A Higher-Order Abstract Syntax Approach to Verified Compilation of Functional Programs

Thesis Proposal

Yuting Wang
Department of Computer Science and Engineering
University of Minnesota, USA

November 7, 2015

1 Introduction

We propose to develop a dissertation in the broad area of verified compilation of programming languages. Our specific focus will actually be on a narrower class of languages called the *functional programming languages* or, more simply, the *functional languages*. *Compilation* in this context is generally visualized as the application of a series of transformations to input programs, resulting eventually in low-level instructions that can be executed directly on existing hardware. Our goal, then, is to show how such transformations can be encoded in programs and how it can be demonstrated that these programs modify or simplify the original code while preserving its meaning. A central thesis underlying our work is that the tasks of implementing and verifying the mentioned transformations is significantly assisted by the use of a particular approach, called the *higher-order abstract syntax* or *HOAS* approach, to representing, manipulating, and reasoning about programs in a functional language. The work that we will undertake in this dissertation is aimed at validating this thesis: in particular, we will further develop an existing framework that supports the HOAS approach and we will demonstrate how this framework can be used effectively in the verified compilation of functional programming languages.

In the rest of this introductory section, we provide a more detailed background to the work we propose. We begin with a motivation for verified compilation in general and for functional languages in particular. In Section 1.3, we explain briefly the relevance of the HOAS approach to the task of describing and verifying transformations on functional programs. In Section 1.4, we outline the framework that supports the HOAS approach that we intend to use in this dissertation and we sketch the specific work that we will carry out in its context. Section 1.5 summarizes the expected contributions of our work. The rest of the thesis proposal is devoted to providing more details regarding the

discussions started in this introduction. Section 1.6 explains the particular role each of the subsequent sections play in this elaboration.

1.1 Formal Verification of Compilers

With the increasing reliance of modern society on software systems, the correct operation of such systems has become a major concern. A commonly used method for gaining confidence in software behavior has been the idea of testing. In this approach, programs are run repeatedly under systematically varying conditions and their results are checked against the expected outcomes. While testing has been used successfully in many situations, it also has a fundamental limitation: in the extreme, testing can only provide *evidence* for correctness, never a water-tight guarantee for the absence of bugs in programs. There are many safety critical software systems for which it is in fact necessary to have such a guarantee of correctness. In such cases, the only known approach to obtaining the requisite assurances is to formally verify the properties of software using solid mathematical principles.

Motivated by the considerations above, a large amount of effort has been devoted over the last several decades towards developing approaches to formally verifying the correctness of programs. Over the years, there has been a convergence on two main ideas in realizing the overall verification goal. First, there has been an emphasis on developing high-level programming languages that make it easy to describe computations and to reason about them based on their descriptions; particular examples of such languages that are commonly used are Java and C++. Second, methodologies have been developed for utilizing the structure of such languages to facilitate the reasoning process. There has been considerable success in realizing these two ideas and they have indeed provided the basis for verifying the properties of many non-trivial programs. However, the combination of these ideas leaves an important gap in the overall verification task in that they assume that programs are eventually executed in the high-level language they are written in. In reality, there is an intervening process called *compilation* that translates these programs into lower level, machine executable code. To complete the verification process, it is also essential to ensure that the *compilers* that carry out such translations actually preserve the behavior of programs.

The existence of this gap has been known for a long time and there has been a focus also on the formal verification of compilers aimed at filling the gap; a survey of such efforts appears, for example, in [15]. Compiler verification is eventually a large, tedious and error-prone task and this has hampered attempts to make it practical. However, the significant strides that have been made in recent years in developing machine-oriented theorem proving techniques and tools have gone a long way towards alleviating these difficulties. Indeed, there has been a mushrooming of efforts related to formal compiler verification using theorem proving systems such as Isabelle [41], HOL [40] and Coq [49]. One of the more successful exercises in this direction has been the CompCert project that has developed and verified a compiler for a large subset of the C language

using the Coq theorem prover [29]. Such efforts have also provided an impetus to more ambitious projects such as the Verified Software Toolchain project [2] related to overall program verification.

1.2 Verified Compilation of Functional Languages

The point of departure for the work to be undertaken in this dissertation is that it will focus on the implementation and verification of compilers for *functional programming languages*.

Commonly used imperative programming languages such as C view computation as the result of the repeated alteration of state, which is given by the values stored in memory cells and the location of control; the building blocks for programs within this paradigm are *statements* or *commands* for modifying program state. In contrast, programs in functional languages are given almost entirely by expressions, computation consisting essentially of evaluating these expressions. This yields a more abstract view of programming, with a focus on describing what has to be done rather than how it should be carried out. The common foundation of functional programming is the λ-calculus, a mathematical system that possesses well-understood logical properties [11]. Because of this basis and also because of the freedom from lower-level implementation details, programs written in functional programming languages are often more concise and easier to reason about than those written in imperative programming languages. Research over the last two decades has shown how programs in these more abstract formalisms can nevertheless be realized with significant efficiency. As a consequence of these results, functional programming languages are starting to play an increasingly important role in modern software practice. Indeed, languages within this paradigm such as Common Lisp, Scheme, Racket, Erlang, Haskell, Standard ML, OCaml, Scala and F# are used today in a wide range of academic, industrial and commercial areas.

As might be expected, the high level of abstraction present in functional programs makes the compilation of these programs a more complex undertaking. A specific aspect contributing to this complexity is the need to objectify functions, allowing them to be embedded in data structures, passed as parameters to other functions and returned as the results of computations. The possibility of such objectification means, in turn, that a compiler itself must build in an understanding of *binding structure* which associates the occurrences of variables with identifiers that define them. In particular, it must be able to differentiate between free and bound variables in a function valued expression, it should not distinguish between expressions that differ only in the names used for their bound variables and it should be able to realize substitutions over expressions in a way that avoids the inadvertent capture of the free variables in the expression being substituted. Indeed, these notions are used in a fundamental way in the traditional approach to compiling functional programs which consists of applying transformations to them that eventually renders them into a form to which compilation techniques that are well-known from the context of imperative programs can be applied.

The difficulties arising from the need to treat variable binding within the structure of a compiler are further exaggerated when it comes to verifying the compilation process. This may be rationalized as follows: whereas the understanding of binding can be left implicit in the implementation of a compiler, the task of showing that specific transformations preserve the meanings of programs must make this understanding explicit. Overall, the task of verifying compilers for functional languages becomes a more complex one at a conceptual level. However, this increased complexity represents a well-motivated tradeoff: the compiler for a language needs to be verified only once, whereas the benefits of a more abstract formalism can be reaped repeatedly in reasoning about the properties of particular programs.

1.3 The Treatment of Object-Level Binding Structure

The need to represent and analyze the syntax of object languages is not limited to the realm of compiler implementation and verification: it arises uniformly in the context of varied systems that manipulate formal objects such as programs, formulas, types and proofs. There is, in fact, a common core of issues related to binding structure that must be treated in these different systems. Recent research has led to an understanding of the essential issues as well as to approaches towards handling them in a satisfactory manner.

One class of approaches that has been developed builds on what is basically a first-order treatment of syntax augmented with special devices for treating binding related aspects. In the simplest form, this augmentation consists simply of a special representation of bound variables complemented with special library functions that utilize this representation. The most common example of this kind of approaches is the one that uses a scheme devised by De Bruijn that eliminates names for bound variables, using indices for their occurrences which unambiguously indicate the abstractions binding them [16]. This "nameless" representation renders all object language expressions that differ only in the names of bound variables into a unique form, thereby making it trivial to determine equality modulo renaming. Substitution and other relevant operations can be defined relative to such a representation and are typically realized through auxiliary programs. Variants of this approach have also been developed such as the one based on the locally nameless representation of binding structure [3].

A different approach that still uses a first-order representation of syntax is that based on nominal logic [45]. The defining characteristic of this approach is that it builds in a logical treatment of bound variables though the notion of equivalence under permutations of names. As with the De Bruijn based representations, equality under renaming is immediate under this nominal logic based approach. A further virtue of the approach is that it provides a logical treatment of free and bound variables that can be useful in reasoning about the correctness of manipulations of syntactic structure. However, nominal logic representations do not provide an intrinsic treatment of substitution; the realization of this and related aspects has to be encoded in user programs that must then also be explicitly reasoned about.

A more radical approach to treating binding in object languages is to use a higher-order representation of syntax, often referred to as *higher-order abstract syntax* or HOAS [34, 42]. In this approach, formal objects that need to be manipulated are encoded using a variety of the typed λ-calculus. Under such a representation, binding constructs in object language syntax are treated directly through abstraction in the meta-language. A consequence of this choice is that varied binding related operations obtain a logical treatment. To take an example, equivalence under renaming of bound variables is realized immediately through the notion of α-conversion that is part of the theory of the meta-language. Similarly, the operation of β-conversion that accompanies the λ-calculus can be used to provide a logically precise treatment of substitution over object language structures. The fact that these operations are supported by the underlying theory has a further advantage: properties of the meta-language can be exploited in the process of reasoning about the encoding of computations over object language constructs.

1.4 Our Thesis and the Proposed Work

Our goal in this work is to develop an approach that is effective both in implementing compilers for functional languages and in verifying them. Our central thesis is that the achievement of these objectives can be considerably simplified by the use of the HOAS approach. The work we propose is oriented towards verifying this thesis: in particular, we will consider a variety of transformations used in compiling functional languages and show the benefits of an HOAS encoding in realizing these transformations and proving them correct.

Carrying out the work we propose requires a set of tools that actually support the HOAS approach. Towards this end, we will use an existing framework that comprises the following components:

- An executable specification language called λProlog [35] that supports the HOAS approach and that is suitable for encoding syntax-directed and rule-based specifications.
- A theorem-proving system called Abella [4] that, once again, supports the HOAS approach and that can be used to reason about relational specifications.
- An encoding of the specification language within the Abella theorem-prover that allows us to treat descriptions of transformation in the former both as implementations and as the basis for reasoning about their correctness.

We will use this framework in the following fashion. First, we will show how the transformations that are used commonly in compiling functional languages can be encoded succinctly and transparently as λProlog programs. The executability of these encodings means that they can serve directly as implementations of the transformations. We will then show how we can use the Abella system

to reason about the properties of these λProlog specifications, thereby proving the correctness of the compilation process that they realize. Throughout these exercises, our focus will be on developing a methodology for exploiting the HOAS approach and demonstrating the benefits of using this methodology.

1.5 The Expected Contributions

The concrete contributions that we expect to make based on the work we propose include the following:

- We will develop a methodology for implementing compiler transformations that uses the HOAS approach in manipulating program structure. To show the effectiveness of this methodology, we will utilize it in developing a multi-pass compiler in λProlog for a representative (typed) functional language that includes recursion; the particular language we will treat will be a superset of the language commonly known as PCF.

- We will develop a methodology for formally verifying transformations on functional programs expressed in λProlog. Our methodology will make use of the HOAS approach in the reasoning process. An important part of compiler verification is a formalization of meaning preservation for program transformations. We will experiment with two different methods for characterizing program equivalence, based respectively on the ideas of logical relations and simulation. We will demonstrate the effectiveness of our methodology by employing it in a proof in Abella of the correctness of the compiler that we would previously have implemented for our chosen functional language.

- The verification task that we propose to undertake represents a major application that will stress the capabilities of the Abella system that is still in a development phase. As part of our work, we will extend this system in ways that are relevant to making it a versatile tool in efforts such as compiler verification.

1.6 The Structure of the Rest of the Proposal

The rest of the proposal will elaborate on the overview of the dissertation work that we have provided in the preceding part of the introduction. Section 2 introduces the framework that will be used to carry out the work in this dissertation. In particular, it presents the λProlog specification language, the Abella theorem prover, and the two-level logic approach to reasoning about λProlog specifications in Abella. This discussion will make clear the HOAS approach to representing and manipulating binding structure in object language syntax and also how λProlog and Abella support this approach. The following sections elaborate on the actual work that will be carried out in this dissertation. Section 3 presents the methodology for implementing transformations on functional programs using λProlog. In this discussion, we first show how some typical compiler

transformations on functional programs can be described in a rule-based fashion. This presentation will bring out the necessity to represent and manipulate the binding structure of programs. We then show how such transformations can be succinctly formalized in λProlog; given the excutability of λProlog specifications, these formalizations serve also as implementations of the transformations. We conclude this section with a plan for the work related to compiler implementation that will be carried out in this thesis. Section 4 turns to the task of formally verifying compiler transformations. More specifically, it considers the methodology for proving the correctness of such tranformations that are encoded in λProlog using the Abella theorem-proving system. Here, we first describe different ways to characterize the preservation of meaning by compiler transformations and we show how to use them in informal proofs of the correctness of such transformations. We then show how to formalize such proofs using Abella. This discussion will bring out the way in which the logical structure of the λProlog specifications can be exploited to prove properties about compiler transformations and also shows how reasoning about binding structure can be simplified by using the HOAS approach. Once again, we conclude this section by laying out a plan for the work related to compiler verification that we will carry out in this thesis. To effectively carry out this verification task, it is necessary to consider certain extensions to the Abella system. Section 5 describes these extensions to Abella that will also constitute a part of this thesis. The verification of compilers for functional programs has been considered by other researchers using theorem-proving systems like Coq, HOL and Isabelle. The distinguishing characteristic of our work is that it will use and benefit from the HOAS approach. Section 6 describes the previous approaches and contrasts the work we propose towards bringing out the precise nature of our contributions in the context of compiler verification. We conclude this proposal in Section 7 with a summary of the work that will be done in the dissertation and an assessment of its objectives illuminated by the elaboration that is provided in the preceding sections.

2 The Framework

The framework that we will use in this thesis consists of the specification language λProlog and the theorem-proving system Abella. A characteristic of the λProlog language is that specifications written it are executable and therefore they serve also as implementations. This feature of specifications is realized by the Teyjus system that implements λProlog. Abella is an interactive theorem prover for reasoning about relational specifications. It is possible to encode derivability in λProlog as a relation in Abella. In fact, Abella builds in such an encoding of λProlog. Further, it allows us to reason about specifications written in λProlog through this encoding using what is referred to as the two-level logic approach. We will make crucial use of this aspect in this thesis: specifically, we will implement compiler transformations in λProlog and we will prove these implementations correct in Abella using the two-level logic approach. Another

important feature of both λProlog and Abella is that they both support a variant of the HOAS approach known as the *λ-tree syntax approach* [33]. As we have already mentioned, a key part of this thesis is to show that this approach significantly simplifies the task of verified compilation.

The discussions in the following sections will rely on an understanding of these various notions. We therefore describe relevant aspects of the framework in this section. In particular, in successive subsections, we expose λProlog, Abella and the two-level logic approach to reasoning. In the course of this presentation, we will also make clear the idea of λ-tree syntax.

2.1 The Specification Language

The language λProlog is based on the logic of Hereditary Harrop formulas, or HH, which is itself a fragment of Church's Simple Theory of Types (STT) [12]. The expressions in this logic are those of the simply typed λ-calculus (STLC). These expressions are actually split into two categories: the types and the terms.

The type expressions are generated from *atomic types* using the *function* or *arrow* type constructor. Their syntax is given as follows, assuming that τ stands for types and a stands for atomic types:

$$\tau \ ::= \ a \mid (\tau \to \tau)$$

In the context of STT, the atomic types consist of a user-defined collection plus the distinguished type `o` that is used for formulas as explained below. By assumption, there is at least one user-defined atomic type. We drop parentheses in writing arrow types, i.e. types of the second form, by using the convention that the arrow operator $\to$ associates to the right. For instance, $\tau_1 \to \tau_2 \to \tau_3$ stands for $(\tau_1 \to (\tau_2 \to \tau_3))$. Using this associativity convention, every type can be written in the form $\tau_1 \to ... \to \tau_n \to \beta$ where β is an atomic type. When a type is written in this fashion, τ_i for $1 \leq i \leq n$ are called its *argument types* and β is called its *target type*.

In building terms, we assume a vocabulary of constants, denoted below by c, and of variables, denoted below by x. We assume that each constant and variable has a specified type associated with it. Terms are then specified together with their types by the following inductive rules:

- A constant c or a variable x of type τ is a term of type τ.
- An expression of the form $(\lambda x : \tau.t)$ in which t is of type τ' is a term of type $\tau \to \tau'$. Such a term is referred to as an abstraction that has the variable x as its binder and whose body or scope is t. Further, all occurrences of x in t are considered bound by the abstraction.
- An expression of the form $(t_1\ t_2)$ in which t_1 is a term of type $\alpha \to \beta$ and t_2 is a term of type α is a term of type β. Such a term is called an application that has t_1 as its function part and t_2 as its argument.

We drop parentheses in terms by assuming applications associate to the left and applications bind more tightly than abstractions. For instance, $(t_1\ t_2\ t_3)$ represents $((t_1\ t_2)\ t_3)$ and $(\lambda x : \tau.t_1\ t_2)$ represents $(\lambda x : \tau.(t_1\ t_2))$. We will often drop the type τ in an abstraction $(\lambda x : \tau.t)$, abbreviating it as $(\lambda x.t)$, when this type is not important to our understanding or can be inferred from the context. We will also need to refer below to the free variables of a term. These are the variables in the term that are not in the scope of any abstraction occurring in it. We will often need to indicate a term t, such as a variable or constant, together with its type τ. We will do this by writing $t : \tau$.

The underlying logic assumes an equality relation between terms that is explained through the following steps:

- An α-step allows us to rename the variables bound by abstractions. Specifically, two terms are related by an α-step if the second can be obtained from the first by replacing a subterm of the form $(\lambda x.t)$ by $(\lambda y.t')$ where y is a variable that does not occur in t and t' is the result of replacing the free occurrences of x in t by y. Two terms are related by α-conversion if one can be obtained from the other by repeated applications of α-steps.

- The β-conversion relation captures the idea of equivalence under function evaluation. Let us refer to a term of the form $((\lambda x.t_1)\ t_2)$ as a β-redex. Then a term u β-contracts to a term v if v can be obtained by replacing such a β-redex in u by the result of substituting t_2 for the free occurrences of x in t_1 provided that the free variables in t_2 do not occur bound in t_1. Conversely, if v results from u by β-contraction, then u results from v by β-expansion. Finally, β-conversion is the reflexive and transitive closure of the union of the β-contraction, β-expansion and α-step relations.

- The η-rule reflects the idea of extensional equality for functions. We refer to a term of the form $(\lambda x.t\ x)$ where x does not occur in t as an η-redex. A term u η-contracts to v if v can be obtained from u by replacing such a subterm by t. Conversely, v η-expands to u if u η-contracts to v. Finally, λ-conversion is the reflexive and transitive closure of β-conversion, η-contraction and η-expansion relations.

Equality is given by the strongest of these relations, i.e., by λ-conversion. It can be seen that this is an equivalence relation, thereby meeting the basic criterion for an equality relation.

We say that a variable or constant has arity n if its type has n argument types. We also say that its occurrence in a term is *fully applied* if it is applied to as many arguments as its arity. A term is said to be in $\beta\eta$-long normal form if it does not contain a β-redex and, further, every variable or constant in it is fully applied. It is known that every term in the the STLC λ-converts to a $\beta\eta$-long normal form that is unique up to α-conversion. We refer to such a form as a $\beta\eta$-long normal form *for* the term. It is further known that any term can be transformed into (one of) its $\beta\eta$-long normal form through a terminating process. A further point to note is that the different operations such as substitution that we consider on λ-terms commute with the conversion rules. This

allows us to always work with the $\beta\eta$-long normal forms of terms, a fact that we will use implicitly in the following discussions.

We will need to consider substitutions into terms. The logically correct form of this operation must be "capture-avoiding." We formalize this idea as follows. Let $x_1, .., x_n$ be a sequence of variables and let $t_1, .., t_n$ be terms such that, for $1 \leq i \leq n$, t_i has the same type as x_i. Then, given a term t, the expression $t[t_1/x_1, ..., t_n/x_n]$ denotes the term that is obtained by simultaneously replacing the free occurrences of the variables x_i in t by the terms t_i for $1 \leq i \leq n$. Note, however, that in doing this replacement we must rename the binders of abstractions in t where this is necessary to prevent them from inadvertently binding the free variables in $t_1, ..., t_n$. Such a renaming has a logical justification: it corresponds to the use of α-steps that preserve equality between terms.

The definition of λ-terms relies on a collection of constants. In the context of HH, we distinguish between *logical* constants and *nonlogical* constants. The logical constants consist of the symbols $\&$ and $\Rightarrow$, both of type $\mathtt{o} \to \mathtt{o} \to \mathtt{o}$, and, for each type τ that does not contain $\mathtt{o}$, Π_τ of type $(\tau \to \mathtt{o}) \to \mathtt{o}$. The first two constants, also called the logical connectives, correspond to conjunction and implication and are usually written as infix operators. The family of constants Π_τ represent universal quantifiers: the term $(\Pi_\tau\ (\lambda x : \tau.M))$ corresponds to the universal quantification of x over M. In writing this expression, we will often use the suggestive abbreviation $\Pi_\tau x.M$ and will also drop the type annotation—i.e. we will simply write $\Pi x.M$—when the type can be inferred or when its knowledge is not essential to the discussion.

The set of nonlogical constants is also called a *signature*. We shall use the symbol Σ, possibly with subscripts to denote signatures. There is a restriction on the constants allowed in a signature in HH: their argument types must not contain the type $\mathtt{o}$. A nonlogical constant whose target type is $\mathtt{o}$ is called a predicate symbol or predicate constant. Such constants are used to form *atomic formulas* that represent relations. Specifically, a term of the form $(p\ t_1\ \ldots\ t_n)$ in which p is a predicate symbol of arity n constitutes an atomic formula.[1] We use the symbol A to denote atomic formulas.

The terms of type $\mathtt{o}$, that are also called *formulas*, have a special status in the logic: they are the expressions to which the derivation rules pertain. The HH logic is determined by two particular kinds of formulas called *goal formulas*, or simply *goals*, and *program clauses*, or simply *clauses*. These formulas are denoted by the symbols G and D, respectively, and are given by the following syntax rules:

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

Goal formulas of the form $D \Rightarrow G$ are called *hypothetical goals*. Goal formulas of the form $\Pi_\tau x.G$ are called *universal goals*. Note that a program clause has

[1] As previously mentioned, we work only with terms in $\beta\eta$-long normal form. Thus, we apply this and similar terminology to terms only after they have been transformed into their normal forms.

the form $\Pi_{\tau_1} x_1 \ldots \Pi_{\tau_n} x_n.(G \Rightarrow A)$ or $\Pi_{\tau_1} x_1 \ldots \Pi_{\tau_n} x_n.A$. We refer to A as the head of such a formula. Further, we call G the body of a clause of the first form and in the second case we say that the clause has an empty body.

In the intended use of the HH logic, a collection of program clauses and a signature constitutes a specification, also called a *program*. A user provides these collections towards defining specific relations. The relations that are so defined are determined by the atomic formulas that are derivable from a program. We formalize this notion through a sequent calculus in the style of Gentzen [21]. In the context of interest, a sequent has the structure $\Sigma; \Theta; \Gamma \vdash G$, where Σ is a signature, Θ is a multi-set of program clauses that is called the *static context*, Γ is the multi-set of program clauses called the *dynamic context* and G is the goal of the sequent. A program given by the clauses Θ and the signature Σ then specifies the relations represented by all the atomic formulas A such that the sequent $\Sigma; \Theta; \emptyset \vdash A$ is derivable. Note that the dynamic context is empty in the beginning. However, as we shall see presently, the process of constructing derivations may add clauses to this context and may also extend the signature.

The actual rules for deriving sequents of the form described are presented in Figure 1. These rules are of two kinds: the rules $\wedge R$, $\Rightarrow R$ and ΠR are used to simplify non-atomic goals and the rules *succeed* or *backchain* apply once the goal has been reduced to atomic form. The rules $\Rightarrow$R and ΠR are the ones that impart a dynamic character to sequents. In particular, the ΠR rule causes the signature to grow through the addition of a previously unused constant in the course of searching for a derivation and the $\Rightarrow$R rule similarly causes additions to the dynamic context. The *succeed* and *backchain* rules capture the following intuition for solving atomic goals: we look for a clause in the dynamic or static context whose head matches the goal we want to solve and then reduce the task to solving the relevant instance of the body of the clause if this is non-empty.

The λProlog language provides a concrete syntax in which a user can present programs. In this syntax, a multiset of clauses is written as a sequence, with each clause being terminated by a period. Further, universal quantifiers at the outermost level in a clause may be omitted by using names starting with capital letters for the occurrences of variables that they bind. Also, the clause $(G \Rightarrow D)$ is written in λProlog as $(D$ `:-` $G)$. Abstraction is written as an infix operator. More specifically, the term $(\lambda x : \tau.M)$ is written as $(x : \tau \setminus M)$ in λProlog. When the type of the bound variable can be inferred uniquely from the context, this expression can also be simplified to $(x \setminus M)$. The constant `pi` is used to denote the family of constants Π_τ. Thus, (`pi` M) stands for $(\Pi_\tau\ M)$. The particular type intended for `pi` is obtained from the type of its argument M. Finally, conjunction in goals is denoted by a comma: a goal of the form $(G_1$ & $G_2)$ is written as (G_1, G_2).

Program clauses in HH provide a natural way to capture rule-based specifications of relations. In particular, the relation that is being defined can be represented by a predicate name, the conclusion of the rule defining it becomes the head of a clause for that predicate and the premises, if any, become the body of the clause. We illustrate this idea by considering the specification of the append relation on lists of natural numbers. Let [] denote the empty list

$$\frac{\Sigma;\Theta;\Gamma \vdash G_1 \quad \Sigma;\Theta;\Gamma \vdash G_2}{\Sigma;\Theta;\Gamma \vdash G_1 \ \& \ G_2}\ \wedge R \qquad \frac{\Sigma;\Theta;\Gamma, D \vdash G}{\Sigma;\Theta;\Gamma \vdash D \Rightarrow G}\ \Rightarrow R$$

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

$$\frac{\Pi_{\tau_1} x_1 \ldots \Pi_{\tau_n} x_n.A' \in \Gamma \cup \Theta}{\Sigma;\Theta;\Gamma \vdash A}\ \textit{succeed}$$

$$\frac{\Pi_{\tau_1} x_1 \ldots \Pi_{\tau_n} x_n.G \Rightarrow A' \in \Gamma \cup \Theta \quad \Sigma;\Theta;\Gamma \vdash G[t_1/x_1, ..., t_n/x_n]}{\Sigma;\Theta;\Gamma \vdash A}\ \textit{backchain}$$
(where $A'[t_1/x_1, ..., t_n/x_n] = A$ in *succeed* and *backchain*)

Figure 1: Derivation Rules of HH

in our object language and let :: denote the "cons" constructor for lists. The append relation can then be given by the following rules:

$$\frac{}{\textit{append}\ []\ l\ l} \qquad \frac{\textit{append}\ l_1\ l_2\ l_3}{\textit{append}\ (x :: l_1)\ l_2\ (x :: l_3)}$$

In encoding these rules in λProlog, let us assume that we have designated the type `nat` to represent natural numbers and `list` for the type of lists of natural numbers. Further, let us assume that we have introduced into the signature the constants $1, 2, 3, ...$ of type `nat` to represent the natural numbers and the following constants to represent empty lists and the cons constructor:

$$\texttt{nil} : \texttt{list} \qquad\qquad \texttt{cons} : \texttt{nat} \rightarrow \texttt{list} \rightarrow \texttt{list}$$

The λProlog language provides the user a way to define such signatures the details of which we do not discuss here. We then use the predicate symbol $\texttt{append} : \texttt{list} \rightarrow \texttt{list} \rightarrow \texttt{list} \rightarrow \texttt{o}$ to represent the append relation and we encode the rules defining the relation in the following clauses:

```
append nil L L.
append (cons X L1) L2 (cons X L3) :- append L1 L2 L3.
```

The way these clauses would be used in constructing derivations in HH has a transparent relationship to the way in which the rules specifying *append* would be used in establishing a relation between three lists: assuming l_1, l_2 and l_3 to be the encodings of three specific lists, to show that (`append` l_1 l_2 l_3) holds,

we would match it with the head of one of the clauses (that corresponds to the conclusions of the rules) and reduce the task to deriving the corresponding body (that corresponds to the premise of the relevant rule) if it is non-empty. In this sense, the λProlog encoding also reflects the *derivation behavior* of the rule based specification of the append relation.

Our interest in this work is in treating specifications of computational systems such as programming languages and logics. An important characteristic of such systems is that the expressions they treat contain variable binding operators. The traditional approach to encoding such operators is to use a first-order representation and to let the user build in properties of binding through additional specifications or programs. A defining aspect of the λProlog language is that it supports a different, more abstract, approach to treating such binding operators. This language provides us with λ-terms, rather than just first-order terms, to represent expressions in an object language. The abstraction operator present in these terms gives us a meta-language level mechanism to capture the binding notions present in the expressions over which we want to specify and carry out computations. In addition to representing the syntax of objects, we also often need to support the ability to specify properties by recursion over their structure. The λProlog language has mechanisms that are specially geared towards realizing such recursion over binding operators. These mechanisms arise from the presence of universal and hypothetical goals in the language.

The use of the different mechanisms mentioned above is what is referred to as the λ-tree syntax approach to treating binding structure in an object language. This approach will play a central role in this dissertation and we therefore provide an illustration of it here. The example we use for this purpose is that of encoding the typing relation for the simply-typed λ-calculus. Note that we treat the STLC as an *object system* in this example. The syntax of expressions in this system is similar to that underlying λProlog, except that we do not allow for constants in terms:

$$\begin{aligned} T &::= a \mid T_1 \to T_2 \\ M &::= x \mid \lambda x : T.M \mid M_1\ M_2 \end{aligned}$$

The typing relation or judgment that we are interested in is written in the form $\Gamma \vdash M : T$ where M is a term, T is a type and Γ is a *typing context* that has the form $x_1 : T_1, ..., x_n : T_n$ where each x_i is a distinct variable. Intuitively, such a judgment represents the fact that M is a well-formed term of type T, assuming that Γ provides the types for its free variables. The rules for deriving a judgment of this kind are the following:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \texttt{t-var} \qquad \frac{\Gamma \vdash M_1 : T_1 \to T_2 \quad \Gamma \vdash M_2 : T_1}{\Gamma \vdash M_1\ M_2 : T_2}\ \texttt{t-app}$$

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x : T_1.M : (T_1 \to T_2)}\ \texttt{t-abs}$$

where x is not already in Γ

To encode this system in λProlog, we first need to represent the syntax of STLC. Towards this end, we identify the two types `ty` and `tm`: Expressions

in λProlog of these types will correspond, respectively, to types and terms in the object language. respectively. We then identify the following constants for encoding types and terms:

$$\texttt{a} : \texttt{ty} \qquad \texttt{arr} : \texttt{ty} \to \texttt{ty} \to \texttt{ty}$$
$$\texttt{abs} : \texttt{ty} \to (\texttt{tm} \to \texttt{tm}) \to \texttt{tm} \qquad \texttt{app} : \texttt{tm} \to \texttt{tm} \to \texttt{tm}$$

The only constant in this signature that requires special mention is `abs`. This constant is used to represent abstractions in the object language. Note that the "term" component that this constant takes is itself an abstraction in λProlog. The idea underlying this representation is that we will isolate the binding aspect of abstractions in the STLC and allow these to be treated through the understanding of abstraction in λProlog. As a concrete example, the STLC term $(\lambda x : a \to a.\lambda y : a.x\ y)$ will be encoded as

$$\texttt{abs}\ (\texttt{arr}\ \texttt{a}\ \texttt{a})\ (x \setminus \texttt{abs}\ \texttt{a}\ (y \setminus \texttt{app}\ x\ y)).$$

As will become clear from the discussion below, this kind of encoding allows us to use the understanding of abstraction over λ-terms present in λProlog to transparently realize binding related properties such as scoping, irrelevance of bound variable names and (logically correct) substitution over object language expressions.

Having fixed the representation of expressions in the STLC, we can now specify the typing rules. We use the predicate symbol $\texttt{of} : \texttt{tm} \to \texttt{ty} \to \texttt{o}$ to represent the typing relation. The typing rules then translate into the following HH clauses defining the `of` predicate:

$$\begin{array}{lll} \texttt{of}\ (\texttt{app}\ M_1\ M_2)\ T_2 & \texttt{:-} & \texttt{of}\ M_1\ (\texttt{arr}\ T_1\ T_2)\,, \texttt{of}\ M_2\ T_1. \\ \texttt{of}\ (\texttt{abs}\ T_1\ M)\ (\texttt{arr}\ T_1\ T_2) & \texttt{:-} & \texttt{pi}\ y \setminus \texttt{of}\ y\ T_1 \Rightarrow \texttt{of}\ (M\ y)\ T_2. \end{array}$$

The first clause represents a natural translation of the rule for typing applications. The second clause, which encodes the typing rule for abstractions, shows how recursion over abstractions is realized and how substitutions are modeled by (meta-level) β-conversion in λProlog. Specifically, if we use this clause to derive the goal

$$\texttt{of}\ (\texttt{abs}\ t_1\ m)\ (\texttt{arr}\ t_1\ t_2),$$

for particular expressions t_1, t_2 and m, we would have to derive the goal

$$\texttt{pi}\ y \setminus \texttt{of}\ y\ t_1 \Rightarrow \texttt{of}\ (m\ y)\ t_2.$$

The only way to do this in the HH logic is to introduce a new constant c (that will represent the bound variable of the abstraction), to add the clause $(\texttt{of}\ c\ t_1)$ to the dynamic context (that encodes the typing context in this example) and to then derive $(\texttt{of}\ (m\ c)\ t_2)$. Note that m is a (λProlog) abstraction here and that $(m\ c)$ is therefore equal, by virtue of β-conversion, to the result of substituting c for the abstracted variable in the body of m. Note also that the only way to derive a goal of the form $(\texttt{of}\ c\ t)$ is to match t with the type t_1 in the (extended)

dynamic context. Thus, deriving the goal $(\texttt{of}\ (m\ c)\ t_2)$ corresponds to showing that the body of the object-level abstraction has the type t_2 in the context where its bound variable has the type t_1. From these observations, it follows that our λProlog specification faithfully captures the typing rules for STLC.

The discussions above show that specifications in λProlog have an executable character: in the example specifications, we have used the rules for deriving HH sequents essentially as a means for computing relations of interest. The cases that we have considered have been limited to checking if a relation holds. However, we can also extend this mechanism to "computing answers" by including variables in a goal that we expect the derivation mechanism to fill in with an actual expression for which the relation holds. As a concrete example, assuming T to be such a variable, we might submit the following query

$$\texttt{of}\ (\texttt{abs}\ (\texttt{arr a a})\ (x \setminus \texttt{abs a}\ (y \setminus \texttt{app}\ x\ y)))\ T$$

in a context where the program consists of the clauses encoding the typing rules in STLC. The only solution to this "schematic" query is one where T is instantiated with the expression $(\texttt{arr}\ (\texttt{arr a a})\ (\texttt{arr a a}))$. Specifications in λProlog function genuinely as programs in this sense: they can be executed and they can also be used to compute results that are previously not known to the user. Indeed many useful applications have been shown to exist for this kind of "logic programming" interpretation of the language [35] and the Teyjus system [46] has been developed to provide efficient support for such applications. We will make use of these facts in this dissertation. In particular, we will use λProlog to specify compiler transformations that we will then execute as programs using the Teyjus system.

2.2 The Theorem Proving System

The Abella system is an interactive theorem prover for a logic called $\mathcal{G}$ which is also based on Church's Simple Theory of Types. The term language of $\mathcal{G}$ is similar to that of HH. One difference is that the `o` is replaced by `prop` as the type of formulas. Another difference is that different names are used for the logical constants and the set of these constants is also larger. In particular, the logical constants of $\mathcal{G}$ consist of $\top, \bot$: `prop` that represent true and false, $\wedge, \vee, \supset$: $\texttt{prop} \to \texttt{prop} \to \texttt{prop}$ that represent conjunction, disjunction and implication, and, for each type τ not containing `prop`, the constants $\forall_\tau, \exists_\tau : (\tau \to \texttt{prop}) \to \texttt{prop}$ that correspond to (the family of) universal and existential quantifiers. Following the style of HH, we write $\wedge$, $\vee$ and $\supset$ as infix operators. Similarly, we abbreviate the expressions $(\forall_\tau\ (\lambda x : \tau.M))$ and $(\exists_\tau\ (\lambda x : \tau.M))$ by $(\forall x : \tau.M)$ and $(\exists x : \tau.M)$. We also drop the type annotations in abstractions and quantified formulas when they are irrelevant to the discussion or can be inferred uniquely. When stating formulas that represent theorems we would like to prove in Abella, we shall use the concrete syntax of Abella for universal and existential quantified formulas in them. In this concrete syntax, an universal formula of the form $(\forall x_1 : \tau_1 \ldots \forall x_n : \tau_n.M)$ is written as

$(\forall(x_1 : \tau_1) \ \ldots \ (x_n : \tau_n), M)$, or $(\forall x_1 \ \ldots \ x_n, M)$ when the types of bound variables can be inferred uniquely from the context. The concrete syntax for expressing existential formulas is similar to universal ones.

Provability in $\mathcal{G}$ is once again formalized via a sequent calculus. This logic builds on a basic structure provided by intuitionistic derivability. We will assume that the reader is familiar with a sequent calculus formalization of this idea of derivability; it may, for instance, be taken to be given by the LJ calculus of Gentzen [21] with the only additional observation that atomic formulas in $\mathcal{G}$ have simply typed λ-terms, rather than just first-order terms, as arguments. The interesting features of $\mathcal{G}$ from the perspective of this proposal are the ones that extend this basic structure to make it a suitable logic for reasoning about relational specifications using the λ-tree syntax approach to treating binding in object languages. We will describe these features below. Our presentation will be informal, focusing on explaining the features so that we can use them later, rather than showing their precise formalization. A reader interested in a precise description of the derivation rules may consult [19].

One of the key aspects of $\mathcal{G}$ is that it provides a treatment of fixed-point definitions. It does this by interpreting predicate constants as defined symbols rather than primitive ones. Concretely, $\mathcal{G}$ is parameterized by a *definition* that is given by a collection of *definitional clauses* of the form $\forall\bar{x}.A \triangleq B$ where A is an atomic formula whose free variables are bound by $\bar{x}$ and B is a formula all of whose free variables must occur free in A. We call A the head of such a clause and B its body. Let p be the predicate symbol at the head of A in such a clause. We then say that this clause is a clause *for* p. The intuitive understanding of a definition then is that it assigns a meaning to each predicate symbol based on all the clauses in it for that symbol. Thus, suppose that we want to prove an atomic formula that has the predicate symbol p as its head. To prove this formula, we may match it with the head of one of the clauses for p and prove the corresponding instance of the body of that clause. Conversely, suppose that one of the assumptions in the sequent we want to prove is an atomic formula with p as its head. This formula can be true only by virtue of one of the clauses for p. This observation leads to a case analysis style of reasoning: we check the way in which each instance of the atomic formula matches with the head of a clause for p and, in each case, we try to prove the sequent that results from replacing the atomic formula on the lefthand side of the sequent with the corresponding instance of the body of the clause.[2]

Fixed-point definitions provide a natural way to encode rule-based relational specifications in $\mathcal{G}$: predicate symbols can be used to name relations and each rule translates into a definitional clause. We use the append relation on lists of natural numbers again to illustrate the idea. As before, we use the atomic type `list` for representations of lists, the constants `nil` and `cons` to construct such representation and the predicate symbol `append` : `list` $\rightarrow$ `list` $\rightarrow$ `list` $\rightarrow$ `prop`

[2]To guarantee the consistency of a logic of this kind, it is necessary to impose conditions on the structure of definitional clauses. One such condition stratifies them based on the names of the predicate symbols defined [31]. We do not present this condition here, but we work with it implicitly until Section 5 where we discuss extensions to it.

to encode the append relation. The rules defining the relation then translate directly into the following clauses:

$$\texttt{append}\ \texttt{nil}\ L\ L \triangleq \top$$
$$\texttt{append}\ (\texttt{cons}\ X\ L_1)\ L_2\ (\texttt{cons}\ X\ L_3) \triangleq \texttt{append}\ L_1\ L_2\ L_3$$

In showing these clauses, we have adopted the Abella convention of making the outermost universal quantifiers implicit by choosing tokens that begin with uppercase letters for the occurrences of the variables they bind.

To understand the fixed-point nature of the definitions, let us consider using them in derivations. First, suppose that we want to show that the following is a theorem:

$$\forall L, \texttt{append}\ \texttt{nil}\ L\ L$$

To do this, we would have to show that the following holds, regardless of the value of l:

$$\texttt{append}\ \texttt{nil}\ l\ l;$$

in the proof search process, l, which represents a generic list, is what is referred to as an *eigenvariable*. This atomic formula is an instance of the head of the first clause for `append` and so the task reduces to proving $\top$, something that is immediate in the logic.

The reasoning example above shows similarities between clauses in HH and definitional clauses in $\mathcal{G}$ when the latter are used to prove atomic formulas; the transformation is effectively what we would obtain through backchaining in HH. There is, however, a significant difference between the way the two logics treat clauses: as we have previously explained, in $\mathcal{G}$ they have a fixed-point interpretation. This aspect becomes clear when we consider proving the formula

$$\forall L, \texttt{append}\ (1 :: 2 :: \texttt{nil})\ L\ (1 :: 3 :: L) \supset \bot$$

in $\mathcal{G}$. The attempt to prove this formula can be reduced to showing that $\bot$ holds whenever we have $\texttt{append}\ (1 :: 2 :: \texttt{nil})\ l\ (1 :: 3 :: l)$ for any list l. This is the case for the following reason: the assumption is not true for *any* value of l. The way we show this in $\mathcal{G}$ is by considering the different ways the clauses for `append` might apply to the assumption; we are, of course, permitted to specialize l in different ways so as to consider different instances of the assumption in the process. In this particular instance, only the second clause for `append` is applicable and so the "case analysis" yields a single case where the assumption has been transformed to $\texttt{append}\ (2 :: \texttt{nil})\ l\ (3 :: l)$. We now try a further case analysis and easily realize that no clause applies, i.e. the assumption cannot in fact be true. The desired theorem therefore follows.

The above style of reasoning works well in proving theorems when the case analysis has a finite structure. Unfortunately, this property does not hold in

many cases of interest. As an example, consider the formula

$$\forall L_1\ L_2\ L_3\ L_3', \texttt{append}\ L_1\ L_2\ L_3 \supset \texttt{append}\ L_1\ L_2\ L_3' \supset L_3 = L_3'$$

that states that `append` is deterministic. Case analysis in this case exhibits a looping structure. More importantly, the fixed-point interpretation of the definition of `append` is not sufficient to establish the truth of the shown formula. We must also know that an `append` predicate holds only by virtue of a *finite* number of unfoldings using the clauses and we should be able to argue by induction on the length of such unfoldings. In other words, we should be able to give a *least-fixed point* or inductive interpretation to the definition of `append`.

The logic $\mathcal{G}$ allows definitions of particular predicate symbols to be treated in this way; there is also the dual possibility of giving them a *greatest-fixed point* or co-inductive interpretation, an aspect that we do not discuss here. Marking a predicate as an inductively defined one allows for the following kind of reasoning process. Suppose that we are interested in proving a theorem of the form

$$\forall x_1\ \ldots\ x_m, F_1 \supset \ldots \supset A \supset \ldots \supset F_n \supset B$$

where A is an atomic formula whose head is an inductive predicate. We can choose to prove this formula inductively. Doing so adds the formula

$$\forall x_1\ \ldots\ x_m, F_1 \supset \ldots \supset A^* \supset \ldots \supset F_n \supset B.$$

as an *induction hypothesis* to the assumption set. We can now advance proof search by introducing the eigenvariables $x_1, .., x_m$ and adding $F_1, \ldots, A^{@}, \ldots, F_n$ as hypotheses, leaving B as the conclusion to be shown. Note the annotations on A^* and $A^{@}$. Their meaning is the following: A^* will match only with another formula that has a similar annotation and the only way to produce a formula with that annotation is to use a definitional clause to unfold $A^{@}$. In other words, we get to use the induction hypothesis only on an atomic formula that is smaller in the unfolding sequence than the one in the original formula to be proved.

We illustrate this style of inductive argument by using it to show that `append` is deterministic, assuming that `append` is defined inductively. The proof is by induction on the first assumption. More specifically, we add

$$\forall L_1\ L_2\ L_3\ L_3', \texttt{append}\ L_1\ L_2\ L_3^* \supset \texttt{append}\ L_1\ L_2\ L_3' \supset L_3 = L_3'$$

as an induction hypothesis and we transform the original goal into one with

$$\begin{array}{ll} \texttt{H1:} & \texttt{append}\ L_1\ L_2\ L_3^{@} \\ \texttt{H2:} & \texttt{append}\ L_1\ L_2\ L_3' \end{array}$$

as hypotheses and with $L_3 = L_3'$ as the formula we want to show; L_1, L_2 and L_3 are eigenvariables here. We then analyze `H1` as follows:

- It is derived using the first clause for `append`. In this case L_1 must be `nil` and $L_2 = L_3$. A case analysis on `H2` reveals $L_2 = L_3'$ and hence the conclusion follows;

- It is derived using the second clause for `append`. Here, there must be some L_1' and L_3'' such that $L_1 = X :: L_1'$ and $L_3 = X :: L_3''$ and such that `H3` : `append` L_1' L_2 $L_3''^*$ must be derivable. Note the changed annotation on `H3`, corresponding to the fact that it must be derivable in *fewer* steps. Case analysis on the corresponding instance of `H2` leads us to assume that there is some L_3''' such that $L_3' = X :: L_3'''$ and we add

  `H4`: `append` L_1' L_2 L_3'''

  to the assumption set. At this point, we can apply the inductive hypotheses to `H3` and `H4` to get $L_3'' = L_3'''$. The desired conclusion now easily follows.

As is the case with HH, the logic $\mathcal{G}$ provides λ-terms as a means for representing objects. Scope related properties of binding operators in object languages can therefore be captured by abstractions in the terms of $\mathcal{G}$. However, unlike in HH, we cannot use universal quantifiers to realize recursion over the representation of such constructs. The reason for this is that universal quantification in $\mathcal{G}$ has an *extensional* reading. This becomes clear when we consider proving a formula such as $(\forall x(q\ x \supset p\ x))$: one way to do this is to consider each term t that q is true of and to construct a (possibly distinct) proof of $(p\ t)$.

To overcome this deficiency, $\mathcal{G}$ has a different *generic* quantifier called nabla, written as ∇_τ; like the other quantifiers, ∇ is parameterized by a type that is omitted if its identity is not relevant to the discussion or it can be inferred uniquely from the context. The logical meaning of a formula of the form $\nabla_\tau x.F$ can be understood as follows: to construct a proof for it, we pick a new constant of type τ that does not appear in F and then prove the formula that results from instantiating x in F with this constant. The constants that are to be used in this way are called nominal constants and they are depicted by tokens that consist of the letter n followed by digits, i.e., by tokens such as $n1$, $n2$, $n3$, etc. The treatment of $\nabla_\tau x.F$ as an assumption is similar: we get to use F with x replaced by a fresh nominal constant as an assumption instead. An important property of the ∇ quantifier is that each quantifier over the same formula refers to a *distinct* constant. Thus, $(\nabla x.\nabla y.x = y \supset \bot)$ is a theorem of $\mathcal{G}$. This property carries over to nominal constants: two distinct nominal constants in a given formula are treated as being different and, thus, the formula $(n1 = n2 \supset \bot)$ is a theorem.

We illustrate the use of the ∇ quantifier in realizing recursion over binding structure by showing its use in formalizing the typing rules for the simply typed λ-calculus. We use a representation for the terms in this calculus that is identical to the one described in Section 2.1. However, unlike in HH, implications do not have a hypothetical reading in $\mathcal{G}$. For this reason, we cannot treat the typing context implicitly through assumptions. Thus, a typing judgment becomes a three-place relation `of` of the type `clist` $\rightarrow$ `tm` $\rightarrow$ `ty` $\rightarrow$ `prop`. The first argument of it is a list representing the typing context. We use the type `clelem` for classifying the type assignments for variables in typing contexts and the constant `vty` : `tm` $\rightarrow$ `ty` $\rightarrow$ `clelem` for encoding such assignments. We then

identify constants $\texttt{clnil} : \texttt{clist}$ that represents the empty list for `clist` and $\texttt{clcons} : \texttt{clelem} \rightarrow \texttt{clist} \rightarrow \texttt{clist}$ that represents the cons operator for `clist`. To encode type checking of a variable in a typing context, we identify a predicate constant $\texttt{vof} : \texttt{clist} \rightarrow \texttt{tm} \rightarrow \texttt{ty} \rightarrow \texttt{prop}$. The clauses defining this checking is given as follows:

$$\texttt{vof}\ (\texttt{clcons}\ (\texttt{vty}\ X\ T)\ L)\ X\ T \triangleq \top$$
$$\texttt{vof}\ (\texttt{clcons}\ E\ L)\ X\ T \triangleq \texttt{vof}\ L\ X\ T$$

Then the typing rules in STLC can be encoded as the following clauses for `of`:

$$\texttt{of}\ L\ X\ T \triangleq \texttt{vof}\ L\ X\ T$$
$$\texttt{of}\ L\ (\texttt{app}\ M_1\ M_2)\ T_1 \triangleq \exists T_2, \texttt{of}\ L\ M_1\ (\texttt{arr}\ T_2\ T_1) \wedge \texttt{of}\ L\ M_2\ T_2$$
$$\texttt{of}\ L\ (\texttt{abs}\ R)\ (\texttt{arr}\ T_1\ T_2) \triangleq \nabla x, \texttt{of}\ (\texttt{clcons}\ (\texttt{vty}\ x\ T_1)\ L)\ (R\ x)\ T_2$$

An interesting point to note about the manner in which this specification is constructed is the following: in a judgment $(\texttt{of}\ L\ M\ T)$ that we want to prove, M represents a possibly open term whose free variables are represented by nominal constants and L identifies the types associated with these constants. This kind of treatment, where nominal constants play a role in representing open terms, is typical of formalizations in $\mathcal{G}$ that are based on the λ-tree syntax approach.

A property of the typing rules for the STLC is that they assign unique types to terms. We can express this property in $\mathcal{G}$ through the formula

$$\forall L\ T_1\ T_2, \texttt{of}\ L\ M\ T_1 \supset \texttt{of}\ L\ M\ T_2 \supset T_1 = T_2.$$

To prove this formula, though, we need some restrictions on the typing context L: it should assign types only to nominal constants and each of these assignments should be unique. These properties are satisfied by any typing context that arises in deriving a typing judgment with an initially empty context. However, we can prove a unique type assignment theorem only if it makes it explicit that the typing context in fact satisfies them.

The properties described above are ones about the structures of terms and, more specifically, about the occurrences of nominal constants in them. Definitions in $\mathcal{G}$ include a mechanism for making such aspects explicit. The full form of definitional clauses is in fact $\forall \bar{x}.\nabla \bar{z}.A \triangleq B$. In generating instances of such clauses, the ∇ quantifiers at the head must be instantiated by distinct nominal constants. A further point to note is the scope of the universal quantifiers: since the ∇ quantifiers appear within their scope, the nominal constants that instantiate them cannot appear in the instantiations of the universal quantifiers. Using clauses of this general form, we can characterize legitimate typing contexts using the following clauses for the predicate $\texttt{ctx} : \texttt{clist} \rightarrow \texttt{prop}$:

$$\texttt{ctx}\ \texttt{clnil} \triangleq \top \qquad \nabla x.\texttt{ctx}\ (\texttt{clcons}\ (\texttt{vty}\ x\ T)\ L) \triangleq \texttt{ctx}\ L$$

It is easy to see that these clauses define a relation `ctx` such that `ctx` L holds exactly when L assigns unique types to a collection of distinct nominal constants. Based on this definition, the uniqueness of type assignment can be restated in the formula

$$\forall L\ T_1\ T_2, \texttt{ctx}\ L \supset \texttt{of}\ L\ M\ T_1 \supset \texttt{of}\ L\ M\ T_2 \supset T_1 = T_2.$$

that is in fact provable in $\mathcal{G}$ [4].

Our focus in most of this subsection has been on explaining the logical devices present in $\mathcal{G}$. The actual basis for realizing these capabilities in an automated interaction is Abella. In the discussions that follow, we will use $\mathcal{G}$ and Abella interchangeably: we will present formalizations in $\mathcal{G}$ but will assume their use in actual reasoning tasks based on Abella.

2.3 A Two-Level Logic Approach

As we have just explained, Abella is an effective tool for reasoning about rule-based relational specifications. We would like to use it for reasoning about λProlog programs. One obvious approach would be to encode λProlog programs as fixed-points definitions and to reason about the properties of these definitions in Abella; this is the approach we used with the specification of typing for STLC. However, this requires that we construct two versions of the specifications, one in λProlog and one in Abella. Moreover, the reasoning in Abella does not directly apply to the λProlog program. Ideally, we would like to have only *one* specification that serves directly as an implementation and that is also shown to be correct.

The above goal can be accomplished by adopting the so-called *two-level logic approach* to reasoning [32, 20]. In the setting of Abella, this approach translates into the following. We continue to write specifications in HH. What we also do is encode HH as a logic into a fixed-point definition in $\mathcal{G}$. We then lift specifications in HH into $\mathcal{G}$ through the encoding. Finally, we reason about the HH specifications using Abella via the translation. This style of reasoning has the obvious advantage that we are proving properties about actual programs in λProlog. Another benefit of the approach arises from the fact that HH is itself a logic with special meta-theoretic properties that can be useful in reasoning about derivations in it. Since it has been encoded into $\mathcal{G}$, such properties can be proved as theorems in $\mathcal{G}$. Once they have been proved, they become available for use in other reasoning tasks.

The Abella system is actually already structured to support the two-level logic approach described above. To encode derivability in HH within $\mathcal{G}$, it uses the predicate symbol `seq` : `olist` → `o` → `prop`. Here, `olist` is a type that is reserved for lists of HH formulas and such lists are constructed using the constants `nil` and `cons`. An HH sequent $\Sigma; \Theta; \Gamma \vdash G$ is encoded as `seq` $\Gamma\ G$. Note that the static context is omitted in this encoding. A development in Abella begins with the loading of an HH specification. This fixes the static context for the development and Abella manages the use of this context implicitly. The

encoding of HH sequents also omits the signature Σ. One part of Σ consists of the signature arising from the original HH program. Abella treats this part by adding it directly to is own vocabulary of nonlogical constants. The dynamic part of the signature arises from the treatment of universal goals in proof search. In the encoding, universal quantifiers goals in HH are handled through ∇ quantifiers. The "constants" that they add to the signature are therefore represented by nominal constants and so there is already a means for identifying them in the proof development. The encoding of the derivation rules of HH is realized through a collection of clauses for `seq`. It has been proved in [18] that $\texttt{seq}\ \Gamma\ G$ is derivable in $\mathcal{G}$ if and only if $\Sigma; \Theta; \Gamma \vdash G$ is derivable in HH. In this concrete syntax of Abella, the formula $\texttt{seq}\ \Gamma\ G$ is written as $\{\Gamma \vdash G\}$; if Γ is an empty, this can be abbreviated to $\{G\}$.

To illustrate how we can state and prove properties about HH specifications in Abella, suppose that we have included into an Abella development the HH program from Section 2.1 that specifies the typing rules for the STLC. Assuming that L is a list of formulas that encode the dynamic "typing" context that arises in the relevant HH derivation, it is the case that $(\texttt{of}\ M\ T)$ is derivable in HH if and only if $\{L \vdash \texttt{of}\ M\ T\}$ is provable in $\mathcal{G}$.[3] We can actually characterize legitimate representations for typing contexts in $\mathcal{G}$ by using the predicate symbol `ctx` that is defined by the following clauses:

$$\texttt{ctx}\ \texttt{nil} \triangleq \top \qquad\qquad \nabla x.\texttt{ctx}\ (\texttt{cons}\ (\texttt{of}\ x\ T)\ L) \triangleq \texttt{ctx}\ L$$

In this setting, the uniqueness of type assignment in STLC is captured by the following formula:

$$\forall L\ M\ T\ T', \texttt{ctx}\ L \supset \{L \vdash \texttt{of}\ M\ T\} \supset \{L \vdash \texttt{of}\ M\ T'\} \supset T = T'.$$

A proof of this formula can be found, for instance, in the Abella tutorial [4].

In the thesis work, we will use the two-level logic approach to verify compiler transformations implemented in λProlog. As we shall see in Section 4, the two-level logic approach is essential for reasoning about such implementations. Combining it with the other rich reasoning devices in Abella, we will be able to prove deep properties such as preservation of meaning by compiler transformations in an effective way.

3 Implementing Functional Transformations

Transformations on functional programs (or functional transformations) are often presented via rules that relate source and target programs. One of the goals of the thesis is to present a methodology for implementing functional transformations. Under this methodology, we encode functional transformations as λProlog specifications by utilizing the approach described in Section 2.1 for

[3] We use a systematic abuse of notation here and elsewhere, confusing the items introduced into the signature by universal goals in HH derivations with the nominal constants that arise in corresponding $\mathcal{G}$ derivations.

translating rules into clauses. Since λProlog specifications are executable, they serve also as implementations of the transformations. Functional transformations typically involve the manipulation of binding structure. We argue that the HOAS approach that is supported by λProlog provides a convenient way to realize such manipulations. As a result of these features, the λProlog specifications of functional transformations are concise and transparently related to the original rule-based description. Moreover, they have a logical structure that can be exploited in the process of reasoning about their correctness.

We illustrate the above ideas in this section by considering the encoding of a few specific functional transformations. In the first subsection we describe the *closure conversion* transformation that embodies an important compilation pass for functional languages. The discussion about closure conversion will bring out the importance of manipulation of binding structure in functional transformations. In the following subsection, we demonstrate how to succinctly encode closure conversion in λProlog. In particular, we will discuss the benefits of the HOAS approach to encoding functional transformations. To demonstrate the generality of our methodology, we also discuss the encoding of another important transformation for compiling functional languages known as *code hoisting*. Our goal in the thesis is to use the methodology that we outline in implementing a compiler for a simple but representative functional language in λProlog. In the concluding subsection, we elaborate on the work that we intend to do towards fulfilling this goal.

3.1 Rule-Based Specifications of Transformations

Program transformations take programs in one intermediate language as input, perform analysis and transformations on them and output the result in the same language or another (usually lower-level) language. Such transformations can be described as a set of rules defining relations on the source and target programs. For transformations on functional programs, the rules usually involve analysis and manipulation of binding structure.

Closure conversion is a transformation on functional programs whose purpose is to make functions independent of their contexts. Specifically, closure conversion replaces each function in the source program by a *closure*, which is a pair of *code* and *environment*. The code part is obtained by abstracting the original function over an extra environment parameter and replacing the references to free variables in the function body by projections from the environment parameter. The environment part encodes the construction of this parameter in the enclosing context. As a concrete example, closure conversion applied to the following pseudo OCaml code

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

yields

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

Here the notion `<F,E>` represent closures where `F` is the code and `E` is the environment. The `i`th project from an environment parameter `e` is denoted by `e.i`. By making functions independent of context, closure conversion enables further transformations that simplify higher-order programs into first-order forms, as we shall see in Section 3.3.

$$
\begin{aligned}
T &::= \mathbb{N} \mid T_1 \to T_2 \\
M &::= n \mid x \mid \lambda x.M \mid M\ M \\
V &::= n \mid \lambda x.M
\end{aligned}
$$

(a) Syntax of the source language

$$
\frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}
$$

$$
\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x : T_1.M : (T_1 \to T_2)}\ x \notin \Gamma \qquad \frac{\Gamma \vdash M_1 : T_1 \to T_2 \quad \Gamma \vdash M_2 : T_1}{\Gamma \vdash M_1\ M_2 : T_2}
$$

(b) Typing rules for the source language

Figure 2: The Source Language for Closure Conversion

To describe the transformation concretely, we must fix the syntax of the source and target languages that it applies to. The languages that we will use are presented in Figure 2 and Figure 3. In these figures, T, M and V denote types, terms and values, respectively. The source language is a extension of STLC with natural numbers in which the atomic type $\mathbb{N}$ represents the type of natural numbers and a constant n represents the corresponding natural number. The target language extends the source language by including mechanisms to describe environments and closures. Environments can be constructed by using the unit expression $()$ and the pair expressions of the form (M_1, M_2). The selection of items from an environment can be represented by applying **fst** to a sequence of **snd** terms. Specifically, an environment consisting of a list of terms $(M_1, ..., M_n)$ is represented as the term $(M_1, (..., (M_n, ())))$; the selection of the ith element from an environment M is represented by the term (**fst** (**snd** (...(**snd** M)))) where **snd** is applied $(i-1)$ times. We shall write $\pi_i(M)$ to denote this term, i.e., $\pi_i(M)$ represents the selection of the ith element from an environment M. A closure is formed by packing a term representing a function with an environment. Specifically, a closure is a term of the form $\langle M_1, M_2 \rangle$ where M_1 corresponds to the function and M_2 corresponds to the environment. To make use of closures, we need a way to unpack them. The unpacking of a closure is represented by a term of the form (**open** $\langle x_f, x_e \rangle = M_1$ **in** M_2); here M_1 is a term that is expected to evaluate to a closure and x_f and x_e are variables that are bound in the term and that have M_2 as their scope. Intuitively, a term with such a form represents a program that unpacks M_1 after evaluating it,

$$
\begin{aligned}
T &::= \mathbb{N} \mid T \to T \mid T \Rightarrow T \mid \mathbf{unit} \mid T \times T \\
M &::= n \mid x \mid () \mid (M, M) \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \mid \\
&\quad \lambda x.M \mid M\ M \mid \langle M, M \rangle \mid \mathbf{open}\ \langle x, y \rangle = M\ \mathbf{in}\ M \\
V &::= n \mid \lambda x.M \mid () \mid (V, V) \mid \langle V, V \rangle
\end{aligned}
$$

(a) Syntax of the target language

$$
\frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash () : \mathbf{unit}}
$$

$$
\frac{\Gamma \vdash M_1 : T_1 \quad \Gamma \vdash M_2 : T_2}{\Gamma \vdash (M_1, M_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash M : T_1 \times T_2}{\Gamma \vdash \mathbf{fst}\ M : T_1} \qquad \frac{\Gamma \vdash M : T_1 \times T_2}{\Gamma \vdash \mathbf{snd}\ M : T_2}
$$

$$
\frac{\Gamma \vdash M_1 : T_1 \quad \Gamma, x : T_1 \vdash M_2 : T}{\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : T}\ x \notin \Gamma
$$

$$
\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x : T_1.M : (T_1 \to T_2)}\ x \notin \Gamma \qquad \frac{\Gamma \vdash M_1 : T_1 \Rightarrow T_2 \quad \Gamma \vdash M_2 : T_1}{\Gamma \vdash M_1\ M_2 : T_2}
$$

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

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

(b) Typing of the target language

Figure 3: The Target Language of Closure Conversion

binds its function part to x_f and its environment part to x_e, and evaluates M_2 under these bindings. We call such a term an unpacking expression and M_2 the body of this expression. Finally, the target language contains let expressions to allow definition and usage of local expressions. A let expression has the form ($\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$) where x is a variable; this expression binds x and has M_2 as its scope. It represents the program M_2 in which the free variable x is defined by the expression M_1.

Not all expressions that can be formed using the syntax rules for the source and target language are well formed; such expressions must also satisfy typing requirements. These requirements are explained through typing judgments for both the source and the target language. Typing judgments have the form ($\Gamma \vdash M : T$). In such a judgment, Γ represents a typing context, written as $(x_1 : T_1, \ldots, x_n : T_n)$, where each x_i is a distinct variable and each T_i is a type. The typing rules in Figures 2 and 3 specify how such judgments may be derived; we say that a term M is well-formed and has type T in typing context Γ if we can derive the judgment ($\Gamma \vdash M : T$). Most of the typing rules are easy to understand. The somewhat less obvious cases are the rules

for typing closures and unpacking expressions. The code part of a closure is a special kind of function: it must be capable of taking an actual argument, combining it with an environment and then using the resulting environment in evaluating its body. This interpretation of a closure underlies the second last rule in Figure 3b, and the third rule from the end correspondingly shows how the code part of a closure is to be applied to an argument. Note that the type of a closure has the form $(T_1 \to T_2)$ which does not mention the type of the environment; this indicates the environments of closures are not directly accessible. The actual application of the code part takes place by unpacking the closure. The last rule, which provides the means for typing unpacking expressions, makes this structure explicit. Note that in this unpacking process we do not want the expression using the closure to have direct access to its environment; that information should only be available to the body of the code of the closure. This requirement is enforced by requiring l to be a type constant in the last rule that is not mentioned anywhere else in the rule. Note also that, since an unpacking expression binds x_e and x_f, these must also be be fresh with respect to the typing context Γ.

The meaning of functional programs can be explained by how to evaluate them, something that can be described via rules defining evaluation relations. We write $M \hookrightarrow M'$ for the relation that M evaluates to M'. This notation is overloaded for source and target languages. The evaluation rules for the source language are the following:

$$\frac{}{n \hookrightarrow n} \qquad \frac{}{\lambda x : T.M \to \lambda x : T.M}$$

$$\frac{M_1 \hookrightarrow \lambda x : T.M_1' \quad M_2 \hookrightarrow V_2 \quad M_1'[V_2/x] \hookrightarrow V}{M_1\ M_2 \hookrightarrow V}$$

Note that we are assuming a call-by-value semantics for evaluation in both languages. Most of the evaluation rules for the target language have a predictable form and we do not present them here. We only show the non-obvious rules which, as might be expected, correspond to evaluating closures and unpacking expressions:

$$\frac{M_1 \hookrightarrow V_1 \quad M_2 \hookrightarrow V_2}{\langle M_1, M_2 \rangle \hookrightarrow \langle V_1, V_2 \rangle} \qquad \frac{M_1 \hookrightarrow \langle V_1, V_2 \rangle \quad M_2[V_1/x_f][V_2/x_e] \hookrightarrow V}{\textbf{open}\ \langle x_f, x_e \rangle = M_1\ \textbf{in}\ M_2 \hookrightarrow V}$$

As we can see, these rules conform to the intuitive interpretation of closures and unpacking expressions.

Having defined the source and target languages and their semantics, we now present the closure conversion transformation. Given a source term, closure conversion recursively transforms abstractions in the source terms into closures and applications into unpacking expressions. It can also be described as a rule-based relational specification. The relation or judgment for closure conversion is written in the form $\rho \triangleright M \rightsquigarrow M'$ where M is a source term, M' is a target term and ρ is a mapping from variables in the source language to terms in the target language that has the form $(x_1 \mapsto M_1, \ldots, x_n \mapsto M_n)$ and that has $\{x_1, \ldots, x_n\}$

$$\frac{}{\rho \triangleright n \rightsquigarrow n}\ \text{cc-nat} \qquad \frac{(x \mapsto M) \in \rho}{\rho \triangleright x \rightsquigarrow M}\ \text{cc-var}$$

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

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

where $\rho' = (x \mapsto y, x_1 \mapsto \pi_1(x_e), \ldots, x_n \mapsto \pi_n(x_e))$

and p, y and x_e are fresh variables

$$\frac{\rho \triangleright x_1 \rightsquigarrow M_1 \quad \ldots \quad \rho \triangleright x_n \rightsquigarrow M_n}{\rho \triangleright (x_1, \ldots, x_n) \rightsquigarrow_e (M_1, \ldots, M_n)}\ \text{cc-env}$$

Figure 4: Closure Conversion Rules

as its domain. Such a judgment represents the fact that M is transformed to M' by closure conversion, assuming that the domain of ρ contains all the free variables in M and these free variables are transformed to what ρ maps them to. For describing the rules of closure conversion, we need another kind of judgment of the form $\rho \triangleright (x_1, \ldots, x_n) \rightsquigarrow_e (M_1, \ldots, M_n)$, which represents the fact that a tuple consisting of variables $x_1, \ldots, x_n$ is transformed into a tuple consisting of terms $M_1, \ldots, M_n$ by closure conversion with the mapping ρ. The rules defining the two judgments are described in Figure 4. In the base case, the rule cc-nat transforms a natural number to itself and the rule cc-var transforms a variable x to the term ρ maps x to. The rule cc-env transforms a tuple $(x_1, \ldots, x_n)$ into a tuple $(M_1, \ldots, M_n)$ if x_i is transformed to M_i by closure conversion for $1 \leq i \leq n$. The real action of closure conversion happens in the rules cc-abs and cc-app. The rule cc-abs converts an abstraction $(\lambda x.M)$ into a closure. In this rule, the body M of the abstraction is recursively transformed into M' under ρ' which maps the argument x to the argument y in the target closure and a list of variables $(x_1, ..., x_n)$ that contains at least the free variables in $(\lambda x.M)$ to projects from an environment parameter x_e. The code of the closure is then formed as an abstraction that has a pair consisting of y and x_e as its argument and M' as its body. The environment of the closure is formed as a tuple M_e consisting of target terms obtained by transforming variables $(x_1, ..., x_n)$ under the current mapping ρ. The rule cc-app transforms an application $(M_1\ M_2)$ into an unpacking expression. In this rule, M_1 and M_2 are recursively transformed into M_1' and M_2', respectively. The term M_1' is expected to be a closure. The open expression is then formed from M_1' and M_2' which represents the application of the closure M_1' to its argument M_2'. Its operational reading is to apply the code of M_1' to a pair consisting of the argument M_2' and the environment of M_1', which essentially corresponds the operational reading of the original application.

As we have seen from the previous discussion, closure conversion operates on

programs containing objects with variable binding operators such as functions and closures. Because closure conversion is recursively performed over such objects, the rules defining it must deal with open terms and side conditions for free variables introduced by recursion; this is manifested in the rule cc-abs. Furthermore, closure conversion involves non-trivial *analysis* of binding structure: the rule cc-abs requires identifying a list of variables that at least cover all the free variables of the source abstraction. These properties related to the notion of bindings are also shared by other functional transformations. To formally specify functional transformations, it is important to faithfully capture such properties.

3.2 Formalizing Functional Transformations in λProlog

In this section we introduce the methodology for formalizing the rule-based relational specifications of functional transformations in λProlog. As might be expected, in this approach a functional transformation is treated as a relation between a source and target program that is represented by a predicate constant. Each rule defining the relation is translated into a program clause in λProlog, such that its conclusion becomes the head of the clause and its premises (if any) become the body of the clause. By using the HOAS approach, the binding operators in the programs involved in the transformation are represented by meta-level binders. As a result, the notions related to bindings such as scoping and substitutions are naturally captured by the properties of meta-level binders. Recursion over these binding operators is handled by using universal and hypothetical goals such that the side conditions for free variables introduced by recursion are captured by the derivation rules of HH. Moreover, the analysis on binding structure can be represented by rule-based relational specifications and formalized in λProlog.

We demonstrate this methodology by showing how it can be used to encode the closure conversion transformation presented in the last section in λProlog. We first describe the encoding of the syntax of the source and target languages. We identify the type `ty` for the types of the source and target languages. Then we identify constants `nat` and `unit` of type `ty` to represent atomic types $\mathbb{N}$ and **unit**, respectively, and the constants `prod`, `arr` and `arr'` of type $\texttt{ty} \to \texttt{ty} \to \texttt{ty}$ to represent the type operators $\times$, $\to$ and $\Rightarrow$, respectively. Note that `arr` has different interpretations in the encodings of the source and target languages: in the source language it represents the operator for generating the types of abstractions; in the target language it represents the operator for generating the types of closures. The constant `arr'` represents the operator for generating the types of abstractions in the target language. It plays no role in the encoding of the source language.

We designate types `tm` and `tm'` to classify the source and target terms, respectively. To encode terms of natural numbers, we identify the type `nat` for natural numbers. We omit an introduction to the encoding of natural numbers in λProlog which follows a standard practice. To encode the source terms, we

identify the following constants:

$$\texttt{nat} : \texttt{nat} \to \texttt{tm} \qquad \texttt{abs} : (\texttt{tm} \to \texttt{tm}) \to \texttt{tm} \qquad \texttt{app} : \texttt{tm} \to \texttt{tm} \to \texttt{tm}$$

Following the standard practice in the HOAS approach, the binding aspect of the object-level abstractions is isolated and represented by the meta-level abstractions. This is shown by `abs` which takes a meta-level abstraction to form an object-level abstraction. The constants for encoding target terms include ones above, but in a primed version, *e.g.*, $\texttt{nat}' : \texttt{nat} \to \texttt{tm}'$. The rest constants are as follows:

$$\begin{array}{ll} \texttt{unit}' : \texttt{tm}' & \texttt{fst}', \texttt{snd}' : \texttt{tm}' \to \texttt{tm}' \\ \texttt{pair}' : \texttt{tm}' \to \texttt{tm}' \to \texttt{tm}' & \texttt{let}' : \texttt{tm}' \to (\texttt{tm}' \to \texttt{tm}') \to \texttt{tm}' \\ \texttt{clos}' : \texttt{tm}' \to \texttt{tm}' \to \texttt{tm}' & \texttt{open}' : \texttt{tm}' \to (\texttt{tm}' \to \texttt{tm}' \to \texttt{tm}') \to \texttt{tm}' \end{array}$$

Again, meta-level abstractions are used for encoding object-level bindings, as manifested by the types of $\texttt{let}'$ and $\texttt{open}'$. Specifically, assuming M_1' is encoded from M_1 and M_2' is encoded from M_2, a let expression ($\textbf{let}\ x = M_1\ \textbf{in}\ M_2$) is encoded as

$$\texttt{let}'\ (M_1')\ (x \setminus M_2')$$

and an unpacking expression ($\textbf{open}\ \langle x_f, x_e \rangle = M_1\ \textbf{in}\ M_2$) is encoded as

$$\texttt{open}'\ M_1'\ (x_f \setminus x_e \setminus M_2').$$

We then present the encoding of typing rules for the source and target languages. We use two predicate constants $\texttt{of} : \texttt{tm} \to \texttt{ty} \to \texttt{o}$ and $\texttt{of}' : \texttt{tm}' \to \texttt{ty} \to \texttt{o}$ to represent the typing relations in the source and target languages, respectively. We translate the typing rules into program clauses in λProlog by following the approach as described in Section 2.1. Such an encoding is obvious for most typing rules. We only show the program clauses that encode the typing rules for closures and open expressions:

$$\begin{array}{l} \texttt{of}'\ (\texttt{clos}'\ F\ E)\ (\texttt{arr}\ T_1\ T_2)\ \texttt{:-}\ \texttt{of}'\ F\ (\texttt{arr}'\ (\texttt{prod}\ T_1\ T_e)\ T_2)\,, \texttt{of}'\ E\ T_e. \\ \texttt{of}'\ (\texttt{open}'\ M\ R)\ T\ \texttt{:-}\ \texttt{of}'\ M\ (\texttt{arr}\ T_1\ T_2)\,, \\ \qquad \texttt{pi}\ f \setminus \texttt{pi}\ e \setminus \texttt{pi}\ l \setminus \texttt{of}'\ f\ (\texttt{arr}'\ (\texttt{prod}\ T_1\ l)\ T_2) \Rightarrow \texttt{of}'\ e\ l \Rightarrow \texttt{of}'\ (R\ f\ e)\ T. \end{array}$$

In the second clause, the recursive typing on the body of an open expression is represented by the last formula consisting of universal and hypothetical goals. The only way to derive this formula is to introduce the constants f and e to represent the bound variables of the unpacking expression and the constant l to represent the type of e, to add clauses ($\texttt{of}'\ f\ (\texttt{arr}'\ (\texttt{prod}\ T_1\ l)\ T_2)$) and ($\texttt{of}'\ e\ l$), which represent the typing rules for these bound variables, into the dynamic context, and to then derive the goal ($\texttt{of}'\ (R\ f\ e)\ T$). By the derivation rules for universal goals, f, e and l must be fresh constants. This requirement captures the side condition in the rule for typing unpacking expressions. Note that because the typing context is implicit and may not be empty, this specification does not enforce the code of closures to be closed. However, given

the specification of typing rules and closure conversion, which we will present shortly, we can prove that the code component of any closure obtained from closure conversion must be closed. Thus, the closedness of code components of closures becomes a property of this specification that can be verified separately.

Similarly, we use constants $\texttt{eval} : \texttt{tm} \to \texttt{tm} \to \texttt{o}$ and $\texttt{eval}' : \texttt{tm}' \to \texttt{tm}' \to \texttt{o}$ to represent evaluation relations in the source and target languages, respectively. The evaluations rules for the source language are encoded by the following clauses:

$$\texttt{eval}\ (\texttt{nat}\ N)\ (\texttt{nat}\ N).$$
$$\texttt{eval}\ (\texttt{abs}\ R)\ (\texttt{abs}\ R).$$
$$\texttt{eval}\ (\texttt{app}\ M_1\ M_2)\ V\ \texttt{:-}\ \texttt{eval}\ M_1\ (\texttt{abs}\ R)\,, \texttt{eval}\ M_2\ V_2\,, \texttt{eval}\ (R\ V_2)\ V.$$

In the last rule, β-conversion is used to represent substitutions at the object-level. The evaluation rules for the target language can be formalized in a similar manner. We consider only a few interesting cases. For instance, the evaluation rules for closures and unpacking expressions are presented as follows:

$$\texttt{eval}\ (\texttt{clos}'\ M_1\ M_2)\ (\texttt{clos}'\ V_1\ V_2)\ \texttt{:-}\ \texttt{eval}\ M_1\ V_1\,, \texttt{eval}\ M_2\ V_2.$$
$$\texttt{eval}\ (\texttt{open}'\ M\ R)V\ \texttt{:-}\ \texttt{eval}\ M\ (\texttt{clos}'\ F\ E)\,, \texttt{eval}\ (R\ F\ E)\ V.$$

Having describe the encoding of the languages closure conversion operates on, we are finally in the position to discuss the encoding of closure conversion itself. Closure conversion is defined relative to a mapping for source variables. We use the type `map` to classify the elements in such mappings and the constant $\texttt{map} : \texttt{tm} \to \texttt{tm}' \to \texttt{map}$ to form these elements. Mappings are then represented by a list of `map` terms. We identify the type `map_list` for mappings and constants `nil` and `::` (written in infix form) as constructors for `map_list`. We represent the membership relation of `map_list` as the predicate constant

$$\texttt{member} : \texttt{map} \to \texttt{map\_list} \to \texttt{prop}.$$

The program clauses defining the membership relation is as follows:

$$\texttt{member}\ X\ (X\ \texttt{::}\ L).$$
$$\texttt{member}\ Y\ (X\ \texttt{::}\ L)\ \texttt{:-}\ \texttt{member}\ Y\ L.$$

We also encode the lists of source and target terms and their membership property in λProlog by following the same process. The type for the lists of source (target) terms is `tm_list` (`tm'_list`). For simplicity, we overload the constants `nil`, `::` and `member` for different lists. Further, we encode the relation for combining two lists of source terms into one by the following clauses defining the predicate $\texttt{combine} : \texttt{tm\_list} \to \texttt{tm\_list} \to \texttt{tm\_list} \to \texttt{o}$:

$$\texttt{combine}\ \texttt{nil}\ FVs_2\ FVs_2.$$
$$\texttt{combine}\ (X\ \texttt{::}\ FVs_1)\ FVs_2\ FVs\ \texttt{:-}$$
$$\quad \texttt{member}\ X\ FVs_2\,, \texttt{combine}\ FVs_1\ FVs_2\ FVs.$$
$$\texttt{combine}\ (X\ \texttt{::}\ FVs_1)\ FVs_2\ (X\ FVs)\ \texttt{:-}\ \texttt{combine}\ FVs_1\ FVs_2\ FVs.$$

The crux in formalizing the definition of closure conversion is capturing the content of the cc-abs rule. A key part of this rule is identifying a list containing at least all the free variables in the source abstraction. Ideally, we would like this list to be as small as possible, containing only the free variables in the source abstraction. This involves analysis of the structure of the source term, in particular, its binding structure. We show that this kind of analysis can be succinctly formalized as relational specifications. We use a predicate constant

$$\texttt{fvars} : \texttt{tm} \to \texttt{tm\_list} \to \texttt{tm\_list} \to \texttt{o}$$

to represent the relation for identifying free variables in terms. We further identify the constant $\texttt{notfree} : \texttt{tm} \to \texttt{o}$. Then we provided the clauses below that define the predicate `fvars` such that ($\texttt{fvars}\ M\ Vs\ FVs$) holds if and only if Vs contains at least all the free variables in M and FVs contains only the free variables in M. In these clauses we use the placeholder _ to represent a variable whose value is not of interest to us; this is a convention that will recur in the future discussion.

$\texttt{fvars}\ (\texttt{nat}\ \_)\ \_\ \texttt{nil}.$

$\texttt{fvars}\ X\ \_\ \texttt{nil}\ \texttt{:-}\ \texttt{notfree}\ X.$

$\texttt{fvars}\ Y\ Vs\ (Y\ \texttt{::}\ \texttt{nil})\ \texttt{:-}\ \texttt{member}\ Y\ Vs.$

$\texttt{fvars}\ (\texttt{app}\ M_1\ M_2)\ Vs\ FVs\ \texttt{:-}$

$\quad \texttt{fvars}\ M_1\ Vs\ FVs_1\ ,\texttt{fvars}\ M_2\ Vs\ FVs_2\ ,\texttt{combine}\ FVs_1\ FVs_2\ FVs.$

$\texttt{fvars}\ (\texttt{abs}\ R)\ Vs\ FVs\ \texttt{:-}\ \texttt{pi}\ x \setminus \texttt{notfree}\ x \Rightarrow \texttt{fvars}\ (R\ x)\ Vs\ FVs.$

These clauses specify the computation of the free variables in a source term M by recursion on the structure of the term, as follows. When M is an abstraction ($\texttt{abs}\ R$), its free variables are computed by introducing a new constant c to represent the binder of the abstraction, adding the assumption ($\texttt{notfree}\ c$) to marked c as not free, and recursively computing the free variables in the abstraction body ($R\ c$). When M is an application, its free variables are obtained by computing the free variables of its subterms and combining the results together. In the base case, M is either a natural number or a variable. The computation for the former case is obvious and encoded in the first clause. There are two cases to consider when M is a variable x: if x is bound, then ($\texttt{notfree}\ x$) must hold and there are no free variables in M; if x is a free, then x must be in Vs and the only free variable in M is x. The second and third clauses formalize these two cases.

Given the list of free variables in the source abstraction, the rule cc-abs identifies the environment of the resulting closure. To encode this, we identify the predicate constant

$$\texttt{mapenv} : \texttt{tm\_list} \to \texttt{map\_list} \to \texttt{tm}' \to \texttt{o}.$$

We provide the clauses below to define `mapenv` such that ($\texttt{mapenv}\ L\ Map\ ML$) holds if and only if the domain of Map contains variables in L and ML is a tuple

composed of the results of mapping the variables in L by *Map*.

$$\begin{array}{l}\texttt{mapenv nil \_ unit}'.\\ \texttt{mapenv}\ (X\ \texttt{::}\ L)\ \mathit{Map}\ (\texttt{pair}'\ M\ \mathit{ML})\\ \quad \texttt{:-}\ \texttt{member}\ (\texttt{map}\ X\ M)\ \mathit{Map}\,,\texttt{mapenv}\ L\ \mathit{Map}\ \mathit{ML}.\end{array}$$

For the recursive closure conversion on the body of the source abstraction, the cc-abs rule identifies a new mapping that maps the list of free variables in the source abstraction to projects from an environment parameter. To encode this, we identify a predicate

$$\texttt{mapvar} : \texttt{tm\_list} \to (\texttt{tm}' \to \texttt{map\_list}) \to \texttt{o}.$$

We provide the clauses below to define `mapvar` such that (`mapvar` Ł *NMap*) holds if and only if *NMap* is an abstraction ($e \setminus \mathit{Map}$) where e represents the environment parameter and *Map* maps the variables in L to projections from e.

$$\begin{array}{l}\texttt{mapvar nil}\ (e \setminus \texttt{nil}).\\ \texttt{mapvar}\ (X\ \texttt{::}\ L)\ (e \setminus (\texttt{map}\ X\ (\texttt{fst}'\ e))\ \texttt{::}\ (\mathit{Map}\ (\texttt{snd}'\ e)))\ \texttt{:-}\ \texttt{mapvar}\ L\ \mathit{Map}.\end{array}$$

We can now specify closure conversion. The relation of closure conversion is represented by the predicate constant:

$$\texttt{cc} : \texttt{map\_list} \to \texttt{tm\_list} \to \texttt{tm} \to \texttt{tm}' \to \texttt{o}.$$

The following clauses define `cc` such that (`cc` *Map Vs M M*′) holds if and only *Vs* contains all the free variables in M and M' is a translated version of M under the mapping *Map* for variables in *Vs*.

$$\begin{array}{l}\texttt{cc \_ \_}\ (\texttt{nat}\ N)\ (\texttt{nat}'\ N).\\ \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ \mathit{XM}\ \texttt{:-}\ \texttt{member}\ (\texttt{map}\ X\ M)\ \mathit{Map}.\\ \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{app}\ M_1\ M_2)\ (\texttt{open}'\ M_1'\ (f \setminus e \setminus \texttt{app}'\ f\ (\texttt{pair}'\ M_2'\ e)))\ \texttt{:-}\\ \quad \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M_1\ M_1'\,,\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M_2\ M_2'.\\ \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{abs}\ R)\\ \quad (\texttt{clos}'\ (\texttt{abs}'\ (p \setminus \texttt{let}'\ (\texttt{fst}'\ p)\ (y \setminus \texttt{let}'\ (\texttt{snd}'\ p)\ (e \setminus R'\ y\ e))))\ E)\ \texttt{:-}\\ \quad \texttt{fvars}\ (\texttt{abs}\ R)\ \mathit{Vs}\ \mathit{FVs}\,,\texttt{mapenv}\ \mathit{FVs}\ \mathit{Map}\ E\,,\texttt{mapvar}\ \mathit{FVs}\ \mathit{NMap}\,,\\ \quad \texttt{pi}\ x \setminus \texttt{pi}\ y \setminus \texttt{pi}\ e \setminus \texttt{cc}\ ((\texttt{map}\ x\ y)\ \texttt{::}\ (\mathit{NMap}\ e))\ (x\ \texttt{::}\ \mathit{FVs}\ )\ (R\ x)\ (R'\ y\ e).\end{array}$$

The clauses above are directly translated from the rules of closure conversion in Figure 4 in the sense that for each rule its conclusion becomes the head of a clause and its premises (if any) become the body of that clause. The last clause stands for the rule cc-abs and demonstrates the HOAS approach to encoding functional transformations. In the body of this clause, `fvars` is used to identify the free variables of the source abstraction, `mapenv` is used to create the environment of the target closure, and `mapvar` is used to create the new mapping for the recursive closure conversion. The recursion is realized by universal goals

in the body of the last clause. To derive these goals, fresh constants will be introduced for the argument x of the source abstraction, the argument y of the resulting closure and the environment parameter e; the recursive closure conversion corresponds to deriving the relation `cc` on the abstraction body with a new mapping that maps x to y and the free variables to projections from e. As we can see, the λ-tree representation, the universal goals and the meta-level applications are used to encode freshness conditions, renaming requirements and substitutions in a concise and logical way. Further, this specification of closure conversion can be executed in a system like Teyjus and hence also serves directly as an implementation.

3.3 Other Transformations

The methodology of formalizing rule-based description of functional transformations in λProlog is a general one. By using this methodology, we have implemented the *CPS transformation* [13] (which is applied before closure conversion to make the control flow explicit) and *code hoisting* (which is applied after closure conversion to move the closed functions to the top-level) in λProlog. The CPS transformation, closure conversion and code hoisting transform source programs into a first-order form. After that, it is possible to apply the conventional transformations and optimizations for first-order languages such as C.

We use code hoisting as an example to show the generality of this methodology. We assume the source programs for code hoisting are the outputs of closure conversion. The purpose of code hoisting is to move all the functions in a source program to the top-level as a list of closed functions independent of each other. After closure conversion, all functions become closed except that a function may occur in the body of other functions. To make functions independent of each other, we need to eliminate this dependency. This is achieved by recursively moving all the functions in the body of a function f out of its scope and modifying the function f to take the "hoisted" functions as extra arguments.

The source language of code hoisting is that in Figure 3. The target language of code hoisting is the source language extended with "hoisted" terms of the form

$$\textbf{letf } (f_1, ..., f_n) = (M_1, ..., M_n) \textbf{ in } M$$

where M_i is a function that is bound to f_i and M may refer to f_i for $1 \leq i \leq n$. Such a term is used to represent an output program of code hoisting where M_i for $1 \leq i \leq n$ represent the functions in the input program being hoisted to the top-level and M represents the input program with its functions replaced by variables that bind the hoisted functions. We call M_i for $1 \leq i \leq n$ the hoisted functions of such a term and M the body of this term. We will often write it as $(\textbf{letf } \vec{f} = \vec{M} \textbf{ in } M)$ where $\vec{f} = (f_1, ..., f_n)$ and $\vec{M} = (M_1, ..., M_2)$. We will also write $(\vec{f}, \vec{g})$ and $(\vec{F}, \vec{G})$ to represent the concatenation of tuples.

The relation or judgment of code hoisting has the form $(\rho \triangleright M \rightsquigarrow_{ch} M')$. It asserts that M' is the result of hoisting all functions in M to the top-level. The

$$\frac{}{\rho \rhd n \rightsquigarrow_{ch} \textbf{letf } () = () \textbf{ in } n}\ \text{ch-nat}$$

$$\frac{x \in \rho}{\rho \rhd x \rightsquigarrow_{ch} \textbf{letf } () = () \textbf{ in } x}\ \text{ch-var} \qquad \frac{}{\rho \rhd () \rightsquigarrow_{ch} \textbf{letf } () = () \textbf{ in } ()}\ \text{ch-unit}$$

$$\frac{\rho \rhd M_1 \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M_1' \quad \rho \rhd M_2 \rightsquigarrow_{ch} \textbf{letf } \vec{g} = \vec{G} \textbf{ in } M_2'}{\rho \rhd (M_1, M_2) \rightsquigarrow_{ch} \textbf{letf } (\vec{f}, \vec{g}) = (\vec{F}, \vec{G}) \textbf{ in } (M_1', M_2')}\ \text{ch-pair}$$

$$\frac{\rho \rhd M \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M'}{\rho \rhd \textbf{fst } M \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in fst } M'}\ \text{ch-fst}$$

$$\frac{\rho \rhd M \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M'}{\rho \rhd \textbf{snd } M \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in snd } M'}\ \text{ch-snd}$$

$$\frac{\rho \rhd M_1 \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M_1' \quad \rho, x \rhd M_2 \rightsquigarrow_{ch} \textbf{letf } \vec{g} = \vec{G} \textbf{ in } M_2'}{\rho \rhd (\textbf{let } x = M_1 \textbf{ in } M_2) \rightsquigarrow_{ch} \textbf{letf } (\vec{f}, \vec{g}) = (\vec{F}, \vec{G}) \textbf{ in } (\textbf{let } x = M_1' \textbf{ in } M_2')}\ \text{ch-let}$$

where x is not already in ρ and is not a free variable of $\vec{G}$

$$\frac{\rho, x \rhd M \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M'}{\rho \rhd \lambda x.M \rightsquigarrow_{ch} \textbf{letf } (\vec{f}, g) = (\vec{F}, \lambda \vec{f}.\lambda x.M') \textbf{ in } g\ \vec{f}}\ \text{ch-abs}$$

where x is not already in ρ and is not a free variable of $\vec{F}$

$$\frac{\rho \rhd M_1 \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M_1' \quad \rho \rhd M_2 \rightsquigarrow_{ch} \textbf{letf } \vec{g} = \vec{G} \textbf{ in } M_2'}{\rho \rhd M_1\ M_2 \rightsquigarrow_{ch} \textbf{letf } (\vec{f}, \vec{g}) = (\vec{F}, \vec{G}) \textbf{ in } M_1'\ M_2'}\ \text{ch-app}$$

$$\frac{\rho \rhd M_1 \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M_1' \quad \rho \rhd M_2 \rightsquigarrow_{ch} \textbf{letf } \vec{g} = \vec{G} \textbf{ in } M_2'}{\rho \rhd \langle M_1, M_2 \rangle \rightsquigarrow_{ch} \textbf{letf } (\vec{f}, \vec{g}) = (\vec{F}, \vec{G}) \textbf{ in } \langle M_1', M_2' \rangle}\ \text{ch-clos}$$

$$\frac{\rho \rhd M_1 \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M_1' \quad \rho, x_e, x_f \rhd M_2 \rightsquigarrow_{ch} \textbf{letf } \vec{g} = \vec{G} \textbf{ in } M_2'}{\begin{array}{l}\rho \rhd \textbf{open } \langle x_f, x_e \rangle = M_1 \textbf{ in } M_2 \rightsquigarrow_{ch} \\ \textbf{letf } (\vec{f}, \vec{g}) = (\vec{F}, \vec{G}) \textbf{ in open } \langle x_f, x_e \rangle = M_1' \textbf{ in } M_2'\end{array}}\ \text{ch-open}$$

where x_e and x_f are not already in ρ and are not free variables of $\vec{G}$

Figure 5: Code Hoisting

context ρ is used to keep track of the free variables in M. The rules of code hoisting are shown in Figure 5. These rules have a general structure: given a source term M, code hoisting is applied recursively to subterms in M and the output is formed by combining the results of the recursion. Note that every time code hoisting goes under a binder, a fresh variable for the binder is introduced to ρ. Moreover, to move the hoisted functions out of the scope of a binder, we need to make sure that they do not contain free occurrences of the variable bound by that binder. These requirements are captured by the side conditions in rules ch-let, ch-abs and ch-open.

To formalize code hoisting in λProlog, we first need to encode the syntax of hoisted terms. Towards this end, we identify the constant $\texttt{hbase} : \texttt{tm}' \to \texttt{tm}'$ to represent the bodies of hoisted terms, $\texttt{habs} : (\texttt{tm}' \to \texttt{tm}') \to \texttt{tm}'$ to bind hoisted

functions, and $\texttt{htm} : \texttt{tm'\_list} \to \texttt{tm}' \to \texttt{tm}'$ to form a hoisted term from a list of hoisted functions and a term that represent its body and that contains nested `habs` terms to bind hoisted functions if the list is not empty. A hoisted term of the form

$$\textbf{letf}\ (f_1, ..., f_n) = (M_1, ..., M_n)\ \textbf{in}\ M$$

is encoded as

$$\texttt{htm}\ (M_1\ \texttt{::}\ \ldots\ \texttt{::}\ M_n\ \texttt{::}\ \texttt{nil})\ (\texttt{habs}\ (f_1 \setminus \ldots (\texttt{habs}\ (f_n \setminus \texttt{hbase}\ M)))).$$

We often write $(\texttt{habs}\ (f_1, ..., f_n) \setminus M)$ for $\texttt{habs}\ (f_1 \setminus \ldots (\texttt{habs}\ (f_n \setminus \texttt{hbase}\ M)))$.

We use the predicate $\texttt{ch} : \texttt{tm}' \to \texttt{tm}' \to \texttt{o}$ to represent the relation of code hoisting and translate the code hoisting rules into the following clauses. They define `ch` such that $(\texttt{ch}\ M\ M')$ holds if and only if M is transformed into M' by code hoisting.

$\texttt{ch}\ (\texttt{nat}'\ N)\ (\texttt{htm nil}\ (\texttt{hbase}\ (\texttt{nat}'\ N))).$

$\texttt{ch unit}'\ (\texttt{htm nil}\ (\texttt{hbase unit}')).$

$\texttt{ch}\ (\texttt{pair}'\ M_1\ M_2)\ (\texttt{htm}\ FE\ M')\ \texttt{:-}\ \texttt{ch}\ M_1\ (\texttt{htm}\ FE_1\ M_1')\,, \texttt{ch}\ M_2\ (\texttt{htm}\ FE_2\ M_2')\,,$
$\quad \texttt{append}\ FE_1\ FE_2\ FE\,, \texttt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \texttt{pair}'\ x\ y)\ M'.$

$\texttt{ch}\ (\texttt{fst}'\ M)\ (\texttt{htm}\ FE\ M'')\ \texttt{:-}\ \texttt{ch}\ M\ (\texttt{htm}\ FE\ M')\,, \texttt{hconstr}\ M'\ (x \setminus \texttt{fst}'\ x)\ M''.$

$\texttt{ch}\ (\texttt{snd}'\ M)\ (\texttt{htm}\ FE\ M'')\ \texttt{:-}\ \texttt{ch}\ M\ (\texttt{htm}\ FE\ M')\,, \texttt{hconstr}\ M'\ (x \setminus \texttt{snd}'\ x)\ M''.$

$\texttt{ch}\ (\texttt{let}'\ M\ R)\ (\texttt{htm}\ FE\ M'')\ \texttt{:-}\ \texttt{ch}\ M\ (\texttt{htm}\ FE_1\ M')\,,$
$\quad (\texttt{pi}\ x \setminus \texttt{ch}\ x\ (\texttt{htm nil}\ (\texttt{hbase}\ x)) \Rightarrow \texttt{ch}\ (R\ x)\ (\texttt{htm}\ FE_2\ (R'\ x)))\,,$
$\quad \texttt{append}\ FE_1\ FE_2\ FE\,, \texttt{hcombine\_abs}\ M'\ R'\ (x \setminus y \setminus \texttt{let}'\ x\ y)\ M''.$

$\texttt{ch}\ (\texttt{abs}'\ R)\ (\texttt{htm}\ ((\texttt{abs}'\ F) :: FE)\ (\texttt{habs}\ R''))\ \texttt{:-}$
$\quad (\texttt{pi}\ x \setminus \texttt{ch}\ x\ (\texttt{htm nil}\ (\texttt{hbase}\ x)) \Rightarrow \texttt{ch}\ (R\ x)\ (\texttt{htm}\ FE\ (R'\ x)))\,,$
$\quad (\texttt{pi}\ f \setminus \texttt{pi}\ l \setminus \texttt{hoistabs}\ R'\ f\ l\ \texttt{nil}\ (R''\ f)\ (F\ l)).$

$\texttt{ch}\ (\texttt{app}'\ M_1\ M_2)\ (\texttt{htm}\ FE\ M')\ \texttt{:-}\ \texttt{ch}\ M_1\ (\texttt{htm}\ FE_1\ M_1')\,, \texttt{ch}\ M_2\ (\texttt{htm}\ FE_2\ M_2')\,,$
$\quad \texttt{append}\ FE_1\ FE_2\ FE\,, \texttt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \texttt{app}'\ x\ y)\ M'.$

$\texttt{ch}\ (\texttt{clos}'\ M_1\ M_2)\ (\texttt{htm}\ FE\ M')\ \texttt{:-}\ \texttt{ch}\ M_1\ (\texttt{htm}\ FE_1\ M_1')\,, \texttt{ch}\ M_2\ (\texttt{htm}\ FE_2\ M_2')\,,$
$\quad \texttt{append}\ FE_1\ FE_2\ FE\,, \texttt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \texttt{clos}'\ x\ y)\ M'.$

$\texttt{ch}\ (\texttt{open}'\ M\ R)\ (\texttt{htm}\ FE\ M')\ \texttt{:-}\ \texttt{ch}\ M\ (\texttt{htm}\ FE_1\ M_1')\,,$
$\quad (\texttt{pi}\ f \setminus \texttt{pi}\ e \setminus \texttt{ch}\ f\ (\texttt{htm nil}\ (\texttt{hbase}\ f)) \Rightarrow \texttt{ch}\ e\ (\texttt{htm nil}\ (\texttt{hbase}\ e)) \Rightarrow$
$\quad\quad \texttt{ch}\ (R\ f\ e)\ (\texttt{htm}\ FE_2\ (R'\ f\ e)))\,,$
$\quad \texttt{append}\ FE_1\ FE_2\ FE\,, \texttt{hcombine\_abs2}\ M_1'\ R'\ (x \setminus y \setminus \texttt{open}'\ x\ y)\ M'.$

By using the HOAS approach, we are able to formalize recursion over binders and capture the side conditions in the rule of code hoisting in a concise and logical way. Again, we use universal goals to perform recursion over binders and hypothetical goals to introduce the rules for transforming bound variables. As a result, the context of code hoisting is represented implicitly by the dynamic context in HH sequents. The freshness condition is captured by the HH rules for deriving universal goals. The requirement that hoisted functions do not

contain free occurrences of binders which they are hoist over is captured by the ordering of quantifiers and binders. For instance, the recursive code hoisting over abstractions is expressed by the following formula in the clause describing the rule ch-abs:

$$\texttt{pi}\ x \setminus \texttt{ch}\ x\ (\texttt{htm}\ \texttt{nil}\ (\texttt{hbase}\ x)) \Rightarrow \texttt{ch}\ (R\ x)\ (\texttt{htm}\ FE\ (R'\ x))$$

Here the variable FE which represents the hoisted functions cannot depend on the binder x because FE is bound out side of x. Thus, the derivation of this formula will succeed only if the hoisted function do not contain x as a free variable. Same observation can be made for clauses describing the rules ch-let and ch-open.

The predicates `hcombine`, `hconstr`, `hcombine_abs` and `hcombine_abs2` are identified and defined to combine the results of code hoisting on subterms to form the outputs. For instance, if M_1' is $(\texttt{habs}\ (f_1, \ldots, f_n) \setminus M_1)$ and M_2' is $(\texttt{habs}\ (g_1, \ldots, g_m) \setminus M_2)$, then

$$\texttt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \texttt{pair}'\ x\ y)\ M'$$

holds when M' is $(\texttt{habs}\ (f_1, \ldots, f_n, g_1, \ldots, g_m) \setminus \texttt{pair}'\ M_1\ M_2)$.

The definition of `hoistabs` predicate captures the essence of code hoisting and needs some explanation. Intuitively, it eliminates the dependence of a function F on the functions occurring in the body of F by abstracting F over a tuple that binds these functions. Let $(\texttt{htm}\ FE\ (R'\ x))$ be the result of recursive code hoisting on the body of an abstraction. Then R' is a meta-level abstraction of the form

$$x \setminus (\texttt{habs}\ (f_1, \ldots, f_n) \setminus (R\ f_1\ \ldots\ f_n\ x)).$$

where $f_1, \ldots, f_n$ binds the functions in FE. Given such an R', the following formula

$$\texttt{hoistabs}\ R'\ f\ l\ \texttt{nil}\ (R''\ f)\ (F\ l)$$

is derivable if and only if F is

$$l \setminus \textbf{let}\ f_1 = \pi_1(l)\ \textbf{in}\ \ldots \textbf{let}\ f_n = \pi_n(l)\ \textbf{in}\ \texttt{abs}'\ (\lambda x.R\ f_1\ \ldots\ f_n\ x)$$

and R'' is

$$f \setminus (\texttt{habs}\ (f_1, \ldots, f_n) \setminus (f\ (f_1, \ldots, f_n))).$$

As a result, $(\texttt{abs}'\ F)$ represents a closed function obtained by abstracting the body of R' over x and $(f_1, \ldots, f_n)$. As indicated in the final output

$$(\texttt{htm}\ ((\texttt{abs}'\ F) :: FE)\ (\texttt{habs}\ R'')),$$

the function $(\texttt{abs}'\ F)$ is hoisted to the top-level and bound by f in R''. Thus, the dependence of R' on $f_1, \ldots, f_n$ is captured by the application in R''.

3.4 Proposed Work Relating to Implementation

To demonstrate the effectiveness of our approach for implementing functional transformations, we will develop a verified compiler for a PCF-style language; this language extends what we have considered here with recursion, control flow constructs and arithmetic. Our compiler will take programs in the HOAS representation as input; transforming actual programs into this form can be accomplished by standard tools for parsing and extracting an internal representation. Compilation will be achieved through several passes that translate the source programs into successive intermediate languages, and finally producing assembly code as output. A tentative structure for these passes is shown in Figure 6, explained qualitatively as follows: the CPS transformation will make the control flow explicit, closure conversion will make functions independent of the context, code hoisting will lift closed functions to the top level, allocation will map objects and code to memory spaces, and code generation will produce assembly code. The static (typing) and dynamic (evaluation) semantics of the input, output and intermediate languages, and the transformations of each pass will be described as rules in the Structural Operational Semantics (SOS) style. We are going to encode the source and target languages and their semantics and implement these transformations in λ-Prolog following the methodology described in the previous sections and demonstrate the effectiveness of this methodology.

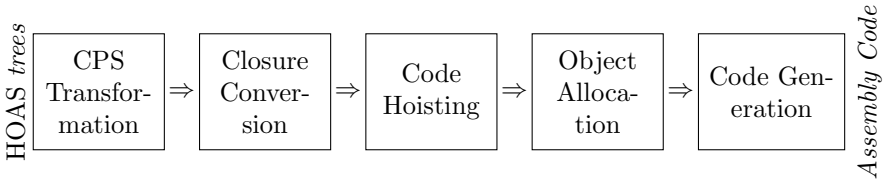

Figure 6: The Compiler for a PCF-style Language

The above describes an initial goal for our work on the implementation. Depending on how things evolve, we will consider exploring the development of transformations on richer functional languages, such as a subset of ML, using our approach. This will require us to extend the language considered earlier with polymorphism and to re-implement some of the transformation passes such as closure conversion and code hoisting on the richer syntax. Based on these exercises, we hope to demonstrate that our approach scales well to more complex languages.

4 Verifying Functional Transformations

In the plan we have presented so far, a compiler will be realized by specifying functional transformations as relations in λProlog that are executed using Teyjus. To show the correctness of the compiler, we would need to show that each transformation preserves the meaning or semantics of the input program.

In the methodology that we are proposing, showing correctness amounts to formally proving properties of the λProlog specifications of functional transformations using Abella. More precisely, we formalize semantics preservation as a relation in Abella between source and target language programs and then show that this relation subsumes the relation that encodes the relevant functional transformation in λProlog. Reasoning about objects with binding structure is an inherent part of the verification of functional transformations. One of our goals is to show that the HOAS approach supported by Abella provides a convenient way to formalize such reasoning.

In the following subsections, we will elaborate on this methodology. We will first present two notions of semantics preservation: one based on *logical relations* and the other based on a weak equivalence relation between values. We will then use a proof of semantics preservation for closure conversion as an example to illustrate the methodology. In particular, for each notion of semantics preservation, we are going to present the informal proof of closure conversion and then describe its formalization in Abella. This discussion will bring out the benefits of the two-level logic approach and the HOAS approach in the semantics preservation proofs. We shall see that no matter what kind of notion of semantics preservation is used, the manipulation and reasoning about binding structure is always an important part of the verification and can be simplified by using the HOAS approach. Finally, we will describe the further work related to compiler verification that we plan to do in this thesis. A major component of the work will be to verify the full compiler for the PCF-style language implemented in λProlog using the methodology introduced in this section.

4.1 Notions of Semantics Preservation

One way to characterize the semantics of programs is to examine how they interact with the outside world. The interaction is characterized by their *behaviors* that have obvious and fixed meanings, such as termination and I/O events. Semantics preservation can then be described as the preservation of behaviors [29]. We write $t \Downarrow B$ to represent the judgment that the program t has the behavior B. If t' is the program that results from transforming the program t, t' preserves the semantics of t if the following property holds:

$$\forall B, t \Downarrow B \iff t' \Downarrow B$$

The strict notion of behavior preservation is often replaced by the notion of *behavior refinement* that corresponds to the following property:

$$\forall B, t' \Downarrow B \supset t \Downarrow B$$

This property allows the target program to choose to follow only *some* behaviors of the source program. For instance, if the evaluation order of expressions e_1 and e_2 is not fixed in the source program, then it may be acceptable if a particular order has been picked in the target program. We are also often not interested

in following specific behaviors of *all* source programs but only those that satisfy some criterion. For example, we may be concerned only about source programs that do not "go wrong." In this case, behavioral refinement can be further refined to

$$WellBehaved(t) \supset \forall B, t' \Downarrow B \supset t \Downarrow B$$

where $WellBehaved$ is defined as

$$WellBehaved(t) \iff (\forall B, t \Downarrow B \supset B \notin Wrong);$$

here $Wrong$ represents the set of "going wrong" behaviors. This definition of behavioral refinement gives the compiler the freedom to pick whatever target program it wants if the source program is one that will go wrong during execution. This is a choice exercised by many C compilers when they encounter source programs with undefined behaviors.

Behavior refinement is not easy to prove in general. However, if the source and target languages have deterministic semantics, then it is easy to show that the refined form of behavioral refinement described above is implied by the following property

$$\forall B \notin Wrong, t \Downarrow B \Rightarrow t' \Downarrow B$$

This property is much easier to prove since we can induct on the evaluation of source programs.

In this thesis, we will be concerned only with languages that have a deterministic semantics and that do not have constructs with side effects. In this context, there are only three kinds of behaviors to consider: either a program gets stuck, it diverges, or it evaluates to some value. We will consider getting stuck and divergence as "going wrong". The property above then reduces to the following statement, commonly known as *forward simulation*:

> *If a source program evaluates to some value v, then its transformed version evaluates to some value v' that is equivalent to v in a sense that is intuitively well-motivated.*

A key part of formalizing the forward simulation property is that of fixing the notion of equivalence between values that is to be used. A variety of definitions have been proposed in the literature, and the particular form has an impact both on the manner in which we may construct correctness proofs for a compiler and the use that we can make of the correctness result. Nies et. al. have proposed the following criteria that may be used to categorize the different approaches [39]:

- *Modularity*: This is a property that allows us to build the correctness proof for a large program in the way we build the program itself, i.e. by composing the correctness proofs for the modules constituting the program. Formally, this means that if we have shown that the target programs T_1 and T_2 preserve the behavior of the source programs S_1 and S_2 from which they were generated, then the code that results from linking T_1 and T_2 should also preserve the behavior of the result of composing S_1 and S_2 at the source level. This property is also called "horizontal composibility."

- *Flexibility*: This criterion amounts to saying that the definition of semantic preservation is fixed solely by the semantics of the source and target languages and is oblivious to the transformations that are performed. Flexibility allows combination of correctness proofs of programs that use the same definition of semantics preservation but are generated by different compilers.

- *Transitivity*: This criterion amounts to saying that semantics preservation proofs for individual transformations can be composed to derive the semantic preservation proof for the full sequence of transformations. Transitivity is necessary for the separate verification of compilation passes in multi-pass compilers. Transitivity is also called "vertical composibility".

In the rest of this subsection, we will introduce two notions of semantics preservation for proving forward simulation for compiler transformations based respectively on logical relations and a weak equivalence relation between values. We will discuss the strength and weakness of these notions based on the three criteria above. The discussion in this section builds up the background for the following subsections, in which we shall discuss the verification of functional transformations using these notions.

4.1.1 Semantics Preservation Based on Logical Relations

A *logical relation* defines a family of relations on λ-terms indexed by their types such that the definition of each relation at a certain type only refers to relations at smaller types. We can use logical relations to describe the forward simulation property for program transformations as follows. Given a program transformation P, we identify the logical relation $\sim$ to describe a simulation relation between terms in the source and target languages and the logical relation $\approx$ to describe an equivalence relation between values in the source and target languages. Both $\sim$ and $\approx$ are indexed by the types of the source terms (written as subscripts). We define $\sim$ such that $t \sim_\tau t'$ holds if and only if

> *for any value v, if t evaluates to v, then there exists some value v' such that t' evaluates to v' and $v \approx_\tau v'$ holds.*

To complete the description of the simulation relation, we need to define when values in the source and target language are considered equivalent. If the source and target languages contain the same atomic objects, then an obvious choice for the notion of equivalence at atomic types is identity. At function types, a natural interpretation of equivalence might be that the two expressions have equivalent behavior on equivalent arguments. Specifically, if τ is an arrow type $\tau_1 \to \tau_2$, $v \approx_\tau v'$ holds if and only if v and v' are functions such that the following holds

> *for any v_1 and v_1', if $v_1 \approx_{\tau_1} v_1'$, then $(v\ v_1) \sim_{\tau_2} (v'\ v_1')$.*

Note that $\sim$ and $\approx$ are mutually recursively defined. In the statement above we are using $\approx$ negatively, i.e. we are assuming it is already defined at a certain type in defining it at another type. This works because the type at which we assume that we already know the relation is smaller in an inductive ordering. In other words, this is a recursive definition based on the inductively defined collection of types.

Given the above interpretation of equivalence between program expressions, the forward simulation property for a transformation P becomes the following:

If a term t of type τ is transformed into t' by P, then $t \sim_\tau t'$ holds.

It is this kind of property that we would want to prove towards showing the correctness of the transformation.

Correctness proofs based on logical relations enjoy the modularity property because of the extensional reading underlying equivalence at function types. If we think of modules with external references as functions whose arguments are these references, then we can think the linking of modules as function applications. By the definition of the equivalence relation for function values, we can easily combine the semantics preservation proofs of individual modules to form the proof for the linked program.

Correctness proofs based on logical relations are also flexible. This is because the definitions of logical relations only depend on the semantics of the programs involved in such definitions. If different transformations use the same logical relation as the notion of semantics preservation, then their correctness proofs can be combined without any problem.

Establishing transitivity of correctness proofs when these use a notion of program equivalence that is based on logical relations is more difficult. Perhaps the most obvious way to obtain this property is to show that the underlying logical relations are composable. In particular, suppose that the logical relations $\sim^1$ and $\sim^2$ underlie the proofs of semantics preservation for two functional transformations and we want to show the correctness of the composition of these two transformations with respect to a third logical relation $\sim^3$. This would trivially be the case if we can show the following:

$$\forall \tau\ t_1\ t_2\ t_3\ , t_1 \sim^1_\tau t_2 \supset t_2 \sim^2_\tau t_3 \supset t_1 \sim^3_\tau t_3.$$

However, this kind of property is generally hard to prove. In fact, it may not hold even if the source and target languages for the transformations are identical and all three logical relations are the same. In [1], Ahmed proposed a way to restrict the permitted forms of logical relations using properties based on types that overcomes this difficulty. Regardless of the merits of this approach, it is not directly usable in the typical setting for compiler verification. The reason for this is that compiler transformations typically modify and simplify programs in a one language into programs in a *different* language that is equipped with specialized constructs motivated by the relevant transformation. As a result, the logical relations that are used to characterize semantics preservation at each stage of a multi-stage transformation are usually different ones and they

also relate programs in different languages. For example, $\sim^1$ may be a logical relation for the continuation-passing style transformation, $\sim^2$ may be a logical relation for the closure conversion transformation, and $\sim^3$ is correspondingly the logical relation that captures the notion of semantics preservation between the source and target languages for the composition of the two transformations. In this case, the definition of $\sim^1$ would have to take into account the way we intend continuations to function, while the definition of $\sim^2$ makes no assumption about continuations. In this kind of situation, it is not clear that we can derive a suitable logical relation between expressions in the initial and final languages simply by composing the the sequence of intermediate logical relations that have been used.

The above discussion shows that if we use a definition of semantics preservation that is based on logical relations, then it becomes difficult to prove the correctness of a multi-stage compiler simply by composing correctness proofs for each stage. We may, however, restrict our attention to those programs that produce a value of atomic type; doing so effectively means that we are focusing on compilation of complete programs and then modularity becomes an irrelevant issue. If we narrow our focus in this way and if equivalence at atomic types is based on a simple one-to-one mapping of values between relevant domains, then correctness results for multiple stages can obviously be composed to obtain a correctness result for the entire sequence of transformations. The example that we consider in Section 4.2 should clarify this point further.

4.1.2 A Weaker Version of Semantics Preservation

The idea of limiting attention to programs that produce values of atomic type has, in fact, been widely used in compiler verification; to take two recent examples, it underlies the CompCert [29] and the CakeML [28] projects. In this more restricted context, the main requirement of the notion of program equivalence that we use is that it accord with intuitions at atomic types. At function types, we may use extensional equality as is done with logical relations style definitions of equivalence, but we may also use other notions that, for example, help us construct correctness proofs more easily. One particular notion of equivalence that we may use is the relation induced by the transformation itself; this typically preserves values at atomic types, thus satisfying the key property we need of program equivalence.

If we use the above idea, the forward simulation property reduces to the following statement for any given transformation P:

> *For any t that is transformed by P into t', if t evaluates to some value v, then there exists an value v' such that t' evaluates to v' and v is transformed by P into v'.*

This statement, which amounts to saying that the transformation and evaluation permute with each other, is usually proved by induction on evaluation sequences in the source language. In order to be able to prove such a statement, we often

have to tune the definitions of the transformation and of evaluation to each other. We will illustrate this aspect in Section 4.3.

We shall refer somewhat loosely to the refined approach for proving semantics preservation that we have just described as an approach based on simulation. Correctness proofs based on this approach are transitively composible for reasons that we have previously explained; note that the justification for this approach is that we are eventually interested only in whole programs that produce a value of atomic type. Proofs in this style are, as a rule, not flexible because they are often dependent on tuning the definition of evaluation to the transformation being proved correct. Finally, the simulation based approach does not automatically guarantee modularity because the notion of equivalence it uses for functions is not sufficiently constrained. Determining ways to apply this approach that yield modularity is an active research topic; [48] presents a recent development along these lines.

4.2 Verification using Logical Relations

We now use closure conversion as an example to demonstrate how to verify functional transformations by using logical relations as the notion of semantics preservation. We will first present the informal proof and then its formalization in Abella. The development of the informal proof follows the presentation in [36]. The discussion of the formal proof will show how the HOAS approach simplifies reasoning with regard to concepts such as substitution and closed terms that must pay attention to binding structure.

4.2.1 Verifying Closure Conversion using a Logical Relation

We present a logical relation consisting of two mutually recursively defined relations called the simulation relation and the equivalence relation.[4] Each consists of a family of relations between the source and target terms of closure conversion and indexed by the types of the source terms. Furthermore, the simulation relation is only defined for closed terms and the the equivalence relation is only defined for closed values. We use $\sim$ to denote the simulation relation and $\approx$ to denote the equivalence relations. Their definitions are given as follows:

$$M \sim_T M' \iff \forall V. M \hookrightarrow V \supset \exists V'. M' \hookrightarrow V' \wedge V \approx_T V';$$
$$n \approx_{\mathbb{N}} n;$$
$$(\lambda x.M) \approx_{T_1 \to T_2} \langle V', V_e \rangle \iff \forall V_1\ V_1'. V_1 \approx_{T_1} V_1' \supset M[V_1/x] \sim_{T_2} V'\ (V_1', V_e).$$

By this definition, the source program M simulates the target program M' if whenever M evaluates to some value V, M' evaluates to a value V' equivalent

[4] The terminologies "simulation" and "equivalence" we use in this section are particular to the semantics preservation proofs by logical relations. In particular, by "equivalence relation" we mean a notion of equivalence between programs and not the mathematical notion of a relation that is reflexive, symmetric and transitive. Similarly, "simulation" relation here should not be confused with simulation as a notion of semantics preservation.

to V. Equivalence between values are defined as follows. At the base type, two values are equivalent if they are identical. At an arrow type $T_1 \to T_2$, a function is equivalent to a closure if given any equivalent arguments V_1 and V_1' at the type T_1, the evaluation of the application of the function to V_1 simulates that of the closure to V_1'. Note that in this case the definition of $\approx$ refers to itself as an assumption. However, this is acceptable: this is a definition by recursion that is based on the inductively defined collection of types.

The logical relation we have just introduced is not enough to describe semantics preservation of closure conversion. This is because in the most general case closure conversion works with open terms while this logical relation only makes sense for closed terms. To describe semantics preservation for open terms, we shall introduce the concept of equivalence between substitutions. Then we assert that the term M' in the target language preserves the semantics of the term M in the source language if the simulation relation holds for M and M' under any equivalent substitutions that bind all the free variables in the respective terms.

The substitutions that we need to consider (both in the source and in the target language) have the form $(V_1/x_1, \ldots, V_n/x_n)$ where, for $1 \leq i \leq n$, each x_i is a distinct variable and V_i is a closed value; such a substitution maps each variable x_i in the finite collection $\{x_1, \ldots, x_n\}$ respectively to the value V_i. We use σ to denote substitutions and $M[\sigma]$ to denote the application of the substitution σ to the term M. For closure conversion, we need to consider the possibility that a collection of free variables in the source substitution may be reified into an environment variable in the target substitution. This motivates the following definition in which γ represents a source language substitution, where $(x_m : T_m, ..., x_1 : T_1)$ assigns types to free variables that form the environment of a closure:

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

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

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

The typing context consists of two parts: $(x_n : T_n, ..., x_1 : T_1)$ for bound variables in the scope of the enclosing function and Γ for the free variables that form the environment. Correspondingly, $(V_1/x_1, ..., V_n/x_n)$ is the source substitution for bound variables and γ is the source substitution for a superset of free variables; $(V_1'/y_1, ..., V_n'/y_n)$ is the target substitution for bound variables and (V_e/x_e) is the target substitution for the environment.

Closure conversion preserves semantics if, for any well-typed source term M, if M is transformed into M' by it and for equivalent substitutions δ and δ' that respectively bind all the free variables in M and M', it is the case that $M[\delta]$ simulates $M'[\delta']$. This is stated as the theorem below:

Theorem 1 *Let* $\delta = (V_1/x_1, ..., V_n/x_n); \gamma$ *and* $\delta' = (V'_1/y_1, ..., V'_n/y_n, V_e/x_e)$ *be source and target language substitutions and let* $\Gamma = (x'_m : T'_m, ..., x'_1 : T'_1, x_n : T_n, ..., x_1 : T_1)$ *be a source language typing context such that* $\delta \approx_\Gamma \delta'$. *Further, let* $\rho = (x_1 \mapsto y_1, ..., x_n \mapsto y_n, x'_1 \mapsto \pi_1(x_e), ..., x'_m \mapsto \pi_m(x_e))$. *If* $\Gamma \vdash M : T$ *and* $\rho \triangleright M \rightsquigarrow M'$, *then* $M[\delta] \sim_T M'[\delta']$.

This theorem can be proced by induction on the derivation of $(\rho \triangleright M \rightsquigarrow M')$. The case when M is a natural number is obvious. When M is a variable, the conclusion is obtained by extracting properties of $\delta \approx_\Gamma \delta'$. The application case can be proved by invoking the inductive hypothesis on the sub-derivations of `cc` and composing the results to form the conclusion. The only complicated one is when M is an abstraction. In this case M' must have the form $\langle M'_1, M'_2 \rangle$. Then $(M'[\delta'] = \langle M'_1[\delta'], M'_2[\delta'] \rangle = \langle M'_1, M'_2[\delta'] \rangle)$ by using the property that the code component of the closure is closed. The proof proceed by forming new typing judgments for the body of the abstraction and applying the inductive hypothesis to get the result to form the conclusion.

From the theorem above we can derive the following corollary that describes the preservation of evaluation for closed programs of atomic types:

Corollary 1 *If* $(\vdash M : \mathbb{N})$ *and* $(\triangleright M \rightsquigarrow M')$, *then for any* N, $M \hookrightarrow (\mathit{nat}\ N)$ *implies* $M' \hookrightarrow (\mathit{nat}'\ N)$.

We can prove semantics preservation theorems similar to Theorem 1 for other transformations and derive corollaries similar to Corollary 1. As we have already mentioned, these semantics preservation theorems are not transitively composible because it is not apparent how to compose the underlying notions of semantic equivalence. However, the composition of equivalence at atomic types is much easier and, in fact, trivial. Thus, the corollaries that rely only on equivalence at such types are obviously composible. By composing these corollaries, we will be able to prove that the full sequence of transformations preserve the semantics for closed programs of atomic types. In other words, the logical relations based approach is transitive if we limit ourselves to the compilation of a program in its entirety.

4.2.2 Formalizing the Logical Relation-Style Verification

We now demonstrate how to formalize the semantics preservation proof of closure conversion in Abella. Before we can formally describe the notion of semantics preservation, we need to formalize the concept of values and closed terms. We characterize the values in the source language by using the predicate constant `val : tm → o` that is defined by the following clauses in λProlog:

$$\texttt{val}\ (\texttt{nat}\ N). \qquad \texttt{val}\ (\texttt{abs}\ R).$$

Similarly, we characterize the values in the target language by using the predicate constant $\texttt{val}' : \texttt{tm}' \to \texttt{o}$ defined by the following clauses:

$$\texttt{val}'\ (\texttt{nat}'\ N). \qquad \texttt{val}'\ \texttt{unit}'. \qquad \texttt{val}'\ (\texttt{abs}'\ R).$$
$$\texttt{val}'\ (\texttt{pair}'\ V_1\ V_2)\ \texttt{:-}\ \texttt{val}'\ V_1\ ,\texttt{val}'\ V_2.$$
$$\texttt{val}'\ (\texttt{clos}'\ F\ E)\ \texttt{:-}\ \texttt{val}'\ F\ ,\texttt{val}'\ E.$$

We characterize the terms in the source and target languages by using respectively the predicate constants $\texttt{tm} : \texttt{tm} \to \texttt{o}$ and $\texttt{tm}' : \texttt{tm}' \to \texttt{o}$ that are defined by the following clauses in λProlog:

$\texttt{tm}\ (\texttt{nat}\ N).$
$\texttt{tm}\ (\texttt{app}\ M_1\ M_2)\ \texttt{:-}\ \texttt{tm}\ M_1, \texttt{tm}\ M_2.$
$\texttt{tm}\ (\texttt{abs}\ R)\ \texttt{:-}\ \texttt{pi}\ x \setminus \texttt{tm}\ x \Rightarrow \texttt{tm}\ (R\ x).$
$\texttt{tm}'\ (\texttt{nat}'\ N). \qquad \texttt{tm}'\ \texttt{unit}'.$
$\texttt{tm}'\ (\texttt{pair}'\ M1\ M2)\ \texttt{:-}\ \texttt{tm}'\ M1\ ,\texttt{tm}'\ M2.$
$\texttt{tm}'\ (\texttt{fst}'\ M)\ \texttt{:-}\ \texttt{tm}'\ M. \qquad \texttt{tm}'\ (\texttt{snd}'\ M)\ \texttt{:-}\ \texttt{tm}'\ M.$
$\texttt{tm}'\ (\texttt{let}'\ M\ R)\ \texttt{:-}\ \texttt{tm}'\ M\ ,\texttt{pi}\ x \setminus \texttt{tm}'\ x \Rightarrow \texttt{tm}'\ (R\ x).$
$\texttt{tm}'\ (\texttt{abs}'\ R)\ \texttt{:-}\ \texttt{pi}\ x \setminus \texttt{tm}'\ x \Rightarrow \texttt{tm}'\ (R\ x).$
$\texttt{tm}'\ (\texttt{app}'\ M_1\ M_2)\ \texttt{:-}\ \texttt{tm}'\ M_1\ ,\texttt{tm}'\ M_2.$
$\texttt{tm}'\ (\texttt{clos}'\ M_1\ M_2)\ \texttt{:-}\ \texttt{tm}'\ M_1\ ,\texttt{tm}'\ M_2.$
$\texttt{tm}'\ (\texttt{open}'\ M\ R)\ \texttt{:-}\ \texttt{tm}'\ M\ ,\texttt{pi}\ f \setminus \texttt{pi}\ e \setminus \texttt{tm}'\ f \Rightarrow \texttt{tm}'\ e \Rightarrow \texttt{tm}'\ (R\ f\ e).$

A source term M is closed if and only if $(\texttt{tm}\ M)$ is derivable with an empty dynamic context in HH. Stated in the two-level logic approach, a source term M is closed if and only if $\{\texttt{tm}\ M\}$ is provable in Abella. Similarly, a target term M' is closed if and only if $\{\texttt{tm}'\ M'\}$ is provable. We can state the property that closed terms in the source language do not have free variables through the following formula in Abella:

$$\forall\ M, \nabla(x : \texttt{tm}), \{\texttt{tm}\ (M\ x)\} \supset \exists M', M = y \setminus M'.$$

In the two-level logic approach, the ∇ quantified variables are used to represent the free variables in the λProlog specifications. The expression $(M\ x)$ represents a term which may contain free occurrences of the variable x. The conclusion states that such a M must be an (meta-level) abstraction of the form $(y \setminus M')$ where M' does not contain free occurrences of y. This is because M' is quantified outside of y and hence cannot be instantiated with y. As a result $(M\ x)$ is equivalent to a term that does not contain free occurrences of x. This formula has a concise proof in Abella thanks to the HOAS representation of terms. A similar formula can be proved for closed terms identified by $\texttt{tm}'$. Once we have shown these two formulas to be theorems, we can use them to simplify dependencies variables and terms that are known to be closed.

Now we can formally define the logical relation for closure conversion. We identify the following predicate constants:

$$\texttt{sim} : \texttt{ty} \to \texttt{tm} \to \texttt{tm}' \to \texttt{prop} \qquad \texttt{equiv} : \texttt{ty} \to \texttt{tm} \to \texttt{tm}' \to \texttt{prop}$$

to respectively represent the simulation relation and the equivalence relation. They are defined by the following clauses:

$$\texttt{sim}\ T\ M\ M' \triangleq \forall V, \{\texttt{eval}\ M\ V\} \supset \exists V', \{\texttt{eval}'\ M'\ V'\} \wedge \texttt{equiv}\ T\ V\ V'$$

$$\texttt{equiv}\ \mathbb{N}\ (\texttt{nat}\ N)\ (\texttt{nat}'\ N) \triangleq \top$$

$$\begin{aligned}&\texttt{equiv}\ (\texttt{arr}\ T_1\ T_2)\ (\texttt{abs}\ R)\ (\texttt{clos}'\ M\ VE) \triangleq \\ &\quad \{\texttt{val}'\ (\texttt{clos}'\ M\ VE)\} \wedge \{\texttt{val}'\ (\texttt{clos}'\ M\ VE)\} \wedge \\ &\quad \{\texttt{tm}'\ (\texttt{clos}'\ M\ VE)\} \wedge \{\texttt{tm}\ (\texttt{abs}\ R)\} \wedge \\ &\quad \forall\ V_1\ V_1', \texttt{equiv}\ T_1\ V_1\ V_1' \supset \texttt{sim}\ T_2\ (R\ V_1)\ (\texttt{app}'\ M\ (\texttt{pair}'\ V_1'\ VE)).\end{aligned}$$

Note that because of the negative occurrence of `equiv` in the last clause, this definition violates the stratification constraints for inductive definitions in $\mathcal{G}$. However, it is a valid definition if we treat it as a *recursive definition* in the sense of [5]; in Section 5.3, we will discuss an extension to Abella that will support such definitions.

To formalize the notion of semantics preservation for open terms, we need a way to encode the substitutions explicitly and the equivalence relation between them. A substitution is represented as a list of pairs of variables and terms. We identify the types `smap` to classify the elements of substitutions in the source language. We overload the constant `map` (which we introduced in Section 3.2) with the type $\texttt{tm} \to \texttt{tm} \to \texttt{smap}$ as the constructor for `smap`. We then identify the types `smap_list` to classify the substitutions in the source language and overload the constants `nil` and `::` as constructors for `smap_list`. We also overload the predicate constant `member` for testing the membership for `smap_list`. Similarly, we introduce the type `cmap_list` to classify the substitutions in the target language and the type `cmap` to classify the elements in them. We overload the constants `map`, `nil` and `::` with appropriate types to form the substitutions in the target language, and overload `member` for test the membership for `cmap_list`.

Not all the the substitutions formed by using the constants above are considered well-formed; such substitutions must map variables to closed and unique values. To characterize variables at the specification level, which are represented as nominal constants at the proof level, we provide the following clause that defines the predicate constant $\texttt{name} : \texttt{tm} \to \texttt{prop}$:

$$\nabla x, \texttt{name}\ x \triangleq \top$$

By this definition, $(\texttt{name}\ M)$ holds if and only if M is a nominal constant of type `tm`. Thus `name` can be used to identify variables in the source language. We give a similar definition for the constant $\texttt{name}' : \texttt{tm}' \to \texttt{prop}$ to characterize variables in the target language.

We then use the predicate constant $\texttt{subst} : \texttt{tm\_list} \to \texttt{prop}$ to characterize well-formed substitutions. It is defined by the following clauses such that $(\texttt{subst}\ LM)$ holds if and only if LM represents a well-formed substitution.

$$\texttt{subst}\ \texttt{nil} \triangleq \top$$

$$\texttt{subst}\ ((\texttt{map}\ X\ V)\ ::\ LM) \triangleq \texttt{subst}\ LM \wedge \texttt{name}\ X \wedge \{\texttt{val}\ V\} \wedge \{\texttt{tm}\ V\} \wedge \forall V', \texttt{member}\ (\texttt{smap}\ X\ V')\ ML \supset V' = V.$$

Note that we use `name`, `val` and `tm` to capture the requirement that a well-formed substitution should map variables to closed values. The uniqueness of this mapping is captured by the last formula.

We use the predicate constant `app_subst` to represent the application of explicit substitutions. We define it by providing the following clauses such that $(\texttt{app\_subst}\ LM\ M\ M')$ holds if and only if M' is the result of replacing the occurrences of the free variables in M with the substitutions for these variables in LM.

$$\texttt{app\_subst}\ \texttt{nil}\ M\ M \triangleq \top$$

$$\nabla x, \texttt{app\_subst}\ ((\texttt{map}\ x\ V)\ ::\ (LM\ x))\ (R\ x)\ M \triangleq \nabla x, \texttt{app\_subst}\ (LM\ x)\ (R\ V)\ M.$$

Observe how the quantifier ordering is used in this definition to create a "hole" where a free variable appears in a term and (meta-level) application is then used to plug the hole with the substitution. Note that the possibility that a variable may occur multiple times in a substitution is captured by the fact that the tail of a non-empty substitution has the form $(LM\ x)$ where x is the variable in the head of this substitution. We can use a constant $\texttt{subst}'$ to characterize the well-formed substitutions in the target language by following a similar development.

Because the explicit substitutions are modeled by using the meta-level substitutions, their properties can be easily proved from the properties of the meta-level substitutions. The following are some examples of such properties stated as formulas that are provable in Abella:

$$\forall LM\ R\ T\ M', \texttt{app\_subst}\ LM\ (\texttt{abs}\ R)\ M' \supset \exists R', M' = \texttt{abs}\ R' \wedge \nabla x, \texttt{app\_subst}\ LM\ (R\ x)\ (R'\ x),$$

$$\forall LM\ M\ M', \{\texttt{tm}\ M\} \supset \texttt{app\_subst}\ LM\ M\ M' \supset M = M',$$

$$\forall LM\ M\ M'\ V, \nabla n, \{\texttt{tm}\ V\} \supset \texttt{app\_subst}\ LM\ (M\ n)\ (M'\ n) \supset \texttt{app\_subst}\ ((\texttt{map}\ n\ V)\ ::\ LM)\ (M\ n)\ (M'\ V),$$

The first formula states that substitutions distribute over abstractions. The second formula states that substitutions have no effect on closed terms. The last formula states that an explicit substitution has the same effect as a meta-level substitution if the substitution value is closed. All three formulas can be proved by induction on the `app_subst` assumption. The proofs for the last two formulas make critical use of the property about `tm` that enables us to simplify dependencies between closed terms and free variables.

We translate the equivalence relation between substitutions into fixed-point definitions in a straightforward manner. We first encode the equivalence relation between source substitutions and target environments by defining the predicate constant

$$\texttt{subst\_env\_equiv} : \texttt{olist} \to \texttt{smap\_list} \to \texttt{tm}' \to \texttt{prop}$$

as follows:

$$\begin{aligned}
&\texttt{subst_env_equiv}\ \texttt{nil}\ LM\ \texttt{unit}' \triangleq \top \\
&\texttt{subst_env_equiv}\ (\texttt{of}\ X\ T :: L)\ LM\ (\texttt{pair}'\ V'\ VE) \triangleq \\
&\quad \exists V, \texttt{subst_env_equiv}\ L\ LM\ VE \wedge \\
&\quad\quad \texttt{member}\ (\texttt{map}\ X\ V)\ LM \wedge \texttt{equiv}\ T\ V\ V'.
\end{aligned}$$

We then encode the equivalence relation between substitutions by defining the predicate constant

$$\texttt{subst\_equiv} : \texttt{olist} \to \texttt{smap\_list} \to \texttt{cmap\_list} \to \texttt{prop}$$

as follows:

$$\begin{aligned}
&\nabla e, \texttt{subst_equiv}\ L\ LM\ ((\texttt{map}\ e\ VE) :: \texttt{nil}) \triangleq \texttt{subst_env_equiv}\ L\ LM\ VE \\
&\nabla x\ y, \texttt{subst_equiv}\ (\texttt{of}\ x\ T :: L)\ ((\texttt{map}\ x\ V) :: LM)\ ((\texttt{map}\ y\ V')\ LM') \triangleq \\
&\quad \texttt{equiv}\ T\ V\ V' \wedge \texttt{subst_equiv}\ L\ LM\ LM'.
\end{aligned}$$

Finally, we can formalize the semantics preservation theorem as follows:

$$\begin{aligned}
&\forall L\ LM\ LM'\ Vs\ Vs'\ Map\ T\ P\ P'\ M\ M', \\
&\quad \texttt{ctx}\ L \supset \texttt{subst}\ LM \supset \texttt{subst}'\ LM' \supset \\
&\quad \texttt{subst_equiv}\ L\ LM\ LM' \supset \texttt{vars_of_ctx}\ L\ Vs \supset \\
&\quad \texttt{vars_of_subst}'\ LM'\ Vs' \supset \texttt{tomapping}\ Vs\ Vs'\ Map \supset \\
&\quad \{L \vdash \texttt{of}\ M\ T\} \supset \{\texttt{cc}\ Map\ Vs\ M\ M'\} \supset \\
&\quad \texttt{app_subst}\ LM\ M\ P \supset \texttt{app_subst}'\ LM'\ M'\ P' \supset \texttt{sim}\ T\ P\ P'.
\end{aligned}$$

Three new predicates are used in this statement. The predicate `ctx` identifies well-formed typing contexts. The judgment ($\texttt{vars\_of\_subst}'\ ML\ Vs'$) "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 $Vs' = (y_1, ..., y_n, x_e)$, the predicate `tomapping` creates in Map the mapping

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

The proof of this theorem is by induction on $\{\texttt{cc}\ Map\ Vs\ M\ M'\}$, the closure conversion derivation, and follows closely the structure of the informal development we outlined in Section 4.2.1. In this formal proof, the uses of properties

about substitutions must be made explicit. They are represented by the applications of the above lemmas about explicit substitutions.

From this theorem, we can easily derive the following corollary about the preservation of evaluations for closed programs at atomic types:

$$\forall\ M\ M'\ N, \{\texttt{of}\ M\ \texttt{nat}\} \supset \{\texttt{cc}'\ M\ M'\} \supset \\ \{\texttt{eval}\ M\ (\texttt{nat}\ N)\} \supset \{\texttt{eval}'\ M'\ (\texttt{nat}'\ N)\}.$$

4.3 Verification using the Simulation Based Approach

In this section, we use closure conversion as an example to demonstrate how to verify functional transformations using the approach that we have earlier labeled as the simulation based approach. We first describe the informal proof and then its formalization in Abella.

4.3.1 The Informal Verification of Closure Conversion

Our goal is to prove that the forward simulation property holds for closure conversion. As we have discussed in Section 4.1.2, this can be reduced to showing that the result of first evaluating a source term to a value and then transforming the value by closure conversion is the same as that of first transforming it into a target term and then evaluating that term. Such a property is not provable if we use the evaluation semantics for the source language of closure conversion as described in Section 3.1 that performs substitutions when evaluating applications. This is because substitutions may eliminate free occurrences of variables under abstractions. Thus, closure conversion on the result of evaluating a term t may generate closures whose environments are smaller than that of their counterparts in the result of closure conversion on t. For example, consider a term t of the form $((\lambda y.(\lambda x.t_1))\ t_2)$, in which the abstraction $(\lambda x.t_1)$ may refer to y as a free variable. Evaluation of t using the evaluation semantics in Section 3.1 will substitute t_2 for the free occurrences of y in $(\lambda x.t_1)$, resulting in $(\lambda x.t_1[t_2/y])$ which no longer has y as a free variable. Thus, closure conversion on this term will generate a closure whose environment is smaller than the environment of the closure for $(\lambda x.t_1)$ in the result of transforming t by closure conversion.

To solve this problem, we adopt an evaluation semantics for the source language that maintains an explicit environment during evaluation. This environment records the bindings for the free variables in the term that is being evaluated. When the application of a function to an argument is encountered, instead of substituting the argument for the binder of the function, the evaluation adds the argument to the environment. When an abstraction is encountered, the evaluation produces a closure at the source language level that consists of the abstraction itself and the current environment. With this evaluation smenatics, occurrences of free variables under abstractions are not affected by evaluations, since the substitutions for the free variables are "suspended" in the environment at the boundaries of abstractions. This treatment parallels what would happen when we closure convert the term first and then evaluate the target

$$\frac{}{\langle\langle n, E\rangle\rangle \hookrightarrow_e n} \qquad \frac{}{\langle\langle x, E\rangle\rangle \hookrightarrow_e E(x)} \qquad \frac{}{\langle\langle \lambda x.M, E\rangle\rangle \hookrightarrow_e \langle \lambda x.M, E\rangle}$$

$$\frac{\langle\langle M_1, E\rangle\rangle \hookrightarrow_e \langle \lambda x.M_3, E'\rangle \quad \langle\langle M_2, E\rangle\rangle \hookrightarrow_e V_2 \quad \langle\langle M_3, (E', x \to V_2)\rangle\rangle \hookrightarrow_e V}{\langle\langle M_1\ M_2, E\rangle\rangle \hookrightarrow_e V}$$

Figure 7: Evaluation with Explicit Environments for the Source Language

language term: the transformation will replace an underlying abstraction with a closure and evaluation will populate the associated binding with actual terms. Thus, it becomes possible to prove that evaluation and closure conversion can be performed in any order.

We now present the evaluation semantics with explicit environments for the source language. We consider only evaluating whole programs each of which consists of a possibly open term M (as described in Figure 2a) and an environment E that binds the free variables in M; such a program is denoted by $\langle\langle M, E\rangle\rangle$. When M is an abstraction, a whole program $\langle\langle M, E\rangle\rangle$ should evaluate to a closure at the source language level which consists of M and E. We write $\langle M, E\rangle$ to denote such a closure. We shall drop the "source language level" qualification for closures when it can be inferred from the context. The values, denoted by V, for this evaluation semantics are either natural numbers or closures. Environments are mappings from variables to values of the form $(x_1 \to V_1, \ldots, x_n \to V_n)$. The evaluation judgment is written as $\langle\langle M, E\rangle\rangle \hookrightarrow_e V$. This judgment asserts that the term M in the environment E evaluates to the value V. The rules for deriving this judgment is shown in Figure 7.

In this new evaluation semantics, closure conversion works on whole programs and values. To state the property that evaluation and closure conversion commute with each other, we need to first define such closure conversion. The relation of closure conversion on values is written in the form $(V \rightsquigarrow_V V')$ where V is a value for the evaluation semantics with explicit environments and V' is a value in the target language (as shown in Figure 3a). It is defined by the following rules:

$$\frac{}{n \rightsquigarrow_V n}$$

$$\frac{V_i \rightsquigarrow_V V_i' \text{ (for } 1 \leq i \leq m) \quad \rho \triangleright \lambda x.M \rightsquigarrow M'}{\langle \lambda x.M, E\rangle \rightsquigarrow_V M'}$$

where $E = (x_1 \to V_1, ..., x_m \to V_m)$ and $\rho = (x_1 \to V_1', ..., x_m \to V_m')$

Note that the second rule make use of closure conversion in Figure 4 to transform closures in the source language into closures in the target language. The relation of closure conversion for whole programs is written in the form $\langle\langle M, E\rangle\rangle \rightsquigarrow_S M'$ where M' is a traget term (shown in Figure 3a). It is defined by the rule below:

$$\frac{(x_1, ..., x_n) \supseteq \mathbf{fvars}(\lambda x_l.M) \quad V_l \rightsquigarrow_V V_l' \quad E(x_i) \rightsquigarrow_V V_i' \text{ (for } 1 \leq i \leq n) \quad \rho' \triangleright M \rightsquigarrow M'}{\langle\langle M, (E, x_l \to V_l)\rangle\rangle \rightsquigarrow_S M'[E'/e][V_l'/x_l]}$$

where $\rho' = (x_l \mapsto x_l, x_1 \mapsto \pi_1(e), ..., x_n \mapsto \pi_n(e))$ and $E' = (V_1', ..., V_n')$

This rule can only be applied to objects of the form $\langle\langle M, (E, x_l \to V_l)\rangle\rangle$ where M is intended to be the body of an abstraction, x_l the binder of the abstraction, and E the environment that binds all the free variables of the abstraction. The substitutions "suspended" by the explicit environment are performed after the closure conversion. Note that because the closure conversion has already made all abstractions closed, such substitutions will not affect the bindings for free variables in abstractions.

The forward simulation theorem for closure conversion is stated as follows:

Theorem 2 *If* $\langle\langle M, (E, x_l \to V_l)\rangle\rangle \hookrightarrow_e V$ *and* $\langle\langle M, (E, x_l \to V_l)\rangle\rangle \rightsquigarrow_S M'$, *then there exists a value* V' *s.t.* $V \rightsquigarrow_V V'$ *and* $M' \hookrightarrow V'$.

This theorem is proved by induction on the source evaluation. The proof for it is much simpler than the proof using logical relations because we do not need to prove equivalence between functions or open terms. Still, it involves manipulation and reasoning about bindings such as that about the explicit environments.

4.3.2 The Formal Verification of Closure Conversion

We first need to encode terms that are bound by explicit environments. We identify the following constants:

$$\texttt{envtm} : \texttt{tm} \to \texttt{tm\_list} \to \texttt{tm} \qquad \texttt{ebase} : \texttt{tm} \to \texttt{tm} \qquad \texttt{eabs} : (\texttt{tm} \to \texttt{tm}) \to \texttt{tm}.$$

Again, we use a higher-order representation for the bindings of free variables. A whole program $\langle\langle M, E\rangle\rangle$ in which the environment $E = (x_1 \to V_1, ..., x_n \to V_n)$ is encoded as

$$\texttt{envtm}\ (\texttt{eabs}\ (x_1 \setminus \ldots \texttt{eabs}\ (x_n \setminus \texttt{ebase}\ M)))\ (V_1\ \texttt{::}\ \ldots\ \texttt{::}\ V_n\ \texttt{::}\ \texttt{nil})$$

We use the same constructors to encode closures in the source language. The exact role of the encoded term is inferred from the context. The evaluation relation $\hookrightarrow_e$ is represented by the predicate constant $\texttt{eeval} : \texttt{tm} \to \texttt{tm} \to \texttt{o}$. The rules in Figure 7 are translated into the the following clauses such that $(\texttt{eeval}\ (\texttt{envtm}\ M\ E)\ V)$ holds if and only if M evaluates to V in the environment E.

$$\begin{aligned}
&\texttt{eeval}\ (\texttt{envtm}\ M\ E)\ V\ \texttt{:-}\ \texttt{eevalnat}\ M\ V.\\
&\texttt{eeval}\ EM\ V\ \texttt{:-}\ \texttt{eevalvar}\ EM\ V.\\
&\texttt{eeval}\ (\texttt{envtm}\ M\ E)\ (\texttt{envtm}\ M\ E)\ \texttt{:-}\ \texttt{isabs}\ M.\\
&\texttt{eeval}\ (\texttt{envtm}\ M\ E)\ V\ \texttt{:-}\ \texttt{isapp}\ M\ M_1\ M_2\,,\\
&\qquad \texttt{eeval}\ (\texttt{envtm}\ M_1\ E)\ (\texttt{envtm}\ V_1\ E_1)\,, \texttt{eeval}\ (\texttt{envtm}\ M_2\ E)\ V_2\,,\\
&\qquad \texttt{abs_to_eabs}\ V_1\ V_1'\,, \texttt{snoc}\ V_2\ E_1\ E_1'\,, \texttt{eeval}\ (\texttt{envtm}\ V_1'\ E_1')\ V.
\end{aligned}$$

The predicate constant $\texttt{snoc} : \texttt{tm} \to \texttt{tm\_list} \to \texttt{tm\_list} \to \texttt{o}$ represents a relation on lists such that $(\texttt{snoc}\ V\ L\ L')$ holds if and only if L' is a list obtained by appending the element V to the list L. Note that to apply the correct clause

to evaluate ($\texttt{envtm}\ M\ E$), we must know what term M represents; for this we need to go under the binders for free variables in M. We use the predicate constants

$$\texttt{eevalnat} : \texttt{tm} \to \texttt{tm} \to \texttt{o} \quad \texttt{isabs} : \texttt{tm} \to \texttt{o}$$
$$\texttt{eevalvar} : \texttt{tm} \to \texttt{tm} \to \texttt{o} \quad \texttt{isapp} : \texttt{tm} \to \texttt{tm} \to \texttt{tm} \to \texttt{o}$$

to identify the terms under the nested binders. They are defined as follows:

$\texttt{eevalnat}\ (\texttt{ebase}\ (\texttt{nat}\ N))\ (\texttt{nat}\ N).$
$\texttt{eevalnat}\ (\texttt{eabs}\ R)\ V\ \texttt{:-}\ \texttt{pi}\ x \setminus \texttt{eevalnat}\ (R\ x)\ V.$
$\texttt{eevalvar}\ (\texttt{envtm}\ (\texttt{ebase}\ X)\ \texttt{nil})\ V\ \texttt{:-}\ \texttt{eevalvar}\ X\ V.$
$\texttt{eevalvar}\ (\texttt{envtm}\ (\texttt{eabs}\ R)\ (\texttt{snoc}\ V\ V_L))\ V'\ \texttt{:-}$
$\quad \texttt{pi}\ x \setminus \texttt{eevalvar}\ x\ V \Rightarrow \texttt{eevalvar}\ (\texttt{envtm}\ (R\ x)\ V_L)\ V'.$
$\texttt{isabs}\ (\texttt{ebase}\ (\texttt{abs}\ R)).$
$\texttt{isabs}\ (\texttt{eabs}\ R)\ \texttt{:-}\ \texttt{pi}\ x \setminus \texttt{isabs}\ (R\ x).$
$\texttt{isapp}\ (\texttt{ebase}\ (\texttt{app}\ M_1\ M_2))\ (\texttt{ebase}\ M_1)\ (\texttt{ebase}\ M_2).$
$\texttt{isapp}\ (\texttt{eabs}\ M)\ (\texttt{eabs}\ M_1)\ (\texttt{eabs}\ M_2)\ \texttt{:-}$
$\quad \texttt{pi}\ x \setminus \texttt{isapp}\ (M\ x)\ (M_1\ x)\ (M_2\ x).$

Among the clauses that represent the evaluation rules, the only interesting one is that for evaluating applications. Given a term M bounded by its environment E, this rule uses `isapp` to test if M is indeed an application. If M is an application of the form $(M_1\ M_2)$, it requires M_1 to evaluate to a closure $(\texttt{envtm}\ V_1\ E_1)$ and M_2 to evaluate to V_2 in the environment E. Then it uses the predicate constant $\texttt{abs\_to\_eabs} : \texttt{tm} \to \texttt{tm} \to \texttt{o}$ which is defined by the following clauses:

$\texttt{abs\_to\_eabs}\ (\texttt{ebase}\ (\texttt{abs}\ R))\ (\texttt{eabs}(x \setminus \texttt{ebase}\ (R\ x))).$
$\texttt{abs\_to\_eabs}\ (\texttt{eabs}\ M)\ (\texttt{eabs}\ M')\ \texttt{:-}\ \texttt{pi}\ x \setminus \texttt{abs\_to\_eabs}\ (M\ x)\ (M'\ x).$

to translate V_1, which should represent an abstraction and has the form

$$\texttt{eabs}\ (x_1 \setminus \ldots \texttt{eabs}\ (x_n \setminus \texttt{ebase}\ (\texttt{abs}\ (x \setminus M))))$$

into V_1 which represents the body of the abstraction and has the form

$$\texttt{eabs}\ (x_1 \setminus \ldots \texttt{eabs}\ (x_n \setminus (\texttt{eabs}\ (x \setminus \texttt{ebase}\ M)))).$$

It then use `snoc` to extend the environment E_1 with the argument V_1 to form E_1'. The final result is obtained by evaluating the abstraction body represented by V_1' in the extended environment E_1'.

We identify the predicate constants

$$\texttt{val\_cc} : \texttt{tm} \to \texttt{tm} \to \texttt{o} \qquad \texttt{sclos\_cc} : \texttt{map\_list} \to \texttt{tm\_list} \to \texttt{tm} \to \texttt{tm}' \to \texttt{o}$$

to encode closure conversion on values. They are defined as follows:

$$\texttt{val\_cc}\ (\texttt{nat}\ N)\ (\texttt{nat}'\ N).$$
$$\texttt{val\_cc}\ M\ M'\ \texttt{:-}\ \texttt{sclos\_cc}\ \texttt{nil}\ \texttt{nil}\ M\ M'.$$
$$\texttt{sclos\_cc}\ \mathit{Map}\ \mathit{FVs}\ (\texttt{envtm}\ (\texttt{ebase}\ M)\ \texttt{nil})\ M'\ \texttt{:-}\ \texttt{cc}\ \mathit{Map}\ \mathit{FVs}\ M\ M'.$$
$$\texttt{sclos\_cc}\ \mathit{Map}\ \mathit{FVs}\ (\texttt{envtm}\ (\texttt{eabs}\ R)\ (V\ E))\ M'\ \texttt{:-}\ \texttt{val\_cc}\ V\ V',$$
$$\quad \texttt{pi}\ x \setminus \texttt{sclos\_cc}\ (\texttt{map}\ x\ V'\ \texttt{::}\ \mathit{Map})\ (x\ \texttt{::}\ \mathit{FVs})\ (\texttt{envtm}\ (R\ x)\ E)\ M'.$$

Note that to transform a closure represented by $(\texttt{envtm}\ M\ E)$, `sclos_cc` iterates on M and E, transforms the values in E recursively, builds up a mapping *Map* for the free variables in the term represented by M and use `cc` to transform the body of M under *Map* to get the final result.

To encode the closure conversion for terms bound by explicit environments, we identify predicate constant

$$\texttt{envtm\_cc} : \texttt{map\_list} \to \texttt{tm\_list} \to \texttt{tm} \to \texttt{tm}' \to \texttt{o}$$

which is defined by the clauses below:

$$\texttt{envtm\_cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{envtm}\ (\texttt{eabs}\ x \setminus \texttt{ebase}\ (M\ x))\ (V\ \texttt{::}\ \texttt{nil}))\ (M'\ E\ V')\ \texttt{:-}$$
$$\quad \texttt{cc}\ \texttt{nil}\ \texttt{nil}\ V\ V',$$
$$\quad \texttt{fvars}\ (\texttt{abs}\ M)\ \mathit{Vs}\ \mathit{FVs}, \texttt{mapenv}\ \mathit{FVs}\ \mathit{Map}\ E, \texttt{mapvar}\ \mathit{FVs}\ \mathit{NMap},$$
$$\quad \texttt{pi}\ x \setminus \texttt{pi}\ y \setminus \texttt{pi}\ e \setminus \texttt{cc}\ ((\texttt{map}\ x\ y)\ \texttt{::}\ (\mathit{NMap}\ e))\ (x\ \texttt{::}\ \mathit{FVs})\ (M\ x)\ (M'\ y\ e).$$
$$\texttt{envtm\_cc}\ \mathit{Map}\ \mathit{FVs}\ (\texttt{envtm}\ (\texttt{eabs}\ x \setminus (\texttt{eabs}\ y \setminus R\ x\ y))\ (V\ \texttt{::}\ E))\ M'\ \texttt{:-}$$
$$\quad \texttt{cc}\ \texttt{nil}\ \texttt{nil}\ V\ V', \texttt{pi}\ x \setminus$$
$$\quad \texttt{envtm\_cc}\ ((\texttt{map}\ x\ V')\ \texttt{::}\ \mathit{Map})\ (x\ \texttt{::}\ \mathit{FVs})\ (\texttt{envtm}\ (\texttt{eabs}\ y \setminus R\ x\ y)\ E)\ M'.$$

The transformation of a term M together with the environment $(E, x_l \to V_l)$ that binds the free variables in M proceeds as follows. If E is not empty, the second clause iterates on E, transforms the values in E to build up a mapping *Map* for the free variables in $\lambda x_l.M$. When E is empty, the term M is expected to be the body of an abstraction and the first clause transforms it accordingly. By this definition, $\langle\langle M, E\rangle\rangle \rightsquigarrow_S P$ if and only if

$$\texttt{envtm\_cc}\ \texttt{nil}\ \texttt{nil}\ (\texttt{envtm}\ M'\ E')\ P'$$

holds where M', E' and P' is respectively encoded from M, E and P.

The forward simulation theorem for closure conversion is stated as follows:

$$\forall M\ V_l\ E\ E', \{\texttt{eeval}\ (\texttt{envtm}\ M\ E)\ V\} \supset \{\texttt{envtm\_cc}\ (\texttt{envtm}\ M\ E)\ M'\} \supset$$
$$\quad \exists V', \{\texttt{eval}'\ M'\ V'\} \wedge \{\texttt{val\_cc}\ V\ V'\}.$$

It is proved by induction on the source evaluation $\{\texttt{eeval}\ (\texttt{envtm}\ M\ E)\ V\}$.

4.4 Proposed Work Relating to Verification

To demonstrate the effectiveness of our approach for verifying functional transformations, we will verify the λProlog implementation of the compilation passes

in the compiler for the PCF-style language that involve manipulation of objects with bindings. Such compilation passes include the CPS transformation, closure conversion and code hoisting. We will prove that they preserve semantics in Abella using the weaker form of program equivalence as well as the one based on logical relations. The PCF-style language supports recursion and, in this case, a simple induction on types does not suffice for a logical relations style definition of type program equivalence. To deal with this issue, we will use logical relations that are indexed by the number of steps in the evaluation. We will investigate more thoroughly the advantages and disadvantages of using logical relations and the weaker form of semantics preservation. Based on this investigation, we will select one approach and use it to verify the full compiler for the PCF-style language.

Depending on how things evolve, we will consider the verification of transformations on richer functional languages, such as a subset of ML, using our approach. For example, we may consider verifying closure conversion on a polymorphic language. Based on these exercises, we hope to demonstrate that that our approach scales to more complicated tasks for verifying functional transformations.

5 Extensions to the Verification Framework

In the two previous sections we have presented out plans for showing that λProlog and Abella provide a strong framework for implementing and verifying functional transformations. The specification language λProlog has been worked on for many years and is at a reasonably mature state of development to be used unchanged in out work. However, the Abella system is newer and is still evolving. In our thesis, we will extend Abella with new features that make it a more flexible verification tool and that also enable or simplify certain tasks in our work on verifying functional transformations.

The extensions we have developed or will develop consist of three parts. The first part is a treatment of high-order relational specifications. Previously, the two-level logic approach in Abella only supported reasoning over a subset of HH. We have given a full encoding of HH in $\mathcal{G}$ and developed a methodology to reason about HH specifications through this encoding. This methodology allows us to reason about λProlog specifications that introduce higher-order formulas dynamically in the derivations. We have used this extension already in several of the proofs that we have carried out related to functional transformations.

The second extension we will work on is to provide support of polymorphism in Abella. The current proof theory of $\mathcal{G}$ is based on STLC which does not include polymorphic types. To encode general data structures such as lists in Abella, we have to define a version of them for every concrete type. Furthermore, we have to prove their properties for every concrete type, even through these proofs are mostly replications of a general pattern under different type instantiations. This is a recurring theme that we have witnessed when we talked about the implementation and verification of functional transformations. We

will develop a schematic treatment of polymorphism in Abella. With this kind of polymorphism, we will be able to develop libraries to support general data structures and to get rid of many duplicated definitions and proofs in Abella.

The last part is the support of recursive definitions. As we have described in Section 4.2.2, we need to define logical relations as *recursive definitions* which are different from fixed-point definitions in $\mathcal{G}$. A treatment of recursive definitions has been developed in [5]. We will implement this treatment in Abella so that we can use the extended form of definitions without compromising soundness.

In the following subsections, we elaborate on these extensions and our plans for work on them in this thesis.

5.1 Treating Higher-Order Relational Specifications

In the two-level logic approach, the specification logic HH is encoded as a fixed-point definition in $\mathcal{G}$ and the specifications in this logic are reasoned about through this encoding. Specifically, an HH sequent $\Sigma; \Theta; \Gamma \vdash G$ is encoded as $(\texttt{seq}\ \Gamma\ G)$ where Θ consists of the static program clauses and Γ is the dynamic context that contains the clauses dynamically added during the derivation. When analyzing a derivation for the sequent encoded by $(\texttt{seq}\ \Gamma\ G)$, we need to consider the possibility that G is proved by backchaining on clauses in Γ. In HH, such a clause can be an arbitrary complicated HH program clause. For example, given $p, q : i \to \texttt{o}$ for some type i and the following clause:

$$p\ X\ \texttt{:-}\ (\Pi x.q\ x \Rightarrow p\ x) \Rightarrow p\ X.$$

Backchaining $(p\ X)$ will introduce the $(\Pi x.q\ x \Rightarrow p\ x)$ to the dynamic context.

To avoid the complexity of reasoning about extensions of arbitrarily complicated program clauses, the previous definition of `seq` is restricted to encode a subset of HH that allows only atomic formulas in dynamic contexts. In this restricted setting, a goal G can be derived from the dynamic context Γ only if it is a member of Γ. As a result, the previous clause will not be accepted as a valid specification.

This restriction on the encoding of HH prevents us from using natural encodings of many relational specifications involving higher-order terms in Abella. We use the transformation of STLC terms into the Debruijn form as an example to illustrate this point. In the Debruijn form, a bound variable is represented by a natural number representing the number of abstractions it must go upwards to reach its binding site. The rules for transforming STLC terms to the Debruijn forms are shown in Figure 8. A judgment $\Gamma \vdash M \equiv_h E$ asserts that M is translated into its Debruijn form E where h is the depth of M (*i.e.*, the number of abstractions above M) and Γ assigns depths to binding variables. As an example, the closed term $(\lambda x.(\lambda y.x\ y)\ x)$ is translated into $(\lambda\ (\lambda\ 2\ 1)\ 1)$ since $(\lambda x.(\lambda y.x\ y)\ x \equiv_0 (\lambda\ (\lambda\ 2\ 1)\ 1))$ holds.

We show how to encode the rules in Figure 8 in HH. We use the encoding of STLC syntax as described in Section 2.1. Given a type `nat` for natural numbers

$$\frac{\Gamma \vdash M_1 \equiv_h E_1 \quad \Gamma \vdash M_2 \equiv_h E_2}{\Gamma \vdash M_1\ M_2 \equiv_h E_1\ E_2}$$

$$\frac{\Gamma, x : h \vdash M \equiv_{h+1} E}{\Gamma \vdash \lambda x.M \equiv_h \lambda E} \qquad \frac{x : h \in \Gamma \quad h + k = i}{\Gamma \vdash x \equiv_i k}$$

Figure 8: Translating λ-terms into Debruijn forms

and `dtm` for Debruijn terms, we introduce the constant: `dapp` : `dtm` $\rightarrow$ `dtm` $\rightarrow$ `dtm` to represent applications of Debruijn terms, `dabs` : `dtm` $\rightarrow$ `dtm` to represent abstractions of Debruijn terms and `dvar` : `nat` $\rightarrow$ `dtm` to represent the Debruijn indexes. Note that the rule in Figure 8 for translating variables into Debruijn indexes can be described as a property for x, *i.e.*, $\forall i\ k.(h + k = i) \supset x \equiv_i k$. This property can be encoded in HH by using universal and hypothetical goals. The rules in Figure 8 are encoded as the following clauses:

$$\texttt{hodb}\ (\texttt{app}\ M_1\ M_2)\ H\ (\texttt{dapp}\ E_1\ E_2)\ \texttt{:-}\ \texttt{hodb}\ M_1\ H\ E_1\ \ \&\ \ \texttt{hodb}\ M_2\ H\ E_2.$$

$$\texttt{hodb}\ (\texttt{abs}\ M)\ H\ (\texttt{dabs}\ D)\ \texttt{:-}$$

$$\Pi x.(\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k)) \Rightarrow \texttt{hodb}\ (M\ x)\ (\texttt{s}\ H)\ D$$

where `add` defines the conventional ternary addition relations between natural numbers. The sub-clause $(\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k))$ describes the rule for translating variables into Debruijn terms. Formulas of this form may be added to the dynamic context during the derivation. This specification will not be accepted by Abella that use the restricted subset of HH.

It is possible to finitely reason about HH relational specifications that introduce higher-order formulas during the derivation. The critical observation is that starting with a finite set of program clauses, clauses that are dynamically introduced must be some sub-formulas of the initial clauses, which can only have finite forms. For instance, the dynamic clauses introduced by the program for `hodb` can only have the form $(\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k))$. By characterizing the dynamic contexts as inductive definitions, we will be able to reason about higher-order relational specifications.

We present a methodology to reason about specifications written in the full HH by encoding a *focused* proof system of HH and using fixed-point definitions to finitely characterize the dynamic contexts. The focused proof system we shall use to encode HH is a slight variant of Figure 1 such that the *backchain* and *succeed* rules are broken into separate rules for analyzing conjunction, forall and implication formulas. In this system, there are two kinds of sequents, the asynchronous sequents of the form $\Sigma; \Theta; \Gamma \vdash G$ and the synchronous (or focused) sequents of the form $\Sigma; \Theta; \Gamma, [F] \vdash A$. To derive an asynchronous sequent $\Sigma; \Theta; \Gamma \vdash G$, we repeatedly reduce the goal until it becomes atomic. At that point, we pick one formula F from Θ or Γ and derive the focused sequent $\Sigma; \Theta; \Gamma, [F] \vdash A$. The derivation proceeds by applying the asynchronous rules

to simplify the focused formula F until the derivation goes back to some asynchronous sequents or the proof is finished.

We identify the predicate constants $\texttt{seq} : \texttt{olist} \to \texttt{o} \to \texttt{prop}$ and $\texttt{bch} : \texttt{olist} \to \texttt{o} \to \texttt{o} \to \texttt{prop}$. An asynchronous sequent $\Sigma; \Theta; \Gamma \vdash G$ is represented by the formula $(\texttt{seq}\ \Gamma\ G)$. An asynchronous sequent $\Sigma; \Theta; \Gamma, [F] \vdash A$ is represented by the formula $(\texttt{bch}\ \Gamma\ F\ A)$. The derivation rules for these sequents are encoded as the fixed-point definition in Figure 9. Here the predicate constant $\texttt{atomic} : \texttt{o} \to \texttt{prop}$ is used to identify atomic formulas. The static program clauses are identified by $\texttt{prog} : \texttt{o} \to \texttt{prop}$ such that $(\texttt{prog}\ D)$ holds if and only if D is a clause in the initial set of program clauses. The first three clauses encode the rules for reducing the asynchronous sequents. The next two clauses encodes the transition from the asynchronous phase into the synchronous phase. The rules for bch captures the rules for deriving asynchronous sequents. Together they define the *backchain* and *succeed* rules in Figure 1.

$$
\begin{array}{ll}
\texttt{seq}\ L\ (G_1\ \&\ G_2) & \triangleq \texttt{seq}\ L\ G_1 \wedge \texttt{seq}\ L\ G_2 \\
\texttt{seq}\ L\ (F \Rightarrow G) & \triangleq \texttt{seq}\ (F \texttt{cons} L)\ G \\
\texttt{seq}\ L\ (\Pi_\tau\ G) & \triangleq \nabla x{:}\tau, \texttt{seq}\ L\ (G\ x) \\
\texttt{seq}\ L\ A & \triangleq \texttt{atomic}\ A \wedge \texttt{member}\ F\ L \wedge \texttt{bch}\ L\ F\ A \\
\texttt{seq}\ L\ A & \triangleq \texttt{atomic}\ A \wedge \texttt{prog}\ F \wedge \texttt{bch}\ L\ F\ A \\
\texttt{bch}\ L\ (F_1\ \&\ F_2)\ A & \triangleq \texttt{bch}\ L\ F_1\ A \vee \texttt{bch}\ L\ F_2\ A \\
\texttt{bch}\ L\ (G \Rightarrow F)\ A & \triangleq \texttt{seq}\ L\ G \wedge \texttt{bch}\ L\ F\ A \\
\texttt{bch}\ L\ (\Pi_\tau\ F)\ A & \triangleq \exists t{:}\tau, \texttt{bch}\ L\ (F\ t)\ A \\
\texttt{bch}\ L\ A\ A & \triangleq \top
\end{array}
$$

Figure 9: Encoding of a Focused Version of HH in $\mathcal{G}$

In Abella, we write $\{\Gamma \vdash G\}$ for $(\texttt{seq}\ \Gamma\ G)$ and $\{\Gamma, [F] \vdash A\}$ for $(\texttt{bch}\ \Gamma\ F\ A)$. The syntax for sequents in different systems is summarized as follows:

HH	$\mathcal{G}$	Abella
$\Sigma; \Theta; \Gamma \vdash G$	$\texttt{seq}\ \Gamma\ G$	$\{\Gamma \vdash G\}$
$\Sigma; \Theta; \Gamma, [F] \vdash A$	$\texttt{bch}\ \Gamma\ F\ A$	$\{\Gamma, [F] \vdash A\}$

Given this encoding of HH, we will be able to reason about higher-order relational specifications. For example, suppose we would like to prove that the translation from λ-terms to Debruijn forms is deterministic. We first identify the predicate constant $\texttt{ctx} : \texttt{o} \to \texttt{prop}$ to characterized the dynamic context, which is defined by the clauses below:

$$\texttt{ctx}\ \texttt{nil} \triangleq \top$$

$$\nabla x.\texttt{ctx}\ ((\Pi i\ k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k))\texttt{cons} L) \triangleq \texttt{ctx}\ L.$$

We need to prove the following lemma about `ctx`:

$$\forall L\ E, \texttt{ctx}\ L \supset \texttt{member}\ E\ L \supset$$
$$\exists x\ H, E = (\Pi i\ k, \texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k)) \wedge \texttt{name}\ x.$$

which identifies the possible forms of formulas in the dynamic context, and another lemma

$$\forall L\ x\ H_1\ H_2, \texttt{ctx}\ L \supset$$
$$\texttt{member}\ (\Pi i\ k, \texttt{add}\ H_1\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k))L \supset$$
$$\texttt{member}\ (\Pi i\ k, \texttt{add}\ H_2\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k))L \supset H_1 = H_2.$$

which states that there is an unique formula for any variable x in the context. The determinacy of the translation into the Debruijn forms can be stated as follows:

$$\forall L\ M\ H\ E_1\ E_2, \texttt{ctx}\ L \supset$$
$$\{L \vdash \texttt{hodb}\ M\ H\ E_1\} \supset \{L \vdash \texttt{hodb}\ M\ H\ E_2\} \supset E_1 = E_2.$$

The proof proceeds by induction on the first assumption. The lemmas about `ctx` will be used to prove the determinancy when M is a variable and must be backchained on some clause in L.

In summary, the combination of an encoding of a focused version of HH and the ability to finitely characterizing dynamic contexts enables inductive proofs for specifications in the full HH. The methodology described here has already been implemented in Abella 2.0. In this section, we have shown one example where the higher-order relational specifications can be elegantly encoded in λProlog and reasoned about in Abella. Another example is that the type preservation for the CPS transformation can be significantly simplified by using higher-order typing contexts. Further examples can be found in [51].

5.2 Supporting Polymorphism in Abella

When specifying and reasoning about computational systems, we often need to define generic data structures such as lists and sets and prove properties about them. Because $\mathcal{G}$ is based on STLC, data structures containing different types of objects need to be defined individually. We have witnessed such phenomenon: when presenting the work on verified transformations, we defined several versions of lists that contain different kinds of elements. Moreover, the operations on different data structures must be defined and their properties must be proved individually, although many of them have exactly the same structure. For example, the append relation on HH formulas can be defined for the predicate `append_ol` : `olist` → `olist` → `olist` → `prop` whose definition is as follows:

$$\texttt{append\_ol}\ \texttt{nil\_ol}\ L\ L \triangleq \top$$
$$\texttt{append\_ol}\ (\texttt{cons\_ol}\ X\ L_1)\ L_2\ (\texttt{cons\_ol}\ X\ L_3) \triangleq \texttt{append\_ol}\ L_1\ L_2\ L_3$$

The property that `append_ol` is deterministic in its third element can be stated as the following theorem:

$$\forall L_1\ L_2\ L_3\ L_3'.\texttt{append\_ol}\ L_1\ L_2\ L_3 \supset \texttt{append\_ol}\ L_1\ L_2\ L_3' \supset L_3 = L_3'$$

We have already proved the same property for the append relation on lists of natural numbers in Section 2.2. It is easy to see the above theorem can be proved by following the same steps as described in Section 2.2.

To avoid the duplication of generic definitions and theorems, we will develop a kind of polymorphism called *schematic polymorphism* in $\mathcal{G}$ following [37]. We discuss the basis of its development in the rest of this section.

The idea is to introduces *type variables* and treat constants, terms and formulas whose types contain type variables as schema that represent the collection of constants, terms and formulas under the instantiation of type variables with ground types. A type variable A is a type that can be instantiated by any other type. A *type constructor* a with *arity* n takes n types $T_1, ..., T_n$ as arguments to form a new type $a\ T_1\ ...\ T_n$. We introduce *kinds* of the form $\texttt{type} \to ... \to \texttt{type}$ to classify type constants with different arities. For example, the list type can be formed from a type constructor $\texttt{list} : \texttt{type} \to \texttt{type}$ of arity 1. We call types containing type variables such as $\texttt{list}\ A$ *polymorphic types* and terms (formulas) of polymorphic types *polymorphic terms (formulas)*. A constant declaration $c : T$ where T is a polymorphic type represents a collection of distinct constants under instantiation of type variables in T with ground types. They are denoted by c indexed with the ground instances for type variables as subscripts. Similarly, a polymorphic term (formula) represents an infinite collection of terms (formulas) under the ground instantiation of type variables. For example, we can define constructors $\texttt{nil} : \texttt{list}\ A$ for forming empty lists and $\texttt{::} : A \to \texttt{list}\ A \to \texttt{list}\ A$ for forming a list from a head element and a tail list.

A definition for a polymorphic constant $c : T$ is interpreted as a mutually recursive definition for all the instances of c containing all definitional clauses under the ground instantiation of type variables. For example, the definition for `append` is interpreted as the following mutually recursive definition for all instance of `append` (suppose the signature only contains `nat`):

$$\texttt{append}_{\texttt{nat}}\ \texttt{nil}\ L\ L \triangleq \top$$
$$\texttt{append}_{\texttt{nat}}\ (X :: L_1)\ L_2\ (X\ L_3) \triangleq \texttt{append}_{\texttt{nat}}\ L_1\ L_2\ L_3$$
$$\texttt{append}_{\texttt{nat}\to\texttt{nat}}\ \texttt{nil}\ L\ L \triangleq \top$$
$$\texttt{append}_{\texttt{nat}\to\texttt{nat}}\ (X :: L_1)\ L_2\ (X\ L_3) \triangleq \texttt{append}_{\texttt{nat}\to\texttt{nat}}\ L_1\ L_2\ L_3$$
....

Because of the schematic interpretation, a polymorphic term or definition has different interpretation in different signatures. For instance, if a new type `tm` is defined, then `append` must include clauses that contain this new type.

We interpret polymorphic theorems containing type variables as a collection of theorems under arbitrary instantiation of type variables. We can also

think of the typing variables in them as implicitly universally quantified at the outermost. For example, the theorem that `append` is deterministic in its third element is stated as the following schematic formula:

$$\forall L_1\ L_2\ L_3\ L_3', \texttt{append}\ L_1\ L_2\ L_3 \supset \texttt{append}\ L_1\ L_2\ L_3' \supset L_3 = L_3'$$

where the variables L_1, L_2, L_3, L_3' have the type `list` A for some type variable A that is implicitly universally quantified at the outermost. It represents a collection of theorems with A instantiated by *knat*, *ktm*, `nat` $\rightarrow$ `nat`, ..., etc.

Since the interpretation of schematic definitions varies depending on the signature they are in, it is necessary to ensure that the provability of a polymorphic theorem is independent of such variance. To achieve this, we are going to develop a set of *schematic proof rules* for deducing polymorphic sequents. A proof rule is schematic if it matches proof rules in $\mathcal{G}$ under ground instantiation of type variables and its structure is maintained under arbitrary instantiation of type variables in any signature. Thus, if a polymorphic theorem has a schematic proof, then its proof holds in any signature.

Most of the proof rules in $\mathcal{G}$ are immediately schematic because their shapes are irrelevant to types. The main problem is the rule for analyzing possible cases of a definition, because it involves the unification of "polymorphic" terms. In general, the shape of the solution to a unification problem will be affected by types. Consider the following unification problem where X has type A and F has types $A \rightarrow$ `list nat`:

$$1 :: 2 :: 3 :: \texttt{nil} = F\ X$$

Depending on the type of F and X, the solutions may be drastically different. Suppose A is instantiated with `nat`, then there are four possible solutions, with $X = 1$, 2 or 3 and F unified with the corresponding project of X into the list, or $F = x \setminus 1 :: 2 :: 3 :: \texttt{nil}$ and X being ignored by F. However, if A is instantiated with `list nat`, then there can be only two solutions: either $F = x \setminus x$ and $X = 1 :: 2 :: 3 :: \texttt{nil}$ or $F = x \setminus 1 :: 2 :: 3 :: \texttt{nil}$. Thus, in general, the polymorphic case analysis rule is not schematic since its shape may change under the instantiation of types. However, in restricted settings, such as in pattern unification, the solution of unifications are not affected by types [38]. Our previous examples falls into this category. We are going to adopt the ideas from [38] to design the polymorphic case analysis rule so that it works in certain restricted settings that still allows a large category of schematic theorems raised in practice to be proved. We are also going to implement the schematic polymorphism in Abella.

5.3 Implementing Recursive Definitions in Abella

The logic $\mathcal{G}$ that underlies Abella requires its fixed-point definitions to satisfy certain stratification conditions: when these conditions are not satisfied, the resulting logic is not guaranteed to be consistent. One of the requirements of these conditions is that a predicate that is being defined must not occur on the

left hand side of an implication in the clauses defining it. The logical relation we used to prove semantics preservation in Section 4.2 violates this condition. When such definitions are used in Abella Version 2.0, the system issues a warning but allows the user to continue. Clearly, this situation is unsatisfactory: we would like to be sure that our correctness proofs are not constructed in an unsound logic.

Fortunately, the particular way in which our logical relation definitions are phrased and used is actually supported by a consistent extension to $\mathcal{G}$. The key idea in this regard is to treat these definitions as *recursive* rather than fixed-point or inductive ones. The distinction is that they may be used to rewrite expressions that match them but they are not permitted to be used in performing case analysis. In [5], Baelde and Nadathur have shown how to combine ideas from logics of fixed-point definitions and the mechanisms provided by a device known as *deduction modulo* [17] to produce an extension of $\mathcal{G}$ that accommodates the kind of treatment of logical relations that we need without losing cut-elimination and, hence, soundness properties.

The work in [5] provides a semantic characterization of the situations in which recursive definitions can be added to $\mathcal{G}$ without losing soundness. A practical system requires a much sharper criterion: the validity of definitions should be something that can be checked via simple syntactic tests. Towards this end, we will investigate reasoning examples to identify a practically checkable condition that satisfies the semantic criterion of [5] and that validates recursive definitions that includes the ones we want to use in our compiler verification work. Once we have identified such a condition, we will incorporate it into the Abella implementation.

6 Related Work

Compiler verification is a very old topic that can be traced back to 1960s [30]. The past decade has witnessed impressive developments on mechanizing compiler verification due to the maturity of formal verification tools. Most of them focus on implementing and verifying compilers for first-order languages (See [15] for a bibliography). The most notable example is the CompCert project that develops a verified compiler for a subset of C using the Coq theorem prover [29]. A distinguishing characteristic of our work is that we are interested in the verification of compiler transformations on higher-order functional languages, in particular, in the compilation passes that analyze and transform the bindings in programs. Our ultimate goal is to demonstrate the effectiveness of our framework with its support of the HOAS approach on implementing verified compilers for functional languages. This section presents the existing work that uses different frameworks and methodologies to mechanize compiler verification for functional programming languages and makes comparison with our work.

Most of the work on verifying functional compiler transformations uses the general theorem provers such as Coq and Isabelle. Because the general theorem provers do not provide a syntactic treatment of binding structure natively,

objects with bindings are usually encoded in first-order forms, such as the Debruijn form or the local nameless form. As a result, manipulation and reasoning about binding structure, such as implementing substitutions and proving their properties, must be made explicit. The following work on verified compilation is formalized by using Coq: In [8] Chlipala describes the verified implementation of a compiler for the STLC where the verification is based on proof by logical relations; In [14] Dargaye describes a verified compiler from a subset of ML into the intermediate language used by CompCert; Hur and Dreyer developed a verified single-pass compiler from a subset of ML to assembly code where verification is based on Kripke logical relations [27]; Neis *et al.* developed a verified multi-pass compiler called Pilsner and a single pass-compiler called Zwickel by using a notion of semantics preservation called Parametric Inter-Languages Simulation (PILS) [39]. The following work is formalized by using Isabelle: The CakeML project implements a verified compiler from a subset of ML to X86 assembly by using Isabelle HOL [28]. The verification follows the simulation approach similar to CompCert. The implementation of the CakeML compiler is boostrapped to generate a verified compiler in x86 assembly code. All those work uses Debruijn indexes to encode the binding structure. As a result, developers must maintain the correct binding structure explicitly and pay close attention to the tedious details related to the notion of bindings. For instance, in Pilsner the higher-order transformations assume that all the bound variables are unique; bound variables must be alpha-renamed to re-establish this uniqueness condition after it is destroyed by a transformation.

The efforts for manipulating and reasoning about binding structure can be greatly reduced by using a higher-order representation for objects with bindings. However, the consistency of theories underlying Coq and Isabelle requires them to exclude a large class of inductive definitions in the HOAS style. To overcome this problem, Chlipala proposed Parametric Higher-Order Abstract Syntax (PHOAS) in [9] to get a pseudo higher-order representation of binding structure in Coq. The idea is to parametrize the type of terms over a variable type. Then types of abstractions have the form $(\tau \to \texttt{tm}\ \tau)$ where τ a variable type and $(\texttt{tm}\ \tau)$ a parametrized term type. Because of the existence of exotic terms in Coq, it is required to assume axioms that assert PHOAS terms are well-formed. Chlipala applied PHOAS to verify a compiler for an impure functional language with recursion and references [10]. PHOAS still requires a lot of explicit manipulation of binding structures. For instance, because variables are not terms and abstractions are over variables insteand of terms, substitutions must be defined explicitly and their properties must be proved by users. Moreover, for certain transformations such as closure conversion, the variable type degenerates to integer and variables degenerate to Debruijn indexes. At this point all the operations related to the notion of bindings become first-order.

Historically, the term HOAS has been misused in a setting where object-level binders are represented by meta-level binders in a system but *analysis* on this representation is not possible (because of the rich equality theory underlying the system). For instance, even if the meta-level binders can be used to represent object-level binders in languages like ML or Haskell, it is not possible to perform

analysis on the binding structure because these languages contain exotic terms (*e.g.*, non-terminating terms) of a certain type that do not correspond to meaningful syntactic objects of that type. We call this approach the *non-canonical HOAS* approach. Guillemette encodes a CPS transformation in Haskell by using the non-canonical HOAS approach and GADT [23]. He argues that the type checking in Haskell ensures typing is preserved by the transformation. Because of the incapability of analyzing variables in Haskell, closure conversion cannot be encoded by using the non-canonical HOAS approach. Instead, he falls back to Debruijn indexes for encoding closure conversion and proves that the transformation is type preserving [22]. Hickey and Nogin proposed a formalization of closure conversion in the MetaPRL logical framework [26] by using the non-canonical HOAS approach. To avoid analysis such as computing free variables and forming fresh environments, they simply combine an abstraction and its enclosing environment to form a closure without any analysis. Because of the inability of analyzing higher-order objects in the non-canonical HOAS approach, none of the work above proved the semantics preservation of the encoded transformations.

A framework based on the HOAS approach does not only support encoding of object-level binders with meta-level binders but also analysis on such encoding. Examples of this kind of frameworks include LF/Twelf [25][43], Beluga [44] and Abella [4]. Hannan and Pfenning demonstrated the specification of a compiler from STLC to abstract machines using the LF logical framework and the verification of the compiler using the Elf meta-language (later renamed to Twelf) [24]. However, their usage of HOAS is limited to a very simple translation from HOAS terms to their Debruijn forms. Tian mechanized a CPS transformation for STLC in LF and proved its correctness in Twelf by using the simulation approach [50]. Recently, Belanger *et al.* developed compiler transformations including CPS transformation, closure conversion and code hoisting in Beluga and proved that they preserve typing [6]. However, to our knowledge, this work does not encompass a proof of semantics preservation for the encoded compiler transformations.

Having discussed the work on verifying functional transformations from the perspective of frameworks, we would like to compare the strengths of the existing frameworks from the perspective of the notions of semantics preservation. We have already discussed two notions of semantics preservation based on simulation and logical relations, respectively, in Section 4.1. Simulation is the most well-known notion for semantics preservation. Most frameworks are capable of mechanizing proofs by simulation. Examples include [14][10] in Coq, the CakeML project [28] in Isabelle HOL and [24][50] in Twelf/LF. Although there is no existing work on semantics preservation via proof by simulation in Beluga, there is nothing fundamental that prevents Beluga to realize this technique. Proof by logical relation is more difficult to formalize because it involves explicit usage of substitutions and an extensional reading of universally quantified variables. Logical relations can be mechanized in general framework such as Coq, as described in [8][9][27], thanks to their strong meta-theory. For framework with a weak meta-theory, such as LF which can only be used to prove "forall-exists"

theorems, proof by logical relation cannot be realized directly. An indirectly encoding of logical relations called *structural logical relations* is proposed in [47] in which logical relations are embedded in a logic which itself is encoded in the LF framework. Structural logical relations incurs another layer of complexity to the proofs using logical relations. Recently, it is shown that Beluga is capable of realizing proof by logical relations [7]. Its usage for compiler verification still needs to be developed. Parametric Inter-Languages Simulation (PILS) is a promising new notion of semantics preservation which addresses the modularity, flexibility and transtivity problems described in Section 4.1. Development of verified compilers using this notion is mechanized in Coq [39]. It has many features similar to logical relations, such as usage of explicit substitution and universal quantifications with extensional reading. Thus, we expect that the formal development of semantics preservation proofs based on PILS can be simplified by using our methodology for proving semantics preservation by logical relations.

7 Conclusion

Programs written in high-level languages need to be transformed into machine code by compilers for execution. To ensure that the properties that are verified for high-level programs still hold after the compilation, it is necessary to verify that compilers preserve the semantics of programs. Our thesis will focus on the problem of implementing verified compiler transformations on programs written in functional languages. Because the manipulation of binding structure is an essential part of functional transformations, the Higher-Order Abstract Syntax (HOAS) which provides a meta-level treatment of binders is useful in both implementation and reasoning about such transformations. We propose an approach to implement and verify compilers using a framework consisting of a specification language λProlog and a theorem prover Abella; both of which support HOAS. In our thesis, we will demonstrate the effectiveness of this approach to implementing verified compilers for functional languages by developing compiler transformations for a PCF-style language and verify their correctness using different methods for characterizing semantics preservation. We will show that the HOAS approach supported by this framework enables concise implementations of functional transformations and greatly simplifies the verification of such implementations. We are also going to extend the Abella theorem prover to either enable or simplify certain verification tasks related to our work on compiler verification. The extensions include the support of high-order relational specifications, polymorphism and recursive definitions.

References

[1] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Proceedings of the 15th European Conference on Pro-*

gramming Languages and Systems, ESOP'06, pages 69–83, Berlin, Heidelberg, 2006. Springer-Verlag.

[2] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

[3] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *35th ACM Symp. on Principles of Programming Languages*, pages 3–15. ACM, January 2008.

[4] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.

[5] David Baelde and Gopalan Nadathur. Combining deduction modulo and logics of fixed-point definitions. pages 105–114. IEEE Computer Society Press, June 2012.

[6] Olivier Savary Belanger, Stefan Monnier, and Brigitte Pientka. Programming type-safe transformations using higher-order abstract syntax. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, pages 243–258, 2013.

[7] Andrew Cave and Brigitte Pientka. A case study on logical relations using contextual types. In *Proceedings of the 2015 ACM SIGPLAN Workshop on Logical Frameworks & Meta-Languages: Theory & Practice, LFMTP'15*, page 1. ACM, August 2015.

[8] Adam Chlipala. 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*, PLDI '07, pages 54–65, New York, NY, USA, 2007. ACM.

[9] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008.

[10] Adam Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 93–106, New York, NY, USA, 2010. ACM.

[11] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[12] Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

[13] Oliver Danvy and Andrzex Filinski. Representing control: a study of the cps transformation. *Mathematical Structures in Computer Science*, 2:361–391, 12 1992.

[14] Zaynah Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. PhD thesis, l'Université Paris 7-Denis Diderot, France, July 2009. Written in French.

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

[16] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[17] Gilles Dowek and Benjamin Werner. Proof normalization modulo. *Journal of Symbolic Logic*, 68(4):1289–1316, 2003.

[18] Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009.

[19] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.

[20] Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.

[21] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935. Translation of articles that appeared in 1934-35. Collected papers appeared in 1969.

[22] Louis-Julien Guillemette and Stefan Monnier. A type-preserving closure conversion in haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 83–92, 2007.

[23] Louis-Julien Guillemette and Stefan Monnier. Type-safe code transformations in haskell. *Electron. Notes Theor. Comput. Sci.*, 174(7):23–39, June 2007.

[24] John Hannan and Frank Pfenning. Compiler verification in LF. In *7th Symp. on Logic in Computer Science*, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[25] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *2nd Symp. on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[26] Jason Hickey and Aleksey Nogin. Formal compiler construction in a logical framework. *Higher Order Symbol. Comput.*, 19(2-3):197–230, September 2006.

[27] Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ml and assembly. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 133–146, New York, NY, USA, 2011. ACM.

[28] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.

[29] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[30] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society, 1967.

[31] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[32] Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

[33] Dale Miller. Abstract syntax for variable binders: An overview. In John Lloyd and *et al.*, editors, *CL 2000: Computational Logic*, number 1861 in LNAI, pages 239–253. Springer, 2000.

[34] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.

[35] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

[36] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. Technical Report CMU-CS-95-171, School of Computer Science, Carnegie Mellon University, July 1995.

[37] Gopalan Nadathur and Frank Pfenning. The type system of a higher-order logic programming language. In Frank Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.

[38] Gopalan Nadathur and Xiaochu Qi. Optimizing the runtime processing of types in polymorphic logic programming languages. In G. Sutcliffe and A. Voronkov, editors, *LPAR: Logic Programming and Automated Reasoning, International Conference*, volume 3835 of *LNCS*, pages 110–124. Springer, December 2005.

[39] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (to appear)*, ICFP '15, New York, NY, USA, 2015. ACM.

[40] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.

[41] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer Verlag, 1994.

[42] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.

[43] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.

[44] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in LNCS, pages 15–21, 2010.

[45] Andrew M. Pitts. Nominal logic, A first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[46] Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. The Teyjus system – version 2, 2015. http://teyjus.cs.umn.edu/.

[47] Carsten Schürmann and Jeffrey Sarnat. Structural logical relations. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 69–80. IEEE Computer Society Press, 2008.

[48] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 275–287, New York, NY, USA, 2015. ACM.

[49] The Coq Development Team. The Coq Proof Assistant Reference Manual Version 7.2. Technical Report 255, INRIA, February 2002. More recent versions may be obtained from the site `http://coq.inria.fr/`.

[50] Ye Henry Tian. Mechanically verifying correctness of cps compilation. In Joachim Gudmundsson and Barry Jay, editors, *Twelfth Computing: The Australasian Theory Symposium (CATS2006)*, volume 51 of *CRPIT*, pages 41–51, Hobart, Australia, 2006. ACS.

[51] Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. Reasoning about higher-order relational specifications. In Tom Schrijvers, editor, *Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming (PPDP)*, pages 157–168, Madrid, Spain, September 2013.