symbol represents a new type constant, different from $\mathbb{N}$ and () and any other type constant used in the typing derivation. This constraint in effect captures the requirement that the environment of a closure should be opaque to its user.

The operational semantics for both the source and the target language is based on a left to right, call-by-value evaluation strategy. We assume that this is given in small-step form and, overloading notation again, we write $M \hookrightarrow_1 M'$ to denote that M evaluates to M' in one step in whichever language is under consideration. The only evaluation rules that may be non-obvious are the ones for applications. For the source language, they are the following:

$$\frac{M_1 \hookrightarrow_1 M_1'}{M_1\ M_2 \hookrightarrow_1 M_1'\ M_2} \qquad \frac{M_2 \hookrightarrow_1 M_2'}{V_1\ M_2 \hookrightarrow_1 V_1\ M_2'} \qquad \frac{}{(\mathbf{fix}\ f\ x.M)\ V \hookrightarrow_1 M[\mathbf{fix}\ f\ x.M/f, V/x]}$$

For the target language, they are the following:

$$\frac{M_1 \hookrightarrow_1 M_1'}{\textbf{open}\ \langle x_f, x_e\rangle = M_1\ \textbf{in}\ M_2 \hookrightarrow_1 \textbf{open}\ \langle x_f, x_e\rangle = M_1'\ \textbf{in}\ M_2}$$

$$\frac{}{\textbf{open}\ \langle x_f, x_e\rangle = \langle V_f, V_e\rangle\ \textbf{in}\ M_2 \hookrightarrow_1 M_2[V_f/x_f, V_e/x_e]}$$

One-step evaluation generalizes in the obvious way to n-step evaluation that we denote by $M \hookrightarrow_n M'$. Finally, we write $M \hookrightarrow V$ to denote the evaluation of M to the value V through 0 or more steps.

The Transformation. In the general case, we must transform terms under mappings for their free variables: for a function term, this mapping represents the replacement of the free variables by projections from the environment variable for which a new abstraction will be introduced into the term. Accordingly, we specify the transformation as a 3-place relation written as $\rho \triangleright M \rightsquigarrow M'$, where M and M' are source and target language terms and ρ is a mapping from (distinct) source language variables to target language terms. We write $(\rho, x \mapsto M)$ to denote the extension of ρ with a mapping for x and $(x \mapsto M) \in \rho$ to mean that ρ contains a mapping of x to M. Figure 3 defines the $\rho \triangleright M \rightsquigarrow M'$ relation in a rule-based fashion; these rules use the auxiliary relation $\rho \triangleright (x_1, \ldots, x_n) \rightsquigarrow_e M_e$ that determines an environment corresponding to a tuple of variables. The cc-let and cc-fix rules have a proviso: the bound variables, x and f, x respectively, should have been renamed to avoid clashes with the domain of ρ. Most of the rules have an obvious structure. We comment only on the ones for transforming fixed point expressions and applications. The former translates into a closure. The function part of the closure is obtained by transforming the body of the abstraction, but under a new mapping for its free variables; the expression $(x_1, \ldots, x_n) \supseteq \textbf{fvars}(\textbf{fix}\ f\, x.M)$ means that all the free variables of $(\textbf{fix}\ f\, x.M)$ appear in the tuple. The environment part of the closure correspondingly contains mappings for the variables in the tuple that are determined by the enclosing context. Note also that the parameter for the function part of the closure is expected to be a triple, the first item of which corresponds to the function being defined recursively in the source language expression. The transformation of a source language application makes clear how this structure is used to realize recursion: the constructed closure application has the effect of feeding the closure to its function part as the first component of its argument.

3.2 A λProlog Rendition of Closure Conversion

Our presentation of the implementation of closure conversion has two parts: we first show how to encode the source and target languages and we then present a λProlog specification of the transformation. In the first part, we discuss also the formalization of the evaluation and typing relations; these will be used in the correctness proofs that we develop later.

Encoding the Languages. We first consider the encoding of types. We will use `ty` as the λProlog type for this encoding for both languages. The constructors `tnat`, `tunit` and `prod` will encode, respectively, the natural number, unit and pair types. There are two arrow types to be treated. We will represent → by `arr` and ⇒ by `arr'`. The following signature summarizes these decisions.

```
tnat,tunit : ty          arr,prod,arr' : ty → ty → ty
```

We will use the λProlog type `tm` for encodings of source language terms. The particular constructors that we will use for representing the terms themselves are the following, assuming that `nat` is a type for representations of natural numbers:

```
nat : nat → tm                    pred,fst,snd : tm → tm        unit : tm
plus,pair,app : tm → tm → tm      ifz : tm → tm → tm → tm
let : tm → (tm → tm) → tm         fix : (tm → tm → tm) → tm
```

The only constructors that need further explanation here are `let` and `fix`. These encode binding constructs in the source language and, as expected, we use λProlog abstraction to capture their binding structure. Thus, **let** $x = n$ **in** x is encoded as `(let (nat n) (x\x))`. Similarly, the λProlog term `(fix (f\x\ app f x))` represents the source language expression (**fix** $f\,x.f\ x$).

We will use the λProlog type `tm'` for encodings of target language terms. To represent the constructs the target language shares with the source language, we will use "primed" versions of the λProlog constants seen earlier; *e.g.*, `unit'` of type `tm'` will represent the null tuple. Of course, there will be no constructor corresponding to `fix`. We will also need the following additional constructors:

```
abs' : (tm' → tm') → tm'     clos' : tm' → tm' → tm'
open' : tm' → (tm' → tm' → tm') → tm'
```

Here, `abs'` encodes λ-abstraction and `clos'` and `open'` encode closures and their application. Note again the λ-tree syntax representation for binding constructs.

Following Sect. 2, we represent typing judgments as relations between terms and types, treating contexts implicitly via dynamically added clauses that assign types to free variables. We use the predicates `of` and `of'` to encode typing in the source and target language respectively. The clauses defining these predicates are routine and we show only a few pertaining to the binding constructs. The rule for typing fixed points in the source language translates into the following.

```
of (fix R) (arr T1 T2) :-
   pi f\ pi x\ of f (arr T1 T2) ⇒ of x T1 ⇒ of (R f x) T2.
```

Note how the required freshness constraint is realized in this clause: the universal quantifiers over `f` and `x` introduce new names and the application `(R f x)` replaces the bound variables with these names to generate the new typing judgment that must be derived. For the target language, the main interesting rule is for typing the application of closures. The following clause encodes this rule.

```
of' (open' M R) T :- of' M (arr T1 T2),
 pi f\ pi e\ pi l\ of' f (arr' (prod (arr T1 T2) (prod T1 l)) T2) ⇒
                          of' e l ⇒ of' (R f e) T.
```

Here again we use universal quantifiers in goals to encode the freshness constraint. Note also how the universal quantifier over the variable `l` captures the opaqueness quality of the type of the environment of the closure involved in the construct.

We encode the one step evaluation rules for the source and target languages using the predicates `step` and `step'`. We again consider only a few interesting cases in their definition. Assuming that `val` and `val'` recognize values in the source and target languages, the clauses for evaluating the application of a fixed point and a closure are the following.

```
step (app (fix R) V) (R (fix R) V) :- val V.
step' (open' (clos' F E) R) (R F E) :- val' (clos' F E).
```

Note here how application in the meta-language realizes substitution.

We use the predicates `nstep` (which relates a natural number and two terms) and `eval` to represent the n-step and full evaluation relations for the source language, respectively. These predicates have obvious definitions. The predicates `nstep'` and `eval'` play a similar role for the target language.

Specifying Closure Conversion. To define closure conversion in λProlog, we need a representation of mappings for source language variables. We use the type `map` and the constant `map : tm → tm' → map` to represent the mapping for a single variable.[3] We use the type `map_list` for lists of such mappings, the constructors `nil` and `::` for constructing such lists and the predicate `member` for checking membership in them. We also need to represent lists of source and target language terms. We will use the types `tm_list` and `tm'_list` for these and for simplicity of discussion, we will overload the list constructors and predicates at these types. Polymorphic typing in λProlog supports such overloading but this feature has not yet been implemented in Abella; we overcome this difficulty in the actual development by using different type and constant names for each case.

The crux in formalizing the definition of closure conversion is capturing the content of the cc-fix rule. A key part of this rule is identifying the free variables in a given source language term. We realize the requirement by defining a predicate `fvars` that is such that if `(fvars M L1 L2)` holds then `L1` is a list that includes all the free variables of `M` and `L2` is another list that contains only the free variables of `M`. We show a few critical clauses in the definition of this predicate, omitting ones whose structure is easy predict.

```
fvars X _ nil :- notfree X.
fvars Y Vs (Y :: nil) :- member Y Vs.
fvars (nat _) _ nil.
```

[3] This mapping is different from the one considered in Sect. 2.3 in that it is from a *source* language variable to a *target* language term.

```
fvars (plus M1 M2) Vs FVs :-
  fvars M1 Vs FVs1, fvars M2 Vs FVs2, combine FVs1 FVs2 FVs.
...
fvars (let M R) Vs FVs :- fvars M Vs FVs1,
  (pi x\ notfree x ⇒ fvars (R x) Vs FVs2), combine FVs1 FVs2 FVs.
fvars (fix R) Vs FVs :-
  pi f\ pi x\ notfree f ⇒ notfree x ⇒ fvars (R f x) Vs FVs.
```

The predicate `combine` used in these clauses is one that holds between three lists when the last is a combination of the elements of the first two. The essence of the definition of `fvars` is in the treatment of binding constructs. Viewed operationally, the body of such a construct is descended into after instantiating the binder with a new variable marked `notfree`. Thus, the variables that are marked in this way correspond to exactly those that are explicitly bound in the term and only those that are not so marked are collected through the second clause. It is important also to note that the specification of `fvars` has a completely logical structure; this fact can be exploited during verification.

The cc-fix rule requires us to construct an environment representing the mappings for the variables found by `fvars`. The predicate `mapenv` specified by the following clauses provides this functionality.

```
mapenv nil _ unit.
mapenv (X::L) Map (pair' M ML) :- member (map X M) Map, mapenv L Map ML.
```

The cc-fix rule also requires us to create a new mapping from the variable list to projections from an environment variable. Representing the list of projection mappings as a function from the environment variable, this relation is given by the predicate `mapvar` that is defined by the following clauses.

```
mapvar nil (e\ nil).
mapvar (X::L) (e\ (map X (fst' e))::(Map (snd' e))) :- mapvar L Map.
```

We can now specify the closure conversion transformation. We provide clauses below that define the predicate `cc` such that `(cc Map Vs M M')` holds if `M'` is a transformed version of `M` under the mapping `Map` for the variables in `Vs`; we assume that `Vs` contains all the free variables of `M`.

```
cc _ _ (nat N) (nat' N).
cc Map Vs X M :- member (map X M) Map.
cc Map Vs (pred M) (pred' M') :- cc Map Vs M M'.
cc Map Vs (plus M1 M2) (plus' M1' M2') :-
   cc Map Vs M1 M1', cc Map Vs M2 M2'.
cc Map Vs (ifz M M1 M2) (ifz' M' M1' M2') :-
   cc Map Vs M M', cc Map Vs M1 M1', cc Map Vs M2 M2'.
cc Map Vs unit unit'.
cc Map Vs (pair M1 M2) (pair' M1' M2') :-
   cc Map Vs M1 M1', cc Map Vs M2 M2'.
cc Map Vs (fst M) (fst' M') :- cc Map Vs M M'.
cc Map Vs (snd M) (snd' M') :- cc Map Vs M M'.
cc Map Vs (let M R) (let' M' R') :- cc Map Vs M M',
  pi x\ pi y\ cc ((map x y) :: Map) (x :: Vs) (R x) (R' y).
```

```
cc Map Vs (fix R) (clos' (abs' (p\ let' (fst' p) (g\
                    let' (fst' (snd' p)) (y\
                    let' (snd' (snd' p)) (e\ R' g y e))))) E) :-
  fvars (fix R) Vs FVs, mapenv FVs Map E, mapvar FVs NMap,
  pi f\ pi x\ pi g\ pi y\ pi e\
   cc ((map x y)::(map f g)::(NMap e)) (x::f::FVs) (R f x) (R' g y e).
cc Map Vs (app' M1 M2)
   (let' M1' (g\ open' g (f\e\ app' f (pair' g (pair' M2' e))))) :-
  cc Map Vs M1 M1', cc Map Vs M2 M2'.
```

These clauses correspond very closely to the rules in Fig. 3. Note especially the clause for transforming an expression of the form `(fix R)` that encodes the content of the cc-fix rule. In the body of this clause, `fvars` is used to identify the free variables of the expression, and `mapenv` and `mapvar` are used to create the reified environment and the new mapping. In both this clause and in the one for transforming a `let` expression, the λ-tree representation, universal goals and (meta-language) applications are used to encode freshness and renaming requirements related to bound variables in a concise and logically precise way.

3.3 Implementing Other Transformations

We have used the ideas discussed in the preceding subsections in realizing other transformations such as code hoisting and conversion to continuation-passing style (CPS). These transformations are part of a tool-kit used by compilers for functional languages to convert programs into a form from which compilation may proceed in a manner similar to that for conventional languages like C.

Our implementation of the CPS transformation is based on the one-pass version described by Danvy and Filinski [13] that identifies and eliminates the so-called administrative redexes on-the-fly. This transformation can be encoded concisely and elegantly in λProlog by using meta-level redexes for administrative redexes. The implementation is straightforward and similar ones that use the HOAS approach have already been described in the literature; *e.g.* see [37].

Our implementation of code hoisting is more interesting: it benefits in an essential way once again from the ability to analyze binding structure. The code hoisting transformation lifts nested functions that are closed out into a flat space at the top level in the program. This transformation can be realized as a recursive procedure: given a function $(\lambda x.M)$, the procedure is applied to the subterms of M and the extracted functions are then moved out of $(\lambda x.M)$. Of course, for this movement to be possible, it must be the case that the variable x does not appear in the functions that are candidates for extraction. This "dependency checking" is easy to encode in a logical way within our framework.

To provide more insight into our implementation of code-hoisting, let us assume that it is applied after closure conversion and that its source and target languages are both the language shown in Fig. 2. Applying code hoisting to any term will result in a term of the form

$$\textbf{let } f_1 = M_1 \textbf{ in } \ldots \textbf{let } f_n = M_n \textbf{ in } M$$

where, for $1 \leq i \leq n$, M_i corresponds to an extracted function. We will write this term below as (**letf** $\vec{f} = \vec{M}$ **in** M) where $\vec{f} = (f_1, \ldots, f_n)$ and, correspondingly, $\vec{M} = (M_1, \ldots, M_n)$.

We write the judgment of code hoisting as $(\rho \triangleright M \rightsquigarrow_{ch} M')$ where ρ has the form $(x_1, \ldots, x_n)$. This judgment asserts that M' is the result of extracting all functions in M to the top level, assuming that ρ contains all the bound variables in the context in which M appears. The relation is defined by recursion on the structure of M. The main rule that deserves discussion is that for transforming functions. This rule is the following:

$$\frac{\rho, x \triangleright M \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M'}{\rho \triangleright \lambda x.M \rightsquigarrow_{ch} \textbf{letf } (\vec{f}, g) = (\vec{F}, \lambda f.\lambda x.\textbf{letf } \vec{f} = (\pi_1(f), \ldots, \pi_n(f)) \textbf{ in } M') \textbf{ in } g\ \vec{f}}$$

We assume here that $\vec{f} = (f_1, \ldots, f_n)$ and, by an abuse of notation, we let $(g\ \vec{f})$ denote $(g\ (f_1, \ldots, f_n))$. This rule has a side condition: x must not occur in $\vec{F}$. Intuitively, the term $(\lambda x.M)$ is transformed by extracting the functions from within M and then moving them further out of the scope of x. Note that this transformation succeeds only if none of the extracted functions depend on x. The resulting function is then itself extracted. In order to do this, it must be made independent of the (previously) extracted functions, something that is achieved by a suitable abstraction; the expression itself becomes an application to a tuple of functions in an appropriate let environment.

It is convenient to use a special representation for the result of code hoisting in specifying it in λProlog. Towards this end, we introduce the following constants:

```
hbase : tm' → tm'
habs : (tm' → tm') → tm'
htm : tm'_list → tm' → tm'
```

Using these constants, the term (**letf** $(f_1, \ldots, f_n) = (M_1, \ldots, M_n)$ **in** M) that results from code hoisting will be represented by

```
htm (M1 :: ... :: Mn :: nil) (habs (f1\ ... (habs (fn\ hbase M)))).
```

We use the predicate `ch : tm' → tm' → o` to represent the code hoisting judgment. The context ρ in the judgment will be encoded implicitly through dynamically added program clauses that specify the translation of each variable `x` as `(htm nil (hbase x))`. In this context, the rule for transforming functions, the main rule of interest, is encoded in the following clause:

```
ch (abs' M) M'' :-
   (pi x\ ch x (htm nil (hbase x)) ⇒ ch (M x) (htm FE (M' x))),
   extract FE M' M''.
```

As in previous specifications, a universal and a hypothetical goal are used in this clause to realize recursion over binding structure. Note also the completely logical encoding of the requirement that the function argument must not occur in the nested functions extracted from its body: quantifier ordering ensures that `FE` cannot be instantiated by a term that contains `x` free in it. We have used

the predicate `extract` to build the final result of the transformation from the transformed form of the function body and the nested functions extracted from it; the definition of this predicate is easy to construct and is not provided here.

4 Verifying Transformations on Functional Programs

We now consider the verification of λProlog implementations of transformations on functional programs. We exploit the two-level logic approach in this process, treating λProlog programs as HOHH specifications and reasoning about them using Abella. Our discussions below will show how we can use the λ-tree syntax approach and the logical nature of our specifications to benefit in the reasoning process. Another aspect that they will bring out is the virtues of the close correspondence between rule based presentations and HOHH specifications: this correspondence allows the structure of informal proofs over inference rule style descriptions to be mimicked in a formalization within our framework.

We use the closure conversion transformation as our main example in this exposition. The first two subsections below present, respectively, an informal proof of its correctness and its rendition in Abella. We then discuss the application of these ideas to other transformations. Our proofs are based on logical relation style definitions of program equivalence. Other forms of semantics preservation have also been considered in the literature. Our framework can be used to advantage in formalizing these approaches as well, an aspect we discuss in the last subsection.

4.1 Informal Verification of Closure Conversion

To prove the correctness of closure conversion, we need a notion of equivalence between the source and target programs. Following [28], we use a logical relation style definition for this purpose. A complication is that our source language includes recursion. To overcome this problem, we use the idea of step indexing [1,2]. Specifically, we define the following mutually recursive simulation relation $\sim$ between closed source and target terms and equivalence relation $\approx$ between closed source and target values, each indexed by a type and a step measure.

$$
\begin{array}{l}
M \sim_{T;k} M' \iff \forall j \leq k. \forall V. M \hookrightarrow_j V \supset \exists V'. M' \hookrightarrow V' \wedge V \approx_{T;k-j} V'; \\
n \approx_{\mathbb{N};k} n; \qquad () \approx_{\mathbf{unit};k} (); \\
(V_1, V_2) \approx_{(T_1 \times T_2);k} (V_1', V_2') \iff V_1 \approx_{T_1;k} V_1' \wedge V_2 \approx_{T_2;k} V_2'; \\
(\mathbf{fix}\ f\, x.M) \approx_{T_1 \to T_2;k} \langle V', V_e \rangle \iff \forall j < k. \forall V_1, V_1', V_2, V_2'. \\
\quad V_1 \approx_{T_1;j} V_1' \supset V_2 \approx_{T_1 \to T_2;j} V_2' \supset M[V_2/f, V_1/x] \sim_{T_2;j} V'\ (V_2', V_1', V_e).
\end{array}
$$

Note that the definition of $\approx$ in the fixed point/closure case uses $\approx$ negatively at the same type. However, it is still a well-defined notion because the index decreases. The cumulative notion of equivalence, written $M \sim_T M'$, corresponds to two expressions being equivalent under *any* index.

Analyzing the simulation relation and using the evaluation rules, we can show the following "compatibility" lemma for various constructs in the source language.

Lemma 1. *1. If* $M \sim_{\mathbb{N};k} M'$ *then* **pred** $M \sim_{\mathbb{N};k}$ **pred** M'. *If also* $N \sim_{\mathbb{N};k} N'$ *then* $M + N \sim_{\mathbb{N};k} M' + N'$.
2. If $M \sim_{T_1 \times T_2;k} M'$ *then* **fst** $M \sim_{T_1;k}$ **fst** M' *and* **snd** $M \sim_{T_2;k}$ **snd** M'.
3. If $M \sim_{T_1;k} M'$ *and* $N \sim_{T_2;k} N'$ *then* $(M, N) \sim_{T_1 \times T_2;k} (M', N')$.
4. If $M \sim_{\mathbb{N};k} M'$, $M_1 \sim_{T;k} M_1'$ *and* $M_2 \sim_{T;k} M_2'$, *then* **if** M **then** M_1 **else** $M_2 \sim_{T;k}$ **if** M' **then** M_1' **else** M_2'.
5. If $M_1 \sim_{T_1 \to T_2;k} M_1'$ *and* $M_2 \sim_{T_1;k} M_2'$ *then* $M_1\ M_2 \sim_{T_2;k}$ **let** $g = M_1'$ **in open** $\langle x_f, x_e \rangle = g$ **in** $x_f\ (g, M_2', x_e)$.

The proof of the last of these properties requires us to consider the evaluation of the application of a fixed point expression which involves "feeding" the expression to its own body. In working out the details, we use the easily observed property that the simulation and equivalence relations are closed under decreasing indices.

Our notion of equivalence only relates closed terms. However, our transformation typically operates on open terms, albeit under mappings for the free variables. To handle this situation, we consider semantics preservation for possibly open terms under closed substitutions. We will take substitutions in both the source and target settings to be simultaneous mappings of closed values for a finite collection of variables, written as $(V_1/x_1, \ldots, V_n/x_n)$. In defining a correspondence between source and target language substitutions, we need to consider the possibility that a collection of free variables in the first may be reified into an environment variable in the second. This motivates the following definition in which γ represents a source language substitution:

$$\gamma \approx_{x_m:T_m,\ldots,x_1:T_1;k} (V_1, \ldots, V_m) \iff \forall 1 \leq i \leq m.\gamma(x_i) \approx_{T_i;k} V_i.$$

Writing $\gamma_1; \gamma_2$ for the concatenation of two substitutions viewed as lists, equivalence between substitutions is then defined as follows:

$$\begin{aligned}(V_1/x_1, \ldots, V_n/x_n); \gamma \approx_{\Gamma, x_n:T_n,\ldots,x_1:T_1;k} (V_1'/y_1, \ldots, V_n'/y_n, V_e/x_e) \\ \iff (\forall 1 \leq i \leq n.V_i \approx_{T_i;k} V_i') \wedge \gamma \approx_{\Gamma;k} V_e.\end{aligned}$$

Note that both relations are indexed by a source language typing context and a step measure. The second relation allows the substitutions to be for different variables in the source and target languages. A relevant mapping will determine a correspondence between these variables when we use the relation.

We write the application of a substitution γ to a term M as $M[\gamma]$. The first part of the following lemma, proved by an easy use of the definitions of $\approx$ and evaluation, provides the basis for justifying the treatment of free variables via their transformation into projections over environment variables introduced at function boundaries in the closure conversion transformation. The second part of the lemma is a corollary of the first part that relates a source substitution and an environment computed during the closure conversion of fixed points.

Lemma 2. *Let* $\delta = (V_1/x_1, \ldots, V_n/x_n); \gamma$ *and* $\delta' = (V_1'/y_1, \ldots, V_n'/y_n, V_e/x_e)$ *be source and target language substitutions and let* $\Gamma = (x_m' : T_m', \ldots, x_1' : T_1', x_n : T_n, \ldots, x_1 : T_1)$ *be a source language typing context such that* $\delta \approx_{\Gamma;k} \delta'$. *Further, let* $\rho = (x_1 \mapsto y_1, \ldots, x_n \mapsto y_n, x_1' \mapsto \pi_1(x_e), \ldots, x_m' \mapsto \pi_m(x_e))$.

1. *If* $x : T \in \Gamma$ *then there exists a value* V' *such that* $(\rho(x))[\delta'] \hookrightarrow V'$ *and* $\delta(x) \approx_{T;k} V'$.
2. *If* $\Gamma' = (z_1 : T_{z_1}, \ldots, z_j : T_{z_j})$ *for* $\Gamma' \subseteq \Gamma$ *and* $\rho \triangleright (z_1, \ldots, z_j) \rightsquigarrow_e M$, *then there exists* V_e' *such that* $M[\delta'] \hookrightarrow V_e'$ *and* $\delta \approx_{\Gamma';k} V_e'$.

The proof of semantics preservation also requires a result about the preservation of typing. It takes a little effort to ensure that this property holds at the point in the transformation where we cross a function boundary. That effort is encapsulated in the following strengthening lemma in the present setting.

Lemma 3. *If* $\Gamma \vdash M : T$, $\{x_1, \ldots, x_n\} \supseteq \mathbf{fvars}(M)$ *and* $x_i : T_i \in \Gamma$ *for* $1 \leq i \leq n$, *then* $x_n : T_n, \ldots, x_1 : T_1 \vdash M : T$.

The correctness theorem can now be stated as follows:

Theorem 4. *Let* $\delta = (V_1/x_1, \ldots, V_n/x_n); \gamma$ *and* $\delta' = (V_1'/y_1, \ldots, V_n'/y_n, V_e/x_e)$ *be source and target language substitutions and let* $\Gamma = (x_m' : T_m', \ldots, x_1' : T_1', x_n : T_n, \ldots, x_1 : T_1)$ *be a source language typing context such that* $\delta \approx_{\Gamma;k} \delta'$. *Further, let* $\rho = (x_1 \mapsto y_1, \ldots, x_n \mapsto y_n, x_1' \mapsto \pi_1(x_e), \ldots, x_m' \mapsto \pi_m(x_e))$. *If* $\Gamma \vdash M : T$ *and* $\rho \triangleright M \rightsquigarrow M'$, *then* $M[\delta] \sim_{T;k} M'[\delta']$.

We outline the main steps in the argument for this theorem: these will guide the development of a formal proof in Sect. 4.2. We proceed by induction on the derivation of $\rho \triangleright M \rightsquigarrow M'$, analyzing the last step in it. This obviously depends on the structure of M. The case for a number is obvious and for a variable we use Lemma 2.1. In the remaining cases, other than when M is of the form (**let** $x = M_1$ **in** M_2) or (**fix** $f\, x.M_1$), the argument follows a set pattern: we observe that substitutions distribute to the sub-components of expressions, we invoke the induction hypothesis over the sub-components and then we use Lemma 1 to conclude. If M is of the form (**let** $x = M_1$ **in** M_2), then M' must be of the form (**let** $y = M_1'$ **in** M_2'). Here again the substitutions distribute to M_1 and M_2 and to M_1' and M_2', respectively. We then apply the induction hypothesis first to M_1 and M_1' and then to M_2 and M_2'; in the latter case, we need to consider extended substitutions but these obviously remain equivalent. Finally, if M is of the form (**fix** $f\, x.M_1$), then M' must have the form $\langle M_1', M_2' \rangle$. We can prove that the abstraction M_1' is closed and therefore that $M'[\sigma'] = \langle M_1', M_2'[\sigma'] \rangle$. We then apply the induction hypothesis. In order to do so, we generate the appropriate typing judgment using Lemma 3 and a new pair of equivalent substitutions (under a suitable step index) using Lemma 2.2.

4.2 Formal Verification of the Closure Conversion Implementation

In the subsections below, we present a sequence of preparatory steps, leading eventually to a formal version of the correctness theorem.

Auxiliary Predicates Used in the Formalization. We use the techniques of Sect. 2 to define some predicates related to the encodings of source and target language types and terms that are needed in the main development; unless explicitly mentioned, these definitions are in $\mathcal{G}$. First, we define the predicates `ctx` and `ctx'` to identify typing contexts for the source and target languages. Next, we define in HOHH the recognizers `tm` and `tm'` of well-formed source and target language terms. A source (target) term `M` is closed if {`tm M`} ({`tm' M`}) is derivable. The predicate `is_sty` recognizes source types. Finally, `vars_of_ctx` is a predicate such that `(vars_of_ctx L Vs)` holds if `L` is a source language typing context and `Vs` is the list of variables it pertains to.

Step indexing uses ordering on natural numbers. We represent natural numbers using `z` for 0 and `s` for the successor constructor. The predicate `is_nat` recognizes natural numbers. The predicates `lt` and `le`, whose definitions are routine, represent the "less than" and the "less than or equal to" relations.

The Simulation and Equivalence Relations. The following clauses define the simulation and equivalence relations.

```
sim T K M M' ≜ ∀J V, le J K → {nstep J M V} → {val V} →
  ∃V' N, {eval' M' V'} ∧ {add J N K} ∧ equiv T N V V';
equiv tnat K (nat N) (nat' N);
equiv tunit K unit unit';
equiv (prod T1 T2) K (pair V1 V2) (pair' V1' V2') ≜
  equiv T1 K V1 V1' ∧ equiv T2 K V2 V2' ∧
  {tm V1} ∧ {tm V2} ∧ {tm' V1'} ∧ {tm' V2'};
equiv (arr T1 T2) z (fix R) (clos' (abs' R') VE) ≜
  {val' VE} ∧ {tm (fix R)} ∧ {tm' (clos' (abs' R') VE)};
equiv (arr T1 T2) (s K) (fix R) (clos' (abs' R') VE) ≜
  equiv (arr T1 T2) K (fix R) (clos' (abs' R') VE) ∧
  ∀V1 V1' V2 V2', equiv T1 K V1 V1' → equiv (arr T1 T2) K V2 V2' →
     sim T2 K (R V2 V1) (R' (pair' V2' (pair' V1' VE))).
```

The formula `(sim T K M M')` is intended to mean that `M` simulates `M'` at type `T` in `K` steps; `(equiv T K V V')` has a similar interpretation. Note the exploitation of λ-tree syntax, specifically the use of application, to realize substitution in the definition of `equiv`. It is easily shown that `sim` holds only between closed source and target terms and similarly `equiv` holds only between closed source and target values.[4]

Compatibility lemmas in the style of Lemma 1 are easily stated for `sim`. For example, the one for pairs is the following.

```
∀T1 T2 K M1 M2 M1' M2', {is_nat K} → {is_sty T1} → {is_sty T2} →
```

[4] The definition of `equiv` uses itself negatively in the last clause and thereby violates the original stratification condition of $\mathcal{G}$. However, Abella permits this definition under a weaker stratification condition that ensures consistency provided the definition is used in restricted ways [5,38], a requirement that is adhered to in this paper.

```
sim T1 K M1 M1' → sim T2 K M2 M2' →
sim (prod T1 T2) K (pair M1 M2) (pair' M1' M2').
```

These lemmas have straightforward proofs.

Representing Substitutions. We treat substitutions as discussed in Sect. 2. For example, source substitutions satisfy the following definition.

```
subst nil;
subst ((map X V)::ML) ≜ subst ML ∧ name X ∧ {val V} ∧ {tm V} ∧
  ∀V', member (map X V') ML → V' = V.
```

By definition, these substitutions map variables to closed values. To accord with the way closure conversion is formalized, we allow multiple mappings for a given variable, but we require all of them to be to the same value. The application of a source substitution is also defined as discussed in Sect. 2.

```
app_subst nil M M;
∇x,app_subst ((map x V)::(ML x)) (R x) M ≜ ∇x,app_subst (ML x) (R V) M.
```

As before, we can easily prove properties about substitution application based on this definition such as that such an application distributes over term structure and that closed terms are not affected by substitution.

The predicates `subst'` and `app_subst'` encode target substitutions and their application. Their formalization is similar to that above.

The Equivalence Relation on Substitutions. We first define the relation `subst_env_equiv` between source substitutions and target environments:

```
subst_env_equiv nil K ML unit';
subst_env_equiv ((of X T)::L) K ML (pair' V' VE) ≜
  ∃V,subst_env_equiv L K ML VE ∧ member (map X V) ML ∧ equiv T K V V'.
```

Using `subst_env_equiv`, the needed relation between source and target substitutions is defined as follows.

```
∇e, subst_equiv L K ML ((map e VE)::nil) ≜ subst_env_equiv L K ML VE;
∇x y, subst_equiv ((of x T)::L) K ((map x V)::ML) ((map y V')::ML') ≜
  equiv T K V V' ∧ subst_equiv L K ML ML'.
```

Lemmas about `fvars`, `mapvar` and `mapenv`. Lemma 3 translates into a lemma about `fvars` in the implementation. To state it, we define a strengthening relation between source typing contexts:

```
prune_ctx nil L nil;
prune_ctx (X::Vs) L ((of X T)::L') ≜ member (of X T) L ∧ prune_ctx Vs L L'.
```

`(prune_ctx Vs L L')` holds if `L'` is a typing context that "strengthens" `L` to contain type assignments only for the variables in `Vs`. The lemma about `fvars` is then the following.

```
∀L Vs M T FVs, ctx L → vars_of_ctx L Vs → {L ⊢ of M T} →
  {fvars M Vs FVs} → ∃L', prune_ctx FVs L L' ∧ {L' ⊢ of M T}.
```

To prove this theorem, we generalize it so that the HOHH derivation of `(fvars M Vs FVs)` is relativized to a context that marks some variables as not free. The resulting generalization is proved by induction on the `fvars` derivation.

A formalization of Lemma 2 is also needed for the main theorem. We start with a lemma about `mapvar`.

```
∀L Vs Map ML K VE X T M' V, ∇e, {is_nat K} → ctx L → subst ML →
  subst_env_equiv L K ML VE → vars_of_ctx L Vs → {mapvar Vs Map} →
  member (of X T) L → app_subst ML X V → {member (map X (M' e)) (Map e)}
  → ∃V', {eval' (M' VE) V'} ∧ equiv T K V V'.
```

In words, this lemma states the following. If `L` is a source typing context for the variables $(x_1, \ldots, x_n)$, `ML` is a source substitution and `VE` is an environment equivalent to `ML` at `L`, then `mapvar` determines a mapping for $(x_1, \ldots, x_n)$ that are projections over an environment with the following character: if the environment is taken to be `VE`, then, for $1 \leq i \leq n$, x_i is mapped to a projection that must evaluate to a value equivalent to the substitution for x_i in `ML`. The lemma is proved by induction on the derivation of `{mapvar Vs Map}`.

Lemma 2 is now formalized as follows.

```
∀L ML ML' K Vs Vs' Map, {is_nat K} → ctx L → subst ML →
  subst' ML' → subst_equiv L K ML ML' → vars_of_ctx L Vs →
  vars_of_subst' ML' Vs' → to_mapping Vs Vs' Map →
  (∀ X T V M' M'', member (of X T) L → {member (map X M') Map} →
    app_subst ML X V → app_subst' ML' M' M'' →
      ∃V', {eval' M'' V'} ∧ equiv T K V V') ∧
  (∀ L' NFVs E E', prune_ctx NFVs L L' →
    {mapenv NFVs Map E} → app_subst' ML' E E' →
      ∃VE', {eval' E' VE'} ∧ subst_env_equiv L' K ML VE').
```

Two new predicates are used here. The judgment `(vars_of_subst' ML' Vs')` "collects" the variables in the target substitution `ML'` into `Vs'`. Given source variables `Vs` = $(x_1, \ldots, x_n, x'_1, \ldots, x'_m)$ and target variables `Vs'` = $(y_1, \ldots, y_n, x_e)$, the predicate `to_mapping` creates in `Map` the mapping

$$(x_1 \mapsto y_1, \ldots, x_n \mapsto y_n, x'_1 \mapsto \pi_1(x_e), \ldots, x'_m \mapsto \pi_m(x_e)).$$

The conclusion of the lemma is a conjunction representing the two parts of Lemma 2. The first part is proved by induction on `{member (map X M') Map}`, using the lemma for `mapvar` when `X` is some $x'_i (1 \leq i \leq m)$. The second part is proved by induction on `{mapenv NFVs Map E}` using the first part.

The Main Theorem. The semantics preservation theorem is stated as follows:

```
∀L ML ML' K Vs Vs' Map T P P' M M', {is_nat K} → ctx L → subst ML →
  subst' ML' → subst_equiv L K ML ML' → vars_of_ctx L Vs →
  vars_of_subst' ML' Vs' → to_mapping Vs Vs' Map → {L ⊢ of M T} →
  {cc Map Vs M M'} → app_subst ML M P → app_subst' ML' M' P' → sim T K P P'.
```

We use an induction on {`cc Map Vs M M'`}, the closure conversion derivation, to prove this theorem. As should be evident from the preceding development, the proof in fact closely follows the structure we outlined in Sect. 4.1.

4.3 Verifying the Implementations of Other Transformations

We have used the ideas presented in this section to develop semantics preservation proofs for other transformations such as code hoisting and the CPS transformation. We discuss the case for code hoisting below.

The first step is to define the step-indexed logical relations $\sim'$ and $\approx'$ that respectively represent the simulation and equivalence relation between the input and output terms and values for code hoisting:

$$M \sim'_{T;k} M' \iff \forall j \leq k. \forall V. M \hookrightarrow_j V \supset \exists V'. M' \hookrightarrow V' \wedge V \approx'_{T;k-j} V';$$

$$n \approx'_{\mathbb{N};k} n;$$
$$() \approx'_{\mathbf{unit};k} ();$$
$$(V_1, V_2) \approx'_{(T_1 \times T_2);k} (V'_1, V'_2) \iff V_1 \approx'_{T_1;k} V'_1 \wedge V_2 \approx'_{T_2;k} V'_2;$$
$$(\lambda x.M) \approx'_{T_1 \Rightarrow T_2;k} (\lambda x.M') \iff \forall j < k. \forall V, V'. V \approx'_{T_1;j} V' \supset M[V/x] \sim'_{T_2;j} M'[V'/x];$$
$$\langle \lambda p.M, V_e \rangle \approx'_{T_1 \to T_2;k} \langle \lambda p.M', V'_e \rangle \iff \forall j < k. \forall V_1, V'_1, V_2, V'_2.$$
$$V_1 \approx'_{T_1;j} V'_1 \supset V_2 \approx'_{T_1 \to T_2;j} V'_2 \supset M[(V_2, V_1, V_e)/p] \sim'_{T_2;j} M'[(V'_2, V'_1, V'_e)/p].$$

We can show that $\sim'$ satisfies a set of compatibility properties similar to Lemma 1.

We next define a step-indexed relation of equivalence between two substitutions $\delta = (V_1/x_1, \ldots, V_m/x_m)$ and $\delta' = (V'_1/x_1, \ldots, V'_m/x_m)$ relative to a typing context $\Gamma = (x_m : T_m, \ldots, x_1 : T_1)$:

$$\delta \approx'_{\Gamma;k} \delta' \iff \forall 1 \leq i \leq m. V_i \approx'_{T_i;k} V'_i.$$

The semantics preservation theorem for code hoisting is stated as follows:

Theorem 5. *Let $\delta = (V_1/x_1, \ldots, V_m/x_m)$ and $\delta' = (V'_1/x_1, \ldots, V'_m/x_m)$ be substitutions for the language described in Fig. 2. Let $\Gamma = (x_m : T_m, \ldots, x_1 : T_1)$ be a typing context such that $\delta \approx'_{\Gamma;k} \delta'$. Further, let $\rho = (x_1, \ldots, x_m)$. If $\Gamma \vdash M : T$ and $\rho \triangleright M \rightsquigarrow_{ch} M'$ hold, then $M[\delta] \sim'_{T;k} M'[\delta']$ holds.*

The theorem is proved by induction on the derivation for $\rho \triangleright M \rightsquigarrow_{ch} M'$. The base cases follow easily, possibly using the fact that $\delta \approx'_{\Gamma;k} \delta'$. For the inductive cases, we observe that substitutions distribute to the sub-components of expressions, we invoke the induction hypothesis over the sub-components and we use the compatibility property of $\sim'$. In the case of an abstraction, δ and δ' must be extended to include a substitution for the bound variable. For this case to work out, we must show that the additional substitution for the bound variable has no impact on the functions extracted by code hoisting. From the side condition for the rule deriving $\rho \triangleright M \rightsquigarrow_{ch} M'$ in this case, the extracted functions cannot depend on the bound variable and hence the desired observation follows.

In the formalization of this proof, we use the predicate constants `sim'` and `equiv'` to respectively represent $\sim'$ and $\approx'$. The Abella definitions of these predicates have by now a familiar structure. We also define a constant `subst_equiv'` to represent the equivalence of substitutions as follows:

```
subst_equiv' nil K nil nil;
∇x, subst_equiv' ((of' x T)::L) K ((map' x V)::ML) ((map' x V')::ML')
  ≜ equiv' T K V V' ∧ subst_equiv' L K ML ML'.
```

The representation of contexts in the code hoisting judgment in the HOHH specification is captured by the predicate `ch_ctx` that is defined as follows:

```
ch_ctx nil;
∇x, ch_ctx (ch x (htm nil (hbase x)) :: L) ≜ ch_ctx L.
```

The semantics preservation theorem is stated as follows, where `vars_of_ctx'` is a predicate for collecting variables in the typing contexts for the target language, `vars_of_ch_ctx` is a predicate such that `(vars_of_ch_ctx L Vs)` holds if `L` is a context for code hoisting and `Vs` is the list of variables it pertains to:

```
∀L K CL ML ML' M M' T FE FE' P P' Vs, {is_nat K} → ctx' L →
  ch_ctx CL → vars_of_ctx' L Vs → vars_of_ch_ctx CL Vs →
  subst' ML → subst' ML' → subst_equiv' L K ML ML' →
  {L ⊢ of' M T} → {CL ⊢ ch M (htm FE M')} → app_subst' ML M P →
  app_subst' ML' (htm FE M') (htm FE' P') → sim' T K P (htm FE' P').
```

The proof is by induction on `{CL ⊢ ch M (htm FE M')}` and its structure follows that of the informal one very closely. The fact that the extracted functions do not depend on the bound variable of an abstraction is actually explicit in the logical formulation and this leads to an exceedingly simple argument for this case.

4.4 Relevance to Other Styles of Correctness Proofs

Many compiler verification projects, such as CompCert [21] and CakeML [20], have focused primarily on verifying whole programs that produce values of atomic types. In this setting, the main requirement is to show that the source and target programs evaluate to the same atomic values. Structuring a proof around program equivalence base on a logical relation is one way to do this. Another, sometimes simpler, approach is to show that the compiler transformations permute over evaluation; this method works because transformations typically preserve values at atomic types. Although we do not present this here, we have examined proofs of this kind and have observed many of the same kinds of benefits to the λ-tree syntax approach in their context as well.

Programs are often built by composing separately compiled modules of code. In this context it is desirable that the composition of correctly compiled modules preserve correctness; this property applied to compiler verification has been called modularity. Logical relations pay attention to equivalence at function types and hence proofs based on them possess the modularity property. Another property that is desirable for correctness proofs is transitivity: we should be able to infer the correctness of a multi-stage compiler from the correctness of each

of its stages. This property holds when we use logical relations if we restrict attention to programs that produce atomic values but cannot be guaranteed if equivalence at function types is also important; it is not always possible to decompose the natural logical relation between a source and target language into ones between several intermediate languages. Recent work has attempted to generalize the logical relations based approach to obtain the benefits of both transitivity and modularity [32]. Many of the same issues relating to the treatment of binding and substitution appear in this context as well and the work in this paper therefore seems to be relevant also to the formalization of proofs that use these ideas.

Finally, we note that the above comments relate only to the formalization of proofs. The underlying transformations remain unchanged and so does the significance of our framework to their implementation.

5 Related Work and Conclusion

Compiler verification has been an active area for investigation. We focus here on the work in this area that has been devoted to compiling functional languages. There have been several projects with ambitious scope even in this setting. To take some examples, the CakeML project has implemented a compiler from a subset of ML to the X86 assembly language and verified it using HOL4 [20]; Dargaye has used Coq to verify a compiler from a subset of ML into the intermediate language used by CompCert [14]; Hur and Dreyer have used Coq to develop a verified single-pass compiler from a subset of ML to assembly code based on a logical relations style definition of program equivalence [19]; and Neis *et al.* have used Coq to develop a verified multi-pass compiler called Pilsner, basing their proof on a notion of semantics preservation called Parametric Inter-Languages Simulation (PILS) [32]. All these projects have used essentially first-order treatments of binding, such as those based on a De Bruijn style representation.

A direct comparison of our work with the projects mentioned above is neither feasible nor sensible because of differences in scope and focus. Some comparison is possible with a part of the Lambda Tamer project of Chlipala in which he describes the verified implementation in Coq of a compiler for the STLC using a logical relation based definition of program equivalence [11]. This work uses a higher-order representation of syntax that does not derive all the benefits of λ-tree syntax. Chlipala's implementation of closure conversion comprises about 400 lines of Coq code, in contrast to about 70 lines of λProlog code that are needed in our implementation. Chlipala's proof of correctness comprises about 270 lines but it benefits significantly from the automation framework that was the focus of the Lambda Tamer project; that framework is built on top of the already existing Coq libraries and consists of about 1900 lines of code. The Abella proof script runs about 1600 lines. We note that Abella has virtually no automation and the current absence of polymorphism leads to some redundancy in the proof. We also note that, in contrast to Chlipala's work, our development treats a version of the STLC that includes recursion. This necessitates the use of a step-indexed logical relation which makes the overall proof more complex.

Other frameworks have been proposed in the literature that facilitate the use of HOAS in implementing and verifying compiler transformations. Hickey and Nogin describe a framework for effecting compiler transformations via rewrite rules that operate on a higher-order representation of programs [18]. However, their framework is embedded within a functional language. As a result, they are not able to support an analysis of binding structure, an ability that brings considerable benefit as we have highlighted in this paper. Moreover, this framework offers no capabilities for verification. Hannan and Pfenning have discussed using a system called Twelf that is based on LF in specifying and verifying compilers; see, for example, [16] and [29] for some applications of this framework. The way in which logical properties can be expressed in Twelf is restricted; in particular, it is not easy to encode a logical relation-style definition within it. The Beluga system [34], which implements a functional programming language based on contextual modal type theory [31], overcomes some of the shortcomings of Twelf. Rich properties of programs can be embedded in types in Beluga, and Belanger *et al.* show how this feature can be exploited to ensure type preservation for closure conversion [7]. Properties based on logical relations can also be described in Beluga [10]. It remains to be seen if semantics preservation proofs of the kind discussed in this paper can be carried out in the Beluga system.

While the framework comprising λProlog and Abella has significant benefits in the verified implementation of compiler transformations for functional languages, its current realization has some practical limitations that lead to a larger proof development effort than seems necessary. One such limitation is the absence of polymorphism in the Abella implementation. A consequence of this is that the same proofs have sometimes to be repeated at different types. This situation appears to be one that can be alleviated by allowing the user to parameterize proofs by types and we are currently investigating this matter. A second limitation arises from the emphasis on explicit proofs in the theorem-proving setup. The effect of this requirement is especially felt with respect to lemmas about contexts that arise routinely in the λ-tree syntax approach: such lemmas have fairly obvious proofs but, currently, the user must provide them to complete the overall verification task. In the Twelf and Beluga systems, such lemmas are obviated by absorbing them into the meta-theoretic framework. There are reasons related to the validation of verification that lead us to prefer explicit proofs. However, as shown in [6], it is often possible to generate these proofs automatically, thereby allowing the user to focus on the less obvious aspects. In ongoing work, we are exploring the impact of using such ideas on reducing the overall proof effort.

Acknowledgements. We are grateful to David Baelde for his help in phrasing the definition of the logical relation in Sect. 4.2. The paper has benefited from many suggestions from its reviewers. This work has been supported by the National Science Foundation grant CCF-0917140 and by the University of Minnesota through a Doctoral Dissertation Fellowship and a Grant-in-Aid of Research. Opinions, findings and conclusions or recommendations that are manifest in this material are those of the participants and do not necessarily reflect the views of the NSF.

References

1. Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 69–83. Springer, Heidelberg (2006)
2. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. **23**(5), 657–683 (2001)
3. Aydemir, B.E., et al.: Mechanized metatheory for the masses: the PoplMark challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
4. Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., Wang, Y.: Abella: a system for reasoning about relational specifications. J. Formalized Reasoning **7**(2), 1–89 (2014)
5. Baelde, D., Nadathur, G.: Combining deduction modulo and logics of fixed-point definitions. In: Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science, pp. 105–114. IEEE Computer Society (2012)
6. Bélanger, O.S., Chaudhuri, K.: Automatically deriving schematic theorems for dynamic contexts. In: Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice. ACM Press (2014)
7. Savary-Belanger, O., Monnier, S., Pientka, B.: Programming type-safe transformations using higher-order abstract syntax. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 243–258. Springer, Heidelberg (2013)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
9. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. Indagationes Mathematicae **34**(5), 381–392 (1972)
10. Cave, A., Pientka, B.: A case study on logical relations using contextual types. In: Proceedings of the Tenth International Workshop on Logical Frameworks and Meta Languages: Theory and Practice, EPTCS, vol. 185, pp. 33–45 (2015)
11. Chlipala, A.: A certified type-preserving compiler from lambda calculus to assembly language. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 54–65. ACM Press (2007)
12. Church, A.: A formulation of the simple theory of types. J. Symb. Logic **5**, 56–68 (1940)
13. Danvy, O., Filinski, A.: Representing control: a study of the CPS transformation. Math. Struct. Comput. Sci. **2**, 361–391 (1992)
14. Dargaye, Z.: Vérification formelle d'un compilateur optimisant pour langages fonctionnels. Ph.D. thesis, l'Université Paris 7-Denis Diderot, France (2009)
15. Gordon, M.J.C.: Introduction to the HOL system. In: Archer, M., Joyce, J.J., Levitt, K.N., Windley, P.J. (eds.) Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications, pp. 2–3. IEEE Computer Society (1991)
16. Hannan, J., Pfenning, F.: Compiler verification in LF. In: 7th Symposium on Logic in Computer Science. IEEE Computer Society Press (1992)
17. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. J. ACM **40**(1), 143–184 (1993)
18. Hickey, J., Nogin, A.: Formal compiler construction in a logical framework. Higher-Order Symb. Comput. **19**(2–3), 197–230 (2006)

19. Hur, C.K., Dreyer, D.: A Kripke logical relation between ML and assembly. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 133–146. ACM Press (2011)
20. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 179–191. ACM Press (2014)
21. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 42–54. ACM Press (2006)
22. McCarthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. In: Proceedings of Symposia in Applied Mathematics. Mathematical Aspects of Computer Science, vol. 19, pp. 33–41. American Mathematical Society (1967)
23. McDowell, R., Miller, D.: Cut-elimination for a logic with definitions and induction. Theoret. Comput. Sci. **232**, 91–119 (2000)
24. Miller, D.: Abstract syntax for variable binders: an overview. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 239–253. Springer, Heidelberg (2000)
25. Miller, D., Nadathur, G.: Programming with Higher-Order Logic. Cambridge University Press, Cambridge (2012)
26. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Trans. Comput. Logic **6**(4), 749–783 (2005)
27. Milner, R., Weyrauch, R.: Proving compiler correctness in a mechanized logic. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7, pp. 51–72. Edinburgh University Press, Edinburgh (1972)
28. Minamide, Y., Morrisett, G., Harper, R.: Typed closure conversion. Technical report CMU-CS-95-171, School of Computer Science, Carnegie Mellon University (1995)
29. Murphy VII, T.: Modal types for mobile code. Ph.D. thesis, Carnegie Mellon University (2008)
30. Nadathur, G., Miller, D.: An overview of λProlog. In: Fifth International Logic Programming Conference, pp. 810–827. MIT Press (1988)
31. Nanevski, A., Pfenning, F., Pientka, B.: Contextual model type theory. ACM Trans. Comput. Logic **9**(3), 1–49 (2008)
32. Neis, G., Hur, C.K., Kaiser, J.O., McLaughlin, C., Dreyer, D., Vafeiadis, V.: Pilsner: a compositionally verified compiler for a higher-order imperative language. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, pp. 166–178. ACM Press (2015)
33. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002)
34. Pientka, B., Dunfield, J.: Beluga: a framework for programming and reasoning with deductive systems (system description). In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 15–21. Springer, Heidelberg (2010)
35. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. Comput. **186**(2), 165–193 (2003)
36. Qi, X., Gacek, A., Holte, S., Nadathur, G., Snow, Z.: The Teyjus system - Version 2. http://teyjus.cs.umn.edu/
37. Tian, Y.H.: Mechanically verifying correctness of CPS compilation. In: Twelfth Computing: The Australasian Theory Symposium. CRPIT, vol. 51, pp. 41–51. ACS (2006)
38. Tiu, A.: Stratification in logics of definitions. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 544–558. Springer, Heidelberg (2012)