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

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Yuting Wang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Gopalan Nadathur

December, 2016

Acknowledgements

Many people have provided me help and support during my doctoral studies. I would like to take this opportunity to express my gratitude to them.

First and foremost, I would like to thank my advisor Gopalan Nadathur for his mentorship throughout my stay at the University of Minnesota, which helped me better understand what research in computer science is all about and how to conduct such research. The dissertation would not have been in its current state without his guidance during the thesis research, his patient examination and revision of the thesis drafts, and his constant encouragement throughout the project. I look forward to continuing to collaborate with him in the future.

I would like to thank Kaustuv Chaudhuri for his guidance and collaboration during my years as a doctoral student. I did two summer internships in the Parsifal team in Inria Saclay under his supervision. The extensions to the Abella system described in the thesis were either direct results of the internships or partially inspired by the work done during that time. To the extent that this work was successful, I owe a great deal to the inspiring discussions I had with Kaustuv and to his supportive attitude towards the research I was interested in doing.

I am grateful to Dale Miller for hosting me in the Parsifal team during my internships and to the other (former or present) members and visitors of the Parsifal team with whom I interacted and learned a great deal from, including but not limited to Anupam Das, Taus Brock-Nannestad, Danko Ilik, Quentin Heath, Tomer Libal, Siddhartha Prasad, Zakaria Chihani, Beniamino Accattoli and Nicolas Guenot.

My interest in exploring the use of the higher-order abstract syntax approach in verified compilation originated from discussions with Michael Whalen. Although the project took quite a different path in the end, Michael has expressed continuous support and interest in my work, for which I am thankful.

Special thanks to Andrew Gacek for creating the original version of the Abella system and for patiently explaining to me the details of the implementation that I have periodically had questions about. I have also learned a great deal of the underlying theory of Abella via interactions with him in the past few years.

I am honored to have had Professors Eric Van Wyk, Stephen McCamant and Karel Prikry on my thesis committee. Comments especially from Eric and Stephen have helped me improve the quality of my dissertation. Special thanks to Eric for taking care of the programming languages lab and providing useful suggestions for being an effective researcher. I am also grateful to Wayne Richter for having served on my thesis proposal committee and for teaching the courses that provided me an introduction to mathematical logic.

I have benefited from discussions with Olivier Savary Belanger, Brigitte Pientka, Derek Dreyer, Chung-Kil Hur and Georg Neis. I would like to thank them all for the patience with which they have explained aspects of their work and for the insights into it that they have thereby provided. Conversations with all of them have helped me understand the big picture in the various areas this thesis relates to.

Finally, I would like to thank the students and friends I worked and studied with at the University of Minnesota, especially Dan DaCosta, Mary Southern, Ted Kaminski, Kevin Williams, Dawn Michaelson, Dongjiang You, Ming Zhou, Peng Liu, Shuo Chang, Xiang Cao, Ziqi Fan, Yuan Li and Yin Liu.

Work on this dissertation has been partially supported by the National Science Foundation through the grants CCF-0917140 and CCF-1617771. Support has also been received from a Doctoral Dissertation Fellowship and a Grant in Aid of Research from the University of Minnesota. Opinions, findings and conclusions or recommendations that are expressed in this dissertation should be understood as mine. In particular, the do not necessarily reflect the views of the National Science Foundation.

Dedication

To my parents and grandparents

Abstract

This thesis concerns the verified compilation of *functional programming languages*. Functional programming languages, or *functional languages* for short, provide a high degree of abstraction in programming and their mathematical foundation makes programs written in them easy to analyze and to be proved correct. Because of these features, functional languages are playing an increasingly important role in modern software development. However, there is a gap that must be closed before we can derive the full benefits of verifying programs written in functional languages. Programs are usually verified with regard to the computational models underlying the functional languages, while the execution of programs proceeds only after they are transformed by compilers into a form that is executable on real hardware. To get programs verified end-to-end, the compilers must also be proved correct.

Significant strides have been taken in recent years towards the goal of verifying compilers. However, much of the attention in this context has been on imperative programming languages. The verification of compilers for functional languages poses some additional challenges. A defining characteristic of such languages is that they treat functions as first-class objects. In describing the compilation of programs written in functional languages and in reasoning about this compilation, it is therefore necessary to treat functions as data objects. In particular, we need to provide a logically correct treatment of the relationship between the arguments of a function and their use within its body, i.e., the *binding structure* of the function. Most existing proof systems for formal verification provide only very primitive support for reasoning about binding structure. As a result, significant effort needs to be expended to reason about substitutions and other aspects of binding structure and this complicates and sometimes overwhelms the task of verifying compilers for functional languages.

We argue that the implementation and verification of compilers for functional languages are greatly simplified by employing a higher-order representation of syntax known as *Higher-Order Abstract Syntax* or *HOAS*. The underlying idea of HOAS is to use a meta-language that provides a built-in and logical treatment of binding related notions. By embedding the meta-language within a larger programming or reasoning framework, it is possible to absorb the treatment of binding structure in the object language into the meta-theory of the system, thereby greatly simplifying the overall implementation and reasoning processes.

We develop the above argument in this thesis. In particular, we present and demonstrate the effectiveness of an approach to the verified implementation of compiler transformations for functional programs that exploits HOAS. In this approach, transformations on functional programs are first articulated in the form of rule-based relational specifications. These specifications are rendered into programs in λProlog, a language that is well-suited to encoding rule-based relational specifications and that supports an HOAS-style treatment of formal objects such as programs. Programs in λProlog serve both as specifications and as executable code. One consequence of this is that the encodings of compiler transformations serve directly as their implementations. Another consequence is that they can be input to the theorem proving system Abella that provides rich capabilities for reasoning about such specifications and thereby for proving their correctness as implementations. The Abella system also supports the use of the HOAS approach. Thus, the λProlog language and the Abella system together constitute a framework that can be used to test out the benefits of the HOAS approach in verified compilation. We use them to implement and verify a compiler for a representative functional programming language that embodies the transformations that form the core of many compilers for such languages. In both the programming and the reasoning phases, we show how the use of the HOAS approach significantly simplifies the representation, manipulation, analysis and reasoning of binding structure.

Carrying out the above exercise revealed some missing capabilities in the Abella system. At the outset, it was possible to reason about only a subset of λProlog programs using the system. Some compiler transformations required the use of features not available in this subset. Another limitation was that it did not support the ability to reason about polymorphic specifications, thereby leading to a loss of modularity in programs and in reasoning. We have addressed these issues as well in this thesis. In particular, we have developed the theoretical underpinnings for introducing polymorphism into Abella and for treating the full range of λProlog specifications. These ideas have also been implemented to yield a new version of Abella with the additional capabilities.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The focus of this thesis is on the verified compilation of functional programming languages. Functional programming languages, or *functional languages* for short, form a class of programming languages that has been the subject of significant research and that is now seeing increasing use in research and industrial settings. One reason for the growing popularity of these languages is the high degree of abstraction they offer both in the representation of data and in the way computations can be expressed. Their close correspondence to mathematical formalisms such as the lambda calculus also makes it easier to analyze programs and even to prove their correctness. There is, however, a catch to these kinds of behavioral analyses. On the one hand, to derive benefit from the structure of the language that is used, the analyses of programs have to be based on the computational model underlying the language. On the other hand, the execution of a program proceeds only after it has been translated into a form that can be run directly on actual hardware. Thus, to leverage the benefits of using functional languages in obtaining assurances of the correctness of a program, it is necessary to ensure that the translation process does not change its meaning, i.e. to verify the compiler.

The importance of compiler verification to overall program correctness has been long recognized and interest in the topic dates back almost to the advent of programming languages (e.g., see [1]). However, the actual task of verifying a compiler is complex, tedious and error-prone and so few such efforts were undertaken in the early years. The availability of sophisticated tools for mechanizing reasoning in recent times has changed this situation dramatically: there have been several well-documented and successful efforts to verify compilers for programming languages that are actually used in practice. Much of this work has been aimed at conventional imperative programming languages such as C or Java—see [2] for a survey on this topic. Because of the high-level of abstraction that functional languages support, the verification of compilers for these languages must deal with some problems that do not arise relative to imperative languages. One particular source of these problems is that functional languages treat functions as first-class objects, allowing them to be passed as arguments and returned as values. As a consequence, compilers for these languages must pay careful attention to the *structure* of functions and, more specifically, to the connection between the arguments of a function and their use within the body. In fact, the initial phases of compilation of functional languages typically involve the explicit manipulation of this kind of structure that we refer to as the *binding structure* of expressions. To prove the correctness of these phases, it becomes necessary also to validate such manipulation of binding structure. Experience has shown this to be a complex task. In fact, it is now well appreciated that without specialized tools or techniques the cost of analyzing, manipulating and reasoning about binding structure can overwhelm the compilation and verification processes.

The need to treat binding structure is not limited to the domain of compiler verification. This kind of need arises in the context of a variety of systems that are geared towards the formal treatment of objects such as programs, formulas, proofs and types. One of the approaches that has been proposed for simplifying the programming and reasoning tasks in these contexts is that of *higher-order abstract syntax* or *HOAS* [3, 4]. The underlying idea in this approach is to use a meta-language for representing formal objects that provides a built-in, logical means for capturing binding related notions. By embedding this kind of a meta-language within a larger programming or reasoning framework, it is possible to absorb the treatment of binding structure in the object language into the meta-theory of the system, thereby greatly simplifying the overall implementation and reasoning processes. Of course, the choice of meta-language must be properly calibrated so as to ensure that these benefits actually flow in practice. There have been two successful realizations of this approach that possess this characteristic. One of these approaches, that is embodied in the Twelf system [5] and its successor Beluga [6], is based on the use of a dependently typed lambda calculus [7] to encode and to reason about formal objects. The second approach, which has resulted in the specification and programming language λProlog [8, 9] and the reasoning system Abella [10, 11], uses a predicate logic over simply typed lambda terms to realize similar capabilities.

Our objective in this thesis is to show that the HOAS approach can be used to significantly simplify the implementation and verification of compilers for functional programming languages. Towards this end, we show how the second realization of the HOAS approach that is described above can be utilized to benefit in this task. Specifically, we show how some of the complex transformations involved in compiling functional programs can be elegantly encoded in λProlog. We then show how these implementations can be effectively reasoned about using the Abella system. In carrying out these tasks, we expose a methodology that we believe to be broadly applicable in this domain. In the course of this work, we have discovered ways to strengthen the Abella system so as to make it more suitable to such applications. We present these ideas as well in this thesis.

We elaborate on the broad ideas described above in this introductory chapter, as a prelude to their technical development in the rest of the thesis. Section 1.1 motivates the general compiler verification endeavor. Section 1.2 discusses the specific difficulties related to the treatment of binding structure that arise in the verified compilation of functional languages. Section 1.3 exposes some of the approaches that have been proposed and used for dealing with these issues; this discussion also introduces the HOAS approach. In Section 1.4, we expand on the idea of using the HOAS approach to simplify verified compilation of functional programs and we present the specific goals for this thesis in this setting. Section 1.5 summarizes the contributions that we make through this thesis. Section 1.6 concludes this chapter by explaining how each of the following chapters fit into the overall dissertation.

1.1 Program Correctness and Compiler Verification

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: at its very best, testing can only provide *evidence* for the correctness of a program, never a water-tight guarantee for the absence of bugs in it. There are many safety critical software systems for which it is in fact necessary to have a guarantee of correctness. The only way in which such an assurance can be provided is by formally verifying the properties of software using principles of mathematical reasoning.

The above considerations have led to a large amount of effort being invested towards developing approaches for 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 programming languages that make it possible to describe computations at a level at which they are amenable to mathematical analysis. Second, methods have been developed for utilizing the structure of these high-level languages to support the process of reasoning about programs written in them. 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.

Functional languages are a particular class of high-level programming languages that have been investigated in this context. The structure of these languages is best understood in comparison with the more commonly used *imperative programming languages* such as C,C++ and Java. The latter class of languages views 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, a view in which the focus is 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 [12]. Because of the mathematical basis of functional languages that also implies freedom from lower-level implementation details, programs written in these languages are more concise and easier to reason about than those written in imperative programming languages. Despite the abstract nature of these languages, research over the last three decades has shown that programs written in them can be translated into a machine understandable form that can be executed with considerable efficiency. As a consequence of these results, functional languages are starting to play an increasingly important role in modern software practice. Indeed, there are a variety of languages within this paradigm such as Common Lisp, Scheme, Racket, Clojure, Erlang, Haskell, Standard ML, OCaml, Scala, Swift and F# that are being used today in a range of academic, industrial and commercial settings (*e.g.*, see [13, 14, 15, 16]).

As noted above, one of the benefits of high-level programming languages is that they make it easier to reason about program behavior. However, there is a gap that must be closed before we can derive the full benefits of verifying programs written in these languages. On the one hand, to take advantage of the structure of the language in arguments about program correctness, it is necessary to reason within the high-level computational model associated with the language. On the other hand, in order to actually execute programs in these languages, they must be translated by an intervening process known as *compilation* into code at a lower level that can be run directly on a machine. Now, it is possible for this compilation process to be buggy, leading thereby to executable versions of programs whose behavior is different from the ones intended of the original versions. Traditionally, the consensus has been to assume the correctness of compilers, with confidence in this assessment being built over time and repeated use. However, in an absolute sense, this approach is no different from testing. To obtain the full force of verification, it is necessary also to establish the correctness of the compilers that are used.

The importance of proving the correctness of compilers to the program verification endeavor has long been recognized and there has also been much work aimed at understanding how this can be done; a survey of the early efforts appears, for example, in [2]. Compilers are complex pieces of software and their verification is therefore too large, tedious and error-prone a task to carry out effectively by hand. This has hampered several past attempts at showing its practicality. However, significant strides have been taken in recent years towards providing computer-based support for the theorem proving task and this has dramatically altered the situation. Indeed, there has been a mushrooming of efforts related to formal compiler verification using theorem proving systems such as Isabelle [17], HOL [18] and Coq [19]. 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 [20]. Such efforts have also provided an impetus to more ambitious projects such as the Verified Software Toolchain project [21] related to overall program verification.

A majority part of the existing work on verified compilation has been devoted to verifying compilers for imperative programming languages. In this thesis, we focus on verified compilation for functional programming languages. With the increasingly important role functional programming languages play in software development, their verified compilation has also been a research topic on the rise in recent years. Notable work in this area includes the Lambda Tamer project [22, 23], a verified compiler for a subset of Standard ML called CakeML [24] and the Pilsner project [25]. There are some new difficulties that arise in the verified compilation of functional languages in contrast to that for imperative programming languages. We discuss this task and some of the issues involved in it in the next section.

1.2 Verified Compilation of Functional Languages

One important source of power in functional languages is that they allow functions to be treated as data objects. Functions can be embedded in data structures or functions themselves, passed as parameters to other functions and returned as results of computations. These feature are used often in applications, leading to concise programs that are easy to understand and to reason about.

Although the presence of the above features provides high-level abstraction capabilities, it also makes compilation of functional programs a more complex undertaking. The traditional approach to compiling functional languages is to put the source programs through a series of transformations that replace program constructs for supporting the high-level features with simpler devices and eventually render the original programs into a form to which compilation techniques that are well-known from the context of imperative programs can be applied. In describing these transformations and in reasoning about them, it is therefore necessary to have a clear and flexible way of treating functions as objects. The difficulty with objectifying functions is that they have inputs or *arguments* and it is necessary for compilers to build in an understanding of the logical relation that these arguments have with their occurrences within the body of the function; we will refer to this logical relation as the *binding structure* of the function. Given a particular expression, a compiler for a functional language must be able to differentiate between variables bound by some arguments and variables not bound by any argument, *i.e. free variables*, in a function valued expression. The compiler should also not distinguish between expressions that differ only in the names used for their bound variables and it should be able to realize substitution over expressions in a way does not replace bound variables and also avoids the inadvertent capture of free variables in the expression being substituted. Manipulating objects with binding structure adds an additional layer of complexity to compilers for functional languages on top of the existing complexity of program transformations common to all compilers. This complexity percolates also into the process of reasoning about the correctness of the transformations. Most programming languages and theorem-proving systems provide only rudimentary support for manipulating and reasoning about binding structure. The person implementing and verifying the compiler must therefore build an additional infrastructure for treating binding related issues. This effort is, in a sense, orthogonal to the main focus in compiler verification and, without suitable support, it has often been known to overwhelm the real objective.

To concretely expose the complexity of dealing with binding structure in implementing and verifying compilers for functional programs, let us consider a typical transformation in such compilers known as *closure conversion* [26, 27, 28, 29, 30]. This transformation replaces each expression that has a functional structure with a version of that expression that is parameterized by its free variables paired with an environment that provides binding for these variables; such a combination is referred to as a *closure*. Closure conversion is an important step in the compilation process partly because it enables further simplification steps; for example, it enables the elimination of nested functions because a closed expression can be moved out to the top level of a program without changing its meaning. To realize the closure conversion transformation, it is necessary to identify all the free variables of an expression, to transform this collection into an environment and to add an environment as an extra parameter to the expression. For example, when closure conversion is applied to the following pseudo-code in a functional language

$$\texttt{let}\ x = 2\ \texttt{in}\ \texttt{let}\ y = 3\ \texttt{in}$$
$$\quad \texttt{fun}\ z \rightarrow z + x + y$$

it will yield the code fragment

$$\texttt{let}\ x = 2\ \texttt{in}\ \texttt{let}\ y = 3\ \texttt{in}$$
$$\quad \langle(\texttt{fun}\ z\ e \rightarrow z + e.1 + e.2), (x, y)\rangle$$

We write $\langle F, E\rangle$ here to represent a closure whose function part is F and environment part is E. Further, we represent an environment as a tuple and we write $e.i$ to indicate the selection of the i-th element of a tuple e. Intuitively, a closure stands for the application of its function part to its environment, but an application that is "suspended" until the closure is provided with a value for its "actual" argument which, in the case of the example considered, is z.

A key part of implementing the closure conversion transformation is, as we have indicated above, the identification of the free variables of an expression and the replacement of these variables by indexed selections from a new environment parameter that is added to the expression. Carrying out these steps obviously requires us to perform a non-trivial analysis of the binding structure of expressions. This can be a complicated task in an environment that does not support an explicit representation of binding structure. This difficulty gets further amplified when we have to reason about the correctness of the transformation: properties about binding structure that are implicit in the representation and treated through user code must be explicitly proved in the reasoning process. For example, to make the intuition that closure conversion preserves the meaning of programs formal, we need to explicitly prove a theorem that describes this property in a reasoning system. To formalize the meaning of programs we often need a formal notion of substitution. As a part of proving the meaning preservation theorem, we will need to show that the substitution operations do not change the function part of a closure. This property is implied by the description of closure conversion because closures generated by it must be *closed*, *i.e.* they must not contain free variables, and substitution has no effect on closed terms. To play out this argument in our proof, we will need to formally describe the property of being a closed term and then show that substitutions have no effect on such terms. If the implementation and theorem-proving framework do not have an effective way to deal with binding structure, formalizing and reasoning about these aspects can be difficult and can become a major part of the verification effort, thereby blurring the essential content of the proofs.

1.3 The Treatment of Object-Level Binding Structure

The need to represent and analyze binding structure arises in a number of contexts that involve the manipulation of formal objects such as programs, formulas, types and proofs. A common core can be identified to the issues that have to be treated in a formalization framework that is well-suited to these varied contexts. First, the framework must enable the representation of the objects of interest in a way that makes the binding constructs within them explicit. Second, it must provide a simple and verifiably correct encapsulation of logical notions relating to binding structure; these notions include, at the very least, equivalence modulo renaming of bound variables and logically correct substitution. Third, it must provide a means for analyzing the representations of the formal objects in a way that takes into account the binding constructs within them. Finally, the framework must provide a way to prove properties about the represented objects that incorporates an understanding of the binding structure that is manifest in them.

Recent research has highlighted the need to deal with issues of the kind described above and a few different approaches have been developed towards this end. We discuss some of the more prominent approaches that have emerged from this work below with the goal of putting the approach that will be the subject of this thesis in context.

1.3.1 Approaches based on first-order representations

The basis for this class of approaches is a first-order representation of syntax that is augmented with special devices for interpreting the parts that deal with binding notions within such a representation. In their simplest form, these special devices consist of library functions that can be used to realize binding sensitive operations over the representations of objects. The most common example of this approach is 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 [31]. A virtue of such a "nameless" representation is that it 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. When it comes to reasoning about properties of the represented objects in a way that takes into account the binding constructs within them, it becomes necessary of course to incorporate into the process the task of reasoning about the programs that realize substitution and related operations. It has been noted that considerable effort can be taken up in proving "boilerplate" properties related to binding in this context. For example, it is observed in [32] that when the nameless representation is used for programs in a typed language , it becomes necessary to prove weakening lemmas for typing in the course of proving type soundness for the language and that this requirement significantly complicates the overall proof.

A few variants of the basic approach discussed above have been developed to solve specific problems. For example, it is sometimes useful to treat the free variables in an expression differently from the bound variables and a representation called the *locally nameless* representation has been developed towards this end [32, 33]. Although these variants simplify the treatment of some aspects of binding structure, they do not alter the first-order nature of the representation and hence do not overcome its primary drawback.

1.3.2 The nominal logic approach

A different approach that still uses a first-order representation of syntax is that based on nominal logic [34]. The defining characteristic of this approach is that it provides a logical treatment of equivalence of expressions under a renaming of bound variables; the technical device it uses to realize this capability is that of equivalence of expressions under permutations of names. This representation obviously subsumes the benefits of the nameless representation discussed earlier. 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 and hence do not also intrinsically support the analysis of expressions with binding structure under substitution. The realizations of these and related aspects have to be encoded in user programs that must then also be explicitly reasoned about.

1.3.3 The functional higher-order approach

When the formalization framework is based on a functional language, it is possible to use expressions of function type to represent objects that need to be manipulated or reasoned about. In this case we can potentially use abstraction in the meta-language to represent binding structure in formal objects. Such an idea has in fact been investigated and we refer to it here as the *functional higher-order approach*.

One of the benefits of using meta-language abstraction to encode binding is that it leads to a simple treatment of substitution: the need to avoid inadvertent capture of free variables in the expression being substituted and to replace the right variable occurrences are both built into the treatment of function evaluation at the meta-level. Benefit has been derived from this observation in a variety of tasks. We mention two that are closely related to the work in this thesis. In [35], Guillemette has used this idea in implementing the Continuation Passing Style (CPS) transformation—a common transformation in the compilation of functional languages—in Haskell. Similarly, Hickey and Nogin have exploited this idea in implementing a version of closure conversion in their MetaPRL logical framework [36].

There is, however, a serious limitation to the functional higher-order approach: it is not capable of supporting the examination of the structure of objects that embed binding constructs. The reason for this is that the notion of equality of expressions in the context of a functional language includes a lot more than just bound variable renaming and β-conversion. To support computation in an adequate fashion, such a language must include at least the conditional and fixed-point combinators together with their associated notions of evaluations. Further, the programmer has the ability to add to the equality theory by defining new functions or combinators. It is not surprising, therefore, that the uses of the functional higher-order approach have not included serious analyses of syntactic structure modulo binding, such as the computation of free and bound variables in expressions. As a concrete example, the closure conversion implementation of Hickey and Nogin makes the simplifying assumption that every variable bound by an external abstraction appears free in an expression of function type that they are wanting to convert into a closure.

1.3.4 The higher-order abstract syntax approach

One way to derive the benefits of the functional higher-order approach while still retaining the ability to examine the structure of expressions is to use a λ-calculus for representation that has very weak computational power. One way to do this is to use a typed λ-calculus that does not include built-in combinators or the provision to define them. This idea underlies what has been called the *higher-order abstract syntax* or HOAS approach [3, 4]. Similar to the functional higher-order approach, binding constructs in object language syntax can be encoded directly via abstraction in the meta-language. The critical difference from the functional higher-order approach is that equality of expressions is governed solely by bound variable renaming and β-conversion which, in this weak setting, corresponds only to substitution. The result of this is that the HOAS approach is capable of supporting an analysis of the binding structure of objects. In contrast to the first-order approaches, these capabilities derive from features of the meta-language and hence do not need to be reasoned about explicitly. Moreover, with a properly configured logic, it is possible also to reason about objects that embody binding notions using rich principles such as case analysis and induction.

The systems that have been developed for specifying and reasoning about formal systems based on the HOAS approach include Twelf [5], Beluga [6], Hybrid [37], λProlog [8, 9] and Abella [10, 11]. These specification and reasoning systems have been used in many formalization efforts related to objects that embody binding constructs and the benefits described above have been shown to be real through these applications. A survey of these efforts can be found in [38, 39].

1.4 Verified Compilation Using the HOAS Approach

The focus of this thesis, as we have previously noted, is on the implementation and verification of compilers for functional languages. Our contention is that these tasks are considerably simplified by the use of the HOAS approach. We provide evidence for this claim by carrying out the exercise of implementing and verifying a compiler for a representative functional language within a framework that provides support for the approach.

The framework we use in this work comprises the executable specification language λProlog [8, 9] and the theorem-proving system Abella [11, 40]. A natural way to characterize a formal system is to describe it through relations that are defined via inference rules based on the syntactic structure of the objects of interest in the system. The λProlog language is a suitable vehicle for formalizing such descriptions: the language is based on a fragment of intuitionistic logic that allows for the transparent encoding of rule-based relational descriptions. A further virtue of λProlog is that it supports the use of a version of HOAS, which has been labeled *λ-tree syntax* in [41], in constructing these relational specifications; we discuss this variant of HOAS in more detail in Chapter 2. The λProlog language has an executable interpretation. As a consequence, specifications written in it can also serve as implementations. The Abella theorem-prover is also based on an intuitionistic logic that supports relational specifications exploiting the λ-tree syntax approach. However, this logic is more richly configured than the one underlying λProlog: in particular, it provides mechanisms for case analysis and induction based reasoning. While Abella can be used to reason directly about relational specifications, there is also the intriguing possibility that it could be used to do this indirectly by reasoning about programs written in λProlog. In fact, features have been built into Abella to make this possibility a reality: the logic underlying λProlog has been embedded into Abella via a definition and particular specifications in λProlog can then be reasoned about via this encoding. In summary, the combination of λProlog and Abella gives us a framework for both implementing relational specifications and reasoning about such implementations.

The methodology that we will use to realize verified compilation is the following. We will first articulate each transformation that underlies the compilation of our functional language in the form of a rule-based relational specification. We will then render this specification into a λProlog program. The executability of λProlog specifications means that our program serves directly as an implementation of the transformation. We then use the Abella system to reason about the properties of the λProlog specification, thereby proving the correctness of the compiler transformation it embodies. In both the specification/programming and the reasoning phases, we will show how the support the framework provides for the λ-tree syntax approach can be used to advantage.

To keep the exposition manageable, the exercise we describe above will be applied to a functional language that is restricted but still contains all the features needed to make the results convincing. The particular language that we will treat will be a superset of the language commonly known as PCF [42] that includes recursion. The sequence of transformations that we will consider will render source language programs into ones in a language similar to Cminor, which is the back-end language for the CompCert project [43]. There are still some transformation steps that need to be carried out in order to obtain actual machine-executable code from programs in our Cminor-like language. However, we do not investigate these steps in this thesis for two reasons. First, the verified implementation of these steps has been studied elsewhere in the literature and we do not have new ideas to add to that discussion; in particular, we do not feel that the HOAS approach holds significant benefits after code in the Cminor-like language has been produced. Second, narrowing the scope of our work in this way allows us to focus more sharply on the part of compilation that is novel to functional languages and where we believe the HOAS approach really shines.

Part of the work we describe also has the objective of enhancing the framework we use to make it more suitable for the relevant application domain. These enhancements are essentially to the Abella theorem-prover. The version of the system that we started with had limited the collection of λProlog specifications that could be reasoned about. We have removed this limitation so that the full logical structure of λProlog specifications can now be used to encode compiler transformations. The second extension incorporates polymorphism into Abella. A concrete result of this extension is that the λProlog programs about which we reason can also be polymorphic. This actually has an important practical benefit: Polymorphism allows us to share code that pertain, for example, to manipulation of generic data structures. Perhaps even more importantly, the sharing of code also allows us to prove its properties just once and to then use these properties in whichever place we eventually use the code.

1.5 The Contributions of The Thesis

In summary, the work underlying this thesis contributes to the state-of-the-art in three ways:

- It leads to extensions of the Abella theorem prover that make it a more versatile tool for applications such as compiler verification. At a conceptual level, those extensions provide a means for reasoning about the full range of λProlog specifications and they lead to logically sound support for polymorphism in specifications (which also double up as implementations in the context of λProlog) as well as in the theorems that are proved about the specifications. We have also incorporated these extensions into the implementation of the Abella system.

- It demonstrates the benefits of the HOAS approach in implementing compilers for functional languages. This goal is achieved by developing a methodology for implementing compiler transformations in λProlog that exploits and also show-cases the facilities the language possesses for supporting the HOAS approach. An auxiliary effect of this exercise is that it yields a compiler that produces intermediate code similar to that in a popular low-level language for a representative functional language that includes recursion.

- It demonstrates the benefits of the HOAS approach in verifying compilers for functional languages. To achieve this objective, it develops a methodology for verifying compiler transformations expressed in λProlog using Abella. It also applies this methodology to verify the compiler for the representative functional language developed in this work, in the process illuminating the way in which the logical structure of λProlog implementations and the HOAS techniques can conspire to simplify correctness proofs.

We note that the work in this thesis is not the first one to undertake the verification of properties related to compilation-oriented transformations using the HOAS approach. In [44], Hannan and Pfenning have exploited this approach in specifying some simple transformations—such as the translation of conventional λ-terms to their de Bruijn forms—using the dependently typed λ-calculus LF and in verifying these specifications using the Elf meta-language (later renamed to Twelf). In [45], Tian has mechanized a CPS transformation for the simply typed λ-calculus in LF and proved its correctness in Twelf. Finally, in [46], Belanger *et al.* have developed a collection of compiler transformations that includes the CPS transformation, closure conversion and code hoisting in Beluga; their specification of these transformations is such that the fact that it passes type checking ensures that types are preserved between the source and target language versions of the program. What distinguishes our work is that it is a systematic study of the implementation and verification of compilers for functional programs that examines the use of the HOAS approach in verifying deep properties such as meaning preservation. We also note that while we conduct our work in the context of a framework defined by λProlog and Abella, the ideas we develop should be applicable to compiler verification using systems like Twelf and Beluga, to the extent that these systems support reasoning principles that are strong enough to carry out the relevant tasks.

1.6 An Overview of the Thesis

The rest of this thesis develops the ideas that we have discussed in this chapter. In Chapter 2, we introduce the λProlog language and the Abella theorem prover. The discussion in this chapter also brings out the λ-tree syntax approach and its use in simplifying both the implementation and the verification of rule-based relational specifications. We then present the two extensions to the Abella theorem prover in Chapter 3. With the enriched framework in place, we are ready to start the discussion of our verified compilation work. Chapter 4, provides an overview of this work. We begin this chapter by presenting the compilation model that we will use and the approaches to describing compilers in the rule-base and relational fashion and to verifying the correctness of compilers described in this manner. We then show how our framework can be used to formalize such implementation and verification and how the λ-tree syntax approach can be used to simplify these tasks. Although in this thesis we will characterize the correctness of compiler transformations using *logical relations* [47], we discuss in Chapter 4 some other notions of meaning preservation that could have been used and also how the HOAS approach can simplify the verification task in their context as well. We conclude this chapter with an overview of the exercise that we will carry out towards demonstrating the effectiveness of our approach to verified compilation of functional languages. That is, we outline the structure of a compiler for the source language that we have chosen and we present an overview of the steps in its verified implementation. In Chapters 5, 6, 7 and 8, we describe how the compiler transformations constituting this compiler are implemented and verified using our approach. We describe the related work in Chapter 9. Finally, we conclude the thesis and discuss future work in Chapter 10.

Chapter 2

A Framework for Verified Implementation

The formal systems that are of interest in the thesis can be naturally described via rules for deriving relations on syntactic objects. To specify and to reason about such *rule-based relational specifications*, we use the framework consisting of the specification language λProlog [8, 9] and the theorem-proving system Abella [11, 40]. λProlog is a language suitable for encoding rule-based relational specifications. A characteristic of the λProlog language is that specifications written in it are executable and therefore they serve also as implementations; such specifications can, for example, be executed using the Teyjus system that implements λProlog [48]. 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 [49, 50]. We will show in later chapters how this structure can be exploited to realize the goal of verified compilation: 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 support a realization of the HOAS approach that has been called the *λ-tree syntax approach* [41]. A key part of this thesis is to show that this approach can help simplify the task of verified compilation for functional programs.

We devote this chapter to exposing the various aspects of the framework that we will make use of in the rest of the thesis. We start by describing λProlog and the logic it is based on and by introducing the methodology they support for implementing rule-based relational specifications in Section 2.1. We then discuss the logic underlying Abella for reasoning about relational specifications in Section 2.2. In Section 2.3, we introduce Abella and the two-level logic style of reasoning. The idea of λ-tree syntax will be implicit throughout the chapter. In the concluding section of the chapter, we discuss explicitly the benefits that can be derived in implementation and in reasoning by using λ-tree syntax.

2.1 The Specification Language

The λProlog language is based on a fragment of a first-order intuitionistic logic known as the logic of Hereditary Harrop formulas [51]. We will call this logic HH$^\omega$ here. HH$^\omega$ is suitable for encoding rule-based description of relations: rules for deriving relations translate naturally into logical formulas that yield the desired kind of proof-theoretic behavior in the logic. The formal systems that are of interest to us usually concern objects that embody binding structure. HH$^\omega$ supports the representation of such objects by using simply typed λ-terms, rather than the more commonly used first-order terms, as "data structures," *i.e.*, as the arguments of predicates. Further, HH$^\omega$ possesses logical capabilities for manipulating these terms in a way that respects and understands the abstraction operation they contain. HH$^\omega$ specifications can be given an operational interpretation and this is, in fact, what λProlog realizes as a programming language.

We expand on the remarks above in the subsections that follow. In the first subsection, we describe HH$^\omega$. We then explain the manner in which HH$^\omega$ can be used to encode relational specifications. Finally, we outline how the λProlog language allows us to execute such specifications.

2.1.1 The specification logic HH$^\omega$

The logic HH$^\omega$ is a fragment of Church's Simple Theory of Types [52]. The expressions in this logic are those of the simply typed λ-calculus or the 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:

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

In this thesis, we shall take the atomic types to correspond to a user-defined collection plus the distinguished type o that is used for formulas as explained below. We assume there to be 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 $\rightarrow$ associates to the right. For instance, $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ stands for $(\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3))$. Using this associativity convention, every type can be written in the form $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_0$ where τ_0 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 τ_0 is called its *target type.*

In building terms, we assume a vocabulary of constants and variables where each constant and variable has a type associated with it. Terms are then specified together with their types by the following inductive rules, assuming that t stands for terms:

- A constant or a variable of type τ is a term of type τ.

- If t is a term of type τ' and x is a variable of type τ, then the expression $(\lambda x{:}\tau.\, t)$ in is a term of type $\tau \rightarrow \tau'$. Such a term is referred to as an *abstraction* that has the variable x as its binder and t as its body or scope. Further, all occurrences of x in t that are not in the scope of any abstraction in t with x as its binder are considered bound by the abstraction.

- If t_1 is a term of type $\tau_1 \rightarrow \tau_2$ and t_2 is a term of type τ_1, then the expression $(t_1\ t_2)$ is a term of type of type τ_2. 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 uniquely 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 bound by any abstraction occurring in it. We will often need to indicate a variable x or a constant c together with its type τ. We do this by by writing $x : \tau$ or $c : \tau$. We will also need to talk about a term t with its type τ in a context where the constants and free variables in t and their types are given. For this we will use the following notation

$$\{c_1 : \tau_1, \ldots, c_m : \tau_m, x_1 : \tau_1', \ldots, x_n : \tau_n'\} \Vdash t : \tau$$

where $\{c_1, \ldots, c_m\}$ and $\{x_1, \ldots, x_n\}$ respectively contain all the constants and free variables in t.

The underlying logic assumes an equality relation between terms that is explained as follows:

- 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 free 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." A term of the form $((\lambda x.\, t_1)\ t_2)$ is referred to as a β-redex. 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 free in t as an η-redex. A term u η-contracts to v if v can be obtained from u by replacing such a η-redex

$(\lambda x.\, t\ x)$ in u by t. Conversely, v η-expands to u if u η-contracts to v. Finally, λ-conversion is the reflexive and transitive closure of the 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 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 in the STLC 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. A substitution, denoted by θ, is a type-preserving mapping from variables to terms that is the identity at all but a finite number of variables. We write $(t_1/x_1, \ldots, t_n/x_n)$ for the substitution that maps $x_1, \ldots, x_n$ to $t_1, \ldots, t_n$, respectively and each of the remaining variables to itself. Given such a substitution, the set of variables $\{x_1, \ldots, x_n\}$ is called its domain and each of the terms in $t_1, \ldots, t_n$ is called the value of the mapping on the variable it corresponds to. We write $dom(\theta)$ for the domain of θ. Substitutions can be lifted to be mappings on terms rather than just on variables. For this lifting to be logically correct, it must be defined in such a way that it is "capture-avoiding." One way to formalize the idea to ensure this is as follows. Given a term t and a substitution θ such that $\theta = (t_1/x_1, \ldots, t_n/x_n)$, the expression $t[\theta]$ denotes the term $((\lambda x_1. \ldots \lambda x_n. t)\ t_1\ \ldots\ t_n)$; observe that this term is equal under λ-conversion to t with each variable x_i replaced by t_i with relevant renamings done within t to avoid inadvertent capture of free variables in t_i.

The composition of two substitutions θ and ρ, denoted by $\theta \circ \rho$, is defined as follows:

$$(\theta \circ \rho)(x) \quad := \quad (\theta(x))[\rho] \qquad x \in dom(\theta)$$

$$(\theta \circ \rho)(x) \quad := \quad \rho(x) \qquad\qquad \text{otherwise.}$$

It is easy to check that $t[\theta][\rho] = t[\theta \circ \rho]$ for any t. Thus, the application of a composition of substitutions to a term corresponds, as one might expect, to their composition as mappings on terms.

The definition of λ-terms relies on a collection of constants. In the context of HH^ω, we distinguish between *logical* constants and *non-logical* constants. The set of logical constants is fixed and consists of *true* of type o, $\&$ and $\Rightarrow$, both of type $\mathsf{o} \to \mathsf{o} \to \mathsf{o}$, and, for each type τ that does not contain o, Π_τ of type $(\tau \to \mathsf{o}) \to \mathsf{o}$. The constants $\&$ and $\Rightarrow$, also called the logical connectives, correspond to conjunction and implication and are usually written as infix operators. The family of constants Π_τ represents universal quantifiers: the term $(\Pi_\tau\ (\lambda x{:}\tau.\, t))$ corresponds to the universal quantification of x over t. In writing this expression, we will often use the suggestive abbreviation $\Pi_\tau x.t$ and will also drop the type annotation—i.e. we will simply write $\Pi x.t$—when the type can be inferred or when its knowledge is not essential to the discussion. Furthermore, we will often abbreviate the formula $\Pi x_1 \ldots \Pi x_n.t$ to $\Pi x_1, \ldots, x_n.t$. Note that given any term $\Pi_\tau\ t$, its subterm t must have the $\beta\eta$-long normal form $\lambda x.\, t'$ for some t'. As a result, $\Pi_\tau\ t$ can always be represented as $\Pi_\tau x.t'$ for some t'.

The set of non-logical constants is also called a *signature*. We shall use the symbol Σ to denote signatures. There is a restriction on the constants allowed in a signature in HH^ω: their argument types must not contain the type o. A non-logical constant whose target type is 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; we shall call p the *predicate head* of the atomic formula.[1] We use the symbol A to denote atomic formulas.

The terms of type 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^ω 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:

[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.

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

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 the form $\Pi_{\tau_1} x_1. \ldots \Pi_{\tau_n} x_n.(G \Rightarrow A)$. We refer to A as the head of such a clause. Further, we call G the body of the clause.

In the intended use of the HH^ω 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 the notion of derivability through a sequent calculus [53]. In the context of interest, a sequent has the structure

$$\Sigma; \Gamma; \Delta \vdash G$$

where Σ is a signature, Γ is a multi-set of program clauses called the *static context* that contains user-defined program clauses, Δ is a multi-set of program clauses called the *dynamic context* that contains clauses dynamically added during the derivation of the sequent, and G is a goal formula called the *goal* of the sequent that is to be shown derivable from the static and dynamic contexts. A program given by the clauses Γ and the signature Σ then specifies the relations represented by all the atomic formulas A such that the sequent $\Sigma; \Gamma; \emptyset \vdash A$ is derivable. If we consider the sequents in a derivation from the perspective of how they arise in the course of searching for a proof, the dynamic context is initially empty. However, as we shall see presently, the process of constructing derivations may add clauses to this context and may also extend the signature.

The rules for deriving sequents of the form described are presented in Figure 2.1. These rules are of three kinds: the rule tR is used to finish the proof, the rules $\wedge R$, $\Rightarrow R$ and ΠR are used to simplify non-atomic goals and the rule *backchain* applies once the goal has been reduced to atomic form. In ΠR we use $(\Sigma, c : \tau)$ to denote $\Sigma \cup \{c : \tau\}$ and in $\Rightarrow$R we write (Δ, D) for $\Delta \cup \{D\}$. 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. These two rules are crucial for specifying systems involving binding operators, as we shall see in Section 2.1.3. The *backchain* rule captures 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.

$$\frac{}{\Sigma; \Gamma; \Delta \vdash \mathit{true}}\ tR \qquad \frac{\Sigma; \Gamma; \Delta \vdash G_1 \quad \Sigma; \Gamma; \Delta \vdash G_2}{\Sigma; \Gamma; \Delta \vdash G_1 \ \&\ G_2}\ \wedge R$$

$$\frac{\Sigma; \Gamma; \Delta, D \vdash G}{\Sigma; \Gamma; \Delta \vdash D \Rightarrow G}\ \Rightarrow R \qquad \frac{\Sigma, c : \tau; \Gamma; \Delta \vdash G[c/x]}{\Sigma; \Gamma; \Delta \vdash \Pi_\tau x.G}\ \Pi R$$

$$\text{(where } c \notin \Sigma\text{)}$$

$$\frac{\Pi_{\tau_1} x_1. \dots \Pi_{\tau_n} x_n.G \Rightarrow A' \in \Gamma \cup \Delta \quad \Sigma; \Gamma; \Delta \vdash G[t_1/x_1, \dots, t_n/x_n]}{\Sigma; \Gamma; \Delta \vdash A}\ \mathit{backchain}$$

$$\text{(where } \Sigma \Vdash t_i : \tau_i \text{ for } 1 \leq i \leq n \text{ and } A'[t_1/x_1, \dots, t_n/x_n] = A \text{ in } \mathit{backchain}\text{)}$$

Figure 2.1: Derivation Rules of HH$^\omega$

An important observation is that a derivation in HH$^\omega$ is guided by the syntactic form of its goal G. If G is true then only tR is applicable. If G is not an atomic formula or true, then exactly one of the $\Rightarrow R$, $\wedge R$ and ΠR rule is applicable. Those rules simplify the goal until it becomes atomic. At that point only *backchain* is applicable. The derivation continues by applying *backchain* which generates a subgoal. This process repeats until all subgoals are proved.

The λProlog language, which is a realization of HH$^\omega$, provides a concrete syntax in which a user can present HH$^\omega$ 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. 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 $(x \setminus M)$. Conjunction in goals is denoted by a comma: a goal of the form $(G_1 \ \&\ G_2)$ is written as (G_1, G_2). The clause $(G \Rightarrow D)$ is written in λProlog as $(D$ `:-` $G)$. Finally, when G is *true* the clause $G \Rightarrow D$ is further simplified to just D.

2.1.2 Encoding rule-based relational specifications

A natural way to present formal systems is to describe them via rules deriving relations on syntactic objects. As an example, we consider the rule-based descriptions of the append relation on lists of natural numbers. Let *nil* denote the empty list in our object language and let $::$ denote the constructor for lists such that $x :: l$ stands for a list where x is the first element of the list, called its head, and l is a list containing the rest of the elements, called its tail. If we limit our attention to lists that are constructed using *nil* and $::$ as the only constructors of list type, then we can define a ternary relation *append* given by the following rules:

$$\frac{}{\mathit{append}\ \mathit{nil}\ l\ l}\ \texttt{appd-nil} \qquad \frac{\mathit{append}\ l_1\ l_2\ l_3}{\mathit{append}\ (x :: l_1)\ l_2\ (x :: l_3)}\ \texttt{appd-cons}$$

The content of this definition is that *append* $l_1\ l_2\ l_3$ holds if and only if it can be derived using these rules. It is then not difficult to see that *append* $l_1\ l_2\ l_3$ holds just in the case that l_1, l_2 and l_3 are lists consisting of ground elements and l_3 contains the elements of l_1 followed by those of l_2.

Program clauses in HH^ω provide a natural way to capture rule-based specifications of relations: a relation can be represented by a predicate constant and each rule of the relation translates naturally to a program clause defining the predicate constant, with the conclusion of the rule becoming the head of the clause and the premises, if any, becoming its body.

We illustrate the above idea by considering the specification of the append relation. In encoding this specification, let us assume that we have designated the type `nat` to represent the type of natural numbers and `list` to represent the type of lists of natural numbers. Further, let us assume that we have introduced into the signature the constants $0, 1, 2, 3, \ldots$ of type `nat` to represent the natural numbers and the following constants to represent the list constructors:

$$\texttt{nil} : \texttt{list} \qquad\qquad \texttt{::} : \texttt{nat} \to \texttt{list} \to \texttt{list}$$

Mirroring the convention in the object language, we will write `::` in infix form also in the λProlog presentation and treat it as right-associative; the λProlog language provides users a way to present such conventions at the time when they identify a signature, but the details of this process are orthogonal to the present discussion. We then use the predicate symbol

$$\texttt{append} : \texttt{list} \to \texttt{list} \to \texttt{list} \to \texttt{o}$$

to represent the append relation and we encode the rules defining the relation in the following clauses where the first clause encodes the `appd-nil` rule and the second clause encodes the `appd-cons` rule:

$\texttt{append}\ \texttt{nil}\ L\ L.$
$\texttt{append}\ (X\ \texttt{::}\ L_1)\ L_2\ (X\ \texttt{::}\ L_3)\ \texttt{:-}\ \texttt{append}\ L_1\ L_2\ L_3.$

Let Γ be a static context consisting of the clauses for `append` shown above, let l_1, l_2 and l_3 be three lists, and let L_1, L_2 and L_3 be their encodings in λProlog. Then, showing that (*append* l_1 l_2 l_3) holds given the specification of *append* is equivalent to showing the following sequent is derivable, where Σ contains the constants we have introduced for encoding lists and the append relation:

$$\Sigma; \Gamma; \emptyset \vdash \texttt{append}\ L_1\ L_2\ L_3$$

Note that the way the program clauses of `append` would be used in constructing derivations for this sequent has a transparent relationship to the way in which the rules specifying *append* would be used in establishing that the relation (*append* l_1 l_2 l_3) holds: for instance, the attempt to construct a derivation would begin by matching ($\texttt{append}\ L_1\ L_2\ L_3$) with the head of one of the clauses for `append` (that corresponds to the conclusions of the rules in the object-language specification) and by reducing the task to deriving the corresponding body (that corresponds to the premise of the relevant rule). In this sense, the λProlog encoding also reflects the *derivation behavior* of the rule based specification of the append relation.

2.1.3 Encoding specifications over objects with binding structure

Our interest in this work is in treating specifications of formal systems such as logics, programming languages and compilers. 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.

To illustrate the different mechanisms mentioned above, we consider the task of encoding the typing relation for the simply-typed λ-calculus. Note that we are thinking of the STLC as an *object system* in this example. We will take the syntax of types and terms in this object system to be given by the following rules:

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

The typing relation that we want to encode is written as

$$\Gamma \vdash M : T$$

where M is a term, T is a type and Γ is a *typing context* that has the form

$$x_1 : T_1, \ldots, x_n : T_n$$

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

The only rule in this collection that is sensitive to binding structure is `t-abs`, the rule for typing abstractions. This rule asserts that the abstraction $\lambda x : T_1.M$ has the type $T_1 \to T_2$ in a typing context Γ if its body M can be given the type T_2 in a typing context that extends Γ with the type T_1 assigned to x, under the assumption, of course, that x is a variable that is not already assigned a type by Γ. This rule exhibits many features that are typical to rules that treat binding constructs that appear in formal objects. To encode rules such as this one, we need a mechanism for capturing recursion that is based on descending into the scope or body of the binding construct, we need to be able to enforce conditions on the variables we introduce to facilitate the recursion (in the case of `t-abs`, this variable, which is named x, must be fresh to Γ), and we need to be able to augment the context with assumptions about the freshly introduced variables (in the case of `t-abs`, x must be assumed to have type T_1). As we shall see below, λProlog provides devices for realizing all these aspects in a logically supported way.

To encode typing in the STLC in λProlog, we first need to represent the expressions it pertains to. 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. We then identify the following constants for encoding types and terms:

$$\begin{array}{ll} \texttt{a} : \texttt{ty} & \texttt{arr} : \texttt{ty} \to \texttt{ty} \to \texttt{ty} \\ \texttt{abs} : \texttt{ty} \to (\texttt{tm} \to \texttt{tm}) \to \texttt{tm} & \texttt{app} : \texttt{tm} \to \texttt{tm} \to \texttt{tm} \end{array}$$

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 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 by

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

The virtue of this kind of encoding is that it 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 (capture-avoiding) substitution over object language expressions.

We can now proceed to encoding the typing rules for the object system. Towards this end, we first introduce the predicate symbol

$$\texttt{of} : \texttt{tm} \to \texttt{ty} \to \texttt{o}$$

to represent the typing relation. Note that the typing context is not treated explicitly in this representation. Instead, it will be realized implicitly via the dynamic context of the HH^ω sequents we try to derive. Specifically, the typing context

$$x_1 : T_1, \ldots, x_n : T_n$$

will be encoded by the dynamic context

$$\texttt{of}\ x_1\ T'_1, \ldots, \texttt{of}\ x_n\ T'_n$$

where, for $1 \leq i \leq n$, T'_i is the encoding of T_i.

The typing rules `t-app` and `t-abs` translate into the following HH^ω clauses defining the `of` predicate:

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

Assuming Γ is the static context consisting of these clauses, then showing that the relation $\Delta \vdash M : T$ holds in the object system is equivalent to showing that the sequent

$$\Sigma; \Gamma, [\Delta] \vdash \texttt{of}\ [M]\ [T].$$

is derivable in HH$^\omega$. We write $[E]$ here to denote the encoding in λProlog of the object language expression E, which might be a typing context, a term or a type. We also assume that Σ contains the constants we have used to encode terms and types and all the constants we would have introduced in the course of an HH$^\omega$ derivation to represent the variables that are assigned types by $[\Delta]$ as we explain below.

It is interesting to note that the way the typing rules are used to establish $\Delta \vdash M : T$ in the object system is transparently related to the way the sequent encoding this judgment is derived in HH$^\omega$. To see this, first observe that application of the `t-var` rule is mirrored in the use of the *backchain* and tR rules to close off a particular path in the HH$^\omega$ derivation: if M is a variable x that is given the type T by the context Δ, there there must be a formula ($\texttt{of}\ x\ [T]$) in $[\Delta]$. It is also easy to see that application of the `t-app` rule is naturally captured by an application in HH$^\omega$ of *backchain* based on the clause that encodes `t-app`. Finally, the clause encoding `t-abs` shows how recursion over binding structure is realized by using universal and hypothetical goals and how substitution is modeled by β-conversion in λProlog. When $[M]$ is ($\texttt{abs}\ [t_1]\ (\lambda x.\,[m])$) and $[T]$ is ($\texttt{arr}\ [t_1]\ [t_2]$) for object language expressions t_1, t_2 and m of the relevant kinds, we would have to derive a sequent which has as its goal the formula

$$\texttt{of}\ (\texttt{abs}\ [t_1]\ (\lambda x.\,[m]))\ (\texttt{arr}\ [t_1]\ [t_2]).$$

The only way this sequent can be derived is by backchaining on the clause encoding `t-abs`. This would yield a sequent that has as its goal the formula

$$\Pi y.\texttt{of}\ y\ [t_1] \Rightarrow \texttt{of}\ ((\lambda x.\,[m])\ y)\ [t_2].$$

To derive this sequent in HH$^\omega$, we would have to introduce a new constant c and add the clause ($\texttt{of}\ c\ [t_1]$) to the dynamic context before trying to derive a sequent that has as its goal the formula ($\texttt{of}\ ((\lambda x.\,[m])\ c)\ [t_2]$). Observe here that the constant c has been introduced to represent the object language bound variable and that the “newness” of

c captures the freshness condition for this variable in the rule `t-abs`. The addition of (`of` c $[t_1]$) to the dynamic context of the sequent encodes the extension of the typing context in the premise of the `t-abs` rule and the λProlog term $((\lambda x.\, [m])\ c)$ is, modulo λ-conversion, a representation of the object language term m in which c is used to represent the variable bound by the abstraction we have descended under. It is easy to see from all this that the sequent we are left to prove corresponds precisely to the premise of the `t-abs` rule.

As an example, the following is a derivation of a typing relation using the typing rules

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : a \to a, y : a \vdash x : a \to a}\ \texttt{t-var} \qquad \dfrac{}{x : a \to a, y : a \vdash y : a}\ \texttt{t-var}}{x : a \to a, y : a \vdash x\ y : a}\ \texttt{t-app}}{x : a \to a \vdash (\lambda y{:}a.\, x\ y) : a \to a}\ \texttt{t-abs}}{\emptyset \vdash (\lambda x{:}a \to a.\, \lambda y{:}a.\, x\ y) : (a \to a) \to a \to a}\ \texttt{t-abs}$$

The corresponding derivation in HH$^\omega$ is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma'; \Gamma; \Delta' \vdash \mathit{true}}\ tR}{\Sigma'; \Gamma; \Delta' \vdash \texttt{of}\ x\ (\texttt{arr a a})}\ \mathit{backchain} \qquad \dfrac{\dfrac{}{\Sigma'; \Gamma; \Delta' \vdash \mathit{true}}\ tR}{\Sigma'; \Gamma; \Delta' \vdash \texttt{of}\ y\ \texttt{a}}\ \mathit{backchain}}{\Sigma, x : \texttt{tm}, y : \texttt{tm}; \Gamma; \texttt{of}\ x\ (\texttt{arr a a}), \texttt{of}\ y\ \texttt{a} \vdash \texttt{of}\ (\texttt{app}\ x\ y)\ \texttt{a}}\ \mathit{backchain}}{\Sigma, x : \texttt{tm}; \Gamma; \texttt{of}\ x\ (\texttt{arr a a}) \vdash \texttt{of}\ (\texttt{abs a}\ (y \setminus \texttt{app}\ x\ y))\ (\texttt{arr a a})}}{\Sigma; \Gamma; \emptyset \vdash \texttt{of}\ (\texttt{abs}\ (\texttt{arr a a})\ (x \setminus \texttt{abs a}\ (y \setminus \texttt{app}\ x\ y)))\ (\texttt{arr}\ (\texttt{arr a a})\ (\texttt{arr a a}))}$$

Here the double line stands for the successive application of *backchain* on the clause encoding `t-abs`, ΠR and $\Rightarrow R$, and Σ' and Δ' are abbreviations of $(\Sigma, x : \texttt{tm}, y : \texttt{tm})$ and $(\texttt{of}\ x\ (\texttt{arr a a}), \texttt{of}\ y\ \texttt{a})$, respectively. Notice how backchaining on program clauses followed by rules for simplifying the goal formula represents the application of the typing rules for non-variable terms and how the application of *backchain* followed by tR represents the application of the rule for typing variables. In general, it should be clear from this discussion that, under the encoding that we have used for the typing rules in the STLC, the derivations we would construct in HH$^\omega$ closely follow the structure of those in the STLC.

2.1.4 λProlog specifications as implementation

It should be clear from the discussion up to this point that the derivation rules for HH$^\omega$ that are shown in Figure 2.1 have an executable character: we proceed to find a derivation for a sequent with a complex goal formula by simplifying the formula and when we arrive at an atomic goal we look for a clause in the static or dynamic context whose head matches the goal. In the cases we have considered previously, the goals have all been closed; in this mode our objective has been limited to checking if a relation holds. However, we can think of extending this mechanism to "compute 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, letting T be a variable of the kind just described, 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 the STLC. The only solution to this "schematic" query is one where T is instantiated with the expression ($\texttt{arr}$ ($\texttt{arr a a}$) ($\texttt{arr a a}$)).

The λProlog language realizes this kind of an executable interpretation. Many useful applications have been shown to exist for this kind of "logic programming" interpretation for the language [8] and the Teyjus system [48] 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 to effect compilation using the Teyjus system.

2.2 A Logic of Fixed-Point Definitions

The logic HH$^\omega$ allows us to encode rule-based relational specifications in such a way that we can reason about what should hold in their context. Thus, using the formalization of typing for the STLC, we could show that the term $(\lambda x{:}a \to a.\, \lambda y{:}a.\, x\ y)$ has the type $(a \to a) \to a \to a$. However, HH$^\omega$ does not provide a means for capturing the fact that these encodings are complete in the sense that if a relation is not derivable from a program then it does not in fact hold. Rule-based specifications are usually intended to be interpreted in this way. For example, based on the specification of typing for the STLC, we might want to conclude that every term in it has a unique type. Implicit to proving this is the fact that a typing judgment of the form

$$\emptyset \vdash (\lambda x{:}a \to a.\, \lambda y{:}a.\, x\ y) : T$$

is *not* derivable for any type T other than $(a \to a) \to a \to a$.

The logic $\mathcal{G}$ has been designed to provide the kind of complete characterization of relational specifications that is discussed above. This logic is the end-point of a sequence of developments that started with work by McDowell and Miller about two decades ago [54, 55, 56, 57]. A defining characteristic of $\mathcal{G}$ is that it interprets atomic predicates as fixed-point definitions. These fixed-point definitions provide a means not only for showing that a relation holds, but also for *analyzing* why it holds. More specifically, they allow us to carry out reasoning based on case analysis that occurs often in proving properties of formal systems. Another important feature of $\mathcal{G}$ is a generic quantifier ∇ (pronounced as "nabla") that, amongst other things, requires us to provide a proof for the formula that it scopes over that is independent of the instance chosen for the quantified variable [56]. Coupled with fixed-point definitions, this quantifier enables us to use case analysis arguments over binding structure that is necessary in many tasks of reasoning about formal systems. Definitions of atomic predicates can also be given a *least fixed-point* interpretation, leading to the ability to reason about these predicates in an inductive fashion. These different capabilities give $\mathcal{G}$ the ability to encode many different forms of arguments that we might want to carry out over relational specifications.

In the rest of this section, we expose the features of $\mathcal{G}$ that are alluded to above. Our presentation is intended only to make our use of the logic in explaining and constructing proofs in Abella understandable. A reader interested in a more complete description of $\mathcal{G}$ may consult [54].

2.2.1 The syntax of $\mathcal{G}$

The logic $\mathcal{G}$ is also based on an intuitionistic version of Church's Simple Theory of Types. The expressions of $\mathcal{G}$ are similar to that of HH^ω, *i.e.*, the STLC terms. One difference is that `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 $\top, \bot : \texttt{prop}$ 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 $\texttt{prop}$, 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^ω, we write $\wedge$, $\vee$ and $\supset$ as infix operators. Similarly, we abbreviate the expressions $(\forall_\tau\ (\lambda x{:}\tau.\, B))$ and $(\exists_\tau\ (\lambda x{:}\tau.\, B))$ by $(\forall_\tau x.B)$ and $(\exists_\tau x.B)$. We also drop the type annotations in abstractions and quantified formulas when they are irrelevant to the discussion or can be inferred uniquely. Furthermore, we will often abbreviate the formula $\forall x_1 \ldots \forall x_n.t$ $(\exists x_1 \ldots \exists x_n.t)$ to $\forall x_1, \ldots, x_n.t$ $(\exists x_1, \ldots, x_n.t)$ or $\forall \vec{x}.t$ $(\exists \vec{x}.t)$ where $\vec{x}$ represents the sequence of variables $x_1, \ldots, x_n$.

2.2.2 The generic quantifier ∇

Universal quantification in $\mathcal{G}$ has an *extensional* reading. That is, to prove $\forall x.B$ in $\mathcal{G}$ we have to prove $B[t/x]$ for each possible term t but these proofs could be different ones for different values of t. However, when describing specifications containing binding constructs, it is often desirable to interpret a statement such as "$B(x)$ holds for x" as $B(t)$ holds for every t for the same *structural* reasons that are independent of the choice of t.

To provide a means for capturing this alternative style of reasoning, $\mathcal{G}$ includes a special *generic* quantifier [56]. Specifically, it includes a constant $\nabla_\tau : (\tau \to \texttt{prop}) \to \texttt{prop}$ for every type τ that does not contain $\texttt{prop}$. As with other quantifiers, $\nabla_\tau x.B$ abbreviates $\nabla_\tau\ (\lambda x{:}\tau.\, B)$. The indexing type τ is omitted if its identity is not relevant to the discussion or can be inferred uniquely from the context. The logical meaning of the formula $\nabla_\tau x.B$ can be understood as follows: to construct a proof for it, we pick a new constant of type τ that does not appear in B and then prove the formula that results from instantiating x in B with this constant. The constants that are to be used in this way are called *nominal constants*. 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 $(a_1 = a_2 \supset \bot)$ where a_1 and a_2 are two different nominal constants is a theorem. Another important property of nominal constants is that their names have significance only in distinguishing between different nominal constants in a single formula. For example, given the predicate symbol p and distinct nominal constants a_1, a_2, a_3, and a_4, the formula $(p\ a_1\ a_2)$ is considered to be logically equivalent to $(p\ a_3\ a_4)$.

2.2.3 Formalizing provability in $\mathcal{G}$

The approach to treating ∇ outlined in Section 2.2.2 was first described in the logic LG^ω [57] and has been adopted by $\mathcal{G}$. Specifically, we assume that there are an infinite number of nominal constants at every type. The collection of all nominal constants is denoted by $\mathcal{C}$, which is disjoint from the collection of usual, non-nominal constants denoted by $\mathcal{K}$. We define the support of a formula or a term t as the set of nominal constants occurring in it, denoted by $supp(t)$. We define a permutation π of nominal constants as a bijection from $\mathcal{C}$ to $\mathcal{C}$. We write $\pi.B$ for the result of applying π to the formula B, which is defined as follows:

$$\pi.c := c \quad \text{where } c \in \mathcal{K} \qquad\qquad \pi.a := \pi(a) \quad \text{where } a \in \mathcal{C}$$
$$\pi.x := x \qquad\qquad \pi.(t_1\ t_2) = \pi.t_1\ \pi.t_2 \qquad\qquad \pi.(\lambda x.\, t) = \lambda x.\, (\pi.t)$$

We use the proposition $B \approx B'$ to denote that the formulas B and B' are equal modulo the permutation of nominal constants occurring in them, that is, there exists some π such that $\pi.B = B'$. Given the fact that we treat the names of nominal constants having scope only over individual terms, substitution must satisfy an extra condition to be logically correct: it should avoid confusion between the names of nominal constants in the terms being substituted and in the terms being substituted into. Specifically, $B[\theta]$ now stands for a formula obtained by first applying a permutation π which maps the nominal constants in B to nominal constants that do not occur in the values of θ and then applying the substitution θ on the resulting formula. Such substitution is ambiguous since the permutation for nominal constants is not unique. However, because formulas are considered to be logically equivalent modulo permutation of nominal constants, the ambiguity turns out to be harmless.

Provability in $\mathcal{G}$ is once again formalized via a sequent calculus. Sequents in this logic have the form

$$\Sigma : \Gamma \longrightarrow B$$

where Γ is a multi-set of formulas called the context of the sequent and the formulas in it are called the assumptions or hypotheses of the sequent, B is a formula called the conclusion of the sequent, and Σ is a signature containing the free variables in Γ and B. The intuitive interpretation of the sequent is that the conclusion B is provable from the set of assumptions in Γ.

The core rules defining provability in $\mathcal{G}$ are shown in Figure 2.2. The structure of these rules as well as the content of many of them should be clear from a familiarity with sequent calculus style formulations of intuitionistic logics. We therefore limit ourselves here to elaborating only those aspects that are peculiar to $\mathcal{G}$. The *id* and the *cut* rules in $\mathcal{G}$ differ from the more familiar versions in that they build in equality under the permutation of nominal constants. The $\nabla\mathcal{R}$ rule formalizes the interpretation of the ∇ quantifier that was described informally in Section 2.2.2; in this rule, a is a "new" nominal constant that does not occur in B. The $\nabla\mathcal{L}$ rule encodes the reading of a ∇-quantified formula as an assumption that follows naturally from what it means to prove such a formula. In the usual reading of universal quantification, to prove that a formula of the form $\forall_\tau x.B$ holds, it suffices to show that B holds for all instantiations of x. The standard formalization of this understanding is based on replacing x with a new variable, called an *eigenvariable*, and then showing that the resulting formula holds no matter what actual value is chosen for that variable. In using this idea in the context of $\mathcal{G}$, we have to be careful to respect the scope of ∇ quantifiers: in particular, we must consider instantiations for the eigenvariable to include the nominal constants already appearing in B but not the ones that may be introduced for ∇ quantifiers appearing within B. This requirement is encoded in $\forall\mathcal{R}$ by using a technique called *raising* [58]: an eigenvariable h is introduced, x is replaced with the application of h to the nominal constants in the support of B, and we only consider instantiations for eigenvariables that do not contain nominal constants. Observe that instances of the quantified variable that use the permitted nominal constants (but not the disallowed ones) can be obtained by substituting a term for the eigenvariable that uses its arguments in a suitable way. The natural counterpart to this interpretation of universal quantification is that to prove a sequent in which $\forall_\tau x.B$ appears as an assumption, it suffices to show that the sequent is derivable when the quantified formula is instantiated by a term that perhaps contains the nominal constants appearing in B. The $\forall\mathcal{L}$ rule encapsulates this idea. In this rule, the typing judgment $\Sigma, \mathcal{K}, \mathcal{C} \Vdash t : \tau$ asserts that t is a term of type τ that is constructed using the variables, nominal constants and regular constants in Σ, $\mathcal{K}$ and $\mathcal{C}$; the nominal constants appearing in t could be further restricted to being ones in the support of B but, as shown in [54], the derivable sequents remain unchanged even without this restriction. Similar explanations can be given for the rules $\exists\mathcal{L}$ and $\exists\mathcal{R}$ which are, respectively, the duals of $\forall\mathcal{R}$ and $\forall\mathcal{L}$.

2.2.4 An informal understanding of fixed-point definitions

The logic $\mathcal{G}$ is actually better thought of as a *family* of logics, each parameterized by a *definition* of the predicate symbols in the vocabulary. Such a definition is given by a possibly infinite collection of *definitional clauses*. In the simplest form, each such clause has the structure[2]

$$\forall \vec{x}.A \triangleq B$$

where A is an atomic formula and B is an arbitrary formula. We call A the head of such a clause and we call B its body. Further, if p is the predicate head of A we say that the clause is *for* p. There are actually some provisos on the form of A and B for such a clause to be considered acceptable. First, neither of them should contain nominal constants. Second, every variable that has a free occurrence in B must also occur in A and all of the free variables of A should appear in $\vec{x}$. We shall think of a definition as consisting of a sequence of blocks of definitional clauses with the requirement that all the clauses for any given predicate symbol be confined to one block. In this context, a further requirement is that all the predicate symbols appearing in the body of a definitional clause must have have their own definitional clauses in the current or preceding blocks. Actually, occurrences of predicates whose definitional clauses appear in the present block must be further constrained to guarantee consistency of the logic. These constraints can be stated in a few different ways and are also somewhat complicated to describe. We

[2]The full form of a definitional clause permits the head of the clause to have ∇ quantifiers over an atomic formula, as we shall explain later in this subsection.

$$\frac{B \approx B'}{\Sigma : \Gamma, B \longrightarrow B'}\ id \qquad \frac{\Sigma : \Gamma \longrightarrow B \quad B \approx B' \quad \Sigma : \Gamma, B' \longrightarrow C}{\Sigma : \Gamma \longrightarrow C}\ cut$$

$$\frac{\Sigma : \Gamma, C, C \longrightarrow B}{\Sigma : \Gamma, C \longrightarrow B}\ c\mathcal{L} \qquad \frac{}{\Sigma : \Gamma, \bot \longrightarrow B}\ \bot\mathcal{L} \qquad \frac{}{\Sigma : \Gamma \longrightarrow \top}\ \top\mathcal{R}$$

$$\frac{\Sigma : \Gamma, B_i \longrightarrow C}{\Sigma : \Gamma, B_1 \wedge B_2 \longrightarrow C}\ \wedge\mathcal{L}, i \in \{1,2\} \qquad \frac{\Sigma : \Gamma \longrightarrow B \quad \Sigma : \Gamma \longrightarrow C}{\Sigma : \Gamma \longrightarrow B \wedge C}\ \wedge\mathcal{R}$$

$$\frac{\Sigma : \Gamma, B \longrightarrow D \quad \Sigma : \Gamma, C \longrightarrow D}{\Sigma : \Gamma, B \vee C \longrightarrow D}\ \vee\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow B_i}{\Sigma : \Gamma \longrightarrow B_1 \vee B_2}\ \vee\mathcal{R}, i \in \{1,2\}$$

$$\frac{\Sigma : \Gamma \longrightarrow B \quad \Sigma : \Gamma, C \longrightarrow D}{\Sigma : \Gamma, B \supset C \longrightarrow D}\ \supset\mathcal{L} \qquad \frac{\Sigma : \Gamma, B \longrightarrow C}{\Sigma : \Gamma \longrightarrow B \supset C}\ \supset\mathcal{R}$$

$$\frac{\Sigma, \mathcal{K}, \mathcal{C} \Vdash t : \tau \quad \Sigma : \Gamma, B[t/x] \longrightarrow C}{\Sigma : \Gamma, \forall_\tau x.B \longrightarrow C}\ \forall\mathcal{L} \qquad \frac{\Sigma, \mathcal{K}, \mathcal{C} \Vdash t : \tau \quad \Sigma : \Gamma \longrightarrow B[t/x]}{\Sigma : \Gamma \longrightarrow \exists_\tau x.B}\ \exists\mathcal{R}$$

$$\frac{\Sigma, h : \tau' : \Gamma, B[(h\ a_1\ \ldots\ a_n)/x] \longrightarrow C}{\Sigma : \Gamma, \exists_\tau x.B \longrightarrow C}\ \exists\mathcal{L} \qquad \frac{\Sigma, h : \tau' : \Gamma \longrightarrow B[(h\ a_1\ \ldots\ a_n)/x]}{\Sigma : \Gamma \longrightarrow \forall_\tau x.B}\ \forall\mathcal{R}$$

assuming that $supp(B) = \{a_1, \ldots, a_n\}$, that, for $1 \le i \le n$, a_i has type τ_i, h is variable of type $\tau_1 \to \ldots \to \tau_n \to \tau$ and $h \notin dom(\Sigma)$ in $\forall\mathcal{R}$ and $\exists\mathcal{L}$

$$\frac{\Sigma : \Gamma, B[a/x] \longrightarrow C}{\Sigma : \Gamma, \nabla_\tau x.B \longrightarrow C}\ \nabla\mathcal{L} \qquad \frac{\Sigma : \Gamma \longrightarrow B[a/x]}{\Sigma : \Gamma \longrightarrow \nabla_\tau x.B}\ \nabla\mathcal{R}$$

provided $a \notin supp(B)$ in $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$

Figure 2.2: The Core Rules of $\mathcal{G}$

therefore do not do this here but refer the interested reader to [59] or [60] for two alternatives. We note that all the definitions that we will use in the thesis will satisfy both forms of restrictions.

The informal understanding of a definition is that it assigns a meaning to each predicate symbol through the clauses it contains. This meaning is obtained intuitively by collecting all the clauses for a predicate and then thinking of any closed atomic formula that has that predicate as its head being true exactly when it is an instance of the head of one of the clauses and the corresponding instance of the body is true. This intuition can be formalized by describing rules for deriving a sequent in which an atomic formula appears as an assumption or as a conclusion in a sequent. In the latter case, it suffices to show that the body of an instance of any clause whose head is identical to the atomic formula follows from the same assumptions. In the former case, we consider all the possible ways in which the atomic predicate could be the head of an instance of a definitional clause and we show that the corresponding instance of the sequent with the atomic formula replaced by the body of the clause instance has a proof. Note that this treatment of an atomic formula that appears on the left of a sequent corresponds to a case analysis style of reasoning: we consider all the possible ways in which the atomic formula could be true based on the definition and we show that the sequent must be derivable in each case.

A simple example of a definition is one that encodes the equality relation between terms. For any given type τ not containing $\mathtt{prop}$, we identify a predicate $\mathtt{eq}_\tau : \tau \to \tau \to \mathtt{prop}$ and add to the definition the following sole clause for it:[3]

$$\mathtt{eq}_\tau\ M\ M \triangleq \top$$

In showing this and other clauses, we use the convention of making the outermost universal quantifiers in the clause implicit by choosing tokens that begin with uppercase letters for the occurrences of the variables they bind. Thus, in a fully explicit form, this clause would be written as

$$\forall M.\mathtt{eq}_\tau\ M\ M \triangleq \top.$$

By this definition, $\mathtt{eq}_\tau\ M_1\ M_2$ is provable if M_1 and M_2 are equal modulo λ-conversion. Conversely, if $\mathtt{eq}_\tau\ M_1\ M_2$ occurs as an assumption, then it must be the case that M_1 and M_2 are equal terms modulo λ-conversion. In the following discussion, we shall write $\mathtt{eq}_\tau\ M_1\ M_2$ as $M_1 =_\tau M_2$, as usual dropping the type annotation if it does not add to the discussion.

[3]As mentioned earlier, definitional clauses must be provided in blocks. That they are presented in this way will be implicit in most of this thesis with one exception: when we discuss the addition of a form of polymorphism to Abella in Chapter 3, we will need to make explicit use of the idea of a block of definitional clauses.

Fixed-point definitions provide a natural way to encode rule-based relational specifications in $\mathcal{G}$: predicate symbols are used to name relations and each rule translates into a definitional clause such that its conclusion becomes the head of the clause and its premises (if any) become the body of the clause. We use the append relation on lists of natural numbers again to illustrate this idea. As in the case of the encoding in HH$^\omega$, we use the atomic type `list` for representations of lists, the constants `nil` and `::` to construct such representations and the predicate symbol

$$\texttt{append} : \texttt{list} \to \texttt{list} \to \texttt{list} \to \texttt{prop}$$

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

$$\begin{array}{lcl} \texttt{append}\,\texttt{nil}\ L\ L & \triangleq & \top \\ \texttt{append}\ (X :: L_1)\ L_2\ (X :: L_3) & \triangleq & \texttt{append}\ L_1\ L_2\ L_3 \end{array}$$

To understand the way definitions are meant to be treated in $\mathcal{G}$, let us consider using the definition of `append` in derivations. First, suppose that we want to show that the following is a theorem, *i.e.*, that it is provable in an empty context:

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

To do this, we would have to show that the following holds, regardless of what actual term of type `list` we put in for l:

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

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$^\omega$ 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$^\omega$. The difference between the two logics is brought out by considering the formula

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

This formula states that $\texttt{append}\ (1::2::\texttt{nil})\ L\ (1::3::L)$ is false in the sense that it *must not* hold for any value of L. Such formulas that show the falsity of particular assumptions cannot be proved by using the "positive" interpretation of fixed-point definitions, *i.e.*, by considering how the assumption can be derived from the definitional clauses. However, they are provable in $\mathcal{G}$ because this logic also encodes the closed-world nature of fixed-point definitions. More specifically, the attempt to prove the particular formula at hand will reduce in $\mathcal{G}$ to showing that $\perp$ holds whenever we have $\texttt{append}\ (1::2::\texttt{nil})\ l\ (1::3::l)$ for any list l. This must the case for the following reason: the assumption formula cannot be true for *any* value of l because the elements of the list that is its first argument cannot be the initial elements of a list that is its third argument. This form of reasoning is realized in $\mathcal{G}$ 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 situation, 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 must not in fact be true. The desired theorem therefore follows.

The above style of reasoning works well in proving theorems when the "unfolding" of an assumption via definitions terminates in a finite number of steps. However, this property does not hold in many reasoning contexts that are of interest. For 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 in its third argument. Case analysis of either the first or the second assumption in this case exhibits a looping structure, reflecting the fact that the lists that we have to consider may be of *a priori* undetermined lengths. To prove this formula, we must actually know that the append predicate holds only by virtue of a *finite* number of unfoldings using the clauses and we must have a means for using this knowledge in an argument. It is possible to do this in $\mathcal{G}$ by giving a *least-fixed point* or inductive interpretation to the definition of append. The logic $\mathcal{G}$ allows definitions to be treated in this way. More specifically, we can mark the definition of `append` as inductive in $\mathcal{G}$, and this will give us the ability to apply an induction principle to the first or second assumption in the determinacy formula above. We will give a formal account of how this is done in Section 2.2.5 and will introduce an effective way to construct inductive proofs in Section 2.3.2.[4]

Like HH^ω, the logic $\mathcal{G}$ provides a set of mechanisms for encoding relational specifications over syntactic objects that embody binding constructs. Meta-level abstraction can be used as before to represent object-level binding operators and α- and β-conversions capture the binding related notions such as renaming and substitution. Further, the ∇ quantifier enables recursion over binding structure by providing a means for moving binding in term structure to formula level and, eventually to proof level binding. We use the formalization of the typing rules for the STLC in $\mathcal{G}$ as an example to illustrate these ideas. We use a representation for the terms in this calculus that is identical to the one described in Section 2.1.3. Since implication has a different reading in $\mathcal{G}$ from that in HH^ω, we cannot use it to treat typing contexts implicitly in the encoding of typing judgments. We therefore represent these judgments via the three-place relation

$$\mathtt{of} : \mathtt{clist} \to \mathtt{tm} \to \mathtt{ty} \to \mathtt{prop}$$

in which the first argument is intended to be an encoding of the typing context. We use the type `clelem` for classifying the type assignments for variables in typing contexts and the constant $\mathtt{vty} : \mathtt{tm} \to \mathtt{ty} \to \mathtt{clelem}$ for encoding such assignments. We then identify the following constants to represent the empty list and the cons operator for `clist`, respectively.

$$\mathtt{clnil} : \mathtt{clist} \qquad \mathtt{clcons} : \mathtt{clelem} \to \mathtt{clist} \to \mathtt{clist}$$

To encode type checking of a variable in a typing context, we identify a predicate constant $\mathtt{vof} : \mathtt{clist} \to \mathtt{tm} \to \mathtt{ty} \to \mathtt{prop}$. The clauses defining this checking is given as follows:

[4]There is also the dual possibility of giving the definition of particular predicates a *greatest-fixed point* or co-inductive interpretation. We do not use co-induction in this thesis and hence do not discuss it further here.

$$\begin{array}{lcl}
\texttt{vof}\ (\texttt{clcons}\ (\texttt{vty}\ X\ T)\ L)\ X\ T & \triangleq & \top \\
\texttt{vof}\ (\texttt{clcons}\ E\ L)\ X\ T & \triangleq & \texttt{vof}\ L\ X\ T
\end{array}$$

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

$$\begin{array}{lcl}
\texttt{of}\ L\ X\ T & \triangleq & \texttt{vof}\ L\ X\ T \\
\texttt{of}\ L\ (\texttt{app}\ M_1\ M_2)\ T_1 & \triangleq & \exists T_2.\texttt{of}\ L\ M_1\ (\texttt{arr}\ T_2\ T_1) \wedge \texttt{of}\ L\ M_2\ T_2 \\
\texttt{of}\ L\ (\texttt{abs}\ R)\ (\texttt{arr}\ T_1\ T_2) & \triangleq & \nabla x.\texttt{of}\ (\texttt{clcons}\ (\texttt{vty}\ x\ T_1)\ L)\ (R\ x)\ T_2
\end{array}$$

It is easy to see that the first and second clauses capture the typing rules for variables and applications. To see that the last rule captures the typing rule for abstractions, assume that we would like to prove the atomic formula $(\texttt{of}\ l\ (\texttt{abs}\ r)\ (\texttt{arr}\ t_1\ t_2))$ given particular encoded expressions l, r, t_1 and t_2. Then, by the definition of `of`, we must prove the following formula:

$$\nabla x.\texttt{of}\ (\texttt{clcons}\ (\texttt{vty}\ x\ t_1)\ l)\ (r\ x)\ t_2$$

The only way to do this is to introduce a new nominal constant a for x and prove the following formula:

$$\texttt{of}\ (\texttt{clcons}\ (\texttt{vty}\ a\ t_1)\ l)\ (r\ a)\ t_2$$

Like in HH$^\omega$, the β-redex $(r\ a)$ represents the result of substituting a (which represents the bound variable of the abstraction) for the occurrence of x in the body of the abstraction. The only way to prove $(\texttt{of}\ a\ t)$ is to match t with t_1 by using the first clause since the nominal constant a is different from both `app` and `abs` and it is therefore impossible to match $(\texttt{of}\ a\ t)$ with the second or the third clause. As a result, deriving the formula above is equivalent to showing that the body of the abstraction has type t_2 in the extended typing context, which matches exactly the behavior of the typing rule for abstractions.

As we have seen in the `append` example, we can prove properties of the relations through their encoding as fixed-point definitions in $\mathcal{G}$. The situation becomes more complicated when dealing with relational specifications with binding structure. Proving properties of such specifications often requires the ability to characterize the binding structure in them. To see this, consider proving the property that the typing rules for the STLC assign unique types to terms. Using the encoding of typing rules as described above, we may express this property through the formula

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

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 typing context. However, to prove the unique type assignment theorem it does not suffice that these properties are true. We also need to make their truth explicit so that we can exploit that knowledge in the argument.

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. Specifically, $\mathcal{G}$ allows the head of a definitional clause to include ∇ quantifiers over atomic formulas. Thus, the full form of definitional clauses is in fact

$$\forall \vec{x}.(\nabla \vec{z}.A) \triangleq B$$

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.

As an example of the use of this extended form of definitional clauses, consider the following clause defining the predicate constant $\texttt{name} : \texttt{tm} \to \texttt{prop}$:

$$\nabla x : \texttt{tm}.\texttt{name}\ x \triangleq \top.$$

An atomic formula $\texttt{name}\ M$ is derivable if and only if it matches with this clause. For this to be possible, M must be a nominal constant of type $\texttt{tm}$. As another example, consider the following clauses defining the predicate constant $\texttt{fresh} : \texttt{tm} \to \texttt{tm} \to \texttt{prop}$:

$$\nabla x : \texttt{tm}.\texttt{fresh}\ x\ M \triangleq \top.$$

The formula $\texttt{fresh}\ X\ M$ holds true if and only if X is a nominal constant and M is a term that does not contain this nominal constant. Thus, this second clause encodes the property of a nominal constant not occurring in, or being fresh to, a term.

Using clauses of this general form, we can characterize legitimate typing contexts using the following clauses for the predicate $\texttt{ctx} : \texttt{clist} \rightarrow \texttt{prop}$:

$$\begin{array}{lcl} \texttt{ctx}\ \texttt{clnil} & \triangleq & \top \\ \nabla x.\texttt{ctx}\ (\texttt{clcons}\ (\texttt{vty}\ x\ T)\ L) & \triangleq & \texttt{ctx}\ L \end{array}$$

The first clause asserts that an empty list encodes a valid typing context. In any instance of the second clause, x must be a nominal constant and the list L cannot contain the nominal constant for x since it is bound outside of x. Furthermore, L itself must encode a valid typing context. These clauses thus define a relation $\texttt{ctx}$ such that $\texttt{ctx}\ L$ 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

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

that is in fact provable by induction on the second or third assumption in $\mathcal{G}$. The detailed proof can be found in [11].

2.2.5 Formalizing fixed-point definitions

The informal exposure that we have provided to fixed-point definitions should suffice for most of the discussions in this thesis. However, we will need a more formal understanding of how these definitions are realized within $\mathcal{G}$ when we introduce a schematic polymorphism capability in reasoning into Abella in Chapter 3. To facilitate that discussion, we now present the proof rules that formalize the treatment of definitions in $\mathcal{G}$.

As might be expected, the treatment of definitions is characterized by rules for introducing atoms on the left and the right sides of sequents. The *definition left* or *def$\mathcal{L}$* rule captures the case analysis style reasoning based on the clauses defining the atom. The definition right rule or *def$\mathcal{R}$* captures backchaining on definitional clauses. To describe these rules formally, we need the notion of an instance of a definitional clause. Given a definitional clause $\forall \vec{x}.(\nabla \vec{z}.A) \triangleq B$ and a substitution θ that assigns distinct nominal constants to $\vec{z}$ and terms not containing such constants to $\vec{x}$, we say that $A[\theta] \triangleq B[\theta]$ is an instance of the original clause. Then the rules for definitions are shown in Figure 2.3. In *def*$\mathcal{L}$, $\Sigma\theta'$ stands for the signature obtained from Σ by removing variables in the domain of θ' and adding free variables in the values of θ'; $\Gamma[\theta']$ stands for the context $\{B[\theta'] \mid B \in \Gamma\}$.

$$\frac{\Sigma : \Gamma \longrightarrow B}{\Sigma : \Gamma \longrightarrow p\,\vec{t}}\ \mathit{def}\mathcal{R}$$

where $p\,\vec{t} \triangleq B$ is an instance of a definitional clause for p

$$\frac{\{\Sigma\theta' : \Gamma[\theta'], B \longrightarrow C[\theta'] \mid (p\,\vec{t})[\theta'] \approx A \text{ and } A \triangleq B \in \mathcal{D}_p\}}{\Sigma : \Gamma, p\,\vec{t} \longrightarrow C}\ \mathit{def}\mathcal{L}$$

where $\mathcal{D}_p = \{A \triangleq B \mid A \triangleq B \text{ is an instance of a definitional clause for } p\}$

Figure 2.3: Rules for Definitions

Note that the set of premises in any given instance of the *def*$\mathcal{L}$ rule could be infinite; this would be the case if there are an infinite number of instances of definitional clauses that match with the atomic formula $p\,\vec{t}$ in the manner indicated. In determining provability in practice, it would be useful to be able to cover this kind of infinite branching possibility in a finitary way. Towards this end, we can provide an alternative formulation of the *def*$\mathcal{L}$ rule that makes use of the idea of a *complete set of unifiers* or *CSU* between the heads of clauses and the atomic formula $p\,\vec{t}$.

To present this alternative formulation, we first make precise what is meant by a complete set of unifiers.

Definition 1. *Given two terms or formulas A and B, a complete set of unifiers for A and B, denoted by $CSU(A, B)$, is a set of substitutions such that*

- *for any $\theta \in CSU(A, B)$, $A[\theta] = B[\theta]$;*
- *for any substitution ρ such that $A[\rho] = B[\rho]$ there is a substitution $\theta \in CSU(A, B)$ and a substitution γ such that $\rho = \theta \circ \gamma$.*

Intuitively, a CSU for A and B consists of a set of substitutions that unify A and B and that "covers" all the unifiers of A and B in the sense that any such unifier can be obtained by further instantiating a substitution in the CSU. Note that there need not exist a unique set of substitutions satisfying this requirement and, in this sense, the notation $CSU(A, B)$ is ambiguous. However, this ambiguity will be harmless in the discussions below in the following sense: we will use the notation to select some complete set of unifiers for A and B and there will be no sensitivity to the choice that is actually made.

Looking at the *def$\mathcal{L}$* rule in Figure 2.3, we see that when matching an assumption $(p\ \vec{t})$ of the sequent with the clause $\forall\vec{x}.(\nabla\vec{z}.A) \triangleq B$, we must allow the variables in $\vec{x}$ to be instantiated with nominal constants in $(p\ \vec{t})$. Similarly, we must allow the variables in the sequent to be instantiated with nominal constants that we choose for $\vec{z}$. We use raising to build in both possibilities. First, given a clause $\forall x_1, \ldots, x_n.(\nabla\vec{z}.A) \triangleq B$, we define a version of it raised over the sequence of nominal constants $\vec{a}$ away from a signature Σ to be a clause of the form

$$\forall h_1, \ldots, h_n.\nabla\vec{z}.A[(h_1\ \vec{a})/x_1, \ldots, (h_n\ \vec{a})/x_n] \triangleq B[(h_1\ \vec{a})/x_1, \ldots, (h_n\ \vec{a})/x_n],$$

where $h_1, \ldots, h_n$ are variables that do not appear in Σ. Next, we define a version of the sequent $\Sigma : \Gamma \longrightarrow B$, where $\Sigma = \{y_1 : \tau_1, \ldots, y_m : \tau_m\}$, raised over the sequence of nominal constants $\vec{c}$ to be a sequent of the form

$$\Sigma' : \Gamma[(y'_1\ \vec{c})/y_1, \ldots, (y'_m\ \vec{c})/y_m] \longrightarrow B[(y'_1\ \vec{c})/y_1, \ldots, (y'_m\ \vec{c})/y_m]$$

where, for $1 \leq i \leq m$, y'_i is an variable of suitable type and $\Sigma' = \{y'_1, \ldots, y'_m\}$. Finally, we combine these notions together with the idea of complete sets of unifiers to identify a set of premises arising from a definitional definitional clause that is useful in formulating an alternative version of the *def$\mathcal{L}$* rule.

Definition 2. *Let H be the sequent $\Sigma : \Gamma, p\ \vec{t} \longrightarrow F$ and let C be the definitional clause $\forall\vec{x}.(\nabla\vec{z}.A) \triangleq B$. Further, let supp$(p\ \vec{t})$ be $\{\vec{a}\}$ and let $\vec{c}$ be a sequence of nominal constants that is of the same length as $\vec{z}$ and such that each constant in the sequence has a type identical to that of the corresponding variable in $\vec{z}$ and is also distinct from the constants in $\vec{a}$. Finally, let $\forall\vec{h}.(\nabla\vec{z}.A') \triangleq B'$ be a version of the clause C raised over $\vec{a}$ away from Σ and let $\Sigma' : \Gamma', p\ \vec{t'} \longrightarrow F'$ be a version of H raised over $\vec{c}$. Then*

$$\mathit{defl\_csu\_premise}(H, p\ \vec{t}, C) = \{\Sigma'[\theta] : \Gamma'[\theta], \pi.B'[\theta] \longrightarrow F'[\theta] \mid$$
$$\pi \textit{ is a permutation of the nominal constants in}$$

$$\{\vec{c}, \vec{a}\} \ \mathit{and}\ \theta \in CSU(p\ \vec{t'}, \pi.A'[\vec{c}/\vec{z}])\}.$$

Intuitively, $\mathit{defl\_csu\_premise}(H, p\ \vec{t}, C)$ corresponds to the premise sequents we would get from unfolding the assumption formula $(p\ \vec{t})$ in the sequent H based on the clause C and using only the substitutions in the complete set of unifiers for the head of C and $(p\ \vec{t})$. In the process, we have to instantiate the ∇ quantifiers in the head of C by nominal constants. Further, we have to take care to allow all possible substitutions for the universally quantified variables in C and the variables in Σ, an aspect that is treated by raising.

The definition of *defl_csu_premise* that we have presented is actually ambiguous: the set of sequents that it identifies is dependent on the the variables we select when raising the sequent and the clause, the names we use for the nominal constants that instantiate the ∇ quantifiers in the head of the clause and the particular complete set of unifiers we choose. The ambiguity arising from the last aspect will be harmless for the reasons already noted. The ambiguity arising from how we make the other choices is also inconsequential. It is easy to see that differences in these choices give rise to sets of sequents that are identical under a "renaming" of the variables and nominal constants. The inconsequentiality of how these choices are made then follows from an easily proved property of $\mathcal{G}$ that two sequents that differ only in the names of the variables and the nominal constants appearing in them are equi-derivable in a strong sense: their proofs have an identical structure and can in fact be obtained one from the other by using the same renaming.

We can finally present an alternative to the $\mathit{def}\mathcal{L}$ rule that uses the idea of complete sets of unifiers. This rule is shown in Figure 2.4. Using an approach similar to that in [55], this rule can be shown to be inter-admissible with the $\mathit{def}\mathcal{L}$ rule in the context of the other rules defining $\mathcal{G}$. As a result, we can replace $\mathit{def}\mathcal{L}$ with $\mathit{def}\mathcal{L}_{CSU}$ without affecting the provability of sequents in $\mathcal{G}$. If we limit ourselves to using $\mathit{def}\mathcal{L}_{CSU}$ only when the CSU is finite, then the proofs constructed by using the rule have a finite branching character. It turns out that we can construct proofs in many interesting situations under this limitation. In fact, all the examples that we consider in this thesis will use the $\mathit{def}\mathcal{L}_{CSU}$ with finite CSUs.

The last rule for definitions we consider is induction. This rule is shown in Figure 2.5; this rule is applicable only to those predicate that are specifically marked as being

$$\frac{\{H \in \mathit{defl\_csu\_premise}(\Sigma : \Gamma, p\,\vec{t} \longrightarrow F, p\,\vec{t}, C) \mid C \in \mathcal{D}\}}{\Sigma : \Gamma, p\,\vec{t} \longrightarrow F}\ \mathit{def}\mathcal{L}_{CSU}$$

where $\mathcal{D}$ is the definition parameterizing $\mathcal{G}$

Figure 2.4: The Definition Left Rule using CSU

inductively defined, as described in the previous subsection. The intuition underlying this rule is the following. If some property S satisfies the clauses that define p then $S\ t$ must hold whenever $p\ t$ holds; this follows by virtue of p being the least fixed-point of its defining clauses. But then if some formula C follows from assuming $S\ t$, it must also follow any time $p\ t$ holds.

Since the property S can be very verbose, it is not very convenient from a user's perspective to use $\mathcal{IL}$ directly in proof construction. We will introduce a more natural approach to do induction in Section 2.3.2. This approach is what we use for constructing proofs via induction in this thesis.

$$\frac{\{\vec{x} : B[S/p] \longrightarrow \nabla\vec{z}.S\ \vec{t_i} \mid \forall\vec{x}.\nabla\vec{z}.p\ \vec{t} \triangleq B \in \mathcal{D}\} \quad \Sigma : \Gamma, S\ \vec{t} \longrightarrow C}{\Sigma : \Gamma, p\ \vec{t} \longrightarrow C}\ \mathcal{IL}$$

provided p is inductively defined by the set of clauses $\mathcal{D}$
and S is a term with no nominal constants and of the same type as p

Figure 2.5: The Induction Rule

2.3 The Abella Theorem Proving System

Abella is a tactics-based interactive theorem prover like Coq [19] or Isabelle [17] that helps in the construction of proofs in $\mathcal{G}$. A user types in commands known as *tactics* to incrementally build up proofs for the theorems she/he want to prove. Tactics are designed to correspond to reasoning steps that are more natural to mathematical arguments but that, at the same time, can translate into a combination of proof rules of $\mathcal{G}$. Thus, with the help of tactics, developing a proof in Abella can be made to have a flavor similar to developing proofs on paper.

We would like to use Abella to prove properties of specifications written in λProlog. This is realized via the *two-level logic approach* to reasoning [49, 50]. In this approach, the logic of λProlog is itself encoded as a fixed-point definition in $\mathcal{G}$ and the λProlog specifications are then reasoned about through this encoding. With the two-level logic approach, we can use λProlog programs both as implementations and as inputs to Abella in which their properties are stated and proved.

We will introduce these features of Abella in the rest of this section. We successively describe the interactive approach to proof construction in Abella, an effective way to construct induction proofs by using tactics and the two-level logic approach to reasoning about λProlog specifications.

2.3.1 Interactive proof construction

In Abella, a user initiates a proof construction process by stating a formula, called a theorem or a lemma, he/she wants to prove. At any moment, the proof state is represented by a set of *subgoals* all of which must be proved to prove the original theorem. A subgoal essentially represents a sequent whose proof is still to be found. It consists of a multi-set of formulas called its *hypotheses* or *assumptions*, which constitute the context of the corresponding sequent, and a formula called its *conclusion*, which corresponds to the conclusion of the sequent. Initially, there is only one subgoal with no hypotheses and whose conclusion is the theorem to be proved.

At any moment, a user applies some tactic to one of the subgoals to make progress to the proof state.[5] A tactic corresponds to a scheme for applying a collection of rules in $\mathcal{G}$. The changes to the proof state depend on the applied tactic. The following are some examples: the subgoal may be proved by the tactic and disappear, new hypotheses may be added to the subgoal, the subgoal may be replaced by several other subgoals as a result of case analysis, or a warning that the tactic is not applicable may be issued and the subgoal remains unchanged. These changes reflect the effects of applying the proof rules represented by the tactic on the sequent corresponding to the subgoal. The proof construction process ends when there are no subgoals left. At that point, we have

[5]In practice, the subgoals are listed in sequence. The first subgoal in the sequence must be proved before a user can move on to the next one. This continues until there is no subgoal left.

essentially constructed a proof for the original theorem in $\mathcal{G}$.

As an example, consider the following theorem whose proof we have informally described in Section 2.2.4

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

This theorem is proved by using tactics as follows. Initially, we have a single subgoal with the formula above as its conclusion and with no hypothesis. We first apply an "introduction" tactic to this subgoal which introduces the variable L and the following hypothesis

$$\texttt{append}\ (1 :: 2 :: \texttt{nil})\ L\ (1 :: 3 :: L)$$

and makes $\bot$ as the new conclusion; application of this tactic mirrors the application of the $\forall\mathcal{R}$ and $\supset\mathcal{R}$ rules to the corresponding sequent. We then apply the "case analysis" tactic to this hypothesis, which corresponds to applying the *def*$\mathcal{L}$ rule to the corresponding sequent. Since the hypothesis can only be derived from the second clause for `append`, case analysis replaces the subgoal with a new one whose hypothesis is reduced to

$$\texttt{append}\ (2 :: \texttt{nil})\ L\ (3 :: L)$$

and whose conclusion is $\bot$. Applying another case analysis tactic to this reduced hypothesis results in no more subgoals because there are no clauses whose heads match the hypothesis. Since there are no subgoals to be solved, the proof is concluded.

A common approach to constructing a large proof is to break it down into smaller lemmas and build towards a final result. Abella provides this capability by allowing for proving theorems separately and using them in proof construction as lemmas. An established theorem can be freely used as a hypothesis at any point of proof construction. Such usage of a theorem corresponds to applying the *cut* rule to introduce the theorem as a new hypothesis.

Theorems in Abella often have the form

$$\forall x_1, \ldots, x_n.H_1 \supset \ldots \supset H_m \supset B$$

Suppose we have such a theorem, we can then use the "apply" tactic to match H_i with some hypotheses to generate a new hypothesis B under proper instantiation of x_i. This corresponds to application of a collection of $\forall\mathcal{R}$, $\supset\mathcal{L}$ and *id* rules [40]. For example, consider the following theorem:

$$\texttt{append}\,(1 :: 2 :: \texttt{nil})\,(4 :: \texttt{nil})\,(1 :: 3 :: 4 :: \texttt{nil}) \supset \bot.$$

It is proved as follows. We first introduce the following hypothesis:

$$\texttt{append}\,(1 :: 2 :: \texttt{nil})\,(4 :: \texttt{nil})\,(1 :: 3 :: 4 :: \texttt{nil})$$

and try to prove the conclusion $\bot$. We then apply the following theorem

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

which has been proved above, with L substituted by $(4 :: nil)$ to match the above hypothesis and generate $\bot$ as a new hypothesis. The proof is concluded by matching the conclusion $\bot$ with the generated hypothesis, which corresponds to applying the *id* rule.

Later we will see theorems of the following form in which the top-level universal quantifiers embed some ∇ quantifiers which further embed an implication formula:

$$\forall x_1, \ldots, x_n.\nabla z_1, \ldots, z_m.H_1 \supset \ldots \supset H_k \supset B.$$

We can apply such theorems in a way similar to what we have described above. The only important point to note is that $z_1, \ldots, z_m$ must be instantiated with distinct nominal constants and $x_1, \ldots, x_n$ must be instantiated with terms that do not contain the nominal constants for $z_1, \ldots, z_m$.

2.3.2 An annotated style of induction

Up to now we have only discussed interactive proofs with case analysis that has a finite structure. When case analysis has a looping structure, we need induction to complete the proof. As we have described in Sections 2.2.4 and 2.2.5, $\mathcal{G}$ supports inductive reasoning by interpreting definitions as least fixed-points and by providing a rule to perform induction on inductively defined predicate symbols. However, the induction rule is not easy to use since it requires explicitly identifying and using an inductive invariant, which can be rather complicated, in proof construction.

To solve this problem, Abella implements an annotated style of induction that, on the one hand, mimics how we perform inductive reasoning in practice and, on the other hand, translates into a sequence of rule applications in $\mathcal{G}$ that includes the induction rule [40]. The annotated style of induction works as follows. 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 inductively defined predicate. We can choose to prove this formula inductively by applying the "induction" tactic with respect to A. 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 and changes the formula to be proved to the following

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

We can now advance proof search by introducing the variables $x_1, \ldots, x_m$ 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.

As an example, consider proving the theorem that the inductive predicate `append` is deterministic, something we considered in Section 2.2.4:

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

The proof is by induction on the first assumption. More specifically, we apply the induction tactic to 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 subgoal into one with the following hypotheses respectively named H1 and H2:

$\texttt{H1}:\quad \texttt{append}\ L_1\ L_2\ L_3^{@}$
$\texttt{H2}:\quad \texttt{append}\ L_1\ L_2\ L_3'$

and with $L_3 = L_3'$ as the conclusion we want to show; L_1, L_2 and L_3 are variables here. We then analyze the possible cases of H1 as follows:

- It holds by virtue of 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 holds because of 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

  $\texttt{H3}:\quad \texttt{append}\ L_1'\ L_2\ L_3''^*$

  holds. Note the changed annotation on H3, corresponding to the fact that it is obtained by an unfolding of the definition. 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

  $\texttt{H4}:\quad \texttt{append}\ L_1'\ L_2\ L_3'''$

  to the hypothesis 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 we can see from this example, inductive reasoning using the annotated style is very similar to what we would have done on paper.

Abella also supports the annotated style of induction for theorems of the following form:

$$\forall x_1, \ldots, x_m.\nabla z_1, \ldots, z_k.F_1 \supset \ldots \supset A \supset \ldots \supset F_n \supset B$$

in a way similar to what we have described above.

2.3.3 The two-level logic approach to reasoning

We are eventually interested in reasoning about the specifications of compiler transformations. One possible approach might be to encode these specifications via fixed-point definitions and to reason based on this encoding. The problem with this approach is that it would not then yield a correctness proof of an *implementation* of the transformations.

To obtain a proof of a specification that is also an implementation, we use the so-called *two-level logic approach* to reasoning [49, 50]. In the setting of Abella, this approach translates into the following. We continue to write specifications in HH^ω. What we also do is encode HH^ω into a fixed-point definition in $\mathcal{G}$ that captures the derivability relation of HH^ω. We then lift specifications in HH^ω into $\mathcal{G}$ through the encoding. Finally, we reason about the HH^ω specifications using Abella via the translation. This style of reasoning yields the desired goal: we end up proving properties about actual, executable programs in λProlog. An auxiliary benefit to the approach comes from the fact that HH^ω 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 in fact specifically structured to support the two-level logic approach described above. Prior to the work in this thesis, Abella actually encoded a weaker logic than HH^ω called HH^2. The difference between HH^2 and HH^ω is that the syntax of goal formulas is limited in the former: specifically, the antecedents of implications in such formulas are required to be atomic. In other words, the syntax of goal formulas in HH^2 is given by the following rule:

$$G \quad ::= \quad \mathit{true} \mid A \mid G \,\&\, G \mid A \Rightarrow G \mid \Pi_\tau x.G$$

As a result of this restriction, the dynamic program context is limited to being a collection of atomic formulas. The initial reason for restricting attention to the HH^2 specification logic was that it is simpler to realize the two-level logic approach with this kind of a dynamic program context. This restriction also turned out to be acceptable because a large class of rule-based relational specifications naturally fall within the structure of HH^2. We shall describe the realization of two-level logic approach with respect to HH^2 in this section. A contribution of this thesis is that it extends the two-level logic approach to deal with the full HH$^\omega$ specifications. We discuss this extension in Chapter 3.

The embedding HH^2 within Abella is achieved by encoding the derivability relation of HH^2 in a fixed-point definition. In order to exploit the meta-logical properties of the HH^2 logic in reasoning, we want to be able to prove theorems about the derivability relation in Abella. These proofs typically rely on a measure of the size of a derivation. To facilitate their construction, we parameterize the encoding of the derivability relation by a natural number. Concretely, we identify `nat` as the type of natural numbers that we encode using the constants `z` : `nat` for representing 0 and `s` : `nat` $\to$ `nat` for representing the successor function. The inductive nature of this representation of natural numbers is formalized by introducing the predicate `nat` : `nat` $\to$ `prop` that is defined as the least fixed point of the following clauses:

$$\begin{array}{lcl} \texttt{nat}\;\texttt{z} & \triangleq & \top \\ \texttt{nat}\;(\texttt{s}\;N) & \triangleq & \texttt{nat}\;N \end{array}$$

Finally, we use the predicate symbol `seq` : `nat` $\to$ `olist` $\to$ `o` $\to$ `prop` to encode the derivability of HH^2. Here, `olist` is a type that is reserved for lists of HH^2 formulas and such lists are constructed using the constants `nil` and `::`. An HH^2 sequent $\Sigma; \Gamma; \Delta \vdash G$ is encoded as $(\texttt{seq}\;N\;\Delta\;G)$; intuitively, this predicate holds if the corresponding sequent has a derivation whose size is at most N.

An aspect to be noted about the encoding of HH^2 sequents is that it treats the static context and the signature implicitly. How exactly this implicit treatment is realized will become clear when we present an actual fixed-point definition for `seq` in Chapter 3. However, we sketch the general idea that is used here. A development in Abella that is about an HH^2 specification begins with loading in that specification. This loading action fixes the static context for the development and Abella manages the use of this context separately. With regard to the signature, one part of it arises from the HH^2 specification that is loaded. Abella treats this part by adding it directly to is own vocabulary of non-logical constants. The other, dynamic part of the signature arises from the treatment of universal goals in proof search. The encoding of derivability in HH^2 handles universally quantified goals in HH^2 by transforming them into formula-level ∇ quantifiers in $\mathcal{G}$. The "constants" that are added to the signature when these ∇ quantifiers are processed therefore end up being represented by nominal constants and so there is already a means for identifying them in the proof development. It has been shown in [40] that ($\texttt{seq}\, N\ \Delta\ G$) is derivable for some N in $\mathcal{G}$ if and only if $\Sigma; \Gamma; \Delta \vdash G$ is derivable in HH^2. This result provides the basis for our thinking of properties of our encoding directly as properties of derivability in HH^2.

We often want to prove properties of HH^2 derivations based on an induction on their sizes. Towards this end, we use the notation $\{\Delta \vdash G\}$ introduced by [40] to represent the formula $\exists N.\texttt{nat}\ N \wedge \texttt{seq}\ N\ \Delta\ G$ in the propositions we want to prove in Abella. We abbreviate $\{\vdash G\}$ as $\{G\}$. Given a formula T of the form

$$\forall x_1, \ldots, x_m.F_1 \supset \ldots \supset \{\Delta \vdash G\} \supset \ldots \supset F_n \supset B.$$

applying the induction tactic on $\{\Delta \vdash G\}$ is equivalent to first transforming T into the following logically equivalent formula:

$$\forall x_1, \ldots, x_m, N.F_1 \supset \ldots \supset \texttt{nat}\ N \supset \texttt{seq}\, N\ \Delta\ G \supset \ldots \supset F_n \supset B$$

and then applying the induction tactic with respect to ($\texttt{nat}\ N$). The annotated style of induction is extended to represent this process in a user-friendly way. Specifically, induction on $\{\Delta \vdash G\}$ in T introduces the following inductive hypothesis

$$\forall x_1, \ldots, x_m.F_1 \supset \ldots \supset \{\Delta \vdash G\}^* \supset \ldots \supset F_n \supset B$$

which represents

$$\forall x_1, \ldots, x_m, N.F_1 \supset \ldots \supset (\texttt{nat}\ N)^* \supset \texttt{seq}\, N\ \Delta\ G \supset \ldots \supset F_n \supset B$$

and changes the conclusion to

$$\forall x_1, \ldots, x_m.F_1 \supset \ldots \supset \{\Delta \vdash G\}^{@} \supset \ldots \supset F_n \supset B$$

which represents

$$\forall x_1, \ldots, x_m, N.F_1 \supset \ldots \supset (\texttt{nat}\ N)^{@} \supset \texttt{seq}\, N\ \Delta\ G \supset \ldots \supset F_n \supset B.$$

When the notation $\{\Delta \vdash G\}$ is used as a hypothesis, it represents the packet of formulas $\texttt{nat}\ N$ and $\texttt{seq}\ N\ \Delta\ G$ for some variable N. This representation coincides with the fact that applying the proof rules to eliminate the logical constants in the hypothesis $\exists N.\texttt{nat}\ N \wedge \texttt{seq}\ N\ \Delta\ G$ will result in the packet we just described. When this notation is used with an annotation in a hypothesis, *i.e.*, when the hypothesis has the form $\{\Delta \vdash G\}^{\Box}$ with $\Box$ being either @ or $*$, then it represents $(\texttt{nat}\ N)^{\Box}$ and $\texttt{seq}\ N\ \Delta\ G$ for some N. Case analysis on $\{\Delta \vdash G\}$ treats the packet as a whole. Specifically, it has the effect of first analyzing the cases of $\texttt{seq}\ N\ \Delta\ G$ and then analyzing $\texttt{nat}\ N$ if N has been instantiated with $(\texttt{s}\ N')$ for some N'. As a result, case analysis on $\{\Delta \vdash G\}^{@}$ might produce branches that have hypotheses of the form $\{\Delta' \vdash G'\}^{*}$ for some Δ' and G'. The inductive hypothesis then becomes applicable to such hypotheses.

To illustrate how we can state and prove properties about HH^2 specifications in Abella, suppose that we have included into an Abella development the program from Section 2.1.3 that specifies the typing rules for the STLC.[6] Assuming that L is a list of formulas that encode the dynamic typing context that arises in the relevant HH^2 derivation, it is the case that $(\texttt{of}\ M\ T)$ is derivable in HH^2 if and only if $\{L \vdash \texttt{of}\ M\ T\}$ is provable in $\mathcal{G}$.[7] We can actually characterize legitimate representations for typing contexts in $\mathcal{G}$ by using the predicate symbol $\texttt{ctx} : \texttt{olist} \rightarrow \texttt{prop}$ that is defined by the following clauses:

$$\begin{array}{lcl} \texttt{ctx}\ \texttt{nil} & \triangleq & \top \\ \nabla x.\texttt{ctx}\ (\texttt{of}\ x\ T :: L) & \triangleq & \texttt{ctx}\ L \end{array}$$

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

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

It is provable by induction on the assumption $\{L \vdash \texttt{of}\ M\ T\}$. A proof of this formula can be found, for instance, in the Abella tutorial [11].

[6]This program, which was presented as one in HH$^\omega$, is also one in HH^2. We use this fact systematically to refer to the development here as one related to HH^2 rather than in HH$^\omega$.

[7]We use a systematic abuse of notation here and elsewhere, confusing the items introduced into the signature by universal goals in HH^2 derivations with the nominal constants that arise in corresponding $\mathcal{G}$ derivations.

In the thesis work, we will use the two-level logic approach to verify compiler transformations implemented in λProlog. As we shall see in Chapters 5, 6, 7 and 8, 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 manner.

2.4 A Higher-Order Abstract Syntax Approach

In the previous sections of this chapter, we have exposed the support that both λProlog and Abella provide for the λ-tree syntax version of the HOAS approach. We have also provided some examples that should have motivated the benefits of this approach to representing, manipulating and reasoning about objects that embody binding constructs. In this section, we provide a few more directed examples that illustrate particular paradigms related to the use of the λ-tree syntax approach in programming and reasoning within the framework of interest. These paradigms will turn out to have repeated applications when we consider the implementation and verification of compilers for functional languages in later chapters in the thesis.

2.4.1 Analyzing object structure in specifications

In Section 2.1.3 we have explained the key idea underlying the representation of formal objects that underlies the use of the λ-tree approach in specifications within our framework: object-level binding translates into abstraction in the λProlog representation. We have also discussed several benefits that flow for this approach, including the fact that the programmer derives help from the meta-language in realizing operations such as bound variable renaming, the correct implementation of substitution and recursion over binding structure. One aspect that we have not explicitly touched upon that we will use repeatedly later is the capabilities of structure analysis that we get from unification in the meta-language.

We illustrate this capability by considering the encoding of η-conversion over the object-language of λ-terms. Suppose that we have designated the λProlog type `tm` for the representations of these terms and that we use the constants `app` : `tm` $\to$ `tm` $\to$ `tm` and `abs` : (`tm` $\rightarrow$ `tm`) $\rightarrow$ `tm` to encode the object language applications and abstractions. Now, let `eta` be a λProlog predicate symbol of type `tm` $\rightarrow$ `tm` $\rightarrow$ `o` and consider the following clause:

$$\texttt{eta}\ (\texttt{abs}\ (x \setminus \texttt{app}\ M\ x))\ M.$$

This clause succinctly captures the basic component of the object language η-conversion relation: the atomic formula `eta` $t\ t'$ is derivable from it exactly when t encodes an η-redex and t' encodes its contracted form. To see this, observe that the binding order requires that in generating an instance of the clause M must not be instantiated with a term that contains x. To derive `eta` $t\ t'$, we need to unify it with the clause. This unification must respect the constraint just described and therefore it will succeeds only if $t = (\texttt{abs}\ (x \setminus \texttt{app}\ M\ x))$ for some M not containing x and t is equal to M.

A closer examination of the example shows that we have used the idea of unification to tell us when a particular part of an object language expression does not depend on a bound variable. This and related forms of structure analysis will be a recurring theme in the specification of compiler transformations that we provide in later chapters. For example, this particular kind of structure analysis will play an important role in specifying the code hoisting transformation in Chapter 7.

2.4.2 Exploiting λ-tree syntax in reasoning

The ability to analyze the structure of terms via unification also plays an essential role in reasoning, specifically, in case analysis. In effect, the case analysis style of reasoning in Abella examines the possible ways an atomic formula can unify with the head of clauses that define it. This was manifest in the examples that make use of case analysis in Section 2.2.4.

By exploiting the logical structure of the λ-tree syntax representations and the rich reasoning capabilities provided by Abella, we are able to prove properties related to binding structure in an effective manner. We illustrate this point via the example of defining substitution as an explicit relation and proving properties about this relation. Assume that we are working with the STLC and the encoding of its syntax that we have already described. We now use the type `map` and the constant `map` : `tm` $\rightarrow$ `tm` $\rightarrow$ `map` to represent mappings from variables (encoded as nominal constants) to terms and a list of such mappings to represent a substitution. We use the Abella type $\mathtt{maplist}$ to represent the type of lists of mappings and the constants $\mathtt{mlnil} : \mathtt{maplist}$ and $\mathtt{mlcons} : \mathtt{map} \to \mathtt{maplist} \to \mathtt{maplist}$ as the constructors of such lists. Then the predicate $\mathtt{app\_subst} : \mathtt{maplist} \to \mathtt{tm} \to \mathtt{tm} \to \mathtt{prop}$ such that $\mathtt{app\_subst}\ S\ M\ M'$ holds exactly when M' is the result of applying the substitution S to M can be defined by the following clauses:

$$
\begin{array}{lcl}
\mathtt{app\_subst}\,\mathtt{mlnil}\ M\ M & \triangleq & \top \\
\nabla x.\mathtt{app\_subst}\ (\mathtt{mlcons}\ (\mathtt{map}\ x\ V)\ S)\ (R\ x)\ M & \triangleq & \mathtt{app\_subst}\ S\ (R\ V)\ M
\end{array}
$$

For obvious reasons, we shall take $\mathtt{app\_subst}$ to be inductively defined. The first clause encodes the fact that an empty substitution has no effect on a term. The second clause applies to a non-empty substitution that maps x to V and that has additional mappings given by S. This clause uses the pattern $(R\ x)$ to match the term to be substituted into. The quantification ordering ensures that R cannot contain x. For this reason, R must be bound to the result of abstracting over all the free occurrences of x in the "source term" for the substitution. The $(R\ V)$ then is equal modulo λ-conversion to the result of replacing all the free occurrences of x in the source term with V. The body of the clause represents the application of the remainder of the substitution to the resulting term.

This style of encoding of the substitution relation that takes advantage of the meta-level understanding of binding structure makes it extremely easy to prove structural properties of the relation. For example, the facts that substitution distributes over applications and abstractions can be stated as follows:

$$
\begin{array}{l}
\forall S, M_1, M_2, M'.\mathtt{app\_subst}\ S\ (\mathtt{app}\ M_1\ M_2)\ M' \supset \\
\qquad \exists M_1', M_2'.M' = \mathtt{app}\ M_1'\ M_2' \wedge \mathtt{app\_subst}\ S\ M_1\ M_1' \wedge \mathtt{app\_subst}\ S\ M_2\ M_2'. \\
\forall S, R, T, M'.\mathtt{app\_subst}\ S\ (\mathtt{abs}\ T\ R)\ M' \supset \\
\qquad \exists R'.M' = \mathtt{abs}\ T\ R' \wedge \nabla x.\mathtt{app\_subst}\ S\ (R\ x)\ (R'\ x).
\end{array}
$$

These properties are easily proved by induction on the only assumption, as follows. The base cases are immediately proved. In the inductive cases, we apply the induction hypothesis to the result of unfolding the first assumption to generate new hypotheses, from which the conclusion is immediately derived. The simplicity of these proofs comes from the fact that the actual substitution in the definition of `app_subst` is carried out via meta-level β-reduction. As a result, the properties above are just explicit statements of the corresponding properties of β-reduction which are immediately true given the meta-language.

Further benefits in reasoning about binding structure can be derived by combining definitions in Abella with the two-level logic approach. As an example, we may want to characterize relationships between closed terms and substitutions. For this, we can first define the well-formed STLC terms through the following HH$^\omega$ clauses:

$$\begin{array}{lll} \texttt{tm}\,(\texttt{app}\ M\ N) & \texttt{:-} & \texttt{tm}\ M\,,\texttt{tm}\ N. \\ \texttt{tm}\,(\texttt{abs}\ T\ R) & \texttt{:-} & \Pi x.\texttt{tm}\ x \Rightarrow \texttt{tm}\ (R\ x). \end{array}$$

This definition is similar to the definition of typing rules for the STLC except that `tm` does not record the type information of terms. We characterize the context used in `tm` derivations in Abella as follows:

$$\begin{array}{lll} \texttt{tm\_ctx}\ \texttt{nil} & & \\ \nabla x.\texttt{tm\_ctx}\ (\texttt{tm}\ x :: L) & \triangleq & \texttt{tm\_ctx}\ L. \end{array}$$

Intuitively, if `tm_ctx` L and $\{L \vdash \texttt{tm}\ M\}$ hold, then M is a well-formed term whose free variables are given by L. Clearly, if $\{\texttt{tm}\ M\}$ holds, then M is closed. We can also state a "pruning" property about a term that says that if a term is well-formed in a context that does not include a particular variable, then the variable does not occur in that term:

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

This formulation of the property of interest makes critical use of λ-tree syntax. The pattern $(M\ x)$ expresses the *possible* dependence of a term on the variable x but with M being the result of abstracting over *all* free occurrences of x in the term; this is the case because of the order of the quantification over M and x. Also because of the order of the quantifiers, L cannot contain x. The conclusion $\exists M'.M = y \setminus M'$ states that M is a vacuous abstraction since M' cannot contain any occurrences of the bound variable y because of the order of the binders. The reason why the conclusion must be true is that if `tm` $(M\ x)$ is derivable from L in the specification language, it must be the case that x

does not appear free in $(M\ x)$. As might be expected from this observation, the pruning property can be proved by an induction on the judgment $\{L \vdash \texttt{tm}\ (M\ x)\}$. The details of the argument are considerably simplified by using the meta-language treatment of binding.

As a special case of the pruning property, we derive that a closed term do not depend on any free variable.

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

Now we can state the fact that substitution has no effect on closed terms as follows:

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

This property is proved by an induction on the judgment $\texttt{app\_subst}\ S\ M\ M'$. In the inductive case, we make use of the pruning property that we have proved to discharge the dependence of the closed term M on variables in S, thereby discharging the effect of the substitution.

Chapter 3

Extensions to the Abella System

The framework consisting of λProlog and Abella is a powerful tool for specifying and reasoning about rule-based relational descriptions of formal systems. 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 our work on verified compilation. However, the Abella system is newer and is still evolving. In this chapter we describe some extensions to Abella that make it a more flexible and powerful reasoning tool. These extensions make it easier for Abella to carry out large-scale proof developments such as the verification compilation work we will describe in later chapters.

The extensions we have developed consist of two parts. The first part is a treatment of λProlog specifications in the full HH^ω. Previously, the two-level logic approach in Abella only supported reasoning over a subset of HH^ω, *i.e.*, HH^2. We have given a full encoding of HH^ω in $\mathcal{G}$ and developed a methodology to reason about HH^ω specifications through this encoding. This methodology allows us to reason about rule-based relational specifications that involve higher-order contexts, *i.e.*, contexts with not only atomic but also universal and implicational assumptions. We demonstrate the usefulness of this extension by proving non-trivial properties of one such specification that encodes a common compiler transformation on functional programs.

The second extension we have developed is to provide support of polymorphism in Abella. The current proof theory of $\mathcal{G}$ is based on the simply typed λ-calculus which does not include polymorphic types. To encode general data structures such as lists in Abella, we have to define a separate version of them at every concrete type where the data structure is needed. Furthermore, we have to prove their properties individually at each such concrete type, even through these properties and their proofs are mostly replications of a general pattern under different type instantiations. This is a recurring theme that we have witnessed when working on the implementation and verification of compiler transformations. To solve these problems, we have developed support for an approach to describing data structures, to writing definitions and statements we want to prove and to providing proofs that is schematic at the type level. Using this kind of "schematic polymorphism," we are able to avoid a lot of the duplication of code, definitions and proofs that we have alluded to above.[1]

We describe the two extensions in detail in the rest of this chapter. We first describe the extension to the two-level logic approach that enables reasoning about the full HH^ω specifications in Section 3.1. We then describe the extension to support polymorphism in Abella in Section 3.2.

3.1 Embedding HH^ω in Abella

We have described a two-level logic approach to reasoning about λProlog specifications in Section 2.3.3. The idea is to embed the logic of λProlog as a fixed-point definition in $\mathcal{G}$ and to reason about λProlog specifications through the embedding. The two-level logic approach adopted by Abella previously, as described in Section 2.3.3, embeds a subclass of HH^ω called HH^2 that allows for only dynamic addition of atomic formulas during derivations. It is good enough for proving properties of rule-based relational specifications that use contexts that contain only atomic assumptions, as demonstrated in the proof of determinacy of typing for the STLC in Section 2.3.3. On the other hand, a lot of formal systems are naturally described via rules that operate with contexts containing non-atomic formulas, or *higher-order contexts*. For example, a common compiler transformation that converts λ-terms into their de Bruijn forms can be elegantly described via rules that record the rule for transforming variables into de Bruijn indexes as implicational formulas in the contexts. Such rule-based specifications can be elegantly encoded in HH^ω. However, they do not have a direct representation in HH^2 and thereby

[1]A related approach has been developed by Kaustuv Chaudhuri and has already been incorporated into the Abella system. However, that approach does not support a type schematic treatment of data structures or of HH^ω specifications which is the main focus of the work here.

cannot be reasoned about using the two-level logic approach described previously.

One reason for the previous two-level logic approach to embed HH^2 instead of HH^ω was for simplifying the case analysis based reasoning about derivability in λProlog. To analyze a derivation in λProlog, we need to take into account of the possibility that the goal formula is derived from the dynamically added program clauses, or *dynamic clauses* in short. In HH^2, this case is possible only if the goal formula is a member of the dynamic context because the dynamic context contains only atomic formulas and such an atomic formula can be used to prove the goal only if it matches exactly with the goal. Analyzing this case is thus equivalent to checking membership. However, in HH^ω, since the dynamic clauses can have the form of arbitrary program clauses, the derivation may proceed by backchaining on some dynamic clause and further yielding subgoals. Analysis of such cases becomes much more complicated and may not even be possible if no proper constraint is placed on the form of the dynamic context and the structure of derivations.

In this section, we describe an extension of the two-level logic approach to support reasoning about the full class of HH^ω specifications [61]. The critical observation that enables this extension is that clauses added dynamically during a derivation must take the form of some subformulas in the original specification. Since we always fix the λProlog specification at the beginning of proof developments, the dynamic clauses must have finite forms. We can therefore give the dynamic context an inductive definition in $\mathcal{G}$. To support reasoning over backchaining steps that result from using dynamic clauses, we embed a version of HH^ω based on the technique called *focusing* [62] as a fixed-point definition in $\mathcal{G}$. With this extension, we are able to reason about rule-based relational specifications that involve higher-order contexts in Abella.

We elaborate on the above ideas in the following subsections. We first introduce the transformation of λ-terms into the de Bruijn form as an example to motivate the extension in Section 3.1.1. We then explain the difficulties with case analysis on higher-order contexts in Section 3.1.2 and present our solution in Section 3.1.3. We conclude the discussion by demonstrating the power of the extended two-level logic approach through proving non-trivial properties about the motivating example in Section 3.1.4.

3.1.1 A motivating example

To understand the issues with reasoning about rule-based descriptions of relations involving higher-order contexts, consider the transformation of λ-terms into their de Bruijn forms as an example. The de Bruijn form is an alternative notion of λ-terms due to de Bruijn in which bound variables are not named and their occurrences are represented instead by indexes that count the abstractions up to the one binding them [31]. Its syntax is given as follows where t denotes a term in this form and i is a natural number denoting the index of a variable occurrence:

$$t \ ::= \ i \mid t\,t \mid \lambda.t$$

Note that the binding structure is already implicitly captured by the representation of variables as de Bruijn indexes. As a result, there is no need to represent binding variables in abstractions.

There is a natural mapping between closed λ-terms and their de Bruijn representation. The λ-terms can be translated into the nameless form by converting every variable occurrences into de Bruijn indexes and dropping the binding variables, and vice versa. For example, the term $(\lambda x.\,(\lambda y.\,x\ y)\ x)$ is mapped to the de Bruijn term $(\lambda.(\lambda.2\ 1)\ 1)$. We give a rule-based description of such mapping. Writing $\Gamma \vdash m \equiv_h d$ to denote the correspondence between the λ-term m that occurs at *depth* h (*i.e.*, under h λ-abstractions) and the de Bruijn term d where Γ determines the mapping between free variables in the two representations, we can define this relation via the rules in Figure 3.1. The rule for relating applications is straightforward. To relate $(\lambda x.\,m)$ to a De Bruijn term at depth h, we must relate each occurrence of x in m, which must be at a depth $h + k$ for some $k > 0$, to the de Bruijn index k. To encode this correspondence, the context is extended in the premise of rule `db-abs` with a (universally quantified) implicational formula. Note also that this rule carries with it the implicit assumption that the name x used for the bound variable is fresh to Γ, the context for the concluding judgment. Eventually, when the λ-term on the right of the turnstile is a variable, the rule `db-var` provides the means to complete the derivation by using the relevant assumption from the context Γ.

The above rule-based description can be elegantly encoded in λProlog. We first

$$\frac{\Gamma \vdash m \equiv_h d \quad \Gamma \vdash n \equiv_h e}{\Gamma \vdash m\, n \equiv_h d\, e}\ \texttt{db-app}$$

$$\frac{\Gamma, \forall i, k.((h + k = i) \supset x \equiv_i k) \vdash m \equiv_{h+1} d}{\Gamma \vdash \lambda x.\, m \equiv_h \lambda.d}\ \texttt{db-abs}$$

$$\frac{\forall i, k.((h + k = i) \supset x \equiv_i k) \in \Gamma \quad h + k = i}{\Gamma \vdash x \equiv_i k}\ \texttt{db-var}$$

Figure 3.1: The Rules Relating λ-Terms and their de Bruijn Forms

encode the syntax of the λ-terms and the de Bruijn terms in λProlog as follows. We use the λProlog type `tm` to represent the type of λ-terms and build their encoding around the two constructors $\texttt{app} : \texttt{tm} \to \texttt{tm} \to \texttt{tm}$ and $\texttt{abs} : (\texttt{tm} \to \texttt{tm}) \to \texttt{tm}$. Using the type `dtm` for the representation of λ-terms in the de Bruijn form, we can encode them via the constructors $\texttt{dvar} : \texttt{nat} \to \texttt{dtm}$, $\texttt{dapp} : \texttt{dtm} \to \texttt{dtm} \to \texttt{dtm}$ and $\texttt{dabs} : \texttt{dtm} \to \texttt{dtm}$. Compared the encoding of λ-terms, variables in this form are formed explicitly by applying `dvar` to de Bruijn indexes. Abstractions are formed by applying `dabs` to terms representing their body.

The rules in Figure 3.1 make use of addition of natural numbers. We use `nat` to represent the type of natural numbers and encode natural numbers by using the constants $\texttt{z} : \texttt{nat}$ for representing 0 and $\texttt{s} : \texttt{nat} \to \texttt{nat}$ for constructing the successor of a natural number. We then encode addition as a relation $\texttt{add} : \texttt{nat} \to \texttt{nat} \to \texttt{nat} \to \texttt{o}$ such that given natural numbers n_1, n_2, n_3 and their encoding N_1, N_2 and N_3, $n_1 + n_2 = n_3$ if and only if $\texttt{add}\ N_1\ N_2\ N_3$ holds. This relation is defined through the following clauses:

$$\texttt{add}\ \texttt{z}\ N\ N.$$
$$\texttt{add}\ (\texttt{s}\ N_1)\ N_2\ (\texttt{s}\ N_3) \quad \texttt{:-} \quad \texttt{add}\ N_1\ N_2\ N_3.$$

We then designate the predicate constant $\texttt{hodb} : \texttt{tm} \to \texttt{nat} \to \texttt{dtm} \to \texttt{o}$ to represent the ternary relation $\equiv$. The rules in Figure 3.1 translate into the following clauses:

$$\texttt{hodb}\ (\texttt{app}\ M_1\ M_2)\ H\ (\texttt{dapp}\ E_1\ E_2) \quad \texttt{:-} \quad \texttt{hodb}\ M_1\ H\ E_1\,, \texttt{hodb}\ M_2\ H\ E_2.$$
$$\texttt{hodb}\ (\texttt{abs}\ M)\ H\ (\texttt{dabs}\ D) \quad \texttt{:-}$$
$$\Pi x.(\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k)) \Rightarrow \texttt{hodb}\ (M\ x)\ (\texttt{s}\ H)\ D.$$

The first and second clause encode the `db-app` and `db-abs` rules, respectively. Assume we use the dynamic contexts in HH$^\omega$ sequents to represent the contexts in the original rules. Then backchaining $\texttt{hodb}\ M_1\ H\ M_2$ on any one of the clauses followed by simplification of the goal formula corresponds to applying the relevant rule to the relation encoded by $\texttt{hodb}\ M_1\ H\ M_2$. This is easy to see for the first clause. In the case of the second clause, backchaining succeeds if $M_1 = \texttt{abs}\ M$ and $M_2 = \texttt{dabs}\ D$ and results in the following new goal:

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

To derive this goal we need to introduce a new constant c for x and derive the goal

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

The only way to derive this goal is to extend the dynamic context with the assumption $(\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ c\ i\ (\texttt{dvar}\ k))$ and then derive the goal $\texttt{hodb}\ (M\ c)\ (\texttt{s}\ H)\ D$. This corresponds to deriving the premise of `db-abs`. Like the encoding of typing rules for the STLC described in Section 2.1.3, this example uses universal goals to realize recursion over abstractions and to capture the side condition and it uses hypothetical goals to introduce assumptions about binding variables. The only difference is that the dynamically introduced assumptions here are not atomic formulas but have the full form of program clauses. Note that an explicit encoding of the `db-var` rule is not necessary as the application of this rule is implicitly captured by backchaining on the relevant dynamic clause.

We use a concrete example to further demonstrate how the above specification works. Let Σ be the signature containing constants we have defined so far and Γ the static context containing the clauses defining `add` and `hodb`. Consider showing that the λ-term $\lambda x.\, \lambda y.\, (y\ x)$ corresponds to the De Bruijn term $\lambda.\lambda.(1\ 2)$. This amounts to proving the following HH$^\omega$ sequent:

$$\begin{aligned}\Sigma; \Gamma; \emptyset \vdash\ & \texttt{hodb}\ (\texttt{abs}\ (x \setminus \texttt{abs}\ (y \setminus \texttt{app}\ y\ x)))\ \texttt{z} \\ & \qquad (\texttt{dabs}\ (\texttt{dabs}\ (\texttt{dapp}\ (\texttt{dvar}\ (\texttt{s}\ \texttt{z}))\ (\texttt{dvar}\ (\texttt{s}\ (\texttt{s}\ \texttt{z})))))).\end{aligned}$$

The only choice is backchaining on the second clause for `hodb`, which changes the proof obligation to:

$$\begin{aligned}&\Sigma, x : \texttt{nat}; \Gamma; (\Pi i, k.\texttt{add z } k\ i \Rightarrow \texttt{hodb } x\ i\ (\texttt{dvar } k)) \vdash\\ &\qquad\qquad \texttt{hodb (abs } (y \setminus \texttt{app } y\ x))\ (\texttt{s z})\\ &\qquad\qquad\qquad (\texttt{dabs (dapp (dvar (s z)) (dvar (s (s z)))))}.\end{aligned}$$

Attempting to backchain the new dynamic clause will fail because the new signature constant x does not unify with `abs`. Hence, the sole possibility that remains is backchaining on the second clause for `hodb` again, yielding:

$$\begin{aligned}&\Sigma, x : \texttt{nat}, y : \texttt{nat}; \Gamma; (\Pi i, k.\texttt{add z } k\ i \Rightarrow \texttt{hodb } x\ i\ (\texttt{dvar } k)),\\ &\qquad\qquad\qquad (\Pi i, k.\texttt{add (s z) } k\ i \Rightarrow \texttt{hodb } y\ i\ (\texttt{dvar } k)) \vdash\\ &\qquad \texttt{hodb (app } y\ x)\ (\texttt{s (s z)})\ (\texttt{dapp (dvar (s z)) (dvar (s (s z))))}.\end{aligned}$$

Now we can only backchain on the first clause for `hodb` to yield two new proof obligations, the first of which is:

$$\begin{aligned}&\Sigma, x : \texttt{nat}, y : \texttt{nat}; \Gamma; (\Pi i, k.\texttt{add z } k\ i \Rightarrow \texttt{hodb } x\ i\ (\texttt{dvar } k)),\\ &\qquad\qquad\qquad (\Pi i, k.\texttt{add (s z) } k\ i \Rightarrow \texttt{hodb } y\ i\ (\texttt{dvar } k)) \vdash\\ &\qquad \texttt{hodb } y\ (\texttt{s (s z)})\ (\texttt{dvar (s z)}).\end{aligned}$$

The only clause that we can select for backchaining is the second dynamic clause for y; none of the other clauses have a matching head. This modifies the goal to:

$$\begin{aligned}&\Sigma, x : \texttt{nat}, y : \texttt{nat}; \Gamma; (\Pi i, k.\texttt{add z } k\ i \Rightarrow \texttt{hodb } x\ i\ (\texttt{dvar } k)),\\ &\qquad\qquad\qquad (\Pi i, k.\texttt{add (s z) } k\ i \Rightarrow \texttt{hodb } y\ i\ (\texttt{dvar } k)) \vdash\\ &\qquad \texttt{add (s z) (s z) (s (s z))}.\end{aligned}$$

This sequent is then proved by backchaining on clauses for `add`. The other proof obligation is handled similarly.

3.1.2 Difficulty with dynamic contexts

We have shown how relational specifications involving higher-order contexts can be faithfully encoded in λProlog. We are interested in reasoning about these specifications through their encoding by using Abella. For example, we might be interested in showing that the relation that we have defined above identifies a bijective mapping between the two representations of λ-terms. One part of establishing this fact is proving that the relation is deterministic from left to right, *i.e.*, that every term in the named notation is related to at most one term in the nameless notation. Writing $\{\Gamma \vdash m \equiv_h d\}$ to denote derivability of the judgment $\Gamma \vdash m \equiv_h d$ by virtue of the rules `db-app`, `db-abs` and `db-var`, this property is stated as follows:

$$\forall \Gamma, m, h, d, e. \{\Gamma \vdash m \equiv_h d\} \supset \{\Gamma \vdash m \equiv_h e\} \supset d = e.$$

Adopting the two-level logic notation provided in Section 2.3.3, we denote derivability of the HH$^\omega$ sequent $\Sigma; \Theta; \Delta \vdash F$ by $\{\Delta \vdash F\}$ where Θ contains the clauses for `add` and `hodb` and Σ is the signature for this encoding. Then the property above translates into the following theorem in Abella:

$$\forall \Gamma, m, h, d, e. \{\Gamma \vdash \texttt{hodb}\ m\ h\ d\} \supset \{\Gamma \vdash \texttt{hodb}\ m\ h\ e\} \supset d = e. \qquad (\texttt{db-det})$$

A proof of the property `db-det` must obviously be based on an induction of derivability of the HH$^\omega$ sequents. A particular difficulty in articulating such inductive arguments relative to HH$^\omega$ specifications is that they need to analyze the cases of derivations that rely on assumptions in changing dynamic contexts. For example, a proof of `db-det` must accommodate the fact that Γ can be dynamically extended in a derivation of $\{\Gamma \vdash \texttt{hodb}\ m\ h\ d\}$ and that the particular content of Γ influences the derivation in the variable case via backchaining. Without well-defined constraints on Γ, it is difficult to predict how the dynamical added clauses might be used and indeed the above property may not even be true.

The difficulties described above can be summarized in two parts: First, it is necessary to finitely characterize the structure of the dynamic contexts; Second, it is necessary to have an approach to structuring reasoning about backchaining steps on the clauses in the dynamic contexts. These problems have straightforward solutions when λProlog specifications only introduce atomic formulas into the dynamic contexts, *i.e.*, when they are HH^2 specifications. In this case, we observe that the dynamic clauses must come from atomic formulas on the left of the implication symbols in program clauses. Since there are only finitely many such formulas in a λProlog specification, the dynamic contexts can be characterized as fixed-point definitions. Moreover, backchaining the goal formula on a dynamic clause succeeds if and only if it matches exactly with that clause. Thus analysis of such backchaining steps is equivalent to analysis of membership of the dynamic context. The two-level logic approach described in Section 2.3.3 is based on those observations and provides an elegant way to reason about HH^2 specifications. The proof of determinacy of typing in that section illustrated this approach.

The above two problems are much more difficulty to solve when dealing with full HH^ω specifications. The HH^ω logic allows for arbitrarily nested universal and implicational formulas in program clauses. As a result, derivations may dynamically introduce implicational or universal clauses. Backchaining on such clauses may yield subgoals that require sub-derivations to be constructed. These sub-derivations may further add clauses into the dynamic context. Since this process can be repeated in derivations, it is not clear how to characterize the structure of the dynamically added clauses and how to effectively perform case analysis over backchaining steps on these clauses.

However, in the example under consideration, there is an easy resolution to these problems. We observe that initially clauses can be introduced dynamically only by backchaining on the clause encoding `db-abs`. These clauses all have the form

$$\Pi i, k.\mathtt{add}\ h\ k\ i \Rightarrow \mathtt{hodb}\ x\ i\ (\mathtt{dvar}\ k)$$

where h is some natural number and x is some fresh variable. Moreover, backchaining further on such clauses does not add new clauses. The elements of the dynamic context Γ must therefore all be of the above form. Thus, the structure of Γ can be encoded into an inductive definition in $\mathcal{G}$ and treated in a finitary fashion by the machinery that $\mathcal{G}$ already provides for reasoning about backchaining steps.

3.1.3 The solution to the problem

We provide a solution to the problems of reasoning about the full HH^ω specifications by extending the two-level logic approach described in Section 2.3.3. The key insight underlying our solution is that the observation we made for the example in the last section generalizes cleanly to other reasoning situations that involve dynamic contexts with nested universal and implicational formulas. Concretely, the dynamic contexts that need to be considered in these situations are completely determined by the additions that can be made to them. Further, the structure of such additions must already be manifest in the original specification because the way derivations are constructed determines only subformulas that have the form of program clauses in the original specification can be added dynamically. Since there are only a finite number of such forms, the dynamic context can always be encapsulated in an inductive definition. To take advantage of this observation we encode a version of HH^ω that is based on the technique called focusing as a fixed-point definition in Abella to support reasoning also over the backchaining steps that result from using dynamically added assumptions.

We shall elaborate on this extension of the two-level logic approach in this section. We first present a focused version of HH^ω. We then provide an encoding of it in $\mathcal{G}$. We lastly describe how to use the inductive definitions of dynamic contexts and the HH^ω encoding to analyze backchaining steps on dynamic contexts. We will illustrate the power of this approach through a complete example in the next section.

A focused version of HH^ω

Focusing is a technique proposed in [62] for formulating sequent calculi in a way that non-determinism in searching for proofs is reduced while the completeness of the focused systems with respect to the original calculi is retained. Here we shall use focusing to characterize backchaining in a fine-grained fashion to facilitate the analysis of backchaining on dynamic contexts. The presentation of HH^ω in Section 2.1.1 already implicitly built in the notion of focusing. To derive a HH^ω sequent $\Sigma;\Gamma;\Delta \vdash A$ where A is an atomic goal, we pick some formula D from Γ or Δ and backchain A on D. If D exactly matches the atomic formula A, then the proof is completed. If D has an universal or implicational structure, then we match the head of D with A and try to derive its body (if any). We observe that backchaining is completed in one big step without any divergence. Using the terminology of focusing, the formula D is "focused" on throughout the backchaining step. After that, we simplify the body to atomic formulas and repeat the "focusing" phase (backchaining) again. This process continues until there are no more subgoals left to prove.

Based on the above observations, we give a focused version of HH^ω that makes the steps of "focusing" on formulas explicit and breaks up the "big-step" backchaining into “small-step” rules. Besides sequents of the form $\Sigma; \Gamma; \Delta \vdash G$ which we call *goal-reduction sequents* and which we have already seen in Section 2.1.1, we also introduce *focused sequents* of the form

$$\Sigma; \Gamma; \Delta, [D] \vdash A$$

where A is an atomic formula and D is the formula being focused on. The rules deriving such sequents are shown in Figure 3.2. The rules $\wedge$R, $\Rightarrow$R and ΠR reduce the goal formula when it is not atomic. The rules *prog* and *dyn* respectively focus on a clause in the static and dynamic contexts when the goal formula is atomic. The rules *match*, ΠL and $\Rightarrow$L reduce the focused formula; together they realize the backchaining steps on it. If we compare the system in Figure 3.2 and the one in Figure 2.1, we will see that they have the same reasoning power. Specifically, notice that the right rules are the same in the two systems; successive application of *prog*, *dyn*, ΠL, $\Rightarrow L$, tR and *match* for an appropriate number of times is equivalent to applying *backchain*.

$$\frac{}{\Sigma; \Gamma; \Delta \vdash \mathit{true}}\ tR \qquad \frac{\Sigma; \Gamma; \Delta \vdash G_1 \quad \Sigma; \Gamma; \Delta \vdash G_2}{\Sigma; \Gamma; \Delta \vdash G_1 \mathbin{\&} G_2}\ \wedge R$$

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

$$\frac{\Sigma; \Gamma; \Delta, [D] \vdash A \quad (D \in \Gamma)}{\Sigma; \Gamma; \Delta \vdash A}\ \mathit{prog} \qquad \frac{\Sigma; \Gamma; \Delta, [D] \vdash A \quad (D \in \Delta)}{\Sigma; \Gamma; \Delta \vdash A}\ \mathit{dyn}$$

$$\frac{}{\Sigma; \Gamma; \Delta, [A] \vdash A}\ \mathit{match}$$

$$\frac{\Sigma \Vdash t : \tau \quad \Sigma; \Gamma; \Delta, [D[t/x]] \vdash A}{\Sigma; \Gamma; \Delta, [\Pi x : \tau.D] \vdash A}\ \Pi L \qquad \frac{\Sigma; \Gamma; \Delta \vdash G \quad \Sigma; \Gamma; \Delta, [A'] \vdash A}{\Sigma; \Gamma; \Delta, [G \Rightarrow A'] \vdash A}\ \Rightarrow L$$

Figure 3.2: A Focused Version of HH$^\omega$

Encoding of HH$^\omega$ in $\mathcal{G}$

To encode HH$^\omega$ sequents in $\mathcal{G}$, we first note that $\mathcal{G}$ and HH$^\omega$ share the same type system. The HH$^\omega$ signatures can therefore be imported transparently into $\mathcal{G}$, so the signatures of HH$^\omega$ sequents will not be explicitly encoded. The contexts of HH$^\omega$ are represented in $\mathcal{G}$ as lists of HH$^\omega$ formulas (*i.e.*, lists of terms of type `o`). The type `olist` with constructors $\texttt{nil} : \texttt{olist}$ and $\texttt{::} : \texttt{o} \to \texttt{olist} \to \texttt{olist}$ is used for these lists, and, per tradition, the `::` constructor is written infix. Membership in a context is defined inductively as a predicate $\texttt{member} : \texttt{o} \to \texttt{olist} \to \texttt{prop}$ with these clauses:

$$\begin{array}{lcl} \texttt{member}\, E\, (E :: L) & \triangleq & \top \\ \texttt{member}\, E\, (F :: L) & \triangleq & \texttt{member}\, E\, L. \end{array}$$

Observe that the two clauses have overlapping heads; there will be as many ways to show $\texttt{member}\ E\ L$ as there are occurrences of E in L. This validates the view of HH$^\omega$ contexts as multisets.

The encoding of HH$^\omega$ is parameterized by the λProlog specification that we would like to reason about. We identify the predicate symbol $\texttt{prog} : \texttt{o} \to \texttt{o} \to \texttt{prop}$ for representing this λProlog specification and translate every program clause $\Pi x_1. \ldots \Pi x_n. G \Rightarrow A$ in the λProlog specification to the definitional clause $\texttt{prog}\, G\, A$.

The sequents of HH$^\omega$ are then encoded in $\mathcal{G}$ using the predicates $\texttt{seq} : \texttt{nat} \to \texttt{olist} \to \texttt{o} \to \texttt{prop}$ and $\texttt{bch} : \texttt{nat} \to \texttt{olist} \to \texttt{o} \to \texttt{o} \to \texttt{prop}$. A goal-reduction sequent $\Sigma; \Gamma; \Delta \vdash G$ is encoded as $\texttt{seq}\ N\ \Delta\ G$ for some N and a focused sequent $\Sigma; \Gamma; \Delta, [F] \vdash A$ is encoded as $\texttt{bch}\, N\ \Delta\ F\ A$ for some N, where N encodes the size of a derivation for the sequent. Here we omit the static context Γ which is the parameterizing λProlog specification and encoded explicitly in `prog` as described above. We also omit the signature Σ which is absorbed into the signature of $\mathcal{G}$.

The rules of the HH$^\omega$ proof system in Figure 3.2 are used to build mutually recursive definitions of the `seq` and `bch` predicates. This definition is depicted in Figure 3.3. The goal reduction rules are systematically translated into the clauses for `seq`, the only novelty being that universally quantified variables of the specification logic are represented as nominal constants in $\mathcal{G}$ using the ∇ quantifier. This use of ∇ accurately captures the rule for reducing universally quantified goal formulas in HH$^\omega$: to derive such a goal, a new (nominal) constant c that is different from any other constructs or constants must be introduce for its binding variable x and the formula obtained by replacing the occurrences of x in its body with c must be derivable. The backchaining rules of HH$^\omega$ are encoded as clauses for `bch` in a straightforward manner. For the *dyn* and *match* rules of HH$^\omega$, we have to enforce the invariant that the goal of the sequent is atomic. This is achieved by means of a predicate $\texttt{atomic} : \texttt{o} \to \texttt{prop}$ defined by the following clause:

$$\begin{array}{lcl}
\texttt{atomic}\,(\Pi_\tau\; G) & \triangleq & \bot \\
\texttt{atomic}\,(G_1 \;\&\; G_2) & \triangleq & \bot \\
\texttt{atomic}\,(G_1 \Rightarrow G_2) & \triangleq & \bot \\
\texttt{atomic}\,\mathit{true} & \triangleq & \bot
\end{array}$$

Effectively, `atomic` characterizes atomic formulas negatively by saying that an atomic formula cannot be constructed with an HH$^\omega$ connective or *true*. The *dyn* rule transparently translates into a clause for `seq`. The *prog* rule does not have a corresponding clause. Instead, it is absorbed into a direct encoding of backchaining on program clauses. Specifically, because the exact forms of the program clauses in the λProlog specification is known, backchaining on such clauses can be analyzed in one big step. Such backchaining is encoded in the following clause which combines the application of *prog* to focus on a clause in the static context and the application of ΠL, $\Rightarrow L$ and *match* for a number of times to realize backchaining on the clause:

$$\texttt{seq}\,(\texttt{s}\;N)\;L\;A \quad\triangleq\quad \texttt{atomic}\;A \wedge \exists G.\texttt{prog}\;G\;A \wedge \texttt{seq}\;N\;L\;G \qquad (\texttt{prog-bc})$$

$$\begin{array}{lcl}
\texttt{seq}\;N\;L\;\mathit{true} & \triangleq & \top \\
\texttt{seq}\,(\texttt{s}\;N)\;L\;(G_1 \;\&\; G_2) & \triangleq & \texttt{seq}\;N\;L\;G_1 \wedge \texttt{seq}\;N\;L\;G_2 \\
\texttt{seq}\,(\texttt{s}\;N)\;L\;(D \Rightarrow G) & \triangleq & \texttt{seq}\;N\;(D :: L)\;G \\
\texttt{seq}\,(\texttt{s}\;N)\;L\;(\Pi_\tau\;G) & \triangleq & \nabla x:\tau.\texttt{seq}\;N\;L\;(G\;x) \\
\texttt{seq}\,(\texttt{s}\;N)\;L\;A & \triangleq & \texttt{atomic}\;A \wedge \exists D.\texttt{member}\;D\;L \wedge \texttt{bch}\;N\;L\;D\;A \\
\texttt{seq}\,(\texttt{s}\;N\;L\;A & \triangleq & \texttt{atomic}\;A \wedge \exists G.\texttt{prog}\;G\;A \wedge \texttt{seq}\;N\;L\;G \\
\texttt{bch}\,(\texttt{s}\;N)\;L\;(D_1 \;\&\; D_2)\;A & \triangleq & \texttt{bch}\;N\;L\;D_1\;A \vee \texttt{bch}\;N\;L\;D_2\;A \\
\texttt{bch}\,(\texttt{s}\;N)\;L\;(G \Rightarrow A')\;A & \triangleq & \texttt{seq}\;N\;L\;G \wedge \texttt{bch}\;N\;L\;A'\;A \\
\texttt{bch}\,(\texttt{s}\;N)\;L\;(\Pi_\tau\;D)\;A & \triangleq & \exists t:\tau.\texttt{bch}\;N\;L\;(D\;t)\;A \\
\texttt{bch}\;N\;L\;A\;A & \triangleq & \top
\end{array}$$

Figure 3.3: Encoding of HH$^\omega$ Rules as Inductive Definitions in $\mathcal{G}$

We note that while the definitions of `seq`, `bch`, and `atomic` that we have presented above are sensible at a schematic level, they are not actually ones that can be written explicitly in $\mathcal{G}$. The reason for this is that some of the clauses that we have shown—such as the one for `seq` that pertains to an HH$^\omega$ goal of the form $(\Pi_\tau\; G)$—are parameterized by a type. In the present context, an explicit rendition of such a "clause" would involve writing down a separate clause for exact distinct type and there may be an infinite collection of such types. The schematic form of polymorphism that we introduce in Section 3.2 will overcome this difficulty.

The faithfulness of our encoding allows us to state and prove known properties of HH^ω in $\mathcal{G}$. For example, the cut-elimination property of HH^ω can be proved as a theorem in $\mathcal{G}$ and freely used in reasoning. Employing such meta-properties can greatly simplify the reasoning about λProlog specifications in many circumstances [40, 61].

Reasoning about HH^ω specifications through their encoding

We are now in a position to talk about the approach to reasoning about HH^ω specifications through the above encoding of HH^ω. We shall use the curly brace notations described in Section 2.3.3 for encoding derivability in HH^ω. That is, we use $\{L \vdash G\}$ and $\{L, [D] \vdash G\}$ to respectively represent the formula $\exists E.\mathtt{nat}\ N \land \mathtt{seq}\ N\ L\ G$ and $\exists E.\mathtt{nat}\ N \land \mathtt{bch}\ N\ L\ D\ A$ in describing theorems. When they occur as hypotheses of a proof state, they respectively represent a packet of formulas $\mathtt{nat}\ N$ and $\mathtt{seq}\ N\ L\ G$ and a packet of formulas $\mathtt{nat}\ N$ and $\mathtt{bch}\ N\ L\ D\ A$. The annotated style of induction and case analysis on such notations work in a way similar to what we have described in Section 2.3.3. That is, induction on $\{L \vdash G\}$ or $\{L, [D] \vdash G\}$ is actually induction on the size measures associated with them and case analysis of them is actually case analysis on the packets they represent.

The major difficulty in reasoning about HH^ω derivations is in performing case analysis on backchaining steps, *i.e.*, on derivations of $\{L \vdash A\}$ where A can either backchain on a clause in the λProlog specification or in L. Analysis of the former case is easy because the form of program clauses in the original specification is already known and the way backchaining proceeds is fixed; this is also reflected in the clause `prog-bc` that encodes backchaining on such clauses. The difficulty part is to analyze backchaining on clauses in the dynamic context L. By the encoding of HH^ω, $\{L \vdash A\}$ is derivable only if there exists some D such that $\mathtt{member}\ D\ L$ holds—*i.e.*, D is a member of L—and $\{L, [D] \vdash A\}$ is derivable. Further analysis is not possible without knowing the form of D. Fortunately, as we have observed previously, clauses in L must come from the original specification, thereby must have a fixed number of forms. As a result, we can give L a fixed-point definition and prove this structural property as a theorem about the definition. By applying the structural property of L, we can reveal the possible forms of D. Once this is done, the way A can backchain on D is fixed and we can reduce the formula $\{L, [D] \vdash A\}$ to one representing a goal-reduction sequent. Now we have finished analysis of the backchaining step and can go on with subsequent reasoning.

3.1.4 An illustration of the extended system

We demonstrate the power of the extension we described above through its use in explicitly proving the bijectivity property of the transformation between λ-terms and de Bruijn terms presented in Section 3.1.1. We do this by showing that `hodb` is deterministic in both its first and third arguments. As expected, we work within Abella with the encoding of HH$^\omega$ described in the previous section. We also assume that the program clauses defining `hodb` and `add` have been reflected into the definition of `prog` in this context.

In the rest of this section, we describe the proof of determinacy of `hodb`. The discussion is organized as follows: We first present the inductive definition of the dynamic contexts for `hodb` derivations and prove their structural properties through the definition, we then present the determinacy theorems and their proofs; the structural properties of dynamic contexts play a critical role in constructing these proofs.

Formalizing the dynamic contexts and their structural properties

As mentioned in the previous section, we will need to finitely characterize the dynamic contexts during the derivation of `hodb`. They are given through the following clauses that inductively define the predicate $\texttt{ctx} : \texttt{olist} \to \texttt{prop}$:

$$\begin{array}{lcl} \texttt{ctx}\,\texttt{nil} & \triangleq & \top \\ \nabla x.\texttt{ctx}\ ((\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ x\ i\ (\texttt{dvar}\ k))\ ::\ L) & \triangleq & \texttt{ctx}\ L. \end{array}$$

Note again that capitalized symbols are implicitly universally quantified over the entire clause. In the second clause, the ∇ quantification guarantees that x must match a nominal constant and the quantification ordering guarantees that x does not occur in L. Therefore, in any L for which $\texttt{ctx}\ L$ holds, it must be the case that there is exactly one clause $(\Pi i, k.\mathtt{add}\ H\ k\ i \Rightarrow \mathtt{hodb}\ x\ i\ (\mathtt{dvar}\ k))$ for each such $x \in \mathit{supp}(L)$. It is easy to establish this fact in terms of a pair of lemmas.

The first of these lemmas, called `ctx-inv`, characterizes the form of clauses in L.

$$\begin{array}{l} \forall L, E.\mathtt{ctx}\ L \supset \mathtt{member}\ E\ L \supset \\ \qquad \exists x, H.E = (\Pi i, k.\mathtt{add}\ H\ k\ i \Rightarrow \mathtt{hodb}\ x\ i\ (\mathtt{dvar}\ k)) \wedge \mathtt{name}\ x. \end{array}$$

Here, `name` is defined by the single clause $\nabla x.\mathtt{name}\ x \triangleq \top$. Therefore, `name` x is a predicate that asserts that x is a nominal constant. To prove `ctx-inv`, we proceed by induction on the first hypothesis, `ctx` L. As mentioned in Section 2.3.2, this is achieved by assuming a new *inductive hypothesis* `IH`:

$$\begin{array}{l} \forall L, E.(\mathtt{ctx}\ L)^* \supset \mathtt{member}\ E\ L \supset \\ \qquad \exists x, H.E = (\Pi i, k.\mathtt{add}\ H\ k\ i \Rightarrow \mathtt{hodb}\ x\ i\ (\mathtt{dvar}\ k)) \wedge \mathtt{name}\ x. \end{array}$$

Moreover, the proof state is transformed to have the following hypotheses converted from the assumptions of the lemma where L and E are promoted to variables at the proof level:

$$\begin{array}{ll} \mathtt{H1}: & \mathtt{ctx}\ L^{@} \\ \mathtt{H2}: & \mathtt{member}\ E\ L \end{array}$$

and have

$$\exists x, H.E = (\Pi i, k.\mathtt{add}\ H\ k\ i \Rightarrow \mathtt{hodb}\ x\ i\ (\mathtt{dvar}\ k)) \wedge \mathtt{name}\ x$$

as the new conclusion.

The `IH` cannot be immediately used because the annotation of `H1` does not match. To make progress, we need to unfold `H1` to get `ctx` L' for some L' whose annotation matches with `IH`. This amounts to finding all ways of unifying `ctx` L with the heads of the clauses in the definition of `ctx`. There are two cases to consider here: when $L = \mathtt{nil}$ and when $L = ((\Pi i, k.\mathtt{add}\ H\ k\ i \Rightarrow \mathtt{hodb}\ \mathtt{n}\ i\ (\mathtt{dvar}\ k)) :: L')$ for some new variables H and L' and a nominal constant `n`. In the latter case we also have a new hypothesis, $(\mathtt{ctx}\ L')^*$, that comes from the body of the second clause for `ctx`. There are two things to note: first, the ∇ quantified x at the head of the second clause of `ctx` is turned into a nominal constant in the proof obligation, and the $*$ annotation indicates that $(\mathtt{ctx}\ L')$ is derivable in fewer steps and hence suits `IH`.

In each case for L, the argument proceeds by analyzing the second hypothesis, $\texttt{member}\ E\ L$. The case of $L = \texttt{nil}$ is vacuous, because there is no way to infer $\texttt{member}\ E\ \texttt{nil}$, making that hypothesis equivalent to false. In the case of

$$L = ((\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ \texttt{n}\ i\ (\texttt{dvar}\ k)) :: L'),$$

we have two possibilities for $\texttt{member}\ E\ L$: either

$$E = (\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ \texttt{n}\ i\ (\texttt{dvar}\ k))$$

or $\texttt{member}\ E\ L'$. The former possibility is exactly the conclusion that we seek, so this branch of the proof finishes. The latter possibility lets us apply IH to the hypotheses $(\texttt{ctx}\ L')^*$ and $\texttt{member}\ E\ L'$, which also yields the desired conclusion.

The second necessary lemma, called ctx-det asserts that there is at most a single clause for each variable in the dynamic context.

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

Note that from $H_1 = H_2$, we are able to conclude that the two dynamic clauses relating x to a de Bruijn index must be the same. Like the previous lemma, it is proved by induction on the hypothesis $\texttt{ctx}\ L$.

Proving the determinacy theorems for hodb

We can now show both directions of determinacy for hodb by using the proved lemmas of ctx. In the forward direction the statement is as follows.

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

We prove this by induction on $\{L \vdash \texttt{hodb}\ M\ H\ D\}$. The induction introduces the IH below:

$$\forall L, M, H, D, E.\texttt{ctx}\ L \supset \{L \vdash \texttt{hodb}\ M\ H\ D\}^* \supset \{L \vdash \texttt{hodb}\ M\ H\ E\} \supset D = E.$$

Moreover, the proof state is changed to have the following hypotheses

H1 : $\texttt{ctx}\, L$
H2 : $\{L \vdash \texttt{hodb}\, M\ H\ D\}^@$
H3 : $\{L \vdash \texttt{hodb}\, M\ H\ E\}$

where L, M, H, D, E are new variables and the conclusion to be proved becomes $D = E$.

Now, $\{L \vdash \texttt{hodb}\ M\ H\ D\}^@$ is just a notation for the packet of formulas $(\texttt{nat}\ N)^@$ and $\texttt{seq}\ L\ (\texttt{hodb}\ M\ H\ D)$ whose definition is given by the clauses in Figure 3.3. Unfolding the definition amounts to finding all the clauses in Figure 3.3 whose heads match $\texttt{seq}\ L\ (\texttt{hodb}\ M\ H\ D)$. Only the final two clauses of `seq`, corresponding to focusing on a dynamic clause and backchaining on a static program clause, are therefore relevant.

Let us consider backchaining the static clauses first. Unfolding $\{L \vdash \texttt{hodb}\ M\ H\ D\}^@$ introduces the following new hypotheses for some new variable G:

H4 : $\texttt{prog}\, G\ (\texttt{hodb}\, M\ H\ D)$
H5 : $\{L \vdash G\}^*$

There are only a finite number of static clauses, so the assumption $\texttt{prog}\ G\ (\texttt{hodb}\ M\ H\ D)$ can be immediately turned into a branched tree with one case for every static program clause. The cases for the static clauses of `add` are immediately dismissed because their heads do not match with $(\texttt{hodb}\ M\ H\ D)$. We are left with cases for the static clauses of `hodb`.

For the first static clause of `hodb`, analysis of $\texttt{prog}\ G\ (\texttt{hodb}\ M\ H\ D)$ instantiates M with $(\texttt{app}\ M'\ N')$, D with $(\texttt{dapp}\ D'\ E')$ and G with $(\texttt{hodb}\ M'\ H\ D'\ \&\ \texttt{hodb}\ N'\ H\ E')$ for fresh variables M', N', D', E'. The goal reduction sequent $\{L \vdash G\}^*$ becomes:

H5 : $\{L \vdash (\texttt{hodb}\, M'\ H\ D'\ \&\ \texttt{hodb}\, N'\ H\ E')\}^*$

which is reduced by the second clause for `seq` to:

H6 : $\{L \vdash (\texttt{hodb}\, M'\ H\ D')\}^*$
H7 : $\{L \vdash (\texttt{hodb}\, N'\ H\ E')\}^*$

We can almost apply the induction hypothesis `IH`—we know $\texttt{ctx}\ L$ and `H6` already—but we still must find the third argument. To get this argument we need to case analyze the other hypothesis, $\{L \vdash \texttt{hodb}\ M\ H\ E\}$, which becomes $\{L \vdash \texttt{hodb}\ (\texttt{app}\ M'\ N')\ H\ E\}$

as a result of the previous instantiation. It has no size annotations because the induction was on the first hypothesis. Nevertheless, we can perform a case analysis of its structure by unfolding its definition. Once again, we have a choice of using a static program clause or a dynamic clause from L. If we use a static clause, then by a similar argument to the above we will get the following fresh hypotheses, for new variables D'' and E'' such that $E = \texttt{dapp}\ D''\ E''$:

$$\begin{aligned} &\texttt{H8}: \quad \{L \vdash (\texttt{hodb}\ M'\ H\ D'')\} \\ &\texttt{H9}: \quad \{L \vdash (\texttt{hodb}\ N'\ H\ E'')\} \end{aligned}$$

We can now apply the `IH` twice, one to `ctx` L, `H6`, `H8` and one to `ctx` L, `H7`, `H9`, yielding $D' = D''$ and $E' = E''$, so $D = \texttt{dapp}\ D'\ E' = \texttt{dapp}\ D''\ E'' = E$.

If, on the other hand, we use a dynamic clause in L, then the two fresh hypotheses we get are:

$$\begin{aligned} &\texttt{H8}: \quad \texttt{member}\ D\ L \\ &\texttt{H9}: \quad \{L, [D] \vdash \texttt{hodb}\ (\texttt{app}\ M'\ N')\ H\ E\} \end{aligned}$$

for some new variable D. This is the first place where the context characterization hypothesis `ctx` L becomes useful. By applying the lemma `ctx-inv` above to `ctx` L and `H8`, we should be able to conclude that D is of the following form

$$(\Pi i, k.\texttt{add}\ H'\ k\ i \Rightarrow \texttt{hodb}\ \texttt{n}\ i\ (\texttt{dvar}\ k))$$

for some term H' and some nominal constant `n`. By looking at the clauses for `bch` in Figure 3.3, it is clear that there is no way to prove the sequent `H9`, because the term `n` will never unify with `app` $M'\ N'$. Hence this hypothesis is vacuous, which closes this branch. We have now accounted for the cases of backchaining the first static clause of `hodb` for the inductive assumption $\{L \vdash \texttt{hodb}\ M\ H\ D\}^{@}$.

For backchaining the second static clause of `hodb`, analysis of `prog` G (`hodb` $M\ H\ D$) instantiates M with (`abs` M'), D with (`dabs` D') and G with

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

for fresh variables M' and D'. The goal reduction sequent $\{L \vdash G\}^*$ is then reduced first by the fourth and then by the third clause for `seq` to:

$$\texttt{H6}: \quad \{L, (\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ \texttt{n}\ i\ (\texttt{dvar}\ k)) \vdash (\texttt{hodb}\ (M'\ \texttt{n})\ (\texttt{s}\ H)\ D')\}^*$$

where n is a nominal constant. To apply IH, we need another assumption that accompanies H6. For this, we do a case analysis of H3, which has become $\{L \vdash \texttt{hodb}\ (\texttt{abs}\ M')\ H\ E\}$ as a result of the previous instantiation, by unfolding its definition. Again, there are two cases to consider here: either H3 is proved by backchaining on the second static clause of hodb or on the dynamic context. In the former case, E is instantiated with $(\texttt{dabs}\ E')$ for some E' and we got the following assumption from the case analysis followed by goal reduction steps:

$$\texttt{H7}: \quad \{L, (\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ \texttt{n}\ i\ (\texttt{dvar}\ k)) \vdash (\texttt{hodb}\ (M'\ \texttt{n})\ (\texttt{s}\ H)\ E')\}^*$$

Because ctx L holds, by the definition of ctx, we know the following hypothesis holds:

$$\texttt{H8}: \quad \texttt{ctx}\ ((\Pi i, k.\texttt{add}\ H\ k\ i \Rightarrow \texttt{hodb}\ \texttt{n}\ i\ (\texttt{dvar}\ k)) :: L)$$

We now apply IH to H8, H6 and H7, yielding $D' = E'$, so $D = (\texttt{dabs}\ D') = (\texttt{dabs}\ E') = E$. The case that H3 is proved by backchaining on the dynamic context L is dismissed by following the same argument described previously when M is an application: by applying ctx-inv to ctx L we observe that $(\texttt{abs}\ M')$ needs to unify with a nominal constant, which is impossible. We have now accounted for all the cases of backchaining a static clause for the inductive assumption $\{L \vdash \texttt{hodb}\ M\ H\ D\}^@$.

We are left with only backchaining on dynamic clauses in L for $\{L \vdash \texttt{hodb}\ M\ H\ D\}^@$. This corresponds to reducing $\{L \vdash \texttt{hodb}\ M\ H\ D\}^@$ by the second to last clause of seq, yielding the following pair of new hypotheses:

$$\texttt{H4}: \quad (\texttt{member}\ D\ L)^*$$
$$\texttt{H5}: \quad \{L, [D] \vdash \texttt{hodb}\ M\ H\ D\}^*$$

As these hypotheses come from unfolding an inductive assumption, they are *-annotated. Once again, we can apply the ctx-inv lemma to conclude that

$$D = (\Pi i, k.\texttt{add}\ H'\ k\ i \Rightarrow \texttt{hodb}\ \texttt{n}\ i\ (\texttt{dvar}\ k))$$

for some variable H' and nominal constant n. We then perform a case analysis on H5. The only possibility here is to use the definitional clauses for bch to unify M with n and D with $(\texttt{dvar}\ k)$ for some variable k. Moreover, the case analysis generates the following fresh hypothesis:

$$\texttt{H6}: \quad \{L \vdash \texttt{add}\ H'\ k\ H\}^*$$

We now analyze how the other hypothesis related to `hodb`, *i.e.* $\{L \vdash \texttt{hodb}\ \texttt{n}\ H\ E\}$, can be derived. Since `n` cannot unify with the normal constants `abs` or `app`, the only way to prove $\{L \vdash \texttt{hodb}\ \texttt{n}\ H\ E\}$ would be to use a dynamic clause D' in L. From this analysis we get the following hypotheses:

$$\texttt{H7}: \quad \texttt{member}\ D'\ L$$
$$\texttt{H8}: \quad \{L, [D'] \vdash \texttt{hodb}\ \texttt{n}\ H\ E\}$$

Once again, by the lemma `ctx-inv` and unfolding the definition of `bch` as above, we can derive that

$$D' = (\Pi i, k.\texttt{add}\ H''\ k\ i \Rightarrow \texttt{hodb}\ \texttt{n}\ i\ (\texttt{dvar}\ k))$$
$$E = (\texttt{dvar}\ k')$$

for some variable H'' and k' and obtain the following new hypothesis:

$$\texttt{H9}: \quad \{L \vdash \texttt{add}\ H''\ k'\ H\}$$

We can now apply the lemma `ctx-det` to `H4` and `H7` to show that $H' = H''$. To finish this case, we need the following lemma that shows `add` is deterministic in its second argument:

$$\forall L, N_1, N_2, N_3, N_2'.$$
$$\texttt{ctx}\ L \supset \{L \vdash \texttt{add}\ N_1\ N_2\ N_3\} \supset \{L \vdash \texttt{add}\ N_1\ N_2'\ N_3\} \supset N_2 = N_2'.$$

This lemma is proved by induction on the second assumption. Note that in the case that `add` $N_1\ N_2\ N_3$ backchains on L, we derive a contradiction by applying the lemma `ctx-inv` to show that `add` must match with `hodb`, which is not possible. We apply the above lemma to `H6` and `H9`, yielding $k = k'$, so $D = (\texttt{dvar}\ k) = (\texttt{dvar}\ k') = E$. We now have finished proving the theorem that `hodb` is deterministic in its third arguments.

The `hodb` relation is also deterministic in its first argument, *i.e.*, given a de Bruijn indexed term, there is at most a single HOAS term it corresponds to. This is stated as the following theorem which is proved in a similar fashion:

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

Thus, the `hodb` relation is manifestly an isomorphism between the two representations of λ-terms.

The example we have considered here has dynamic clauses with implicational formulas only at the top level. It is possible for HH^ω specifications to dynamically introduce clauses that have nested implications. The same approach described in this section applies to reasoning abut such specifications. Such an example can be found in [61].

3.2 Schematic Polymorphism in Abella

The logic $\mathcal{G}$ is based on the simply typed λ-calculus. As a consequence, we cannot identify data structures such as lists and sets within it that will work at any type. Instead we have to define such data structures separately for each type at which they are needed and prove properties about each of these versions independently. This drawback also affects the way in which data structures must be defined and used in the specification logic if we are to be able to reason about them in Abella: to be embedded into Abella, the specifications must also be provided separately for each concrete type. All this has two adverse consequences for a large development such as that involved in verified compilation. First, it leads to code duplication and loss of modularity at the implementation level. Second, it requires proofs that have an identical structure to be repeated at different types, thereby expanding the proof development effort.

These problems can be alleviated by utilizing a schematic form of polymorphism. Such an approach is, in fact, already used in λProlog. Constructors for data structures such as lists can be parameterized by types in λProlog. This kind of type parameterization is extended to predicates and thereby also to the HH^ω clauses that define them. However, in the end, the underlying logic and language remain simply typed and the parameterization functions simply as a notational shorthand for identifying separate constructors, predicate symbols and clauses at each of their respective type instances.

Our goal in this section is to extend this kind of schematic polymorphism to Abella so as to allow for the kind of programming in λProlog that is described above and also to modularize the construction of proofs in Abella. We achieve this goal through the following steps:

- as in λProlog, we allow constants—which could be constructors or predicate names—to have types that are parameterized by other types, with the interpretation that they each actually stand for a family of constants that are obtained by instantiating the type parameters with concrete types;

- we allow definitional clauses to be parameterized by types, with the interpretation that each such clause stands for a collection of clauses in $\mathcal{G}$, each member of which is obtained by instantiating the type parameters with concrete types;
- we allow blocks of definitional clauses to be parameterized by types, with the interpretation that a block of this kind actually stands for a collection of blocks in $\mathcal{G}$ that are obtained by instantiating the type parameters with concrete types;
- we permit theorems to be parameterized by types, with the interpretation that each such "theorem" actually stands for the collection of theorems in $\mathcal{G}$ that are obtained by instantiating the type parameters with concrete types; and
- we lift the proof rules for $\mathcal{G}$ to permit the construction of proofs for type parameterized theorems that are such that instantiating the proof with concrete types will yield a proof in $\mathcal{G}$ for the "concrete" theorem.

We refer to the proofs that are constructed in this context as *schematic proofs*.

We elaborate on the steps described above in the rest of this section. We begin by providing motivating examples for the extension. We then describe a "polymorphic" form for definitions and theorems in Section 3.2.2. In Section 3.2.3 we present a lifted form of the proof rules for $\mathcal{G}$ and show that using them indeed leads to schematic proofs for polymorphic theorems under the intended interpretation of polymorphic definitions. We conclude the section by showing how the resulting system can be used to construct proofs for the motivating examples.

3.2.1 Some motivating examples

When implementing a compiler, we may need to represent lists of terms in the source language. Assume that the Abella type `tm` is the type of terms in this language, that `tmlist` is the type of lists of such terms and that the constructors for such lists are

$$\texttt{nil} : \texttt{tmlist} \qquad\qquad \texttt{::} : \texttt{tm} \to \texttt{tmlist} \to \texttt{tmlist}$$

As usual, we will write `::` in infix form. In the course of reasoning about these lists, we may need to describe a membership relation based on them. Such a relation may be represented by the predicate $\texttt{member} : \texttt{tm} \to \texttt{tmlist} \to \texttt{prop}$ that is defined by the following clauses:

$$\begin{array}{lcl} \texttt{member}\, X\ (X :: L) & \triangleq & \top \\ \texttt{member}\, X\ (Y :: L) & \triangleq & \texttt{member}\, X\ L \end{array}$$

Now, in the reasoning process, we may need to articulate a property of the following kind concerning list membership: If a term M is a member of the list L and we know that a nominal constant c does not appear in L, then c could not possibly appear in M. This property is an easy consequence of the following statement that we refer to as a *pruning property* for lists of terms:

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

Note that x is of type $\texttt{tm}$, M is of type $\texttt{tm} \to \texttt{tm}$ and L is of type $\texttt{tmlist}$ in this formula. The expression $(M\ x)$ in this formula captures the idea that the term that is a member of L might contain the variable x that represents the nominal constant. Since L is bound outside the scope of x, the term instantiating it cannot contain this nominal constant. As should be clear from previous discussions, $(\exists M'.M = y \setminus M')$, the conclusion of the formula, asserts that M represents a vacuous abstraction under the assumed circumstances.

The pruning property presented above can be proved by induction on the definition of the membership predicate that is used in the antecedent of the property. In the base case, in an assumption of the form $\texttt{member}\ (M\ n)\ L$, $L = (E :: L')$ and $E = (M\ \texttt{n})$ for some variables E and L' and some nominal constant $\texttt{n}$. Since E cannot depend on $\texttt{n}$, the only solution to $E = (M\ \texttt{n})$ is $M = y \setminus E$. Thus, this case is concluded. In the inductive case, $L = (E :: L')$ and $\texttt{member}\ (M\ \texttt{n})\ L'$ holds. Applying the inductive hypothesis to $\texttt{member}\ (M\ \texttt{n})\ L'$ concludes this case.

Suppose that the source and target languages of the compiler are different. We would then have to use a different type, say $\texttt{tm}'$, to represent target language expressions. To represent lists of such expressions, we would need a new type, say $\texttt{tmlist}'$, and also new constructors, say $\texttt{nil}' : \texttt{tmlist}'$ and $::' : \texttt{tm}' \to \texttt{tmlist}' \to \texttt{tmlist}'$. To reason about the properties of such lists, we would again need to reflect this type into $\mathcal{G}$. As we shall see in later chapters, it is also not unusual that we would want to define properties such as membership for lists of target language terms and to prove facts about it such as the pruning property. All this work would be identical in structure to what we have described earlier for source language terms except that it is done now using a different type for terms.

If this kind of repetitive work looks bad, it can get worse. As we shall see in Chapter 4, we are interested in verifying compilers that transform source programs into target code through a sequence of compilation passes. It is often the case that these passes will take code in one intermediate language as input and produce code in a different language as output. We may need to repeat the sequence of steps described above for every distinct intermediate language. This can get to be quite tedious.

The schematic polymorphism we want to introduce into Abella will solve the kind of problem that we have described above. The key is to observe that the constructors for lists differ only in the type of the elements in the list. Similarly, the definition of membership in lists and the pruning property can be parameterized by this type. Finally, the proof of the pruning property does not actually refer to the type, *i.e.*, it has the same structure regardless of the type of the list elements. Thus, if we could parameterize the types of constants, definitions and theorems by types with the interpretation that they stand for the family of their type instances and if we could provide a means for constructing "proofs" that are both independent of this type and will yield a correct proof when the type is instantiated, then we would be able to cover all the cases in one go.

The above discussion motivates the parameterization of a block of definitional clauses by types. Thus, if the clauses defining `member` constitute a block and we parameterize it by the type of the elements of the list, what we would be doing is providing a concise description of a collection of blocks of clauses, one for each possible concrete type. We also want to provide for the parameterization of individual clauses within a given block by types, with the interpretation that such a clause abbreviates a collection of clauses *within* the block, each obtained by replacing the type parameters by concrete types. The primary motivation for this addition is that we want to be able to support a schematic form of polymorphism in the writing of specifications in HH$^{\omega}$. As mentioned earlier, this kind of polymorphism is already realized in λProlog and it is essential to the convenient development of programs in the language.

To understand why this second kind of polymorphism is needed and also some of the issues involved in supporting it, let us suppose that we need a "member" predicate for lists not just for reasoning about compiler transformations but also for specifying or implementing them. Following the approach we have been discussing—which is actually implemented in λProlog—we might identify the predicate $\texttt{memb} : A \to \texttt{list}\ A \to \texttt{o}$ in the specification logic and define it using the HH$^\omega$-style clauses shown below.

$$\begin{array}{lll} \texttt{memb}\ X\ (X :: L). & & \\ \texttt{memb}\ X\ (Y :: L) & \texttt{:-} & \texttt{memb}\ X\ L. \end{array}$$

The token A in the type shown for `memb` is to be interpreted as a type variable that schematizes this constant in the sense that we should think of `memb` as representing a collection of constants, each indexed by a concrete type. This schematization extends to the clauses that define `memb`: each of these clauses also abbreviates a collection of clauses obtained by using `memb` and the list constructor `::` at particular concrete types. Note that if the collection of types is infinite, then what results is in fact an infinite collection of clauses.

Now consider what happens when we think of reflecting specifications provided in such an abbreviated form into $\mathcal{G}$. Following the approach we have described in Section 3.1, we would have to provide a definitional clause for `prog` for each clause in the specification. It can be quite tedious if we have to do this for each type instance of the clauses for `memb`. In fact, it is something that is impossible to do when the collection of types is infinite. Our proposal to permit type parameterization at the level of a definitional clause is intended to solve this problem. Using such a parameterization over suitable definitional clauses for `prog`, we can provide a concise, finite description of what might actually be an infinite collection of clauses in $\mathcal{G}$.

We would, of course, want to reason about the specifications that we allow to be written in the abbreviated way that we have described. For example, we may want to prove a pruning property that is based this time on the membership predicate in the specification logic:

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

In proving this statement, we would eventually have to carry out a case analysis on $\{$`memb` $(M\ x)\ L\}$. This would require us to consider the `prog` clauses that encode the definition of `memb` in the specification logic. It is not too difficult to see how this would proceed if the statement we are trying to prove uses `memb` at a particular type such as `tm` $\rightarrow$ `list tm` $\rightarrow$ `o`. However, the situation changes if the statement that we are interested in proving is actually parameterized by types. In this case, the case analysis would, in principle, need to consider all the different type instantiations. Each of these could match the definitional clauses in different ways, potentially leading to different proofs for each case.

The problem we have described above can show up even if we confine ourselves to reasoning about relations defined within $\mathcal{G}$, albeit in a polymorphic way. Our solution to the issue in both cases is identical. We will consider constructing only those proofs for "polymorphic" theorems that have an identical structure regardless of the type instance that is chosen. This design principle translates into two specific constraints in proof construction. First, we do not permit the instantiations of types in goal sequents in the course of constructing proofs. Second, we permit the use of case analysis in the reasoning process only when the way the atomic assumption matches the head of a definitional clause is independent of the type instantiation; this means, in particular, that the unifiers in each case will have an identical structure, regardless of the type instance. The technical machinery for realizing these constraints is what underlies our lifting of the proof rules of $\mathcal{G}$ to the context of schematic polymorphism.

3.2.2 The schematization of definitions and theorems

Our interest in polymorphism arose from a desire to parameterize data structures by types. The starting point in building in such a facility is to generalize type constants to *type constructors* that could take other types as arguments to form atomic types. With each type constructor must be associated an arity that indicates how many argument types it needs. As examples, we might identify `list` as a constructor of arity 1 that is can be used to form the type of lists of different types of elements. In this terminology, `nat`, which represents the types of natural numbers, would also be a type constructor whose arity is 0.

In addition to type constructors, type expressions will now also include type variables. The syntax of such expressions is given by the rule below, in which we denote type variables by A, a type constructor of arity n by a_n and we use τ with subscripts to denote types.

$$\tau \ ::= \ A \mid (\tau \to \tau) \mid (a_n \ \tau_1 \ \ldots \ \tau_n)$$

We call type expressions not containing type variables *concrete types* or *ground types.* We think of type variables and expressions of the form $(a_n \ \tau_1 \ \ldots \ \tau_n)$ as atomic types and we then extend the notions of argument types and target type in the obvious way to the type expressions defined here. We shall refer to a type that has `prop` as its target type as a predicate type.

We will now associate *type schemata* rather than types with term-level constants. A type schema has the form $([A_1, \ldots, A_n]\tau)$ where τ is a type expression all of whose type variables are contained in the sequence of distinct type variables $A_1, \ldots, A_n$. We shall write $(c : [A_1, \ldots, A_n]\tau)$ to associate such a schema with c. A declaration of this kind identifies c as representing a family of constants each of which has a type obtained by instantiating $(A_1, \ldots, A_n)$ in τ with ground types. We write $c_{\tau_1,\ldots,\tau_n}$ to represent the constant of type $\tau[\tau_1/A_1, \ldots, \tau_n/A_n]$ in the family represented by c. For example, we can declare the following constants for constructing lists where `::` is written in infix form:

$$\texttt{nil} : [A]\texttt{list}\ A \qquad \texttt{::} : [A]A \to \texttt{list}\ A \to \texttt{list}\ A$$

In this context, $\texttt{::}_{\texttt{nat}}$ has the type $\texttt{nat} \to \texttt{list nat} \to \texttt{list nat}$ and similarly $\texttt{::}_{\texttt{tm}}$ has the type $\texttt{tm} \to \texttt{list tm} \to \texttt{list tm}$. As another example, the universal quantifier in HH$^{\omega}$ might be identified by the declaration $(\Pi : [A](A \to \texttt{o}) \to \texttt{o})$ in $\mathcal{G}$. We shall use $\mathcal{K}$ to represent the set of constant declarations that are operative in a given context.

Like constants, terms can also be parameterized by types such that by instantiating the type parameters with ground types they become terms in $\mathcal{G}$. Of course, to function this way they must satisfy some constraints even with variables in type expressions. We formalize this idea via a *schematic typing judgment.* Towards this end, we identify *schematic typing contexts for variables*, denoted by Σ, of the form $(x_1 : \tau_1, \ldots, x_n : \tau_n)$

where x_i are variables and τ_i are type expressions.[2] We also identify *schematic typing contexts for nominal constants*, denoted by $\mathcal{C}$, of the form $(a_1 : \tau_1, \ldots, a_n : \tau_n)$ where a_i are nominal constants and τ_i are type expressions. A schematic typing judgment then has the form $\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t : \tau$ where the free variables and nominal constants in t are respectively bound in Σ and $\mathcal{C}$, and Ψ is a set of type variables that contains all the type variables appearing in Σ, $\mathcal{C}$, t and τ. We refer to the variables in Ψ as the type parameters of the judgment.

In describing the rules for deriving schematic typing judgments we will make use of the notion of a *type substitution* that is written as $(\tau_1/A_1, \ldots, \tau_n/A_n)$. We will use the notation $\Psi' \vdash \Phi : \Psi$ to denote the fact that Φ is substitution $(\tau_1/A_1, \ldots, \tau_n/A_n)$ where Ψ is the set of variables $\{A_1, \ldots, A_n\}$ and, for $1 \leq i \leq n$, all the type variables in the type expression τ_i are contained in Ψ'. The rules for deriving schematic typing judgments are then those shown in Figure 3.4. All the rules except t-cst are similar to the typing rules in the STLC. The rule t-cst assigns the constant $c_{\tau_1,\ldots,\tau_n}$ a type expression obtained by instantiating the type parameters of c with $(\tau_1, \ldots, \tau_n)$. Note that the right premise ensures that each τ_i in this collection must be a well-formed type expression that uses only the variables in Ψ.

$$\frac{d : \tau \in \mathcal{C} \cup \Sigma}{\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} d : \tau} \text{ t-var-norm}$$

$$\frac{c : [A_1, \ldots, A_n]\tau \in \mathcal{K} \quad \Psi \vdash (\tau_1/A_1, \ldots, \tau_n/A_n) : \{A_1, \ldots, A_n\}}{\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} c_{\tau_1,\ldots,\tau_n} : \tau[\tau_1/A_1, \ldots, \tau_n/A_n]} \text{ t-cst}$$

$$\frac{\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t_1 : \tau_1 \to \tau \quad \Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t_2 : \tau_1}{\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t_1\ t_2 : \tau} \text{ t-app} \qquad \frac{\Psi; \mathcal{C}, \Sigma, x : \tau_1 \Vdash_{\mathcal{K}} t : \tau}{\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} \lambda x{:}\tau_1.\, t : \tau_1 \to \tau} \text{ t-abs}$$

Figure 3.4: The Schematic Typing Rules

Given the rules in Figure 3.4 we have the following lemma which states that schematic typing judgments are preserved under the instantiation of type parameters:

Lemma 1. *If* $\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t : \tau$ *and* $\Psi' \vdash \Phi : \Psi$, *then* $\Psi'; \mathcal{C}[\Phi], \Sigma[\Phi] \Vdash_{\mathcal{K}} t[\Phi] : \tau[\Phi]$.

This lemma is proved by an easy induction on $\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t : \tau$. When Ψ' is empty, $t[\Phi]$

[2]We will extend Σ to include type associations for predicate names being defined in a declaration block later in this subsection.

is a well-formed term with a ground type. Consequently, if $\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t : \tau$ holds, we can think if t as a schematic term of type τ parameterized by Ψ. When τ is `prop`, we call t a schematic formula.

We now move on to schematizing definition blocks. As a first step, we identify a *pre-definitional clause* as an expression of the form

$$\forall \vec{x} : \vec{\tau_x}.(\nabla \vec{z} : \vec{\tau_z}.A) \triangleq B$$

in which A must have the structure of an atomic formula that contains no nominal constants and all of whose free variables appear in $\vec{x}$ or $\vec{z}$ and B has the structure of a formula that, once again, does not contain any nominal constants and all of whose free variables also occur free in $(\nabla \vec{z} : \vec{\tau_z}.A)$. If the constant at the head of A is c, then the clause is said to be *for* c.[3] Let C denote a pre-definitional clause. A *schematic definitional clause* has the form $[\Psi]C$ where Ψ is a set of type variables; Ψ is said to be the set of types parameterizing the definitional clause. Finally, a *block of definitional clauses parameterized by types* or a *schematic definition block* is constituted by a finite set of type variables Ψ', a finite set of predicate constants $\{c_1 : \tau_1, \ldots, c_n : \tau_n\}$ and a collection of schematic definitional clauses each of which is for one of the constants in $c_1, \ldots, c_n$. Such a definitional block is said to be parameterized by the type variables in Ψ'.

Not all schematic definition blocks are considered to be well-formed. To have that property, they must satisfy the typing constraints that are described below.

Definition 3. *Given a schematic definitional block parameterized by Ψ' and its associated predicate constants $\{c_1 : \tau_1, \ldots, c_n : \tau_n\}$. Let $\Sigma = (c_1 : \tau_1, \ldots, c_n : \tau_n)$. Then this block is well-defined if for every clause $[\Psi]\forall \vec{x} : \vec{\tau_x}.(\nabla \vec{z} : \vec{\tau_z}.A) \triangleq B$ in it, Ψ is disjoint from Ψ', the typing judgments $\Psi', \Psi; \Sigma, \vec{x} : \vec{\tau_x}, \vec{z} : \vec{\tau_z} \Vdash_{\mathcal{K}} A :$ `prop` and $\Psi', \Psi; \Sigma, \vec{x} : \vec{\tau_x} \Vdash_{\mathcal{K}} B :$ `prop` hold, and all the type variables that occur in B also occur in $\nabla \vec{z} : \vec{\tau_z}.A$.*

Observe that the way we have phrased the definition above, the constants $c_1, \ldots, c_n$

[3]The expressions “atomic formula,” “formula” and the “head” of an atomic formula are being used loosely here because we have not yet enforced typing constraints on A and B. However, the structural properties being imposed should be clear and will become even more apparent after the definition of a well-defined schematic definition block that is provided below.

identified by a schematic definitional block are required to be used at their "defined types" at every occurrence in the block. Thus, these constants are schematic at the block level. When these constants are added to the signature for defining other blocks or in writing formulas to be proved, then each c_i is to be thought of as having the type expression $[\Psi']\tau_i$ associated with it, *i.e.*, it can be used at instances of its schematic type. Note also that we require that all the type variables in the body of a schematic definitional clause occur in its head. This requirement ensures that whenever the type variables in the head are fixed the type variables in the body are also fixed. As we shall see in Section 3.2.3, this is a desired property for case analysis on the schematic definitional clauses in the construction of schematic proofs.

A schematic definition block is intended to be an abbreviated representation of a collection of definition blocks in $\mathcal{G}$ that are obtained via type instantiations as follows. First, we instantiate the type variables that parameterize the block with concrete types. Then, within the structure of each "block" generated in this fashion, we generate all the versions of each schematic definitional clause it contains by instantiating the type variables that parameterize it with all available concrete types. Using Lemma 1, it can be seen that each block that results from this process is a well-formed definition block in $\mathcal{G}$. A point to be noted is that both the collection of definition blocks and the collection of definitional clauses within each block is sensitive to the vocabulary of types in existence at a particular point. However, the schematic proofs whose construction we will support will be such that they will allow us to prove only those statements whose instances have derivations in $\mathcal{G}$ independently of the available type signature.

We consider some examples to illustrate the definition we have presented. Each of the following expressions is a schematic definitional clause:

$$\begin{array}{lcl} \texttt{member}\ X\ (X :: L) & \triangleq & \top \\ \texttt{member}\ X\ (Y :: L) & \triangleq & \texttt{member}\ X\ L \end{array}$$

The set of variables that parameterize each of the clauses above is empty. Assuming the type schema we have provided earlier for `::`, combining these clauses with the predicate signature `member` : $A \rightarrow$ `list` $A \rightarrow$ `prop` and parameterizing the result by A produces a well-formed schematic definition block; in ensuring that schematic definitional clauses within the block "type-check," we will have to use `::` in them at the type $A \to \texttt{list}\ A \to \texttt{list}\ A$. To get definition blocks in $\mathcal{G}$ from this schematic definition block, we would have to instantiate A with concrete types. If we instantiate it with $\texttt{tm}$, we would get the definition block

$$\begin{array}{lcl} \texttt{member}_{\texttt{tm}}\ X\ (X \,\texttt{::}_{\texttt{tm}}\, L) & \triangleq & \top \\ \texttt{member}_{\texttt{tm}}\ X\ (Y \,\texttt{::}_{\texttt{tm}}\, L) & \triangleq & \texttt{member}_{\texttt{tm}}\ X\ L \end{array}$$

We can similarly generate a definition block that, for instance, provides us clauses for $\texttt{member}_{\texttt{nat}}$.

As another example, suppose that our signature contains the following constant

$$\texttt{memb} : [B]B \to \texttt{list}\ B \to \texttt{o}$$

and then consider the following as a schematic definitional clause:

$$[A]\texttt{prog}\ \texttt{true}\ (\texttt{memb}_A\ X\ (X \,\texttt{::}\, L)) \triangleq \top$$

It represents a number of clauses for $\texttt{prog}$, one in fact for each concrete type that can be used to instantiate A.

Given the schematic definitions, we would like to state theorems about them. We adopt a schematic view of such theorems as well. A *schematic theorem* has the form $[A_1, \ldots, A_n]F$ that is such that $A_1, \ldots, A_n; \emptyset \Vdash F : \texttt{prop}$ holds; intuitively, F must be a closed formula such that all the type variables occurring in it are contained in $\{A_1, \ldots, A_n\}$. We say that such a theorem is parameterized by the type variables $A_1, \ldots, A_n$. Given the ground types $\tau_1, \ldots, \tau_n$, we can generate the theorem $F[\tau_1/A_1, \ldots, \tau_n/A_n]$ in $\mathcal{G}$. In effect, a schematic theorem parameterized by a non-empty set of type variables stands for an infinite collection of theorems in $\mathcal{G}$ under the instantiation of its parameterizing type variables with ground types. In other words, a theorem of this kind should be provable with our extended machinery only if every type instance is provable in $\mathcal{G}$. A schematic theorem not parameterized by any type variable coincides with a theorem in $\mathcal{G}$.

As an example, consider the pruning property of $\texttt{member}$ we mentioned in Section 3.2.1. Given variables M of type $(B \to A)$, L of type $\texttt{list}\ A$, x of type B and M' of type A, that theorem can be formulated as the following schematic theorem where $\texttt{member}$ is an instance of the membership predicate and has the type

$A \to \texttt{list}\ A \to \texttt{prop}$:

$$[A, B]\forall M, L.\nabla x.\texttt{member}\ (M\ x)\ L \supset \exists M'.M = y \setminus M'.$$

By instantiating B with $\texttt{tm}$ and A with $\texttt{tm}$, we obtain the pruning property for lists of $\texttt{tm}$, where M is of type ($\texttt{tm} \to \texttt{tm}$), L is of type $\texttt{list tm}$, x and M' is of type $\texttt{tm}$:

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

Similarly, by instantiating B with $\texttt{tm}'$ and A with $\texttt{tm}'$, we obtain the pruning property for lists of $\texttt{tm}'$. We shall discuss an approach to proving such schematic theorems in the next section.

The ability to parameterize each definitional clause by type variables allows us to encode HH^ω in a situation where specifications utilize the schematic polymorphism supported by λProlog. Crucial to the embedding of HH^ω in $\mathcal{G}$ are definitional clauses for $\texttt{prog}$ that encode HH^ω clauses. However, we have already seen how this encoding can be realized. For example, suppose that the HH^ω program has the clause

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

where $\texttt{memb}$ has been defined to have the type $A \to \texttt{list}\ A \to \texttt{o}$ that is polymorphic in the λProlog sense. We would first translate $\texttt{memb}$ into a constant of the same name that has a type scheme associated with it and we would add the clauses

$[A]\texttt{prog}\ \texttt{true}\ (\texttt{memb}_A\ X\ (X :: L)) \triangleq \top$
$[A]\texttt{prog}\ (\texttt{memb}_A\ X\ L)\ (\texttt{memb}_A\ X\ (Y :: L)) \triangleq \top$

to the appropriate schematic definition block.

In Section 3.1.3, we pointed out a problem in the encoding of HH^ω that manifest itself in the following clauses that treat quantifiers:

$$\begin{array}{lcl}
\texttt{seq}\ L\ (\Pi_\tau\ G) & \triangleq & \nabla x : \tau.\texttt{seq}\ L\ (G\ x) \\
\texttt{bch}\ L\ (\Pi_\tau\ D)\ A & \triangleq & \exists t : \tau.\texttt{bch}\ L\ (D\ t)\ A \\
\texttt{atomic}\ (\Pi_\tau\ G) & \triangleq & \bot
\end{array}$$

This problem is easily solved with our extended syntax. In particular, our encoding can use the following schematic definitional clauses:

$$
\begin{array}{lcl}
[\tau]\mathtt{seq}\ L\ (\Pi_\tau\ G) & \triangleq & \nabla x:\tau.\mathtt{seq}\ L\ (G\ x) \\
[\tau]\mathtt{bch}\ L\ (\Pi_\tau\ D)\ A & \triangleq & \exists t:\tau.\mathtt{bch}\ L\ (D\ t)\ A \\
[\tau]\mathtt{atomic}\ (\Pi_\tau\ G) & \triangleq & \bot
\end{array}
$$

Our extension also allows us to formulate theorems about polymorphic specifications in HH$^\omega$. For example, we can state the pruning property of `memb` that we discussed in the previous subsection as the following schematic theorem, where M is of type $(B \to A)$, L is of type `list` A, x is of type B and M' is of type A:

$$[A, B]\forall M, L.\nabla x.\{\mathtt{memb}\ (M\ x)\ L\} \supset \exists M'.M = y \setminus M'.$$

3.2.3 Proving schematic theorems

A schematic theorem stands for a possibly infinite collection of theorems obtained by instantiating its parameterizing type variables with concrete types. One possible way to try to prove such a theorem is by providing a collection of possibly different proofs that cover all the type instances. However, there are two drawbacks with this approach. First, it could involve the kind of duplication of effort that we are wanting to avoid by introducing schematic polymorphism. Second, and perhaps more importantly, we do not intend the schematic theorem to be true for only those concrete type instances that are known in a particular context. In fact, we intend these theorems to hold for any vocabulary of types; the “closed world assumption” applies only to the collection of defining clauses available for a predicate.

To avoid these pitfalls, we propose to support the construction of only those proofs for schematic theorems that work *the same way* at *all* types, *i.e.*, our proofs must be structures that yield concrete proofs for concrete types simply through the process of type instantiation. We refer to such proofs as *schematic proofs.* The basis for realizing our approach is to lift the proof rules for $\mathcal{G}$ to a set of *schematic proof rules*; each of these schematic proof rules must yield actual proof rules in $\mathcal{G}$ under type instantiation. Most of the proof rules for $\mathcal{G}$ are easily made schematic. The one complicated case is that of the *def$\mathcal{L}$* rule that supports case analysis. Much of the discussion in this subsection is devoted to a consideration of this rule.

To begin the formalization process, we will assume that our schematic proof rules derive *schematic sequents* that have the following form

$$\Psi; \Sigma : \Gamma \longrightarrow B.$$

These sequents augment those in $\mathcal{G}$ with a set Ψ of type variables that binds the type variables in Σ, Γ and B. This set will remain unchanged throughout the derivation of the sequent. Thus, the variables in this set will function as placeholders for arbitrary types but will be like "black boxes" in that they will not allow us to look at or use the particular structures of the types that fill them. To prove a schematic theorem $[A_1, \ldots, A_n]F$, we need to derive the sequent $A_1, \ldots, A_n; \emptyset : \emptyset \longrightarrow F$.

The schematic versions of the core rules in Figure 2.2 are shown in Figure 3.5. They are obtained from the rules in Figure 2.2 by abstracting their conclusions and premises over the set Ψ of type variables. It is easy to see the schematic nature of these rules: by instantiating the premises and conclusions of each rule with any substitution of ground types for variables in Ψ, we get a rule in $\mathcal{G}$.

To describe schematic forms of the *def*$\mathcal{L}$ and *def*$\mathcal{R}$ rules, we introduce the notion of *the reduced form* of a schematic definitional clause.

Definition 4. *Let $[\Psi]C$ be a schematic definitional clause in a schematic definition block parameterized by Ψ'. Then the reduced form of $[\Psi]C$ is $[\Psi'']C$ where Ψ'' is the subset of $\Psi \cup \Psi'$ such that $A \in \Psi''$ if and only if A occurs in C.*

Intuitively, the reduced form of a schematic definitional clause is obtained by "pulling down" the type parameters at the block level to the clause level and by further removing type variables that do not occur in the clause. Note that because the type variables in the body of the clause must occur in its head, Ψ'' contains exactly the variables that occur in the clause head.

The schematic version of *def*$\mathcal{R}$ rule can be easily derived by abstracting it over a set Ψ of type variables as before. To present this rule we need to formalize the notion of instances of schematic definitional clauses, as follows. Given a schematic definitional clause $[A_1, \ldots, A_n]\forall \vec{x}.(\nabla \vec{z}.A) \triangleq B$, a type substitution $\Phi = [\tau_1/A_1, \ldots, \tau_n/A_n]$ and a substitution θ that assigns distinct nominal constants to $\vec{z}$ and terms not containing

$$\frac{B \approx B'}{\Psi; \Sigma : \Gamma, B \longrightarrow B'}\ \mathit{s\text{–}id} \qquad \frac{\Psi; \Sigma : \Gamma \longrightarrow B \quad B \approx B' \quad \Psi; \Sigma : \Gamma, B' \longrightarrow C}{\Psi; \Sigma : \Gamma \longrightarrow C}\ \mathit{s\text{–}cut}$$

$$\frac{\Psi; \Sigma : \Gamma, C, C \longrightarrow B}{\Psi; \Sigma : \Gamma, C \longrightarrow B}\ s\text{–}c\mathcal{L} \qquad \frac{}{\Psi; \Sigma : \Gamma, \bot \longrightarrow B}\ s\text{–}\bot\mathcal{L} \qquad \frac{}{\Psi; \Sigma : \Gamma \longrightarrow \top}\ s\text{–}\top\mathcal{R}$$

$$\frac{\Psi; \Sigma : \Gamma, B_i \longrightarrow C}{\Psi; \Sigma : \Gamma, B_1 \wedge B_2 \longrightarrow C}\ s\text{–} \wedge \mathcal{L}, i \in \{1,2\} \qquad \frac{\Psi; \Sigma : \Gamma \longrightarrow B \quad \Psi; \Sigma : \Gamma \longrightarrow C}{\Psi; \Sigma : \Gamma \longrightarrow B \wedge C}\ s\text{–} \wedge \mathcal{R}$$

$$\frac{\Psi; \Sigma : \Gamma, B \longrightarrow D \quad \Psi; \Sigma : \Gamma, C \longrightarrow D}{\Psi; \Sigma : \Gamma, B \vee C \longrightarrow D}\ s\text{–} \vee \mathcal{L}$$

$$\frac{\Psi; \Sigma : \Gamma \longrightarrow B_i}{\Psi; \Sigma : \Gamma \longrightarrow B_1 \vee B_2}\ s\text{–} \vee \mathcal{R}, i \in \{1,2\}$$

$$\frac{\Psi; \Sigma : \Gamma \longrightarrow B \quad \Psi; \Sigma : \Gamma, C \longrightarrow D}{\Psi; \Sigma : \Gamma, B \supset C \longrightarrow D}\ s\text{–} \supset \mathcal{L} \qquad \frac{\Psi; \Sigma : \Gamma, B \longrightarrow C}{\Psi; \Sigma : \Gamma \longrightarrow B \supset C}\ s\text{–} \supset \mathcal{R}$$

$$\frac{\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t : \tau \quad \Psi; \Sigma : \Gamma, B[t/x] \longrightarrow C}{\Psi; \Sigma : \Gamma, \forall_\tau x.B \longrightarrow C}\ s\text{–}\forall\mathcal{L}$$

$$\frac{\Psi; \mathcal{C}, \Sigma \Vdash_{\mathcal{K}} t : \tau \quad \Psi; \Sigma : \Gamma \longrightarrow B[t/x]}{\Psi; \Sigma : \Gamma \longrightarrow \exists_\tau x.B}\ s\text{–}\exists\mathcal{R}$$

$$\frac{\Psi; \Sigma, h : \tau' : \Gamma, B[(h\ a_1\ \ldots\ a_n)/x] \longrightarrow C}{\Psi; \Sigma : \Gamma \exists_\tau x.B \longrightarrow C}\ s\text{–}\exists\mathcal{L}$$

$$\frac{\Psi; \Sigma, h : \tau' : \Gamma \longrightarrow B[(h\ a_1\ \ldots\ a_n)/x]}{\Psi; \Sigma : \Gamma \longrightarrow \forall_\tau x.B}\ s\text{–}\forall\mathcal{R}$$

assuming that $\mathit{supp}(B) = \{a_1, \ldots, a_n\}$, that, for $1 \leq i \leq n$, a_i has type τ_i
h is variable of type $\tau_1 \to \ldots \to \tau_n \to \tau$ and $h \notin \mathit{dom}(\Sigma)$ in $s\text{–}\forall\mathcal{R}$ and $s\text{–}\exists\mathcal{L}$

$$\frac{\Psi; \Sigma : \Gamma, B[a/x] \longrightarrow C}{\Psi; \Sigma : \Gamma, \nabla_\tau x.B \longrightarrow C}\ s\text{–}\nabla\mathcal{L} \qquad \frac{\Psi; \Sigma : \Gamma \longrightarrow B[a/x]}{\Psi; \Sigma : \Gamma \longrightarrow \nabla_\tau x.B}\ s\text{–}\nabla\mathcal{R}$$

where $a \notin \mathit{supp}(B)$ in $\nabla\mathcal{L}$ and $\nabla\mathcal{R}$

Figure 3.5: The Schematic Core Rules

such constants to $\vec{x}$, we say that $A[\theta][\Phi] \triangleq B[\theta][\Phi]$ is an instance of the original clause. Then the schematic definition right rule is depicted in Figure 3.6.

$$\frac{\Psi; \Sigma : \Gamma \longrightarrow B}{\Psi; \Sigma : \Gamma \longrightarrow p\,\vec{t}}\ s\text{–}\mathit{def}\mathcal{R}$$

where $p\,\vec{t} \triangleq B$ is an instance of the reduced form of some schematic definitional clause

Figure 3.6: The Schematic Definition Right Rule

We will consider treating atomic formulas on the assumptions side of a sequent only by a schematic version of the $\mathit{def}\mathcal{L}_{CSU}$ rule. In describing a lifted form of this rule, we have to pay attention to the fact that the structure of unifiers in the context of the simply typed λ-calculus can depend on the particular types assigned to constants [63]. To get around this issue, we will limit the schematic version of the rule to apply only in those cases where the complete set of unifiers can be calculated without paying attention to types. The following definition provides the basis for doing so by lifting the notion of a CSU to schematic terms.

Definition 5. *Given two schematic terms B and C of the same type parameterized by $A_1, \ldots, A_n$, a type generic complete set of unifiers (type generic CSU) for B and C, denoted by $CSU_{gen}(B, C)$, is a set of substitutions for variables in B and C such that for any ground types $\tau_1, \ldots, \tau_n$ the set $\{\theta[\tau_1/A_1, \ldots, \tau_n/A_n] \mid \theta \in CSU_{gen}(B, C)\}$ is a complete set of unifiers for $B[\tau_1/A_1, \ldots, \tau_n/A_n]$ and $C[\tau_1/A_1, \ldots, \tau_n/A_n]$.*

One approach to calculating type generic CSUs for two terms is to use the unification algorithm due to Miller that is known as *pattern unification* [64]. This approach in fact suffices for all the theorems we have considered in the work in this thesis.

A schematic version of the $\mathit{def}\mathcal{L}_{CSU}$ rule will require us to analyze all the ways in which an atomic formula can match with the head of a schematic definitional clause. In doing so, we need to consider the possibility that there are nominal constants in the formula and, conversely, that the ∇ quantifiers in the head will be instantiated with nominal constants. To deal with this issue, we once again use the idea of raising. In Section 2.2.5, we have defined the notions of raising a definitional clause over a sequence of nominal constants and away from the variables in a sequent, and of raising a sequent over a sequence of nominal constants. These notions are, in a sense, orthogonal to typing structure and therefore adapt in an obvious way to the situation of schematized sequents and definitional clauses. We shall assume such an adaptation here.

For us to be able to use case analysis over a particular atomic formula in proving a schematic sequent in a schematic way, it is necessary for the analysis to be generic with respect to *all* available schematic definitional clauses. We identify a set of requirements that ensure this to be the case below.

Definition 6. *Let* $S = (\Psi; \Sigma : \Gamma, p\ \vec{t} \longrightarrow D)$ *be a schematic sequent and let* $supp(p\ \vec{t})$ *be* $\{\vec{a}\}$. *Let* C *be a schematic definitional clause and* $[A_1, \dots, A_n]\forall \vec{x}.(\nabla \vec{z}.A) \triangleq B$ *be* C *raised over* $\vec{a}$ *and away from* Σ. *Also let* $\Psi \cup \{A_1, \dots, A_n\}; \Sigma' : \Gamma', p\ \vec{t'} \longrightarrow D'$ *be a version of the original sequent raised over a sequence of distinct nominal constants* $\vec{c}$ *for* $\vec{z}$. *Then* S *is said to be analyzable in a generic way with respect to the clause* C *and the atomic formula* $(p\ t)$ *if one of the following conditions hold for every permutation* π *of the nominal constants* $\{\vec{c}, \vec{a}\}$*:*

1. $(p\ \vec{t'})$ *and* $(\pi.A[\vec{c}/\vec{z}])$ *are not unifiable under any instantiation of type variables.*
2. *For some type expressions* $\tau_1, \dots, \tau_n$ *whose type variables are bound in* Ψ, *these is a type generic CSU for* $(p\ \vec{t'})[\tau_1/A_1, \dots, \tau_n/A_n]$[4] *and* $(\pi.A[\vec{c}/\vec{z}])[\tau_1/A_1, \dots, \tau_n/A_n]$ *and for any other type substitution* Φ *for* $A_1, \dots, A_n$, *it is the case that the formulas* $(p\ \vec{t'})[\Phi]$ *and* $(\pi.A[\vec{c}/\vec{z}])[\Phi]$ *are not unifiable.*

The sequent S *is said to be amenable to case analysis with respect to the atomic formula* $(p\ t)$ *if it is analyzable with respect to the reduced version of every available definitional clause.*

The condition that the definition above requires to hold with respect to each schematic definitional clause may be understood as follows. The first possibility applies when the formula being analyzed cannot match with the head of the clause no matter how the type variables are instantiated. The second possibility applies when the formula being analyzed can match with the head of the schematic definitional clause. The requirement in this case has two parts. First, case analysis should uniquely fix the type variables

[4]These type substitutions must be applied to $(p\ t')$ as well because the type variables $A_1, \dots, A_n$ may appear in the types of the formula via the nominal constants in $\vec{c}$.

parameterizing the (reduced) schematic definitional clause[5]; without this, we would have to explore different possible instantiations for these type variables for the rule to be sound. Second, case analysis via the schematic definition left rule should be type generic. If one of these possibilities hold, then the effect of the schematic definitional clause on the case analysis is guaranteed to be independent of the types involved.

To actually articulate the schematic version of the $def\mathcal{L}_{CSU}$ rule, we have to identify the set of premises arising from any given definitional clause. Towards this end, we define the relation *defl_gen_premise* which is similar to *defl_csu_premise* used for $def\mathcal{L}_{CSU}$ in Section 2.2.5.

Definition 7. *Let H be the schematic sequent $\Psi; \Sigma : \Gamma, p\ \vec{t} \longrightarrow F$ and let C be the schematic definitional clause $[A_1, \ldots, A_n]\forall\vec{x}.(\nabla\vec{z}.A) \triangleq B$. Further, let $supp(p\ \vec{t})$ be $\{\vec{a}\}$ and let $\vec{c}$ be a sequence of nominal constants that is of the same length as $\vec{z}$ and such that each constant in the sequence has a type identical to that of the corresponding variable in $\vec{z}$ and is also distinct from the constants in $\vec{a}$. Finally, let $[A_1, \ldots, A_n]\forall\vec{h}.(\nabla\vec{z}.A') \triangleq B'$ be a version of the clause C raised over $\vec{a}$ and let $\Sigma' \cup \{A_1, \ldots, A_n\} : \Gamma', p\ \vec{t'} \longrightarrow F'$ be a version of H raised over $\vec{c}$. Then*

$$\begin{aligned} defl\_gen\_premise(H, p\ \vec{t}, C) = \{ & \Sigma'[\theta] : \Gamma'[\theta], \pi.B'[\theta] \longrightarrow F'[\theta] \mid \\ & \pi \text{ is a permutation of the nominal constants in } \{\vec{c}, \vec{a}\} \\ & \text{and } \Phi = (\tau_1/A_1, \ldots, \tau_n/A_n) \text{ is a type substitution} \\ & \text{such that } CSU_{gen}((p\ \vec{t'})[\Phi], (\pi.A'[\vec{c}/\vec{z}])[\Phi]) \\ & \text{exists and } \theta \in CSU_{gen}((p\ \vec{t'})[\Phi], (\pi.A'[\vec{c}/\vec{z}])[\Phi])\}. \end{aligned}$$

The schematic definition left rule $s\text{–}def\mathcal{L}_{CSU}$ is shown in Figure 3.7. This rule has a proviso associated with it: it is only applicable if the lower sequent of the rule is amenable to case analysis with respect to $(p\ \vec{t})$ as formally described in Definition 6.

Up to this point, we have allowed schematic definition blocks only to be interpreted as abbreviations for collections of fixed-point definitions in $\mathcal{G}$. We actually allow such blocks also to be treated as generators of inductive definitions. For a block to be designated in this way, it must be the case that the schematic definitional clauses within

[5]Since these variables are exactly the type variables in the clause head and is a superset of the variables in the clause body, case analysis only needs to fix the type variables in the clause head through unification.

$$\frac{\{H \in \mathit{defl\_gen\_premise}(\Psi; \Sigma : \Gamma, p\ \vec{t} \longrightarrow F, p\ \vec{t}, C) \mid C \in \mathcal{D}_{\mathcal{S}}\}}{\Psi; \Sigma : \Gamma, p\ \vec{t} \longrightarrow F}\ s\text{–}\mathit{def}\mathcal{L}_{CSU}$$

where $\mathcal{D}_{\mathcal{S}}$ is the set of all the reduced forms of the schematic definitional clauses obtained from all the schematic definition blocks

Figure 3.7: The Schematic Definition Left Rule

it are each parameterized by an empty set of type variables; when this constraint is met, each schematic definitional clause gives rise to exactly one clause in the definition block in $\mathcal{G}$ that is generated by instantiating the type variables parameterizing the block with concrete types. Furthermore, for each clause $\forall\vec{x}.\nabla\vec{z}.p\ \vec{t} \triangleq B$ in the block, the type variables in the type of p must contain all the type variables in $\nabla\vec{z}.p\ \vec{t}$. As a result, the type of the entire clause is fixed when the type of p is fixed.

We provide the following auxiliary definition for formalizing a schematic induction rule.

Definition 8. *Let B be an inductive schematic definition block that has associated with it only the predicate constant p. Then the clauses in B for a type instance $p_{\vec{\tau}}$ of p are the instances of clauses in B obtained by instantiating their parameterizing type variables with $\vec{\tau}$.*

The schematic induction rule is then given in Figure 3.8, which is very similar to the induction rule in $\mathcal{G}$.

$$\frac{\{\Psi; \vec{x} : D[S/p_{\vec{\tau}}] \longrightarrow \nabla\vec{z}.S\ \vec{t} \mid \forall\vec{x}.\nabla\vec{z}.p_{\vec{\tau}}\ \vec{t} \triangleq D \in \mathcal{C}\} \quad \Psi; \Sigma : \Gamma, S\ \vec{t} \longrightarrow C}{\Psi; \Sigma : \Gamma, p_{\vec{\tau}}\ \vec{t} \longrightarrow C}\ s\text{–}\mathcal{IL}$$

if B, the schematic block for p, has only p associated with it
and $\mathcal{C}$ is the set of all the clauses in B for $p_{\vec{\tau}}$
and S is a term with no nominal constants and of the same type as $p_{\vec{\tau}}$

Figure 3.8: The Schematic Induction Rule

The "proofs" constructed using the schematic proof rules are schemata for generating actual proofs in $\mathcal{G}$ under the instantiation of type variables. This property is stated as follows:

Theorem 1. *If a schematic sequent $\Psi; \Sigma : \Gamma \longrightarrow B$ is derivable by using the schematic proof rules in Figures 3.5, 3.6, 3.7 and 3.8, then given any type substitution Φ for variables in Ψ such that $\emptyset \vdash \Phi : \Psi$ holds, there exists a proof for $\Sigma[\Phi] : \Gamma[\Phi] \longrightarrow B[\Phi]$ in $\mathcal{G}$.*

Proof. We proceed by induction on the derivation of $\Psi; \Sigma : \Gamma \longrightarrow B$, analyzing the last schematic rule it uses. In most cases, the argument follows a set pattern: we invoke the inductive hypothesis on the premises of the last rule, we then apply the non-schematic version of the rule to conclude. When the last rule is either s–$\exists\mathcal{R}$ or s–$\forall\mathcal{L}$, we apply Lemma 1 to its left premise to get a well-typed term for substituting for the binding variable. The only case that needs further explanation is when the last rule is s–$def\mathcal{L}_{CSU}$. Given any schematic definitional clause, the provisos of this rule ensure that if the matching between the atomic formula being analyzed with the head of the clause fails, then it will also fail under the instantiation of type variables. The provisos also ensure that when the matching succeeds, it also succeeds under the type instantiation. Moreover, the type generic natural of the matching ensures the structure of the premise generated from the matching is preserved under the type instantiation. From these observations, we can follow the set pattern to finish the proof for this case. $\square$

The proof of the above theorem is constructive. Its procedural interpretation provides us a function for constructing proofs in $\mathcal{G}$ from the schematic proofs. An easy consequence of the theorem is also the following:

Corollary 1. *If a schematic theorem $[A_1, \ldots, A_n]F$ is provable by using schematic proof rules, then given any ground types $\tau_1, \ldots, \tau_n$, $F[\tau_1/A_1, \ldots, \tau_n/A_n]$ is provable is $\mathcal{G}$.*

We have implemented the schematic polymorphism extension in Abella. The interactive style of reasoning works mostly in the same way as before with our polymorphism extension. One difference is that Abella now keeps track of the type variables parameterizing schematic theorems during the proof developments, which are like constants in that they cannot be instantiated and are also like unknown objects in that they cannot be compared with other types for equality. Another difference is that when performing case analysis, Abella checks if the unification problems can be solved in a way that satisfies the provisos of the s–$def\mathcal{L}_{CSU}$ rule. If not, for example, if unification needs to compare two different parameterizing variables for equality, then the application of the case analysis tactic fails.

3.2.4 Some example interactive proof developments

To illustrate the usefulness of the schematic polymorphism extension, let us consider proving the pruning property of the membership relation using the extension. The schematic definition of the membership relation has already been given in Section 3.2.2. So does the schematic theorem that states the pruning property. We repeat the theorem as follows where M is of type $(B \to A)$, L is of type `list` A, x is of type B and M' is of type A:

$$[A, B]\forall M, L.\nabla x.\texttt{member}\ (M\ x)\ L \supset \exists M'.M = y \setminus M'.$$

In Section 3.2.1, we have discussed the proofs of the two instances of this pruning property obtained by instantiating A and B with `tm` and with `tm'`, respectively. The proofs for these instances have exactly the same structure and do not rely on the type information. This implies a schematic proof exists for this theorem. We show that this is indeed the case by formally constructing a schematic proof for it.

At the beginning of the proof development, A and B are marked as type variables that need to be tracked implicitly. The theorem to be proved becomes

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

We perform an induction on the only assumption, introducing the following inductive hypothesis `IH`:

$$\forall M, L.\nabla x.(\texttt{member}\ (M\ x)\ L)^* \supset \exists M'.M = y \setminus M'$$

The proof state is changed to have the following hypothesis

$$\texttt{H1}: \quad (\texttt{member}\ (M\ \texttt{n})\ L)^@$$

where L, M are new variables and `n` is a nominal constant. The conclusion to be proved becomes $\exists M'.M = y \setminus M'$. We then unfold `H1`, resulting in two cases.

In the first case, we have $L = E :: L'$ and $(M\ \mathtt{n}) = E$ for some variables E and L'. By solving the unification problem $(M\ \mathtt{n}) = E$, we get $M = y \backslash E$. We can now close this branch by instantiating M' with E.

In the second case, we have $L = E :: L'$ and the following new hypothesis

$$\mathtt{H2}: \quad (\mathtt{member}\ (M\ \mathtt{n})\ L')^*$$

Because H2 comes from the unfolding of H1, it is *-annotated, indicating it is derivable in fewer steps than H1. We can therefore apply IH to H1, yielding $\exists M'.M = y \backslash M'$ which is exactly the conclusion we want to prove. This concludes the proof.

Notice that when we do case analysis in the above proof, we have not inspected the types of terms because the solutions to the unification problems are completely determined by the structure of terms and are oblivious to any type information. Applying the case analysis tactic here corresponds to applying the *s–def*$\mathcal{L}_{CSU}$ rule whose provisos are satisfied because of the type generic nature of these unification problems.

We have also given the encoding of the membership relation in λProlog and the theorem describing its pruning property in Section 3.2.2. We repeat the theorem as follows where M is of type $(B \to A)$, L is of type $\mathtt{list}\ A$, x is of type B and M' is of type A:

$$[A, B]\forall M, L.\nabla x.\{\mathtt{memb}\ (M\ x)\ L\} \supset \exists M'.M = y \backslash M'.$$

We can prove this theorem by induction on the only assumption and by following conceptually the same steps as in the previous proof. However, there is a significant difference in terms of how case analysis is actually carried out. After unfolding $\{\mathtt{memb}\ (M\ \mathtt{n})\ L\}$ (where M and L are variables and $\mathtt{n}$ is a nominal constant), we need to consider unifying the formula $\mathtt{prog}\ G\ (\mathtt{memb}\ (M\ \mathtt{n})\ L)$ where G is some variable with the heads of the following schematic definitional clauses:

$$\begin{array}{lcl} [A']\mathtt{prog}\ \mathtt{true}\ (\mathtt{memb}_{A'}\ X\ (X :: L)) & \triangleq & \top \\ [A'']\mathtt{prog}\ (\mathtt{memb}_{A''}\ X\ L)\ (\mathtt{memb}_{A''}\ X\ (Y :: L)) & \triangleq & \top. \end{array}$$

Note that the unification problems have solutions only when A' and A'' unify with the type A. That is, only the following two clauses among the instances of the above schematic definitional clauses are relevant to our case analysis; all the other cases end in unification failure:

$$\begin{array}{lcl}\texttt{prog}\ \texttt{true}\ (\texttt{memb}_A\ X\ (X::L)) & \triangleq & \top \\ \texttt{prog}\ (\texttt{memb}_A\ X\ L)\ (\texttt{memb}_A\ X\ (Y::L)) & \triangleq & \top.\end{array}$$

As in the previous proof, the case analysis on those two clauses proceeds in a type generic fashion. As a result, the provisos of the *s–def*$\mathcal{L}_{CSU}$ rule are satisfied and a schematic proof can be constructed using our polymorphism extension.

As another example, consider the definition of substitutions as relations. We have given a definition of the substitution relation on the STLC terms in Section 2.4.2. It can be generalized to substitutions at any type as follows. We first identify the type $\texttt{map} : \texttt{type} \to \texttt{type} \to \texttt{type}$ for mappings and its constructor $\texttt{map} : A \to B \to \texttt{map}\ A\ B$. A substitution is represented as a list of mappings. We then identify the substitution relation as the predicate symbol $\texttt{app\_subst} : \texttt{list}\ (\texttt{map}\ A\ A) \to B \to B \to \texttt{prop}$ defined through the following clauses:

$$\begin{array}{lcl}\texttt{app\_subst}\ \texttt{nil}\ M\ M & \triangleq & \top \\ \nabla x.\texttt{app\_subst}\ ((\texttt{map}\ x\ V)::S)\ (R\ x)\ M & \triangleq & \texttt{app\_subst}\ S\ (R\ V)\ M\end{array}$$

By the definition, $\texttt{app\_subst}\ S\ M\ M'$ holds exactly when M' is the result of applying the substitution S to M. Given this schematic definition, we can easily prove general properties about the substitution relation in Abella. The following are some examples:

$$\begin{array}{l}[A,B]\forall(S:\texttt{list}\ (\texttt{map}\ A\ A)),(M:B),M',M''. \\ \quad \texttt{app\_subst}\ S\ M\ M' \supset \texttt{app\_subst}\ S\ M\ M'' \supset M' = M''. \\ [A,B,C]\forall(S:\texttt{list}\ (\texttt{map}\ A\ A)),(M:B),M'.\nabla(x:C). \\ \quad \texttt{app\_subst}\ S\ M\ (M'\ x) \supset \exists M''.M' = y \setminus M''. \\ [A,B,C]\forall(S:\texttt{list}\ (\texttt{map}\ A\ A)),(M_1:C),(M_2:C\to B),M_1',M_2'.\nabla(x:C). \\ \quad \texttt{app\_subst}\ S\ M_1\ M_1' \supset \texttt{app\_subst}\ S\ (M_2\ x)\ (M_2'\ x) \\ \qquad \supset \texttt{app\_subst}\ S\ (M_2\ M_1)\ (M_2'\ M_1').\end{array}$$

The first theorem states that substitution is deterministic. The second theorem is a pruning property of the substitution relation. The last theorem is an “instantiation” theorem for the substitution relation. All of these theorems have schematic proofs because their proof construction does not make use of any type information.

Chapter 4

A Structure for Verified Compilation

A common way to structure the compilation of functional programs is as a *multi-pass* process: Given a particular program, the compiler transforms it through a sequence of steps into successively lower-level forms till eventually it has produced code in the desired target language, which might be an assembler or a low-level intermediate language. It is this kind of a multi-pass compiler that will be considered in this thesis.

A natural way to specify a compiler transformation is to describe it via a set of rules in the Structural Operational Semantics (SOS) style [65] that derive relations on the source and target programs of the transformation. The specification of a multi-pass compiler then comprises the rule-based relational specifications of its transformations. To guarantee the correctness of a compiler, we must prove that the meaning or *semantics* of the source language program is preserved by the target language program that is eventually produced. In this thesis, like in other compiler verification projects such as CompCert [20] or CakeML [24], our interest is in showing that the compiler preserves the meaning of closed programs at atomic types. One could try to achieve this goal by taking on the verification of the entire compiler in one step (*e.g.*, see [66]). We adopt an alternative approach here. In this approach we prove that each individual transformation preserves the semantics of programs and we then compose these results to obtain the correctness property for the full compiler. We feel that this approach is more manageable because it takes advantage of the fact that each individual transformation is designed to achieve one specific objective. As such, semantics preservation for it has a clear definition and its proof is modular.

An obvious way to prove semantics preservation for compiler transformations described via rule-based relational specifications is to define semantics preservation as a relation between programs in the source and target languages and show that it subsumes the transformation relation. Since a compiler transformation in a multi-pass compiler may transform programs in one intermediate language into programs in a different language, the relation denoting semantics preservation must be able to relate programs across multiple languages. We use the device of *logical relations* [47] that have this property to denote semantics preservation. Because we are only interested in verifying compilation of closed programs at atomic types and logical relations are composible at atomic types, proofs of semantics preservation for the individual transformations can be composed to form the correctness proof for the full multi-pass compiler.

As we have discussed in the introduction chapter, a major difficulty in implementing and reasoning about the compilation of functional languages lies in modeling and reasoning about the binding structure of functional objects. Most existing proof systems for formal verification provide only very primitive support for dealing with binding structure. As a result, verification of compilers for functional languages in such systems tends to be unintuitive and laborious; a large amount of effort needs to be expended just to prove the “boilerplate” lemmas about notions related to binding structure.

The main goal of the thesis is to show that our extended framework is suitable for implementing and verifying compiler transformations on functional programs. In particular, we would like to show that the λ-tree syntax approach supported by the framework significantly simplifies the representation, manipulation, analysis and reasoning about binding structure in the implementation and verification of compiler transformations on functional programs. We achieve this goal by presenting a methodology that formalizes the approach to verified compilation described above using our framework. Under this methodology, we encode transformations on functional programs as λProlog specifications by utilizing the approach described in Section 2.1. Since λProlog specifications are executable, they serve also as implementations of the transformations and the compiler. We argue that the λ-tree syntax approach that is supported by λProlog provides a convenient way to realize manipulation and analysis of binding structure in these rules. Thanks to these features, the λProlog specifications of transformations on functional programs are concise and transparently relate to the original rule-based relational descriptions. Moreover, they have a logical structure that can be exploited in the process of reasoning about their correctness. In the methodology that we are proposing, showing the correctness of compiler transformations amounts to formally proving properties of the λProlog specifications of transformations on functional programs using Abella. More precisely, for every compiler transformation, we formalize the relation describing semantics preservation in Abella and then prove a theorem that this relation subsumes the relation that encodes the compiler transformation in λProlog. These theorems are then composed together to form the correctness proof for the full compiler. In the construction of the correctness proofs, we show that the λ-tree syntax approach supported by Abella provides a convenient way to formalize and prove properties about binding structure. To illustrate this methodology, we use it to implement a verified compiler for a representative functional programming language that is an extension of the Programming Computable Functions (PCF) language [42].

In the following sections, we elaborate on the above ideas. In Section 4.1 we present the model for compiling functional languages we work with in this thesis. In Section 4.2 we expand on the approach to verified compilation described in this preamble. In Section 4.3 we elaborate on the methodology for formalizing this approach using our framework. There exist many choices for the notion of semantics preservation beside logical relations. In Section 4.5 we shall compare those different notions and explain why the benefits of our framework in verified compilation on functional programs observed in the thesis can also be derived when other notions of semantics preservation are used. In Section 4.6 we give an overview of the exercise we shall carry out to verify the usefulness of our methodology for verified compilation of functional programs.

4.1 The Compilation Model

There exist two common models for compiling functional languages in multiple passes. One compiles functional programs into abstract machine code through a sequence of transformations [67, 68, 69, 70]. The compiled code is then executed on the run-time infrastructures of the abstract machines. The other model compiles functional programs into executable code for real hardware [71, 72, 73]. Compilation in this model usually comprises two phases. In the first phase, the higher-order functional programs go through several compiler transformations by which their higher-order features are gradually removed. The output of the first phase are programs that resemble those in procedural languages such as C in which all functions are at the top-level (*i.e.*, there are no nested functions) and the control flow is explicit. At this point, the conventional techniques for compiling procedural languages become applicable. In fact, the second phase of the compilation usually consists of a sequence of transformations that resemble those in compilers for procedural languages. It takes the output of the first phase as input and eventually generates executable code in the desired target language such as an assembly language or machine language.

In this thesis, we shall work with the latter model. Within this context, we will focus on the first phase described above for two reasons. First, the transformations in this phase involve complicated manipulation and analysis of binding structure and it is in encoding and verifying such transformations that the λ-tree syntax approach proves most useful. Second, there is already a large body of work devoted to verifying the compiler transformations in the second phase, such as the series of papers from the CompCert project [20, 43, 74]. Since the transformations in the second phase do not involve significant manipulation of binding structure, we do not have many new ideas to add to the work that has already been done in relation to this phase. Rather than repeating a development of existing ideas, we provide a minimal but functional implementation of the second phase and verify this implementation.

4.2 The Approach to Verified Compilation

In this section, we elaborate on our approach to specifying and verifying the multi-pass compilers for functional languages on paper. To specify a multi-pass compiler, we give every compiler transformation a rule-based relational description. Specifically, we characterize a transformation via a relation between programs in its source and target languages and a set of rules for deriving the relation. The characterization must satisfy the following property: a source program is transformed to a target program by the transformation if and only if the relation holds of them. In the setting of compiling functional programs, the transformation rules usually involve manipulation and analysis of binding structure. An example of such a specification is the transformation of λ-terms into their de Bruijn forms described in Section 3.1. The specifications for the sequence of transformations of a multi-pass compiler together constitute the specification of the compiler.

As we have noted already, a compiler that is composed of a sequence of transformations can be verified by showing that each transformation preserves the meaning or semantics of the program it is applied to. How exactly we do the latter will, of course, depend on how we characterize the semantics of programs. One way to do this is to examine how a program interacts with the outside world. This interaction is characterized by their *behaviors* that have obvious and fixed meanings, such as termination and I/O events. Let us write $t \Downarrow B$ to represent the judgment that the program t has the behavior B. We describe a development of the idea of semantics preservation based on such a judgment that has been used in the CompCert project by Xavier Leroy [20]. Let t' be the program that results from transforming the program t. Then we may say at the outset that t' preserves the semantics of t if the following property holds:

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

We write $F \iff F'$ here to denote that $F \Rightarrow F'$ and $F' \Rightarrow F$ hold where $F \Rightarrow F'$ ($F' \Rightarrow F$) itself denotes that F' (F) holds if F (F') holds. Now, the strict notion of behavior preservation presented above is often replaced by the notion of *behavior refinement* that corresponds to the following property:

$$\forall B. t' \Downarrow B \Rightarrow t \Downarrow B.$$

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

$$WellBehaved(t) \Rightarrow \forall B.t' \Downarrow B \Rightarrow t \Downarrow B$$

where $WellBehaved$ is defined as

$$WellBehaved(t) \iff (\forall B.t \Downarrow B \Rightarrow B \notin Wrong);$$

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 are only concerned 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.*

Forward simulation can have many different interpretations depending on the notion of equivalence it uses. In the setting of verifying multi-pass compilers for functional languages, this notion must possess the following properties. First, equivalence needs to be defined between not only atomic values but also function values. Second, since a compiler transformation may transform programs in one language into ones in a different language, the equivalence notion must be definable across multiple languages. We use logical relations that possess these properties as the notion of semantics preservation. We could have made a different choice as we discuss in Section 4.5, but this choice suffices to bring out the main ideas that are of concern in this thesis.

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. Identity would of course not be a good choice for equivalence at function types if a transformation is intended to accomplish something substantial. A better idea might be to base this equivalence on the idea of semantics preservation—which is characterized by simulation—when the two expressions are applied to 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. Note also that the relations $\sim$ and $\approx$ are defined only for closed terms since evaluation semantics does not exist for terms containing free variables.

In this thesis, we consider compiling functional languages with general recursion. In such a language, functions have the form $(\mathbf{fix}\ f\,x.t)$ where f is a binding variable for the function itself, x is the argument of the function and t is the function body that may refer to f and x. Applying $(\mathbf{fix}\ f\,x.t)$ to an argument t' has the effect of replacing f with the function itself and x with t' in t and recursively evaluating the resulting term $(t[(\mathbf{fix}\ f\,x.t)/f, t'/x])$. Since functions are arguments to themselves, the logical relation describing equivalence between them must have itself as an assumption. One may consider defining this equivalence relation as follows: 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 f, f', v_1 and v_1', if $f \approx_\tau f'$ $v_1 \approx_{\tau_1} v_1'$, then $(v\ f\ v_1) \sim_{\tau_2} (v'\ f'\ \ v_1')$.*

However, this relation is not well-defined because $f \approx_\tau f'$ occurs as an assumption: we are assuming that the equivalence relation $\approx_\tau$ is already known in its very definition.

We solve the above problem by using the idea of *step indexing* logical relations [75, 76, 77]. Specifically, we further index a logical relation with a natural number which stands for the maximum number of evaluation steps in which the two terms related by it cannot be distinguished from each other. We mutually recursively define the step-indexed simulation relation $\sim$ and the equivalence relation $\approx$ as follows. The relation $t \sim_{\tau;i} t'$ where i is the step index holds if and only if

> *for any value v and any j such that $j \leq i$, if t evaluates to v in j steps, then there exists some value v' such that t' evaluates to v' and $v \approx_{\tau;i-j} v'$ holds.*

If τ is an atomic type, $v \approx_{\tau;i} v'$ holds if and only if v and v' are identical. If τ is an arrow type $\tau_1 \to \tau_2$, $v \approx_{\tau;i} v'$ holds if and only if v and v' are functions such that

> *for any f, f', v_1 and v_1', and any j such that $j < i$, if $f \approx_{\tau;j} f'$ and $v_1 \approx_{\tau_1;j} v_1'$ hold, then $(v\ f\ v_1) \sim_{\tau_2;j} (v'\ f'\ \ v_1')$.*

Every assumption in this definition is either indexed by a smaller type or by the same type and a smaller evaluation steps. Therefore, we can view it as a recursive definition based on a collection of pairs of types and evaluation steps inductively defined by lexicographical ordering. We say t simulates t' in i steps at type τ if $t \sim_{\tau;i} t'$ holds. For a term t of type τ to simulate t', the simulation relation must hold of them at all steps, *i.e.*, $t \sim_{\tau;i} t'$ holds for any i. We use $t \sim_{\tau} t'$ to represent this relation, which coincides with the notation for simulation when no step-indexing is involved.

Given the above definitions, semantics preservation for a transformation P can be stated as follows:

Property 1. *If a closed term t of type τ is transformed into t' by P, then $t \sim_{\tau} t'$ holds.*

When the transformation P is given as a rule-based description of relations, the property above is equivalent to saying that the simulation relation subsumes the transformation relation. When step-indexing logical relations are used, this property is equivalent to the following

Property 2. *For any i, if a closed term t of type τ is transformed into t' by P, then $t \sim_{\tau;i} t'$ holds.*

This is the property we would like to prove towards showing the correctness of the transformation P.

We may consider proving Property 2 by induction on the derivation of the transformation relation. However, this will not work because transformations on functional programs often manipulate expressions within the scope of an abstraction and, in this sense, work recursively on open terms. In this case the inductive hypothesis is not applicable because it assumes terms involved in the transformation are closed. We solve this problem by generalizing Property 2 to accommodate open terms by relating them under closed substitutions. Specifically, given a transformation P, assume θ is a substitution $(v_1/x_1, \ldots, v_n/x_n)$ in its source language, θ' is a substitution $(v'_1/x'_1, \ldots, v'_n/x'_n)$ in its target language, where $v_1, \ldots, v_n, v'_1, \ldots, v'_n$ are closed values and x_i is mapped to x'_i by P for $1 \leq i \leq n$. We use $\theta \approx_{\Gamma;i} \theta'$ where Γ is a type context $(x_1 : \tau_1, \ldots, x_n : \tau_n)$ to denote that $v_k \approx_{\tau_k;i} v'_k$ holds for $1 \leq k \leq n$. Given two terms t and t' such that the free variables of t are contained in $\{x_1, \ldots, x_n\}$ and the free variables of t' are contained in $\{x'_1, \ldots, x'_n\}$, the generalized semantics preservation property is stated as follows:

Property 3. *For any i, θ, θ' and Γ such that $\theta \approx_{\Gamma;i} \theta'$ holds, if the free variables of t are bound in Γ and t is of type τ, and if t is transformed into t' by P, then $t[\theta] \sim_{\tau;i} t'[\theta']$ holds.*

This property can be proved by induction on the derivation of the transformation relation. In the case that the transformation goes under binders, we extend the substitutions θ and θ' with equivalent values for the binders and apply the inductive hypothesis using the extended substitutions. Since $\theta \sim_{\tau;i} \theta'$ holds vacuously when θ and θ' are empty, Property 2 is just a special case of Property 3.

As we have described at the beginning of this chapter, we are interested in verifying compilation of closed programs at atomic types. Since the equivalence relation at atomic types is identity, we can show that Property 1 is equivalent to the following property when τ is an atomic type:

Property 4. *If a closed term t of atomic type τ is transformed into t' by P and t evaluates to v, then t' evaluates to v.*

This is the forward simulation property of the transformation P we are eventually interested in.

Finally, we would like to compose the forward simulation properties of individual transformations to form the correctness proof of the full compiler. Notice that to apply these properties, we need to show that the input terms of the transformations are well-typed. For this we prove that every transformation preserves the types of its source terms. Thus, if the source program of the compiler is well-typed, then by the type preservation properties, every intermediate result in the compilation sequence is also well-typed. A type preservation property looks like the following:

If a term t has type τ and is transformed into t' by P, then t' has type τ.

Such a property is proved by induction on the transformation relation. This proof is usually straightforward and much simpler than that of semantics preservation.

The semantics preservation property for a compiler C is stated as follows:

Property 5. *If a closed term t of atomic type τ is compiled into t' by C and t evaluates to v, then t' evaluates to v.*

Assume C consists of a sequence of transformations $P_1, \ldots, P_n$, the property above is equivalent to the following:

Property 6. *If a closed term t of atomic type τ is transformed into t' by $P_1, \ldots, P_n$ in sequence and t evaluates to v, then t' evaluates to v.*

We prove it by applying the semantics and type preservation theorems of $P_1, \ldots, P_n$ in sequence, as follows. We know t is transformed into t'' by P_1 for some t'' and t'' is transformed by the rest of the transformations into t'. By semantics preservation of P_1, we know t'' evaluates to v. By type preservation of P_1, we know t'' has type τ. At this point we have gathered enough assumptions for applying the semantics and type preservation theorems of P_2 with t'' as its input. This process is repeated for the rest of the transformations. In the end, we have t' evaluates to v.

Up to now, we have presented the essential ideas for characterizing and proving semantics preservation properties using logical relations. When we talk about compiler transformations concretely in the later chapters, we will notice that they work with languages with richer constructs more than just atomic values and functions. In those situations, the definitions of logical relations, the semantics preservation theorems and their proofs will have more complicated forms than those presented in this section. Nevertheless, the fundamental ideas in this section still apply in those situations.

4.3 Using the Framework in Verified Compilation

In this section, we elaborate on the methodology for formalizing the approach to implementing and verifying compilation of functional programs discussed in the previous section. Under this methodology, we encode compiler transformations as λProlog specifications and prove semantics preservation of the transformations in Abella through the two-level logic approach.

We follow the approach to implementing rule-based relational specifications described in Section 2.1 to implement compiler transformations. Given a compiler transformation, we first encode the source and target languages of the transformation, including their syntax and typing rules in λProlog. For instance, the typing relation of the source language may be represented by the predicate `of` such that `of` M T holds if and only if M has type T; the type relation of the target language is represented by $\texttt{of}'$ in a similar way. We then identify a predicate constant to represent the relation for the compiler transformation and translate the transformation rules into program clauses defining this predicate constant. We have already seen an example illustrating this approach in Section 3.1, *i.e.*, the encoding of the transformation from λ-terms to their de Bruijn forms.

As we have discussed in Section 2.1.4, the λProlog specifications function directly as implementations under a logic programming interpretation. We can therefore use the encoding of compiler transformations as an implementation of the compiler. Suppose that a multi-pass compiler consists of a sequence of transformations $P_1, \ldots, P_n$ and they are represented as binary predicate symbols $p_1, \ldots, p_n$ such that $(p_i\ s\ t)$ holds if and only if the program S is transformed by P_i into T and s and t are respectively the encodings of S and T. Then given a source program S and its encoding m_0, we can identify variables $m_1, \ldots, m_n$ and query the following goal

$$(p_1\ m_0\ m_1)\ \&\ (p_2\ m_1\ m_2)\ \&\ \ldots\ \&\ (p_n\ m_{n-1}\ m_n).$$

Proof search in λProlog will instantiate $m_1, \ldots, m_n$ with terms such that $(p\ m_{i-1}\ m_i)$ holds for $1 \leq i \leq n$. The term for m_n is then the encoding of the result of compiling S.

We verify the compiler by formalizing the verification development described in Section 4.2 in Abella. That is, we encode semantics preservation of every transformation as a theorem in Abella, prove these theorems and compose them to form the correctness proof of the full compiler. The key is to formally prove Property 3 for every compiler transformation. To formalize this property in Abella, we need to formalize the logical relations and the substitution operations. The encoding of substitution as a relation has already been given in Section 3.2.4. We are left with the problem of encoding logic relations.

Before we can talk about encoding logical relations, we have to encode the evaluation semantics of the source and target languages of the transformation. Since we need to keep track of the number of evaluation steps in step-indexing logical relations, we give small-step evaluation semantics to these languages. We define the evaluation semantics as λProlog specifications using the approach described in Section 2.1. Specifically, for the source language we designate a predicate constant `step` to represent a one-step evaluation relation. It is defined through a set of program clauses such that `step` t t' holds if and only if t evaluates t' in one step. We also identify the predicate constant `nstep` such that `nstep` n t t' holds if and only if t evaluates to t' in n steps where $n \geq 0$. It is defined by transitively composing the `step` relation. We use the predicate constant `eval` to denote the big-step evaluation relation such that `eval` t v holds if t evaluates to the value v in a finite number of steps, *i.e.*, `nstep` n t v holds for some n. Similarly, we define the predicates `step`$'$, `nstep`$'$ and `eval`$'$ that represent corresponding evaluations relations for the target language.

Assume P is the transformation under consideration and $\sim$ and $\approx$ represent the logical relations denoting the semantics preservation property of P. We designate the predicate `sim` to represent $\sim$ and `equiv` to represent $\approx$. We then translate the definitions of $\sim$ and $\approx$ into clauses for `sim` and `equiv` such that

- $t \sim_{\tau;i} t'$ holds if and only if `sim` T I M M' holds and
- $t \approx_{\tau;i} t'$ holds if and only if `equiv` T I M M' holds

where T, I, M and M' are encodings of τ, i, t and t', respectively. The translation is straightforward and makes use of the encoding of evaluation semantics. Note that we need to indicate that `sim` and `equiv` are defined only for closed terms. We use the technique presented in Section 2.4.2 to characterized this closedness property. That is, we identify a predicate `tm` defined by a set of program clauses in λProlog such that $\{$`tm` $M\}$ holds if and only if M is a well-formed closed term. Notice that `sim` and `equiv` are not given as a fixed-point definition. Instead, they form a *recursive definition* that is based on an inductively defined set of pairs of types and step indexes. The theoretical justification of recursive definitions in Abella is given in [59].

Given the encoding of logical relations on closed terms, it is easy to extend it to relate closed substitutions. We designate the predicate `subst_equiv` to represent the logical relation on closed substitutions such that `subst_equiv` L I S S' holds if and only if $\theta \approx_{\Gamma;i} \theta'$ where Γ, I, S and S' are encodings of Γ, i, θ and θ'. It is given a fixed-point definition by using `sim` and `equiv` through the following clauses:

$$\texttt{subst\_equiv}\ \texttt{nil}\ I\ \texttt{nil}\ \texttt{nil} \quad \triangleq \quad \top$$

$$\begin{array}{ll}\nabla x.\texttt{subst\_equiv}\ (\texttt{of}\ x\ T\ \texttt{::}\ L)\ I\ (\texttt{map}\ x\ M\ \texttt{::}\ S)\ (\texttt{map}\ x\ M'\ \texttt{::}\ S') & \triangleq \\ \quad \texttt{equiv}\ T\ I\ M\ M' \wedge \texttt{subst\_equiv}\ L\ I\ S\ S' & \end{array}$$

Suppose the relation describing the transformation P is encoded as a predicate constant p such that $(p\ M\ M')$ holds if and only if t is transformed into t' by P where M and M' are respectively the encodings of t and t'. Then we can state Property 3 for P as the following theorem in Abella where $\{L \vdash \texttt{of}\ M\ T\}$ asserts that the source term M has type T in the typing context L:

$$\begin{array}{l}\forall L, I, S, S', M, M', T, N, N'. \\ \quad \texttt{subst\_equiv}\ L\ I\ S\ S' \supset \{L \vdash \texttt{of}\ M\ T\} \supset \{p\ M\ M'\} \supset \\ \quad\quad \texttt{app\_subst}\ S\ M\ N \supset \texttt{app\_subst}\ S'\ M'\ N' \supset \texttt{sim}\ T\ I\ N\ N'.\end{array}$$

This theorem is then proved by induction on $\{p\ M\ M'\}$. When the terms M and M' are closed and the substitutions S and S' are empty, we get the following special case of the above theorem, corresponding to Property 2:

$$\forall I, M, M', T.\{\texttt{of}\ M\ T\} \supset \{p\ M\ M'\} \supset \texttt{sim}\ T\ I\ M\ M'.$$

When T is an atomic type, this theorem further degenerates into the following, corresponding to Property 4:

$$\forall M, M', T.\{\texttt{of}\ M\ T\} \supset \{p\ M\ M'\} \supset \{\texttt{eval}\ M\ V\} \supset \{\texttt{eval}'\ M'\ V\}.$$

To compose the above theorem with that for other transformations, we need to prove the following type preservation property for the transformation:

$$\forall M, M', T.\{\texttt{of}\ M\ T\} \supset \{p\ M\ M'\} \supset \{\texttt{of}'\ M'\ T\}.$$

It is proved by induction on $\{p\ M\ M'\}$, usually in a straightforward manner.

Now, assume the predicate constants $p_1, \ldots, p_n$ encode the transformation relations for the sequence of transformations in the compiler and the predicate constant c represent the compilation relation such that $(c\ \ t_0\ \ t_n)$ holds if and only if there exists $t_1, \ldots, t_{n-1}$ such that $(p_i\ t_{i-1}\ t_i)$ holds for $1 \leq i \leq n$. Then semantics preservation of the full compiler is encoded as follows which corresponds to Property 5:

$$\forall M, M', T.\{\texttt{of}\ M\ T\} \supset \{c\ M\ M'\} \supset \{\texttt{eval}\ M\ V\} \supset \{\texttt{eval}'\ M'\ V\}.$$

Suppose we have proved the semantics and type preservation properties for $p_1, \ldots, p_n$. We can then apply them to prove the above theorem. The proof closely follows the informal one we described in Section 4.2.

4.4 Using λ-Tree Syntax in Verified Compilation

By using the framework consisting of λProlog and Abella in verified compilation of functional programs, we expect to draw on the benefits for the λ-tree syntax approach that we outlined in Section 2.4. In this section, we motivate the way in which these benefits will play out in the concrete developments that we undertake in the next few chapters in the thesis.

In the implementations of transformations on functional programs, we use meta-level λ-abstraction to represent the binding operators in functional objects. As a result, the notions related to binding structure such as renaming and substitution are captured by α- and β-conversions in a logical and precise fashion. For instance, β-conversion will be used to model the "administrative" substitution operations that are an inherent part of the CPS transformation as described in Chapter 5, resulting in a very concise implementation of the transformation. When working on functional objects, the compiler transformations often need to go under their binding operators and recursively transform the function bodies. Furthermore, these transformations often have side conditions for the binding variables introduced by recursion. As described in Sections 2.1.3 and 2.4 we use the universal and hypothetical goals to perform recursion over binding operators and to enforce these side conditions. Compiler transformations often perform non-trivial analysis on the binding structure of function objects. Such analysis can be captured in the λProlog specifications concisely and logically. For example, in Chapter 7 we will present a transformation that extracts closed functions to the top-level. For the extraction to work, it is necessary to show that such functions do not refer to any bound variable. This independence relation can be statically characterized via quantification ordering and dynamically realized via unification. Complicated analysis of binding structure can even be specified through λProlog programs. As an example, consider the closure conversion transformation that was briefly touched in the introduction and that will be presented in detail in Chapter 6. To implement closure conversion, we need to compute free variables in functions. This computation will be presented as a relation between function objects and their free variables and defined through a set of λProlog program clauses.

The above uses of the λ-tree syntax approach also apply when we treat definitions involving binding structure in the verification of compiler transformations on functional programs in Abella. Furthermore, the logical structure of such treatments can be exploited to significantly simplify reasoning about binding structure. For example, we have given a definition of substitution as an explicit relation in Section 3.2.4 which is an essential component for describing the semantics preservation property characterized as logical relations. We have also shown that a lot of properties of the substitution relation can be easily established by observing that they are just manifestation of the properties of β-conversion in the meta-level language.

Rich properties of binding structure can be proved by combining the λ-tree syntax approach and the two-level logic approach. This manifests in the treatment of closedness property of λ-terms. In Section 2.4.2 we have seen that we can characterize closed terms through a λProlog specification for the predicate `tm` such that $\{\texttt{tm}\ M\}$ holds if and only if M is a closed term. This predicate is used in the definition of logical relations which should hold only for closed terms. We can then easily prove the property that substitution has no effect on closed terms by using the explicit definition of substitution and the λProlog specification for closed terms. This property is critical to proving semantics preservation of closure conversion as we shall see in Chapter 6.

Finally, the logical structure of binding related treatments in the implementation of compiler transformations can be exploited to simplify reasoning about binding structure through the two-level logic approach. For instance, the usage of β-conversion to model "administrative" substitution operations in the CPS transformation makes it extremely easy to reason about the effects of such operations, as we shall see in Chapter 5. As another example, in Chapter 6 we will need to prove the following "strengthening" property for typing: if a term t has type τ in a typing context Γ and Γ' is a restriction of Γ that contained only the free variables in t, then t also has type τ in Γ'. We shall see that this property can be easily established by exploiting the logical structure of the λProlog program for computing free variables.

4.5 Nuances in Formalizing Semantics Preservation

We will be using a logical relations style characterization of semantics preservation in the work in this thesis. There are, however, a few other ways in which this notion has been characterized in the literature. We discuss and contrast these different approaches in this section.

A good starting point for the discussion is a set of criteria proposed by Neis *et al.* [25] to assess the different approaches to formalizing semantics preservation:

- *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 semantics 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 semantics 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. Together with modularity, it allows for combination of correctness proofs of modules that use the same definition of semantics preservation but are generated by different compiler transformations.

- *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".

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 reduce the linking of modules to 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 evaluation semantics of the source and target programs they hold of. As a result, the correctness proofs of different transformations that use the same logical relation as the notion of semantics preservation can be combined without any problem.

Establishing transitivity of correctness proofs when these use a notion of semantics preservation 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 transformations on functional programs 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 [76], 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 that will be described in Chapter 5, $\sim^2$ may be a logical relation for the closure conversion transformation that will be described in Chapter 6, 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 the devices called *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 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 and flexibility become irrelevant issues. 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 be composed to obtain a correctness result for the entire sequence of transformations. In fact, this is the perspective we take in this thesis and we have discussed how the composition works in Section 4.2.

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 [20] and the CakeML [24] 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, *i.e.*, equivalence between values is reduced to identity 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.

By the above statement it is easy to show that correctness proofs based on the refined approach we have just described are transitively composible; 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; [78] presents a recent development along these lines.

There has been some recent work led, not surprisingly by Neis and colleagues that is aimed at designing a notion of semantics preservation that actually meets all three desired criteria put forward for such a notion. Most specifically, Neis *et al.* [25] have introduced a relation called parametric inter-language simulation or PILS towards this end. PILS can be thought of as a refinement to logical relations. Similar to logical relations, PILS enjoys modularity and flexibility. Moreover, PILS also enjoys transitivity that logical relations fail to support. The key idea is to break up a single definition of logical relation into two relations called "global knowledge" and "local knowledge" to dismiss the negative self-reference that occurs in the definition of equivalence at function types so that semantics preservation can be characterized as fixed-point definitions. As we have discussed before, the negative self-reference is the main reason that logical relations are not transitively composible. By finessing the need for such a reference, PILS is able to support transitivity.

We have chosen to use a logical relations style definition of semantics preservation rather than one based on PILS mainly because the latter is still in at an evolutionary stage; our focus in this thesis is not so much on developing new and useful ideas concerning semantics preservation as it is on exposing useful approaches to formalizing them. In this regard, we believe many of the lessons to be drawn from this work also carry over to situations in which other notions of semantics preservation are used. The key observation is that representation, manipulation and analysis of binding structure are inherent in compiler transformations of functional programs and properties about them must be explicitly proved in reasoning no matter what notions of semantics preservation are used. Thus, the λ-tree syntax approach can always be used to simplify these reasoning tasks. For example, suppose we prove the correctness of the closure conversion transformation by showing that it permutes with evaluations. For this we need to keep an explicit representation of environments in the evaluation semantics. The evaluation environments can be easily encoded in λProlog by using meta-level abstractions to represent the bindings of the environments and their properties can be easily proved in Abella by exploiting the logical structure of the encoding of environments. As another example, the definition of PILS makes essential use of substitution like logical relations. As a result, reasoning about the substitution operations and their interaction with other program constructs is an inherent part of the correctness proofs based on PILS. Such reasoning can be carried out effectively by using λ-tree syntax just as when logical relations are used as the correctness notion.

4.6 Exercise of Verified Compilation in the Thesis

To demonstrate the effectiveness of our methodology for implementing and verifying transformations on functional programs, we will use it to develop a verified compiler for a representative functional language. This language extends the simply typed λ-calculus with general recursion, conditional and arithmetic expressions; it can be thought of as an extension to the well-known PCF language [42]. Our compiler takes programs in the λ-tree syntax 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 code in a language that is similar to the Cminor language used in the CompCert project [43]. We have chosen this as our target language because many other compiler verification projects have used Cminor as an intermediate language and we can therefore benefit from their work in completing the compiler verification process.

The structure of the compiler that we develop in this thesis and that is shown in Figure 4.1 is explained qualitatively as follows. The CPS transformation makes the control flow such as evaluation ordering explicit. The closure conversion transformation makes (nested) functions independent of their context. It effectively makes functions closed. The code hoisting transformation lifts these closed functions to the top level. After this transformation, all the higher-order features of the source programs are removed. The code closely resembles that in procedural languages such as C. The code generation phase makes the allocation of objects explicit and generates Cminor-like code.

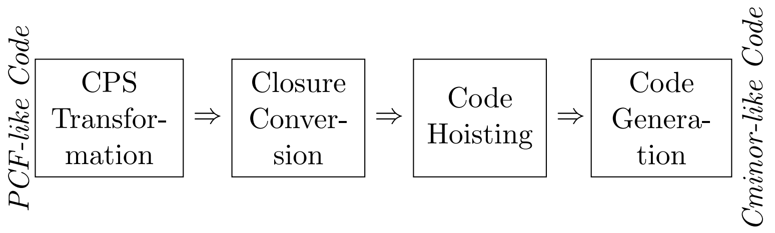

Figure 4.1: The Compiler for a PCF-style Language

We use the methodology described in Section 4.3 to implement a formally verified version of this compiler. Chapter 5,6,7 respectively describe the implementation and verification of the CPS transformation, closure conversion and code hoisting. Chapter 8 describes the code generation transformation and composition of semantics preservation of individual transformations to form the correctness proof for the full compiler.

Chapter 5

The Continuation Passing Style Transformation

Continuation Passing Style (CPS) is a programming style in which control flow is made explicit by using devices called *continuations* that record "the remaining computation". CPS has a long history that dates back to 1960s [79]. Languages in this form are favored as intermediate languages for a lot of compilers for functional languages (*e.g.*, see [27, 80, 81]) because continuations provide an elegant representation of language constructs such as function calls and pattern matching, and expressions in the CPS form resemble statements in procedural languages and thereby can be more easily related to executable code.

A CPS transformation translates functional programs in a direct style of programming into a CPS form [82, 83, 84]. It takes as arguments a source term and a continuation representing an abstraction of the remaining computation over the value of the source term. It then transforms the source term into a more fine-grained form such that control flow is made explicit—*i.e.*, there is no ambiguity in the order of evaluation—by recursively applying the transformation on its subterms and accumulating the continuation in the process. A common approach to describe the CPS transformation is to describe it using λ-calculus [83, 84]. In this approach , it is important to distinguish between β-redexes introduced by the transformation for realizing substitution at the translation time and β-redexes that come from the source term. The former kind of

β-redexes are reduced at the translation time as part of the transformation while the later should not be reduced because that will count as partial evaluation of the source term in the compilation phase. We adopt the terminology in [84] to denote the former as *administrative β-redexes* and the later as *dynamic β-redexes*, *i.e.*, β-redexes that should only be reduced at run time.

The CPS transformation can be easily described in a rule-based and relational style. However, the formalization of such a description can be difficult. The major difficulty lies in correctly capturing the way the administrative and dynamic β-redexes work. Moreover, this difficulty is amplified when proving properties of the transformation such as semantic preservation because the properties of binding related notions for manipulating administrative and dynamic β-redexes must be proved explicitly.

We solve the above problems by using the methodology for implementing and verifying compiler transformations described in Section 4.3. First, we encode the rule-based relational description of the CPS transformation as a λProlog specification. We use meta-level β-redexes to encode the administrative β-redexes. As such, the substitution operations they represent are automatically captured by meta-level β-reduction. The dynamic β-redexes are encoded as program constructs by using the λ-tree syntax approach. Because λProlog specifications are executable, the encoding of the CPS transformation is also its implementation. We then prove that the implementation preserves semantics in Abella by following the logical relation based approach described in Section 4.3. Because the logical structure of the implementation is transparently reflected into Abella via the two-level logic approach, we are able to prove the properties of administrative and dynamic β-redexes easily by exploiting their λ-tree syntax representation. In the end we get a concise and elegant proof of semantics preservation for the implementation.

In the following sections, we illustrate the above ideas by constructing a verified implementation of the CPS transformation in our compiler described in Section 4.6. We first give an overview of the CPS transformation in Section 5.1. We then present the source and target languages and give a rule-based relational description of the transformation in Section 5.2. We then encode the rule-based description as a λProlog specification in Section 5.3. The discussion here will focus on using the λ-tree syntax approach to encode the administrative and dynamic β-redexes. In Section 5.4, we present the informal semantics preservation proof of the CPS transformation. We lastly give the formal semantic preservation proof of the implementation in Section 5.5. The discussion will bring out the benefits of λ-tree syntax approach in reasoning about the administrative and dynamic β-redexes.

5.1 An Overview of the Transformation

The CPS transformation we work with is based on the transformation proposed by Danvy and Filinski in [84] that reduces administrative β-redexes on the fly. To describe the transformation, we need to distinguish between the administrative β-redexes which only manifest themselves during the transformation and are eliminated in the final result, and the dynamic β-redexes that are constituting pieces of the result. We shall use $\hat{\lambda}x.\, t$ and $(@\ t_1\ t_2)$ to represent administrative abstractions and applications and normal abstractions and applications to represent the dynamic ones.

The transformation takes as input a term and a continuation that is an administrative abstraction over the value of the term and that is already in the CPS form. Intuitively, the source term can be thought as a program fragment that represents the "current computation" and eventually evaluates to a value. The continuation can be thought as a program fragment that, when fed with the value of the source term, represents the "remaining computation." The job of the CPS transformation is to convert the source term into a form in which control flow becomes explicit and to combine the result with the continuation to form the complete program. The transformation proceeds by recursion over the structure of the source term. At the beginning of this process, a continuation representing the context in which the source program will be used is given. The CPS transformation is then recursively applied to the sub-expressions of the source term and the results are accumulated into the continuation in an order that reflects the control flow of the source program. The result of such accumulation is the output of the CPS transformation.

We start by illustrating how the CPS transformation works on basic expressions through some examples. Consider the base case of the transformation, *i.e.*, when the source term t is a variable or a constant. In this case, the control flow in t is vacuously fixed. Letting $\hat{\lambda}v.\, t'$ be the input continuation, we can therefore apply $\hat{\lambda}v.\, t'$ to the source term to form the output which is $(@\ (\hat{\lambda} v.\, t')\ t)$. The term $(@\ (\hat{\lambda} v.\, t')\ t)$ is an example of administrative β-redexes which will be immediately reduced by the transformation. To illustrate the transformation on compound expressions, consider the case when the source term is an addition expression $t_1 + t_2$. Let $(\hat{\lambda} v.\, t)$ be the input continuation and assume that the expression is evaluated from left to right. Then we can break down the computation represented by $t_1 + t_2$ into the following sequence: evaluating t_1 to v_1, evaluating t_2 to v_2 and evaluating $v_1 + v_2$. By doing so the control flow becomes explicit and unambiguous. We recursively perform the CPS transformation on t_2 with the continuation

$$c_2 = (\hat{\lambda} v_2.\, \mathbf{let}\ v = v_1 + v_2\ \mathbf{in}\ @\ (\hat{\lambda} v.\, t)\ v)$$

to get a target program t_2'; note that v_1 is a free variable in this expression that will be bound by an administrative abstraction in the input continuation to the transformation of t_1 as we explain presently. In the continuation c_2, we have used a let expression to explicitly state that the addition of v_1 and v_2 must be computed before we can use it for the remaining computation represented by t. This is a common technique for enforcing evaluation ordering that will arise often in the following discussion. Again notice the use of the administrative β-redex $(@\ (\hat{\lambda} v.\, t)\ v)$ to represent a substitution operation. Notice also that c_2 represents the computation after evaluating t_2. As a result, t_2' represents the evaluation of t_2 followed by the remaining computation, which is also the computation after evaluating t_1. We can therefore recursively apply the transformation on t_1 with $\hat{\lambda} v_1.\, t_2'$ as the input continuation to get the complete program in the CPS form.

At this point we know how the basic expressions are transformed into the CPS form. Another important part of the CPS transformation is to convert functions and function applications into the CPS form. A function in the CPS form takes a continuation as an extra argument. This continuation is provided by the caller of the function that represents the future computation after the function returns. Instead of returning a value to the caller, evaluation of the function body ends with calling the continuation with the return value as its argument. As a result, in the CPS form function calls never return and computation always goes forward by invoking other continuations. By making the control flow for function calls and returns explicit, we avoid having to treat a function return as a program construct and enable optimizations that rely on control flow analysis such as elimination of tail-calls.

We give a more concrete account of the transformation on functions and function calls. Let $\lambda x.\, t$ be the source function and k be the input continuation. Letting k' denote the continuation passed over from the caller, we first recursively perform the transformation on t with the continuation $(\hat{\lambda} a.\, k'\ a)$ to get the function body t' in the CPS form. Note that we cannot use k' directly as an input to the recursive transformation because it is a dynamic λ-abstraction. That is why we instead have used $(\hat{\lambda} a.\, k'\ a)$ which is an administrative λ-abstraction equivalent to k' as the input continuation. Then the original function $\lambda x.\, t$ is transformed into the following expression:

$$\textbf{let}\ f = \lambda x.\, \lambda k'.\, t'\ \textbf{in}\ (@\ k\ f)$$

Note that the original λ-abstraction is converted to the dynamic abstraction $\lambda x.\, \lambda k'.\, t'$ and the administrative β-redex $(@\ k\ f)$ represents the remaining computation.

To transform a function application $(t_1\ t_2)$ given the input continuation k, we need to pass k as an extra argument to t_1. We cannot use k directly as an argument because it is an administrative λ-abstraction while a dynamic λ-abstraction is expected here. We therefore convert k into a dynamic abstraction $\lambda a.\, (@\ k\ a)$ and use it as the continuation argument. Let c_2 be the continuation representing the computation after evaluating t_2. This continuation has the form

$$\hat{\lambda} v_2.\, v_1\ v_2\ (\lambda a.\, (@\ k\ a))$$

where v_1 is a free variable that, intuitively, represents the place where the result of evaluating (the transformed version of) t_1 must be filled in to complete the computation. Now, to actually effect the transformation we first transform t_2 with c_2 as the input continuation to get t_2'. Then we transform t_1 with the continuation $\lambda v_1.\, t_2'$ to get the complete result of transforming $(t_1\ t_2)$ into the CPS form.

As an example, consider the following program expression:

$$\textbf{let}\ f = \lambda x.\, x + 2\ \textbf{in}\ f\ 3$$

The input continuation used for transforming this expression, as also any expression at the top-level, is $\hat{\lambda} x.\, x$. Using this continuation, the CPS transformation converts the given expression into the following equivalent one in the CPS form:

$$\textbf{let } f = (\lambda x.\, \lambda k'.\, \textbf{let } v = x + 2 \textbf{ in } (k'\ v)) \textbf{ in } f\ 3\ (\lambda a.\, a)$$

In producing this expression, we have contracted several administrative redexes that arise, as we invite the reader to verify by actually carrying out the steps of the transformation that we have described.

We have now covered all the important aspects of the CPS transformation in the style of Danvy and Filinski. Realistic functional programming languages may have richer program constructs. But the CPS transformation for them follows the essential ideas exposed in this section.

5.2 A Rule-Based Description of the Transformation

The CPS transformation is the first pass of the compiler we described in Section 4.6. We give it a rule-based relational description in this section based on the ideas presented in the last section.

5.2.1 The source and target languages

The source language of the CPS transformation, which is also the source language of the whole compiler, is a slight variant of the PCF language [42], a representative functional programming language that is often studied in the literature on functional programming languages and verification. The target language of the transformation is the same as its source language, *i.e.*, the CPS transformation is a source-to-source transformation that does not introduce any new program constructs. The syntax of this language is shown in Figure 5.1. In this figure, T, M and V stand respectively for the categories of types, terms and the terms recognized as values.

We provide some intuition into the structure of the language whose syntax is described by the rules in Figure 5.1; this intuition will underlie the typing and evaluation judgments that we will present later. The symbol $\mathbb{N}$ represents the type of natural numbers and the symbol **unit** represents a type whose sole constructor is $()$ that is also pronounced as unit. Further, $T_1 \to T_2$ corresponds to the function type and $T_1 \times T_2$ is the type of pairs. The collection of terms in the language is essentially an extension of the terms constituting the simply typed λ-calculus. More specifically, this collection

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

Figure 5.1: The Syntax of the Source/Target Language of the CPS transformation

can be understood as follows. First, it includes the natural numbers; this collection is denoted by n in the syntax rules. Second, it includes the arithmetic operators **pred** and + that represent, respectively, the predecessor and addition functions on natural numbers. Third, it includes constructors and destructors for tuples: () is the unit constructor, (M_1, M_2) is a pair whose first element is M_1 and second element is M_2; **fst** and **snd** are the projection operators on pairs to their first and second elements. Fourth, it includes the conditional expression **if** M_1 **then** M_2 **else** M_3. The behavior of such an expression is based on whether or not the "condition" M_1 is zero: If so, it behaves the same as M_2; Otherwise, it behaves the same as M_3. Fifth, it includes *let expressions* of the form **let** $x = M_1$ **in** M_2 that are convenient for breaking up a program into more manageable pieces; in any given instance of this expression, x represents a local variable that is bound to the value of M_1 and whose scope is limited to M_2. Finally, the collection of terms includes the recursion or fixed-point operator **fix** which abstracts simultaneously the function f and the parameter x over the function body M to form the function **fix** $f\,x.M$ and the usual function application expression $(M_1\ M_2)$. The application of a function **fix** $f\,x.M$ to an argument M' behaves by replacing f with itself and x with M' in its body M and evaluating the resulting term. We use $\lambda x.\, M$ to denote the function **fix** $f\,x.M$ in which f does not occur in M.

A typing judgment for the source/target language is written as $\Gamma \vdash M : T$, where Γ is a list of type assignments for variables. They are derivable by using the rules in Figure 5.2. The typing rules are mostly standard. The only interesting rule is of-fix for typing functions: to give the function **fix** $f\,x.M$ an arrow type $T_1 \to T_2$, we need to show its body M has type T_2 in an extended context that assigns f with the type of the function itself and x with the type of its argument. This coincides with the intuitive interpretation of such expressions that we have presented above.

$$\frac{}{\Gamma \vdash n : \mathbb{N}}\ \text{of-nat} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{of-var}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \mathbf{pred}\ M : \mathbb{N}}\ \text{of-pred} \qquad \frac{\Gamma \vdash M_1 : \mathbb{N} \quad \Gamma \vdash M_2 : \mathbb{N}}{\Gamma \vdash M_1 + M_2 : \mathbb{N}}\ \text{of-plus}$$

$$\frac{\Gamma \vdash M_1 : \mathbb{N} \quad \Gamma \vdash M_2 : T \quad \Gamma \vdash M_3 : T}{\Gamma \vdash \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3 : T}\ \text{of-if}$$

$$\frac{}{\Gamma \vdash () : \mathbf{unit}}\ \text{of-unit} \qquad \frac{\Gamma \vdash M_1 : T_1 \quad \Gamma \vdash M_2 : T_2}{\Gamma \vdash (M_1, M_2) : T_1 \times T_2}\ \text{of-pair}$$

$$\frac{\Gamma \vdash M : T_1 \times T_2}{\Gamma \vdash \mathbf{fst}\ M : T_1}\ \text{of-fst} \qquad \frac{\Gamma \vdash M : T_1 \times T_2}{\Gamma \vdash \mathbf{snd}\ M : T_2}\ \text{of-snd}$$

$$\frac{\Gamma \vdash M_1 : T_1 \quad \Gamma, x : T_1 \vdash M_2 : T}{\Gamma \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : T}\ \text{of-let}$$
(provided x does not occur in Γ)

$$\frac{\Gamma, f : T_1 \to T_2, x : T_1 \vdash M : T_2}{\Gamma \vdash \mathbf{fix}\ f\,x.M : T_1 \to T_2}\ \text{of-fix} \qquad \frac{\Gamma \vdash M_1 : T_1 \to T \quad \Gamma \vdash M_2 : T_1}{\Gamma \vdash M_1\ M_2 : T}\ \text{of-app}$$
(provided f and x do not occur in Γ)

Figure 5.2: Typing Rules for the Source Language of the CPS transformation

5.2.2 The transformation rules

We give the rules for the CPS transformation based on the informal ideas described in Section 5.1. In general, we must transform terms containing free variables. These free variables must be tracked throughout the transformation. Thus, we specify the transformation as a 4-place relation written as $\rho \triangleright M; K \rightsquigarrow_{cps} M'$, where M and M' are the input and output terms, K is the input continuation and ρ is a set of variables that contains all the free variables in M. We write (ρ, x) to denote the extension of ρ with a variable x. Figure 5.3 defines the $\rho \triangleright M; K \rightsquigarrow_{cps} M'$ relation in a rule-based fashion. Note again that we use $\hat{\lambda} x.\, M$ and $(@\ M_1\ M_2)$ to represent administrative abstractions and applications. An administrative β-redex should be considered as representing its λ-normal form, *i.e.*, it is immediately reduced by the transformation. Most of the rules follow directly from the informal description of the CPS transformation given in

Section 5.1. The rules that need further explanation are cps-if for transforming the conditional expressions, cps-fix for transforming functions and cps-app for transforming function applications.

Intuitively, a rule for transforming the conditional expressions can be given as follows:

$$\frac{\rho \triangleright M_2; K \rightsquigarrow_{cps} M_2' \quad \rho \triangleright M_3; K \rightsquigarrow_{cps} M_3' \quad \rho \triangleright M_1; \hat{\lambda} x_1.\, \mathbf{if}\ x_1\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3' \rightsquigarrow_{cps} M'}{\rho \triangleright \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3; K \rightsquigarrow_{cps} M'}$$

That is, we recursively transform both branches of the expression with the input continuation K since K must be the remaining computation no matter which branch is taken in the evaluation. We then recursively transform the condition part M_1 with the continuation

$$\hat{\lambda} x_1.\, \mathbf{if}\ x_1\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3'$$

that is formed from the branches in the CPS form and represents the remaining computation after evaluating M_1 to get the CPS form of the original expression. However, there is a problem with this rule: the continuation K is duplicated for transforming the branches and may result in exponential explosion in the code size when nested conditional expressions are present. The cps-if rule solves this problem by use a variable k as a placeholder of the continuation for the branches. When transforming the condition M_1, it uses the continuation

$$\mathbf{let}\ k = \lambda a.\, (@\ K\ a)\ \mathbf{in}\ (\mathbf{if}\ x_1\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3')$$

that links this placeholder to the actual continuation through a let expression. Since there is only one copy of K used in the recursive transformation, the program grows linearly in space.

The cps-fix rule for transforming functions mostly follows the corresponding informal description given in Section 5.1. To transform a function $\mathbf{fix}\ f\ x.M$ with the continuation K, we first recursively transform the function body M with the continuation $\hat{\lambda} y.\, k\ y$ where k is the continuation argument of the transformed function to get its CPS form

$$\frac{}{\rho \rhd n; K \rightsquigarrow_{cps} @\ K\ n}\ \text{cps-nat} \qquad \frac{x \in \rho}{\rho \rhd x; K \rightsquigarrow_{cps} @\ K\ x}\ \text{cps-var}$$

$$\frac{}{\rho \rhd (); K \rightsquigarrow_{cps} @\ K\ ()}\ \text{cps-unit}$$

$$\frac{\rho \rhd M; \hat{\lambda} x.\, \mathbf{let}\ v = \mathbf{pred}\ x\ \mathbf{in}\ (@\ K\ v) \rightsquigarrow_{cps} M'}{\rho \rhd \mathbf{pred}\ M; K \rightsquigarrow_{cps} M'}\ \text{cps-pred}$$

$$\frac{\rho \rhd M_2; \hat{\lambda} x_2.\, \mathbf{let}\ v = x_1 + x_2\ \mathbf{in}\ (@\ K\ v) \rightsquigarrow_{cps} M_2' \quad \rho \rhd M_1; \hat{\lambda} x_1.\, M_2' \rightsquigarrow_{cps} M'}{\rho \rhd M_1 + M_2; K \rightsquigarrow_{cps} M'}\ \text{cps-plus}$$

(provided x_1 does not occur in ρ, M_2 and K)

$$\frac{\rho \rhd M_2; \hat{\lambda} x_2.\, \mathbf{let}\ v = (x_1, x_2)\ \mathbf{in}\ (@\ K\ v) \rightsquigarrow_{cps} M_2' \quad \rho \rhd M_1; \hat{\lambda} x_1.\, M_2' \rightsquigarrow_{cps} M'}{\rho \rhd (M_1, M_2); K \rightsquigarrow_{cps} M'}\ \text{cps-pair}$$

(provided x_1 does not occur in ρ, M_2 and K)

$$\frac{\rho \rhd M; \hat{\lambda} x.\, \mathbf{let}\ v = \mathbf{fst}\ x\ \mathbf{in}\ (@\ K\ v) \rightsquigarrow_{cps} M'}{\rho \rhd \mathbf{fst}\ M; K \rightsquigarrow_{cps} M'}\ \text{cps-fst}$$

$$\frac{\rho \rhd M; \hat{\lambda} x.\, \mathbf{let}\ v = \mathbf{snd}\ x\ \mathbf{in}\ (@\ K\ v) \rightsquigarrow_{cps} M'}{\rho \rhd \mathbf{snd}\ M; K \rightsquigarrow_{cps} M'}\ \text{cps-snd}$$

$$\frac{\begin{array}{c}\rho \rhd M_2; \hat{\lambda} x.\, k\ x \rightsquigarrow_{cps} M_2' \\ \rho \rhd M_3; \hat{\lambda} x.\, k\ x \rightsquigarrow_{cps} M_3' \\ \rho \rhd M_1; \hat{\lambda} x_1.\, \mathbf{let}\ k = \lambda a.\, (@\ K\ a)\ \mathbf{in}\ (\mathbf{if}\ x_1\ \mathbf{then}\ M_2'\ \mathbf{else}\ M_3') \rightsquigarrow_{cps} M'\end{array}}{\rho \rhd \mathbf{if}\ M_1\ \mathbf{then}\ M_2\ \mathbf{else}\ M_3; K \rightsquigarrow_{cps} M'}\ \text{cps-if}$$

(provided k does not occur in ρ, M_2 and M_3)

$$\frac{\rho, x \rhd M_2; K \rightsquigarrow_{cps} M_2' \quad \rho \rhd M_1; \hat{\lambda} x.\, M_2' \rightsquigarrow_{cps} M'}{\rho \rhd \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2; K \rightsquigarrow_{cps} M'}\ \text{cps-let}$$

(provided x does not occur in ρ and K)

$$\frac{\rho, f, x \rhd M; \hat{\lambda} y.\, k\ y \rightsquigarrow_{cps} M'}{\begin{array}{l}\rho \rhd (\mathbf{fix}\ f\, x.M); K \rightsquigarrow_{cps} \\ \qquad \mathbf{let}\ v = (\mathbf{fix}\ f\, p.\mathbf{let}\ k = \mathbf{fst}\ p\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{snd}\ p\ \mathbf{in}\ M')\ \mathbf{in}\ (@\ K\ v)\end{array}}\ \text{cps-fix}$$

(provided f, x do not occur in ρ and k does not occur in ρ and M)

$$\frac{\begin{array}{c}\rho \rhd M_2; \lambda x_2.\mathbf{let}\ k = \lambda a.\, (@\ K\ a)\ \mathbf{in}\ \mathbf{let}\ p = (k, x_2)\ \mathbf{in}\ (x_1\ p) \rightsquigarrow_{cps} M_2' \\ \rho \rhd M_1; \hat{\lambda} x_1.\, M_2' \rightsquigarrow_{cps} M'\end{array}}{\rho \rhd M_1\ M_2; K \rightsquigarrow_{cps} M'}\ \text{cps-app}$$

(provided x_1 do not occur in ρ, M_2 and K)

Figure 5.3: The Rules for the CPS Transformation

M'. The CPS form of the original function is then the following

$$(\textbf{fix}\ f\, p.\textbf{let}\ k = \textbf{fst}\ p\ \textbf{in}\ \textbf{let}\ x = \textbf{snd}\ p\ \textbf{in}\ M').$$

whose second argument is a pair consisting of the continuation argument and the original argument and whose body is essentially M' except for the let expressions for selecting arguments from the pair. Denoting it by F, we get the output of the transformation $(\textbf{let}\ v = F\ \textbf{in}\ (@\ K\ v))$. The cps-app rule for transforming function applications almost follows its informal description given in Section 5.1. The only difference is that we need to pack the continuation argument and the original argument into a pair argument.

An important aspect of the rules in Figure 5.3 is the freshness side conditions for variables for guaranteeing the correctness of the CPS transformation. Such side conditions can be classified into two categories. One is for the free variables introduced by recursion over binding operators, including x in cps-let and f, x in cps-fix. These free variables must be fresh to avoid accidental capturing of them in the recursive transformations. Another category—and the more interesting one—is for "placeholder" variables, including k in cps-fix and cps-if and x_1 in cps-plus, cps-pair and cps-app. In all those rules, we perform recursive CPS transformations with continuations containing free variables that are placeholders for arguments that will be bound later in construction of other continuations. To avoid accidental capturing of these free variables in the recursive transformations, we must place freshness constraints on them too.

5.3 Implementing the Transformation in λProlog

Our presentation of the implementation of the CPS transformation consists of two parts: we first show how to encode the source (and also target) language in λProlog and we then present a λProlog program for the transformation. In describing the first part, we discuss also the formalization of the typing rules. In describing the second part, we shall discuss how the λ-tree syntax approach is used to capture the reading of administrative and dynamic β-redexes and the side conditions and show the transparent correspondence between our λProlog encoding and the original transformation rules.

5.3.1 Encoding the language

We first consider the encoding of types. We use the λProlog type `ty` to represent the types of the language. The constructors `tnat`, `tunit` and `prod` encode, respectively, the natural number, unit and pair types. We represent the arrow type constructor $\to$ by `arr`. These decisions are summarized in the following λProlog signature.

`tnat, tunit`	:	`ty`
`arr, prod`	:	`ty` $\to$ `ty` $\to$ `ty`

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

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

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

Following Section 2.1.3, we represent typing judgments as relations between terms and types, treating contexts implicitly via dynamically added clauses that assign types to free variables. We use the predicate symbol `of` : `tm` $\to$ `ty` $\to$ `o` to encode typing in the language. Every typing rule in Figure 5.2 is translated into a clause for `of`. These clauses are listed as follows:

```
of (nat N) tnat.
of (pred M) tnat :- of M tnat.
```

$\texttt{of}\ (\texttt{plus}\ M_1\ M_2)\ \texttt{tnat}\ \texttt{:-}\ \texttt{of}\ M_1\ \texttt{tnat}\,,\texttt{of}\ M_2\ \texttt{tnat}.$

$\texttt{of}\ (\texttt{ifz}\ M\ M_1\ M_2)\ T\ \texttt{:-}\ \texttt{of}\ M\ \texttt{tnat}\,,\texttt{of}\ M_1\ T\,,\texttt{of}\ M_2\ T.$

$\texttt{of}\ \texttt{unit}\ \texttt{tunit}.$

$\texttt{of}\ (\texttt{pair}\ M_1\ M_2)\ (\texttt{prod}\ T_1 T_2)\ \texttt{:-}\ \texttt{of}\ M_1\ T_1\,,\texttt{of}\ M_2\ T_2.$

$\texttt{of}\ (\texttt{fst}\ M)\ T_1\ \texttt{:-}\ \texttt{of}\ M\ (\texttt{prod}\ T_1\ T_2).$

$\texttt{of}\ (\texttt{snd}\ M)\ T_2\ \texttt{:-}\ \texttt{of}\ M\ (\texttt{prod}\ T_1\ T_2).$

$\texttt{of}\ (\texttt{let}\ M\ R)\ T\ \texttt{:-}\ \texttt{of}\ M\ T_1\,,(\Pi x.\texttt{of}\ x\ T_1 \Rightarrow \texttt{of}\ (R\ x)\ T).$

$\texttt{of}\ (\texttt{fix}\ R)\ (\texttt{arr}\ T_1\ T_2)\ \texttt{:-}$

$\quad \Pi f, x.\texttt{of}\ f\ (\texttt{arr}\ T_1\ T_2) \Rightarrow \texttt{of}\ x\ T_1 \Rightarrow \texttt{of}\ (R\ f\ x)\ T_2.$

$\texttt{of}\ (\texttt{app}\ M_1\ M_2)\ T\ \texttt{:-}\ \texttt{of}\ M_1\ (\texttt{arr}\ T_1\ T)\,,\texttt{of}\ M_2\ T_1.$

The only interesting clauses are the second and third ones to the last pertaining to the binding constructs `fix` and `let`. Note how the required freshness constraint and the extension of the typing contexts are realized in these clauses: take the clause for `fix` as an example, the universal goals over f and x introduce new names and the application $(R\ f\ x)$ replaces the bound variables with these names, and the hypothetical goals dynamically introduce the typing assignments for these names and generate the new typing judgment that must be derived.

5.3.2 Specifying the CPS transformation

An important task in specifying the CPS transformation is to capture the reading of administrative β-redexes. We shall use meta-level λ-abstraction in λProlog to represent the administrative λ-abstractions. As a result, administrative β-redexes become meta-level β-redexes and substitution denoted by them is realized by meta-level β-reduction.

With this in mind, we encode the rule-based description of the CPS transformation in Section 5.2 as follows. We first identify the predicate symbol

$$\texttt{cps} : \texttt{tm} \to (\texttt{tm} \to \texttt{tm}) \to \texttt{tm} \to \texttt{o}$$

for representing the transformation relation such that $\texttt{cps}\ M\ K\ M'$ holds if and only if given the input continuation K the source term M is transformed into the CPS term M'. Note here we use a meta-level abstraction K to encode the input continuation which is an administrative λ-abstraction. Since M can be an open term, we also need to keep track of the context of the CPS transformation that contains all the free variables in M. Similar to the encoding of the typing rules, this context is implicitly represented by the dynamic context and thus is not explicitly given as an argument of `cps`. The transformation rules in Figure 5.3 then translate into the following clauses defining `cps`:

$\texttt{cps}\ (\texttt{nat}\ N)\ K\ (K\ (\texttt{nat}\ N)).$

$\texttt{cps}\ \texttt{unit}\ K\ (K\ \texttt{unit}).$

$\texttt{cps}\ (\texttt{pred}\ M)\ K\ M'\ \texttt{:-}\ \texttt{cps}\ M\ (x \setminus \texttt{let}\ (\texttt{pred}\ x)\ (v \setminus K\ v))\ M'.$

$\texttt{cps}\ (\texttt{plus}\ M_1\ M_2)\ K\ M'\ \texttt{:-}$

$\quad (\Pi x_1.\texttt{cps}\ M_2\ (x_2 \setminus \texttt{let}\ (\texttt{plus}\ x_1\ x_2)\ (v \setminus K\ v))\ (M_2'\ x_1)), \texttt{cps}\ M_1\ M_2'\ M'.$

$\texttt{cps}\ (\texttt{pair}\ M_1\ M_2)\ K\ M'\ \texttt{:-}$

$\quad (\Pi x_1.\texttt{cps}\ M_2\ (x_2 \setminus \texttt{let}\ (\texttt{pair}\ x_1\ x_2)\ (v \setminus K\ v))\ (M_2'\ x_1)), \texttt{cps}\ M_1\ M_2'\ M'.$

$\texttt{cps}\ (\texttt{ifz}\ M_1\ M_2\ M_3)\ K\ M'\ \texttt{:-}$

$\quad (\Pi k.\texttt{cps}\ M_2\ (x \setminus \texttt{app}\ k\ x)\ (M_2'\ k)), (\Pi k.\texttt{cps}\ M_3\ (x \setminus \texttt{app}\ k\ x)\ (M_3'\ k)),$

$\quad \texttt{cps}\ M_1\ (x_1 \setminus \texttt{let}\ (\texttt{fix}\ f \setminus K)\ (k \setminus \texttt{ifz}\ x_1\ (M_2'\ k)\ (M_3'\ k)))\ M'.$

$\texttt{cps}\ (\texttt{fst}\ M)\ K\ M'\ \texttt{:-}\ \texttt{cps}\ M\ (x \setminus \texttt{let}\ (\texttt{fst}\ x)\ (v \setminus K\ v))\ M'.$

$\texttt{cps}\ (\texttt{snd}\ M)\ K\ M'\ \texttt{:-}\ \texttt{cps}\ M\ (x \setminus \texttt{let}\ (\texttt{snd}\ x)\ (v \setminus K\ v))\ M'.$

$\texttt{cps}\ (\texttt{let}\ M\ R)\ K\ M'\ \texttt{:-}$

$\quad (\Pi x, k.(\texttt{cps}\ x\ k\ (k\ x)) \Rightarrow \texttt{cps}\ (R\ x)\ K\ (R'\ x)), \texttt{cps}\ M\ R'\ M'.$

$\texttt{cps}\ (\texttt{fix}\ R)\ K$

$\quad (\texttt{let}\ (\texttt{fix}\ (f \setminus p \setminus \texttt{let}\ (\texttt{fst}\ p)\ (k \setminus \texttt{let}\ (\texttt{snd}\ p)\ (x \setminus R'\ f\ k\ x))))\ K)\ \texttt{:-}$

$\quad \Pi k, f, x.(\Pi k.\texttt{cps}\ f\ k\ (k\ f)) \Rightarrow (\Pi k.\texttt{cps}\ x\ k\ (k\ x)) \Rightarrow$

$\qquad\qquad \texttt{cps}\ (R\ f\ x)\ (y \setminus \texttt{app}\ k\ y)\ (R'\ f\ k\ x).$

$\texttt{cps}\ (\texttt{app}\ M_1\ M_2)\ K\ M'\ \texttt{:-}$

$\quad (\Pi x_1.\texttt{cps}\ M_2\ (x_2 \setminus \texttt{let}\ (\texttt{fix}\ f \setminus K)(k \setminus \texttt{let}\ (\texttt{pair}\ k\ x_2)\ (p \setminus \texttt{app}\ x_1\ p)))$

$\qquad\qquad (M_2'\ x_1)),$

$\quad \texttt{cps}\ M_1\ M_2'\ M'.$

By using the meta-level λ-abstractions and applications to represent their administrative counterparts, every rule in Figure 5.3 except cps-var transparently translates into one clause for `cps`. Note that in the clauses corresponding to the rules cps-if and cps-app the term $(\texttt{fix}\ f \setminus K)$ (*i.e.*, $(\texttt{fix}\ f \setminus a \setminus K\ a)$) where f does not occur in K is used to represent the function $(\lambda a.\,(@\ K\ a))$ in the original rules. The freshness side conditions of the original rules are captured in a concise and logically precise way by using the λ-tree representation, meta-level applications and universal goals. For example, given the function ($\texttt{fix}\ R$) and the continuation K, to derive the $\texttt{cps}\ (\texttt{fix}\ R)\ K\ M'$ for some M', we must backchain this goal on the penultimate clause for cps and prove the following formula for some R':

$$\Pi k, f, x.(\Pi k.\texttt{cps}\ f\ k\ (k\ f)) \Rightarrow (\Pi k.\texttt{cps}\ x\ k\ (k\ x)) \Rightarrow \\ \texttt{cps}\ (R\ f\ x)\ (y \setminus \texttt{app}\ k\ y)\ (R'\ f\ k\ x).$$

Derivation of this goal must introduce constants for k, f and x that are fresh with respect to ($\texttt{fix}\ R$), K and R'. Those constants will replace k, f and x in the subsequent derivation of $\texttt{cps}\ (R\ f\ x)\ (y \setminus \texttt{app}\ k\ y)\ (R'\ f\ k\ x)$ where the meta-level application $(R\ f\ x)$ represents the body of the function and $(R'\ f\ k\ x)$ represents its CPS form. This example also illustrates how the cps-var rule is represented in our encoding. Similar to the encoding of the transformation between λ-terms and de Bruijn terms in Section 3.1.1, we use hypothetical goals to dynamically introduce program clauses that represent the transformation rule for the binding variables at the point they are introduced via universal goals. In the case of the above example, the binding variables are f and x and the clauses representing cps-var for them are $(\Pi k.\texttt{cps}\ f\ k\ (k\ f))$ and $(\Pi k.\texttt{cps}\ x\ k\ (k\ x))$. When the source term becomes a variable, the transformation makes progress by backchaining on the appropriate clause for that variable in the dynamic context.

5.4 Informal Verification of the Transformation

We give an informal description of the verification of the CPS transformation in this section based on the ideas presented in Section 4.2. It serves as the basis for the formal verification of the CPS transformation in Abella which we shall discuss in Section 5.5.

5.4.1 Type preservation of the transformation

Before we start talking about the semantics preservation property of the CPS transformation, let us first prove that the CPS transformation preserves types. This type preservation property is essential for composing the semantics preservation proofs and have a special use for deriving the property of being closed terms which is useful in the formal verification. The description of type preservation in this section is partially based on [35].

Strictly speaking, the CPS transformation changes the types of terms, albeit in a systematic way. Because the CPS transformation embeds the (transformed) source term into a context (which is the initial input continuation), the type of the output of the CPS transformation is determined by the target type of the context. Moreover, since continuations in the transformed source term represent future computations which always end up in evaluating the embedding of the value of the source term in the initial context, their target type must be the same as the target type of the context. The only (sub)terms from the source term whose types are changed by the transformation are functions because every function after the transformation takes an extra argument which is a continuation that will be called when the function returns. By this observation, we represent a mapping between types of terms before the CPS transformation and that after the transformation as a ternary relation $\simeq$ such that $T_1 \simeq_S T_2$ holds if the type T_1 of a source term is mapped to T_2 given the target type S of the initial context. This relation is defined by the rules depicted in Figure 5.4. The last rule formalizes the mapping between function types based the previous observation: a function of type $T_1 \to T_2$ after the CPS transformation has the type $((T_2' \to S) \times T_1') \to S$ where T_1' is the type of the original argument after the transformation and T_2' is the type of the function body after the transformation; note that the function takes a continuation argument of type $(T_2' \to S)$ and its target type indicates an application of the function should have the same type as that of the continuation argument, which conforms to the way the continuation argument is used in the function.

$$\frac{}{\mathbb{N} \simeq_S \mathbb{N}} \qquad \frac{}{\mathbf{unit} \simeq_S \mathbf{unit}} \qquad \frac{T_1 \simeq_S T_1' \quad T_2 \simeq_S T_2'}{T_1 \times T_2 \simeq_S T_1' \times T_2'}$$

$$\frac{T_1 \simeq_S T_1' \quad T_2 \simeq_S T_2'}{T_1 \to T_2 \simeq_S ((T_2' \to S) \times T_1') \to S}$$

Figure 5.4: The Rules for Mapping Types of Terms in the CPS Transformation

Now we state the type preservation property of the CPS transformation, as follows:

Theorem 2. *Let M be a term whose free variables are contained in the set $\rho = \{x_1, \ldots, x_n\}$, Γ be a typing context $(x_1 : T_1, \ldots, x_n : T_n)$ such that $\Gamma \vdash M : T$ holds for some T, K be a continuation of type $T' \to S$ such that $T \simeq_S T'$ holds, Γ' be the type context $(x_1 : T'_1, \ldots, x_n : T'_n)$ such that $T_i \simeq_S T'_i$ holds for $1 \leq i \leq n$. If $\rho \triangleright M; K \rightsquigarrow_{cps} M'$ holds for some M', then $\Gamma' \vdash M' : S$ holds.*

In essence, this theorem states that if a source term M is CPS transformed into M' in the context K, then M' has the target type of K. We prove it by induction on the derivation of $\rho \triangleright M; K \rightsquigarrow_{cps} M'$ and analyzing the last rule of the derivation. When the last rule is cps-nat, cps-var or cps-unit, the proof is obvious. The rest of the cases are proved by following a set pattern, as follows. We examine the premises of the last rule from left and right. For each premise, we apply the induction hypothesis to get the type of its output. Once this type is known, it is easy to derive the type of the input continuation in the next premise (if any). We can then repeat the previous process on the next premise. This continues until we have derived the type of the output of the conclusion. At that point this case is finished.

Given Theorem 2, it is easy to show the following type preservation property for closed terms:

Corollary 2. *If $\vdash M : \mathbb{N}$, K is a continuation of type $\mathbb{N} \to S$ and $\emptyset \triangleright M; K \rightsquigarrow_{cps} M'$, then $\vdash M' : S$.*

By choosing K to be $\hat{\lambda} x.\, x$, we get the following corollary:

Corollary 3. *If $\vdash M : \mathbb{N}$ and $\emptyset \triangleright M; (\hat{\lambda} x.\, x) \rightsquigarrow_{cps} M'$, then $\vdash M' : \mathbb{N}$.*

5.4.2 Semantics preservation of the transformation

We informally describe the proof of semantics preservation for the CPS transformation in this section. We first describe the operational (evaluation) semantics of the source/target language of the transformation, then the logical relations for denoting equivalence between the source and target programs and their properties, and finally the semantics preservation theorem and its proof.

Operational semantics of the source/target language

To describe semantics preservation, we must first define the operational semantics of the languages involved the CPS transformation. Since the source language and the target languages of the CPS transformation are the same, they share the same operational semantics. The operational semantics is based on a left to right, call-by-value evaluation strategy. We assume that this is given in a small-step form and, we write $M \hookrightarrow_1 M'$ to denote that M evaluates to M' in one step. The evaluation rules are defined in a standard way, as shown in Figure 5.5. The only interesting rule is the last one which shows how recursion works in the language: evaluating an application of a recursive function to a value causes the occurrences of the function parameter in the function body to be replaced by the function itself. One-step evaluation generalizes to n-step evaluation that we denote by $M \hookrightarrow_n M'$ and that holds if M reduces to M' by n one-step evaluations. We shall write $M \hookrightarrow_* M'$ to denote that there exists some n such that $M \hookrightarrow_n M'$ holds. Finally, we write $M \hookrightarrow V$ to denote the evaluation of M to the value V through 0 or more steps.

Logical relations and their properties

Following the ideas in Section 4.2, we use step-indexing logical relations to characterize the semantics preservation property between source and target programs of the CPS transformation. We define the mutually recursive simulation relation $\sim$ between pairs of closed source terms and input continuations and target terms and the equivalence relation $\approx$ between closed source and target values, each indexed by a type and a step measure, in Figure 5.6. We give an intuitive explanation of $\sim$ and $\approx$ as follows.

The simulation relation $M; K \sim_{T;i} M'$ holds between the term M, the continuation K and the term M' where K is the context in which the CPS transformed M will be used and M' can be thought of as the program obtained by combining the CPS form of M and the application of K to the value of M. When the relation holds, we say M simulates M' in the context K at the type T within i steps. Its definition is interpreted as follows: if M evaluates to some value V, then its CPS form must evaluate to an equivalent value V' and hence M' must evaluates to the result of applying K to V', which is represented by the administrative β-redex @ $K\ V'$. The uses of indexing types

$$\frac{M \hookrightarrow_1 M'}{\textbf{pred } M \hookrightarrow_1 \textbf{pred } M'}\ \text{step-pred} \qquad \frac{n' \text{ is the predecessor of } n}{\textbf{pred } n \hookrightarrow_1 n'}\ \text{step-pred-base}$$

$$\frac{M_1 \hookrightarrow_1 M_1'}{M_1 + M_2 \hookrightarrow_1 M_1' + M_2}\ \text{step-plus-left} \qquad \frac{M_2 \hookrightarrow_1 M_2'}{V_1 + M_2 \hookrightarrow_1 V_1 + M_2'}\ \text{step-plus-right}$$

$$\frac{n_3 \text{ is the sum of } n_1 \text{ and } n_2}{n_1 + n_2 \hookrightarrow_1 n_3}\ \text{step-plus-base}$$

$$\frac{M_1 \hookrightarrow_1 M_1'}{\textbf{if } M_1 \textbf{ then } M_2 \textbf{ else } M_3 \hookrightarrow_1 \textbf{if } M_1' \textbf{ then } M_2 \textbf{ else } M_3}\ \text{step-if-cond}$$

$$\frac{}{\textbf{if } 0 \textbf{ then } M_1 \textbf{ else } M_2 \hookrightarrow_1 M_1}\ \text{step-if-left} \qquad \frac{n > 0}{\textbf{if } n \textbf{ then } M_1 \textbf{ else } M_2 \hookrightarrow_1 M_2}\ \text{step-if-right}$$

$$\frac{M_1 \hookrightarrow_1 M_1'}{(M_1, M_2) \hookrightarrow_1 (M_1', M_2)}\ \text{step-pair-left} \qquad \frac{M_2 \hookrightarrow_1 M_2'}{(V_1, M_2) \hookrightarrow_1 (V_1, M_2')}\ \text{step-pair-right}$$

$$\frac{M \hookrightarrow_1 M'}{\textbf{fst } M \hookrightarrow_1 \textbf{fst } M'}\ \text{step-fst} \qquad \frac{}{\textbf{fst } (V_1, V_2) \hookrightarrow_1 V_1}\ \text{step-fst-base}$$

$$\frac{M \hookrightarrow_1 M'}{\textbf{snd } M \hookrightarrow_1 \textbf{snd } M'}\ \text{step-snd} \qquad \frac{}{\textbf{snd } (V_1, V_2) \hookrightarrow_1 V_2}\ \text{step-snd-base}$$

$$\frac{M_1 \hookrightarrow_1 M_1'}{\textbf{let } x = M_1 \textbf{ in } M_2 \hookrightarrow_1 \textbf{let } x = M_1' \textbf{ in } M_2}\ \text{step-let-arg}$$

$$\frac{}{\textbf{let } x = V \textbf{ in } M \hookrightarrow_1 M[V/x]}\ \text{step-let-body}$$

$$\frac{M_1 \hookrightarrow_1 M_1'}{M_1\ M_2 \hookrightarrow_1 M_1'\ M_2}\ \text{st-app-fun} \qquad \frac{M_2 \hookrightarrow_1 M_2'}{V_1\ M_2 \hookrightarrow_1 V_1\ M_2'}\ \text{st-app-arg}$$

$$\frac{}{(\textbf{fix } f\, x.M)\ V \hookrightarrow_1 M[\textbf{fix } f\, x.M/f][V/x]}\ \text{st-app-base}$$

Figure 5.5: Evaluation Rules for the Source Language of the CPS Transformation

and steps in the definition of $\sim$ indicate that M simulates M' within at least i steps of evaluation at the type T.

The equivalence relation $V \approx_{T;i} V'$ holds if and only if the values V and V' cannot be distinguished from each other (*i.e.*, considered as equivalent) in any context within at least i steps of evaluation at the type T. Its definition is explained as follows. At the types of natural numbers and unit, two values are equivalent exactly when they are identical. Two pairs of values are considered equivalent within at least i steps exactly when their constituting elements are respectively equivalent within at least i steps.

The equivalence relation at the functions types holds between two functions $\mathbf{fix}\ f\,x.M$ and $\mathbf{fix}\ f\,p.M'$ where the latter can be thought of as the CPS form of the former whose parameter p denotes a pair consisting of the input continuation and the actual argument of the function. The two functions are equivalent within at least i steps exactly when the following holds: for any j less than i, given any continuation K and any arguments that are equivalent within at least j steps, the result of applying $\mathbf{fix}\ f\,x.M$ to the source arguments simulates the result of applying $\mathbf{fix}\ f\,p.M'$ to the target arguments in the context K within at least j steps. Note that the term $\hat{\lambda}x.\,@\ K\ x$ (which is a term equivalent to K in the object language) is fed to $\mathbf{fix}\ f\,p.M'$ as its input continuation, which conforms to how functions in CPS form should work: when such a function finishes its evaluation, computation proceeds by applying the continuation representing the remaining computation—in our case, K—to the evaluation result. Note that the definition of $\approx$ in the function case uses $\approx$ negatively at the same type. However, it is still a well-defined notion because the index decreases.

$$\begin{array}{l}
M; K \sim_{T;i} M' \iff \\
\quad \forall j \leq i. \forall V. M \hookrightarrow_j V \supset \exists V'. M' \hookrightarrow_* @\ K\ V' \wedge V \approx_{T;i-j} V'; \\
n \approx_{\mathbb{N};i} n; \\
() \approx_{\mathbf{unit};i} (); \\
(V_1, V_2) \approx_{T_1 \times T_2;i} (V_1', V_2') \iff V_1 \approx_{T_1;i} V_1' \wedge V_2 \approx_{T_2;i} V_2'; \\
\mathbf{fix}\ f\,x.M \approx_{T_1 \to T_2;i} \mathbf{fix}\ f\,p.M' \iff \\
\quad \forall j < i. \forall V_1, V_1', V_2, V_2', K. V_1 \approx_{T_1;j} V_1' \supset V_2 \approx_{T_1 \to T_2;j} V_2' \supset \\
\qquad M[V_2/f, V_1/x]; K \sim_{T_2;j} M'[V_2'/f, (\hat{\lambda}x.\,@\ K\ x, V_1')/p].
\end{array}$$

Figure 5.6: The Logical Relations for Verifying the CPS Transformation

A property we will need about the above step-indexing logical relations is that $\approx$ is closed under decreasing indexes. It is stated as the following lemma:

Lemma 2. *If $V \approx_{T;i} V'$ holds, then for any j such that $j \leq i$, $V \approx_{T;j} V'$ holds.*

Note that we cannot prove this lemma by induction on the relation $V \approx_{T;i} V'$ because, as we have explained in Section 4.2, $\approx$ is not an inductive definition. Instead, $\approx$ is a recursive definition based on inductive defined indexing types and steps. We therefore prove this lemma by a (nested) induction first on the types and then on the step indexes of $\approx$. The proof itself is obvious. However, it is important to keep in mind of the following recurring theme about proving properties of logical relations: These properties cannot be proved by induction on logical relations but instead are usually proved by induction on their indexes or other inductively defined relations.

We show that simulation relations can be composed similar to how the transformation relations are built by using the transformation rules. Those properties, commonly known as "compatibility lemmas", are shown as follows:

Lemma 3. *1. If* $M; (\hat{\lambda}x.\, \textbf{let}\ v = \textbf{pred}\ x\ \textbf{in}\ @\ K\ v) \sim_{\mathbb{N};i} M'$ *then* $(\textbf{pred}\ M); K \sim_{\mathbb{N};i} M'$;

2. If $M_2; (\hat{\lambda}x_2.\, \textbf{let}\ v = x_1 + x_2\ \textbf{in}\ @\ K\ v) \sim_{\mathbb{N};i} M'_2$ *and* $M_1; (\hat{\lambda}x_1.\, M'_2) \sim_{\mathbb{N};i} M'$ *then* $(M_1 + M_2); K \sim_{\mathbb{N};i} M'$.

3. If $M; (\hat{\lambda}x.\, \textbf{let}\ v = \textbf{fst}\ x\ \textbf{in}\ @\ K\ v) \sim_{T_1 \times T_2;i} M'$ *then* $(\textbf{fst}\ M); K \sim_{T_1;i} \textbf{fst}\ M'$.

4. If $M; (\hat{\lambda}x.\, \textbf{let}\ v = \textbf{snd}\ x\ \textbf{in}\ @\ K\ v) \sim_{T_1 \times T_2;i} M'$ *then* $(\textbf{snd}\ M); K \sim_{T_2;i} \textbf{fst}\ M'$.

5. If $M_2; (\hat{\lambda}x_2.\, \textbf{let}\ v = (x_1, x_2)\ \textbf{in}\ @\ K\ v) \sim_{T_2;i} M'_2$ *and* $M_1; \hat{\lambda}x_1.\, M'_2 \sim_{T_1;i} M'$ *then* $(M_1, M_2); K \sim_{T_1 \times T_2;i} M'$.

6. If $M_2; K \sim_{T;i} M'_2$, $M_3; K \sim_{T;i} M'_3$ *and* $M_1; (\hat{\lambda}x_1.\, \textbf{if}\ x_1\ \textbf{then}\ M'_2\ \textbf{else}\ M'_3) \sim_{\mathbb{N};i} M'$, *then* $(\textbf{if}\ M_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3); K \sim_{T;k} M'$.

7. If $M_2; (\hat{\lambda}x_2.\, \textbf{let}\ k = \lambda a.\, (@\ K\ a)\ \textbf{in}\ \textbf{let}\ p = (k, x_2)\ \textbf{in}\ (x_1\ p)) \sim_{T_1;i} M'_2$ *and* $M_1; \hat{\lambda}x_1.\, M'_2 \sim_{T_1 \to T;i} M'$ *then* $(M_1\ M_2); K \sim_{T_2;i} M'$.

These lemmas are proved by analyzing the simulation relation and using the evaluation rules. The main complication involved in those proofs is caused by calculating and comparing the step measures, which is an inherent difficulty in any proof techniques that make use of step-indexing. Some proofs of these properties need the property that the equivalence relation is closed under decreasing indices, which have already proved as Lemma 2. For instance, the proof of the last of these properties requires us to consider the evaluation of the application of fixed point expressions which involves "feeding" these expressions that are equivalent at a lower step measure as arguments to their own body. These arguments are obtained by applying Lemma 2.

Informal proof of semantics preservation

Our notion of equivalence only relates closed terms. However, the CPS transformation typically operates on open terms. To handle this situation, we consider semantics preservation for possibly open terms under closed substitutions. We will take substitutions in both the source and target settings to be simultaneous mappings of closed values for a finite collection of variables, written as $(V_1/x_1, \ldots, V_n/x_n)$. The equivalence between substitutions θ and θ' is indexed by a typing context Γ and a step measure i and written as $\theta \approx_{\Gamma;i} \theta'$. Its definition is given as follows:

$$(V_1/x_1, \ldots, V_n/x_n) \approx_{(x_1:T_1,\ldots,x_n:T_n);k} (V_1'/x_1, \ldots, V_n'/x_n) \iff (\forall 1 \leq i \leq n. V_i \approx_{T_i;k} V_i')$$

The semantics preservation theorem for the CPS transformation can now be stated as follows, which is a realization of Property 3 for the CPS transformation:

Theorem 3. *Let $\Gamma = (x_1 : T_1, \ldots, x_n : T_n)$, $\rho = (x_1, \ldots, x_n)$, $\theta \approx_{\Gamma;i} \theta'$ and $\Gamma \vdash M : T$. If $\rho \triangleright M; K \rightsquigarrow_{cps} M'$, then $M[\theta]; K[\theta'] \sim_{T;i} M'[\theta']$.*

Proof. We outline the main steps in the argument for this theorem: these will guide the development of a formal proof in Section 5.5. We proceed by induction on the derivation of $\rho \triangleright M; K \rightsquigarrow_{cps} M'$, analyzing the last step in it. This obviously depends on the structure of M. The cases for a natural number, the unit constructor or a variable are obvious. In the remaining cases, other than when M is of the form **let** $x = M_1$ **in** M_2 or **fix** $f\,x.M_1$, the argument follows a set pattern: we observe that substitutions distribute to the sub-components of expressions, we invoke the induction hypothesis over the sub-components and then we use Lemma 3 to conclude. If M is of the form **let** $x = M_1$ **in** M_2, then we have $\rho, x \triangleright M_2; K \rightsquigarrow_{cps} M_2'$ and $\rho \triangleright M_1; \hat{\lambda} x.\, M_2' \rightsquigarrow_{cps} M'$ for some M_2'. Here again the substitutions distribute over M_1, M_2, M_2' and M'. We can therefore apply the inductive hypothesis (I.H.) to $\rho \triangleright M_1; \hat{\lambda} x.\, M_2' \rightsquigarrow_{cps} M'$ to get a simulation relation S. However, we cannot directly apply I.H. to $\rho, x \triangleright M_2; K \rightsquigarrow_{cps} M_2'$ since the transformation on M_2 introduces x as a new variable into ρ. For this we need to extend the substitutions with equivalent values for x. These values are derived from the previous simulation relation S. This case is concluded by following the pattern for proving Lemma 3 after we got the simulation relations from applying I.H. to M_1 and M_2.

Finally, if M is of the form **fix** $f\,x.M_1$, then we have $\rho, f, x \triangleright M_1; \hat{\lambda} y.\, k\ y \rightsquigarrow_{cps} M_1'$. By the definition of $\sim$, the simulation relation we need to prove reduces into an equivalence relation R on function values. Similar to the last case, to apply I.H. to M_1 we need to extend the substitutions with equivalent values for f and x which are obtained from the assumptions in R. The process of proving R after applying I.H. is like in other cases. Another important yet subtle point we need to show is that the functions related by R must be closed, which is equivalent to showing that $M[\theta]$ and $M'[\theta']$ are closed. It is easy to show that $M[\theta]$ is closed by observing that the values of θ are closed and the domain of θ is the same as that of Γ which contains all the free variables of M. To show $M'[\theta']$ is closed, we need to apply the type preservation theorem—*i.e.*, Theorem 2—to M to show that the free variables of M' are also contained in Γ. $\square$

An immediate corollary of Theorem 3 is the following which correspond to Property 2 in the setting of CPS transformation:

Corollary 4. *If* $\vdash M : T$ *and* $\emptyset \triangleright M; K \rightsquigarrow_{cps} M'$*, then* $M; K \sim_{T;i} M'$ *for any* i*.*

From this corollary, it is easy to derive the following correctness property of the CPS transformation for closed programs at atomic types:

Corollary 5. *If* $\vdash M : \mathbb{N}$*,* $\emptyset \triangleright M; K \rightsquigarrow_{cps} M'$ *and* $M \hookrightarrow V$*, then* $M' \hookrightarrow_* @\ K\ V$*.*

By choosing K to be the identity function $\hat{\lambda} x.\, x$, we get the following corollary that corresponds to Property 4 in the setting of the CPS transformation:

Corollary 6. *If* $\vdash M : \mathbb{N}$*,* $\emptyset \triangleright M; \hat{\lambda} x.\, x \rightsquigarrow_{cps} M'$ *and* $M \hookrightarrow V$*, then* $M' \hookrightarrow V$*.*

5.5 Verifying the λProlog Program in Abella

In this section, we formally verify the correctness of the CPS transformation in Abella based on the informal proof given in Section 5.4. The most significant difference between the formal and informal proof is that the binding related properties in the former must be made explicit, often stated and proved formally as theorems. We show that the λ-tree syntax approach can be used to significantly alleviate this difficulty, producing a formal correctness proof that closely follows the informal one.

5.5.1 Type preservation of the transformation

We prove that the λProlog implementation of the CPS transformation given in Section 5.3.2 preserves typing by following the informal argument given in Section 5.4.1. We start by giving an inductive definition of the types of the source (target) language, which will be useful later when induction on types are needed. We identify a constant $\texttt{is\_sty} : \texttt{ty} \to \texttt{o}$ such that $\texttt{is\_sty}\ T$ holds if and only if T is a well-formed type. It is defined through the following λProlog clauses:

$\texttt{is\_sty}\ \texttt{tnat}.$
$\texttt{is\_sty}\ \texttt{tunit}.$
$\texttt{is\_sty}\ (\texttt{prod}\ T_1\ T_2)\ \texttt{:-}\ \texttt{is\_sty}\ T_1\ ,\texttt{is\_sty}\ T_2.$
$\texttt{is\_sty}\ (\texttt{arr}\ T_1\ T_2)\ \texttt{:-}\ \texttt{is\_sty}\ T_1\ ,\texttt{is\_sty}\ T_2.$

The typing context of the source (target) language is characterized as the following fixed-point definition for $\texttt{sctx} : \texttt{olist} \to \texttt{prop}$:

$$\begin{array}{lcl}
\texttt{sctx}\ \texttt{nil} & \triangleq & \top \\
\nabla x.\texttt{sctx}\ (\texttt{of}\ x\ T :: L) & \triangleq & \texttt{sctx}\ L \land \{\texttt{is\_sty}\ T\}.
\end{array}$$

By this definition, $\texttt{sctx}\ L$ holds if and only if L consists of formulas of the form $\texttt{of}\ x\ T$ where T is well-formed and assigns types to unique nominal constants. If $\texttt{sctx}\ L$ holds, then $\{L \vdash \texttt{of}\ M\ T\}$ holds if and only if M is a well-formed term of type T whose free variables are represented by the nominal constants in L.

The mapping between types before and after the CPS transformation is defined through the following clauses for $\texttt{cps\_ty} : \texttt{ty} \to \texttt{ty} \to \texttt{ty} \to \texttt{prop}$ such that $T_1 \simeq_S T_2$ holds if and only if $\texttt{cps\_ty}\ S\ T_1\ T_2$ holds.

$$\begin{array}{l}
\texttt{cps\_ty}\ S\ \texttt{nat}\ \texttt{nat} \quad \triangleq \quad \top \\
\texttt{cps\_ty}\ S\ \texttt{unit}\ \texttt{unit} \quad \triangleq \quad \top \\
\texttt{cps\_ty}\ S\ (\texttt{prod}\ T_1\ T_2)\ (\texttt{prod}\ T_1'\ T_2') \quad \triangleq \quad \texttt{cps\_ty}\ S\ T_1\ T_1' \land \texttt{cps\_ty}\ S\ T_2\ T_2' \\
\texttt{cps\_ty}\ S\ (\texttt{arr}\ T_1\ T_2)\ (\texttt{arr}\ (\texttt{prod}\ (\texttt{arr}\ T_2'\ S)\ T_1')\ S) \quad \triangleq \\
\qquad \texttt{cps\_ty}\ S\ T_1\ T_1' \land \texttt{cps\_ty}\ S\ T_2\ T_2'.
\end{array}$$

These clauses are transparently translated from the rules in Figure 5.4. The mapping between types can be generalized to typing contexts, as follows:

$$\begin{array}{l}\texttt{cps\_sctx}\, S\, \texttt{nil}\, \texttt{nil} \quad \triangleq \quad \top \\ \nabla x.\texttt{cps\_sctx}\, S\, (\texttt{of}\; x\; T :: L)\; (\texttt{of}\; x\; T' :: L') \quad \triangleq \\ \qquad \texttt{cps\_ty}\; S\; T\; T' \wedge \texttt{cps\_sctx}\; S\; L\; L'.\end{array}$$

The λProlog implementation of the transformation makes use of the dynamic context to bind the free variables of the source term and to represent the rules for transforming these variables. We characterized such a dynamic context by the following definition for $\texttt{cctx} : \texttt{olist} \rightarrow \texttt{prop}$:

$$\begin{array}{lcl}\texttt{cctx}\, \texttt{nil} & \triangleq & \top \\ \nabla x.\texttt{cctx}\, ((\Pi k.\texttt{cps}\; x\; k\; (k\; x)) :: L) & \triangleq & \texttt{cctx}\, L.\end{array}$$

We can now state the type preservation theorem of the transformation in Abella as follows, called `cps_typ_pres`:

$$\begin{array}{l}\forall TL, CL, M, T, K, M', T', TL', S. \\ \quad \{\texttt{is\_sty}\; T\} \supset \{\texttt{is\_sty}\; S\} \supset \texttt{sctx}\; TL \supset \texttt{cctx}\; CL \supset \\ \quad \{TL \vdash \texttt{of}\; M\; T\} \supset \{CL \vdash \texttt{cps}\; M\; K\; M'\} \supset \\ \quad \texttt{cps\_ty}\; S\; T\; T' \supset \texttt{cps\_sctx}\; S\; TL\; TL' \supset \\ \qquad \{TL', \Pi x.\texttt{of}\; x\; T' \Rightarrow \texttt{of}\; (K\; x)\; S \vdash \texttt{of}\; M'\; S\}.\end{array}$$

This is a direct translation from Theorem 2. Note that $\texttt{sctx}\; TL$ and $\{TL \vdash \texttt{of}\; M\; T\}$ together assert that M has type T in the context TL, and $\texttt{cctx}\; CL$ and $\{CL \vdash \texttt{cps}\; M\; K\; M'\}$ together assert that M whose free variables are bound in CL with the input continuation K is transformed into M'. The (higher-order) formula $\Pi x.\texttt{of}\; x\; T' \Rightarrow \texttt{of}\; (K\; x)\; S$ in the dynamic context of the conclusion asserts that K has the type $T' \rightarrow S$. The theorem is proved by induction on $\{CL \vdash \texttt{cps}\; M\; K\; M'\}$. By the transparent encoding of HH^ω in Abella and the transparent relation between the λProlog implementation and the rules its encodes, the proof corresponds to induction on the derivation of the HH^ω judgment $CL \vdash \texttt{cps}\; M\; K\; M'$ and furthermore on the derivation using the transformation rules. It has the same structure as its informal rendition.

From the above theorem, it is easy to prove the following rendition of Corollary 2 in Abella, called `cps_typ_pres_closed`:

$$\begin{array}{l}\forall S, M, K, M'.\{\texttt{is\_sty}\; S\} \supset \{\texttt{of}\; M\; \texttt{tnat}\} \supset \{\texttt{cps}\; M\; K\; M'\} \supset \\ \quad \{\Pi x.\texttt{of}\; x\; \texttt{tnat} \Rightarrow \texttt{of}\; (K\; x)\; S \vdash \texttt{of}\; M'\; S\}.\end{array}$$

Letting $K = x \setminus x$, it is easy to prove the following rendition of Corollary 3:

$$\forall M, M'.\{\texttt{of}\ M\ \texttt{tnat}\} \supset \{\texttt{cps}\ M\ (x \setminus x)\ M'\} \supset \{\texttt{of}\ M'\ \texttt{tnat}\}.$$

Now we have formally proved that the implementation of the CPS transformation preserves types.

5.5.2 Semantics preservation of the transformation

In this section, we formally develop the semantics preservation proof of the CPS transformation in Abella. We place a focus on showing how the λ-tree syntax approach can alleviate the difficulties in reasoning about binding structure. Specifically, we show how the property of being closed can be elegantly encoded as a λProlog program, how the definition of substitution as a relation can be used to derive "boilerplate" properties with very little effort, how the definition of substitution is used to prove the closedness property, and how the logical structure of binding related treatments in the λProlog can be exploit to greatly simplify the correctness proof of the transformation.

Formalizing the operational semantics

We formalize the small-step operational semantics as a λProlog program. For this we need to formally define what are values. We identify the constant $\texttt{val} : \texttt{tm} \to \texttt{o}$ such that $\texttt{val}\ V$ holds if and only if V is a value. It is defined by the following program clauses:

$\texttt{val}\ (\texttt{nat}\ N).$
$\texttt{val}\ \texttt{unit}.$
$\texttt{val}\ (\texttt{pair}\ V_1\ V_2)\ \texttt{:-}\ \texttt{val}\ V_1, \texttt{val}\ V_2.$
$\texttt{val}\ (\texttt{fix}\ R).$

We also need to define addition and predecessor operations on natural numbers. They are given through the following clauses with obvious meanings:

$\texttt{npred}\ \texttt{z}\ \texttt{z}.$
$\texttt{npred}\ (\texttt{s}\ N)\ N.$
$\texttt{add}\ \texttt{z}\ N\ N.$
$\texttt{add}\ (\texttt{s}\ N_1)\ N_2\ (\texttt{s}\ N_3)\ \texttt{:-}\ \texttt{add}\ N_1\ N_2\ N_3.$

We designate the constant step : tm $\to$ tm $\to$ o to represent one-step evaluation such that step M_1 M_2 holds if M_1 evaluates to M_2 in one step. It is defined by the following program clauses:

step (pred M) (pred M') :- step M M'.
step (pred (nat N)) (nat N') :- npred N N'.
step (plus M_1 M_2) (plus M_1' M_2) :- step M_1 M_1'.
step (plus V_1 M_2) (plus V_1 M_2') :- val V_1 , step M_2 M_2'.
step (plus (nat N_1) (nat N_2)) (nat N) :- add N_1 N_2 N.
step (ifz M M_1 M_2) (ifz M' M_1 M_2) :- step M M'.
step (ifz (nat z) M_1 M_2) M_1.
step (ifz (nat (s N)) M_1 M_2) M_2.
step (pair M_1 M_2) (pair M_1' M_2) :- step M_1 M_1'.
step (pair V_1 M_2) (pair V_1 M_2') :- val V_1 , step M_2 M_2'.
step (fst M) (fst M') :- step M M'.
step (fst (pair V_1 V_2)) V_1 :- val (pair V_1 V_2).
step (snd M) (snd M') :- step M M'.
step (snd (pair V_1 V_2)) V_2 :- val (pair V_1 V_2).
step (let M R) (let M' R) :- step M M'.
step (let V R) (R V) :- val V.
step (app M_1 M_2) (app M_1' M_2) :- step M_1 M_1'.
step (app V_1 M_2) (app V_1 M_2') :- val V_1 , step M_2 M_2'.
step (app (fix R) V) (R (fix R) V) :- val V.

These program clauses are transparently translated from the evaluation rules in Figure 5.5. Note here how application in the meta-language realizes substitution. We designate the constant nstep : nat $\to$ tm $\to$ tm $\to$ o to represent n-step evaluation, defined as follows:

nstep z M M.
nstep (s N) M M'' :- step M M' , nstep N M' M''.

Finally, we use the constant eval : tm $\to$ tm $\to$ o to represent the evaluation relation such that eval M V if and only if M evaluates to the value V in a finite number of steps, defined by the following clause:

$\texttt{eval}\ M\ V\ \texttt{:-}\ \texttt{nstep}\ N\ M\ V\ ,\texttt{val}\ V.$

Formalizing the closedness property

A notion related to bindings that is important for our verification task is the property of being closed terms, *i.e.*, terms that do not have any free variable. We have seen in Section 2.4.2 how the closed STLC terms are characterized in Abella. The same idea carries over to our setting. We first defined the predicate $\texttt{tm} : \texttt{tm} \to \texttt{o}$ such that $\{\texttt{tm}\ M\}$ holds if M is a well-formed term in our source (target) language through the following program clauses in λProlog:

$\texttt{tm}\ (\texttt{nat}\ N).$
$\texttt{tm}\ (\texttt{pred}\ M)\ \texttt{:-}\ \texttt{tm}\ M.$
$\texttt{tm}\ (\texttt{plus}\ M_1\ M_2)\ \texttt{:-}\ \texttt{tm}\ M_1\ ,\texttt{tm}\ M_2.$
$\texttt{tm}\ (\texttt{ifz}\ M\ M_1\ M_2)\ \texttt{:-}\ \texttt{tm}\ M\ ,\texttt{tm}\ M_1\ ,\texttt{tm}\ M_2.$
$\texttt{tm}\ \texttt{unit}.$
$\texttt{tm}\ (\texttt{pair}\ M_1\ M_2)\ \texttt{:-}\ \texttt{tm}\ M_1\ ,\texttt{tm}\ M_2.$
$\texttt{tm}\ (\texttt{fst}\ M)\ \texttt{:-}\ \texttt{tm}\ M.$
$\texttt{tm}\ (\texttt{snd}\ M)\ \texttt{:-}\ \texttt{tm}\ M.$
$\texttt{tm}\ (\texttt{let}\ M\ R)\ \texttt{:-}\ \texttt{tm}\ M\ ,\Pi x.\texttt{tm}\ x \Rightarrow \texttt{tm}\ (R\ x).$
$\texttt{tm}\ (\texttt{fix}\ R)\ \texttt{:-}\ \Pi f, x.\texttt{tm}\ f \Rightarrow \texttt{tm}\ x \Rightarrow \texttt{tm}\ (R\ f\ x).$
$\texttt{tm}\ (\texttt{app}\ M_1\ M_2)\ \texttt{:-}\ \texttt{tm}\ M_1\ ,\texttt{tm}\ M_2.$

These clauses resembles the clauses encoding the typing rules, except that they do not have any type information. Similar to the typing rules, we can define the context of well-formedness as follows:

$$\begin{array}{lcl} \texttt{tm\_sctx}\ \texttt{nil} & \triangleq & \top \\ \texttt{tm\_sctx}\ (\texttt{tm}\ X\ \texttt{::}\ L) & \triangleq & \texttt{tm\_sctx}\ L \wedge \texttt{name}\ X. \end{array}$$

Now, $\texttt{tm\_sctx}\ L$ and $\{L \vdash \texttt{tm}\ M\}$ hold if and only if M is a well formed term whose free variables are bound in L. We can therefore use the judgment $\{\texttt{tm}\ M\}$ to denote that M is closed.

We usually do not explicitly prove that a term is well-formed or closed. Instead, because we only deal with typed terms, we can derive well-formedness by making use of the property that well-typed terms are also well-formed. This property is stated as the following theorem, named `sof_to_tm`, and proved by induction on $\{L \vdash \texttt{of}\ M\ T\}$:

$$\begin{array}{l}\forall L, Vs, SL, M, T, \{\texttt{is\_sty}\ T\} \supset \texttt{sctx}\ L \supset \texttt{vars\_of\_sctx}\ L\ Vs \supset \texttt{tm\_sctx}\ SL \supset \\ \quad \texttt{vars\_of\_tm\_sctx}\ SL\ Vs \supset \{L \vdash \texttt{of}\ M\ T\} \supset \{SL \vdash \texttt{tm}\ M\}.\end{array}$$

Here the predicate constant $\texttt{vars\_of\_tm\_sctx} : \texttt{olist} \to \texttt{list tm} \to \texttt{prop}$ is used to collect the variables in the contexts for `tm`. It is defined through the following clauses:

$$\begin{array}{lcl}\texttt{vars\_of\_tm\_sctx}\ \texttt{nil}\ \texttt{nil} & \triangleq & \top \\ \nabla x.\texttt{vars\_of\_tm\_sctx}\ (\texttt{tm}\ x :: L)\ (x :: L') & \triangleq & \texttt{vars\_of\_tm\_sctx}\ L\ L'.\end{array}$$

Similarly, the predicate constant $\texttt{vars\_of\_sctx} : \texttt{olist} \to \texttt{list tm} \to \texttt{prop}$ is used to collect variables from the typing contexts represented by `sctx`. It is defined through the following clauses:

$$\begin{array}{lcl}\texttt{vars\_of\_sctx}\ \texttt{nil}\ \texttt{nil} & \triangleq & \top \\ \nabla x.\texttt{vars\_of\_sctx}\ (\texttt{of}\ x\ T :: L)\ (x :: L') & \triangleq & \texttt{vars\_of\_sctx}\ L\ L'.\end{array}$$

An important property of closed terms that will be used very often later is that they do not depend on any nominal constant, which is stated as the following theorem named `sclosed_tm_prune`:

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

Again, this is an example of the pruning properties we have discussed before. It is a special case of the following theorem

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

This theorem is proved easily by induction on $\{L \vdash \texttt{tm}\ (M\ x)\}$.

Formalizing the logical relations

We describe some auxiliary predicates that are necessary for formalizing the logical relations. We define the predicate $\texttt{le} : \texttt{nat} \to \texttt{nat} \to \texttt{prop}$ such that $\texttt{le}\ N_1\ N_2$ holds if and only if $N_1 \leq N_2$ through the following clause:

$$\texttt{le}\ N_1\ N_2 \quad \triangleq \quad \exists N.\{\texttt{add}\ N_1\ N\ N_2\}.$$

We also define a predicate $\texttt{step}*: \texttt{tm} \to \texttt{tm} \to \texttt{prop}$ as a short cut of n-step evaluation as follows:

$$\texttt{step}*\ M\ M' \quad \triangleq \quad \exists N.\{\texttt{nstep}\ N\ M\ M'\}.$$

We designate the predicate constants $\texttt{sim\_cps} : \texttt{ty} \to \texttt{nat} \to \texttt{tm} \to (\texttt{tm} \to \texttt{tm}) \to \texttt{tm} \to \texttt{prop}$ and $\texttt{equiv\_cps} : \texttt{ty} \to \texttt{nat} \to \texttt{tm} \to \texttt{tm} \to \texttt{prop}$ to respectively represent the simulation and equivalence relations such that $\texttt{sim\_cps}\ T\ I\ M\ K\ M'$ holds if and only if $M; K \sim_{T;I} M'$ holds and $\texttt{equiv\_cps}\ T\ I\ M\ M'$ holds if and only if $M \approx_{T;I} M'$ holds. Note that how the meta-level abstraction K is used to represent the input continuation for $\texttt{sim\_cps}$. The two predicates are defined as follows:

$$\begin{array}{l}
\texttt{sim\_cps}\ T\ I\ M\ K\ M' \quad \triangleq \\
\quad \forall J, V. \texttt{le}\ J\ I \supset \{\texttt{nstep}\ J\ M\ V\} \supset \{\texttt{val}\ V\} \supset \\
\quad\quad \exists N, V'. \texttt{step}*\ M'\ (K\ V') \wedge \{\texttt{add}\ J\ N\ I\} \wedge \texttt{equiv\_cps}\ T\ N\ V\ V' \\
\texttt{equiv\_cps}\ \texttt{tnat}\ I\ (\texttt{nat}\ N)\ (\texttt{nat}\ N) \quad \triangleq \quad \top \\
\texttt{equiv\_cps}\ \texttt{tunit}\ I\ \texttt{unit}\ \texttt{unit} \quad \triangleq \quad \top \\
\texttt{equiv\_cps}\ (\texttt{prod}\ T_1\ T_2)\ I\ (\texttt{pair}\ V_1\ V_2)\ (\texttt{pair}\ V_1'\ V_2') \quad \triangleq \\
\quad \texttt{equiv\_cps}\ T_1\ I\ V_1\ V_1' \wedge \texttt{equiv\_cps}\ T_2\ I\ V_2\ V_2' \wedge \\
\quad \{\texttt{tm}\ V_1\} \wedge \{\texttt{tm}\ V_2\} \wedge \{\texttt{tm}\ V_1'\} \wedge \{\texttt{tm}\ V_2'\} \\
\texttt{equiv\_cps}\ (\texttt{arr}\ T_1\ T_2)\ \texttt{z}\ (\texttt{fix} f \backslash x \backslash R\ f\ x)\ (\texttt{fix} f \backslash p \backslash R'\ f\ p) \quad \triangleq \\
\quad \{\texttt{tm}\ (\texttt{fix}\ R)\} \wedge \{\texttt{tm}\ (\texttt{fix}\ R')\} \\
\texttt{equiv\_cps}\ (\texttt{arr}\ T_1\ T_2)\ (\texttt{s}\ I)\ (\texttt{fix} f \backslash x \backslash R\ f\ x)\ (\texttt{fix} f \backslash p \backslash R'\ f\ p) \quad \triangleq \\
\quad \forall V_1, V_1', V_2, V_2', K. \\
\quad\quad \texttt{equiv\_cps}\ T_1\ I\ V_1\ V_1' \supset \texttt{equiv\_cps}\ (\texttt{arr}\ T_1\ T_2)\ I\ V_2\ V_2' \supset \\
\quad\quad \texttt{sim\_cps}\ T_2\ I\ (R\ V_2\ V_1)\ K\ (R'\ V_2'\ (\texttt{pair}\ (\texttt{fix}\ f \backslash K)\ V_1'))
\end{array}$$

These clauses are translated from the definition in Figure 5.6. Notice how the $\texttt{tm}$ predicate is used to enforce the closedness constraint and how meta-level β-redexes are used to model administrative β-redexes and substitutions. For example, the administrative β-redex $@\ K\ V'$ in Figure 5.6 is represented by the meta-level β-redex $(K\ V')$ in the definition of $\texttt{sim\_cps}$. Note also that $(\texttt{fix}\ f \backslash K)$ encodes the non-recursive function $\lambda x. (K\ x)$ which is a shorthand for $\textbf{fix}\ f\, x.(K\ x)$.

The definition of the `equiv_cps` relation uses itself negatively in the last clause. As such, we cannot view this as a fixed-point definition in $\mathcal{G}$. However, we use it only as a recursive definition, *i.e.*, as a definition based on which we can do unfolding or rewriting but not case analysis. In fact, this is the reason why we "build" the relation up over the natural numbers rather than mirroring directly the structure of the informal definition.

The property that `equiv_cps` is closed under decreasing indices is stated as follows

$$\forall T, I, J, V, V'.\{\texttt{is\_sty}\ T\} \supset \{\texttt{is\_nat}\ I\} \supset \\ \quad \texttt{equiv\_cps}\ T\ I\ V\ V' \supset \texttt{le}\ J\ I \supset \texttt{equiv\_cps}\ T\ J\ V\ V'.$$

where $\texttt{is\_nat} : \texttt{nat} \to \texttt{prop}$ is defined by the following λProlog program:

$\texttt{is\_nat}\ \texttt{z}.$

$\texttt{is\_nat}\ (\texttt{s}\ N)\ \texttt{:-}\ \texttt{is\_nat}\ N$

It is proved by a nested induction on $\{\texttt{is\_sty}\ T\}$ and $\{\texttt{is\_nat}\ I\}$ and unfolding of $\texttt{equiv\_cps}\ T\ I\ V\ V'$ in each case.

The compatibility lemmas for the simulation relation are formalized as the following theorems in Abella:

$$\forall I, M, M', K.\texttt{sim\_cps}\ \ \texttt{tnat}\ I\ M\ (x \setminus \texttt{let}\ (\texttt{pred}\ x)\ (v \setminus K\ v))\ M' \supset \\ \quad \texttt{sim\_cps}\ \ \texttt{tnat}\ I\ (\texttt{pred}\ M)\ K\ M'.$$

$$\forall I, M_1, M_2, M_2', M', K.\nabla x_1.\{\texttt{is\_nat}\ I\} \supset \texttt{sim\_cps}\ \texttt{tnat}\ I\ M_1\ M_2'\ M' \supset \\ \quad \texttt{sim\_cps}\ \texttt{tnat}\ I\ M_2\ (x_2 \setminus \texttt{let}\ (\texttt{plus}\ x_1\ x_2)\ (v \setminus K\ v))\ (M_2'\ x_1) \supset \\ \qquad \texttt{sim\_cps}\ \texttt{tnat}\ I\ (\texttt{plus}\ M_1\ M_2)\ K\ M'.$$

$$\forall T_1\ T_2\ K\ I\ M\ M'.\{\texttt{is\_nat}\ I\} \supset \{\texttt{is\_sty}\ (\texttt{prod}\ T_1\ T_2)\} \supset \\ \quad \texttt{sim\_cps}\ (\texttt{prod}\ T_1\ T_2)\ I\ M\ (x \setminus \texttt{let}\ (\texttt{fst}\ x)\ (v \setminus K\ v))\ M' \supset \\ \qquad \texttt{sim\_cps}\ T_1\ I\ (\texttt{fst}\ M)\ K\ M'.$$

$$\forall T_1, T_2, K, I, M, M'.\{\texttt{is\_nat}\ I\} \supset \{\texttt{is\_sty}\ (\texttt{prod}\ T_1\ T_2)\} \supset \\ \quad \texttt{sim\_cps}\ (\texttt{prod}\ T_1\ T_2)\ I\ M\ (x \setminus \texttt{let}\ (\texttt{snd}\ x)\ (v \setminus K\ v))\ M' \supset \\ \qquad \texttt{sim\_cps}\ T_2\ I\ (\texttt{snd}\ M)\ K\ M'.$$

$$\forall I, T_1, T_2, M_1, M_2, M_2', M', K.\nabla x_1.\{\texttt{is\_nat}\ I\} \supset \{\texttt{is\_sty}\ T_1\} \supset \{\texttt{is\_sty}\ T_2\} \supset \\ \quad \texttt{sim\_cps}\ T_1\ I\ M_1\ M_2'\ M' \supset \\ \quad \texttt{sim\_cps}\ T_2\ I\ M_2\ (x_2 \setminus \texttt{let}\ (\texttt{pair}\ x_1\ x_2)\ (v \setminus K\ v))\ (M_2'\ x_1) \supset \\ \qquad \texttt{sim\_cps}\ (\texttt{prod}\ T_1\ T_2)\ I\ (\texttt{pair}\ M_1\ M_2)\ K\ M'.$$

$\forall I, T, K', M_1, M_2, M_3, M_2', M_3', M'.\{\texttt{is\_nat}\ I\} \supset \{\texttt{is\_sty}\ T\} \supset$
$\quad \texttt{sim\_cps}\ T\ I\ M_2\ (x \setminus \texttt{app}\ (\texttt{fix}\ f \setminus K')\ x)\ (M_2'\ (\texttt{fix}\ f \setminus K')) \supset$
$\quad \texttt{sim\_cps}\ T\ I\ M_3\ (x \setminus \texttt{app}\ (\texttt{fix}\ f \setminus K')\ x)\ (M_3'\ (\texttt{fix}\ f \setminus K')) \supset$
$\quad \texttt{sim\_cps}\ \texttt{tnat}\ I\ M_1\ (x_1 \setminus \texttt{let}\ (\texttt{fix}\ f \setminus K')\ (k \setminus \texttt{ifz}\ x_1\ (M_2'\ k)\ (M_3'\ k)))\ M' \supset$
$\quad\quad \texttt{sim\_cps}\ T\ I\ (\texttt{ifz}\ M_1\ M_2\ M_3)\ K'\ M'.$
$\forall T_1, T, I, K', M_1, M', M_2, M_2'.\nabla x_1.\{\texttt{is\_nat}\ I\} \supset \{\texttt{is\_sty}\ (\texttt{arr}\ T_1\ T)\} \supset$
$\quad \texttt{sim\_cps}\ (\texttt{arr}\ T_1\ T)\ I\ M_1\ M_2'\ M' \supset$
$\quad \texttt{sim\_cps}\ T_1\ I\ M_2\ (x_2 \setminus \texttt{let}\ (\texttt{fix}\ (f \setminus K'))\ (k \setminus \texttt{let}\ (\texttt{pair}\ k\ x_2)\ (p \setminus \texttt{app}\ x_1\ p)))$
$\quad\quad\quad (M_2'\ x_1) \supset$
$\quad\quad \texttt{sim\_cps}\ T\ I\ (\texttt{app}\ M_1\ M_2)\ K'\ M'.$

These theorems are directly translated from Lemma 3. They are proved by following arguments similar to the informal ones. Therefore the closedness property of `equiv_cps` we proved previously is used here.

Representing substitutions

We treat substitutions as discussed in Section 3.2.4. We identify the type $\texttt{map} : \texttt{type} \to \texttt{type} \to \texttt{type}$ as the type of mappings and its constructor $\texttt{map} : A \to B \to \texttt{map}\ A\ B$. A substitution is then represented as a list of objects of the form $\texttt{map}\ x\ V$ where x is a variable and V is a closed value in which the variables are distinct. The substitutions for the source language is characterized by the predicate $\texttt{subst} : \texttt{tm} \to \texttt{prop}$ with the following definition such that $\texttt{subst}\ M$ holds if and only if M is a substitution:

$$\begin{array}{lcl} \texttt{subst}\ \texttt{nil} & \triangleq & \top \\ \nabla x.\texttt{subst}\ (\texttt{map}\ x\ V :: ML) & \triangleq & \texttt{subst}\ ML \wedge \{\texttt{val}\ V\} \wedge \{\texttt{tm}\ V\}. \end{array}$$

As described in Section 3.2.4, application of substitution is identified by the predicate symbol $\texttt{app\_subst} : \texttt{list}\ (\texttt{map}\ A\ A) \to B \to B \to \texttt{prop}$ defined through the following clauses such that $\texttt{app\_subst}\ S\ M\ M'$ holds exactly when M' is the result of applying the substitution S to M:

$$\begin{array}{lcl} \texttt{app\_subst}\ \texttt{nil}\ M\ M & \triangleq & \top \\ \nabla x.\texttt{app\_subst}\ ((\texttt{map}\ x\ V) :: S)\ (R\ x)\ M & \triangleq & \texttt{app\_subst}\ S\ (R\ V)\ M \end{array}$$

Given the above definitions, we can easily prove properties about substitution. The first important property that is useful in the semantics preservation proof is that substitution application distributes over term structure. The following are some examples:

$$\begin{array}{l}\forall ML, M, M'.\texttt{app\_subst}\ ML\ (\texttt{pred}\ M)\ M' \supset \\ \quad \exists M''.M' = \texttt{pred}\ M'' \wedge \texttt{app\_subst}\ ML\ M\ M''. \\ \forall ML, M_1, M_2, M'.\texttt{app\_subst}\ ML\ (\texttt{app}\ M_1\ M_2)\ M' \supset \\ \quad \exists M_1', M_2'.M' = \texttt{app}\ M_1'\ M_2' \wedge \texttt{app\_subst}\ ML\ M_1\ M_1' \wedge \texttt{app\_subst}\ ML\ M_2\ M_2'. \\ \forall R, M'.\texttt{app\_subst}\ ML\ (\texttt{fix}\ R)\ M' \supset \\ \quad \exists R'.M' = \texttt{fix}\ R' \wedge \nabla f, x.\texttt{app\_subst}\ ML\ (R\ f\ x)\ (R'\ f\ x).\end{array}$$

All such distribution lemmas have exactly the same proof. We do induction on the only assumption and analyze its cases. The base case is immediately concluded. In the inductive case we apply the inductive hypothesis on the smaller derivations for app_subst and use the results to conclude the case. Because of the λ-tree syntax approach, the above lemmas are just explicit statements of corresponding distribution properties for β-reductions. So it is no surprise they have such simple proofs.

Another important property that will be used very often is that substitution has no effect on closed terms. It is stated as follows:

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

This lemma shows how the closedness property defined at the implementation level coordinates with the reasoning. It is easily proved by induction on app_subst $ML\ M\ M'$ and using sclosed_tm_prune to discharge the dependence of M on the variables in the substitution ML.

The last important and useful property of substitution is that if all free variables of a term M are bound in a substitution ML, then the result of applying ML to M is a closed term. This is stated as the following theorem, called subst_result_closed_tm:

$$\begin{array}{l}\forall ML, L, M, M', Vs.\texttt{tm\_sctx}\ L \supset \{L \vdash \texttt{tm}\ M\} \supset \texttt{vars\_of\_tm\_sctx}\ L\ Vs \supset \\ \quad \texttt{subst}\ ML \supset \texttt{vars\_of\_subst}\ ML\ Vs \supset \texttt{app\_subst}\ ML\ M\ M' \ \supset \{\texttt{tm}\ M'\}.\end{array}$$

Here we use the predicate constant vars_of_subst : list (map tm tm) $\rightarrow$ list tm $\rightarrow$ prop such that vars_of_subst $ML\ Vs$ holds if and only Vs is the domain of the substitution ML. Its definition is similar to vars_of_sctx and is elided here. Again, the above theorem is easily proved by induction on $\texttt{app\_subst}\ \mathit{ML}\ M\ M'$ thanks to the λ-tree syntax representation of substitutions.

The equivalence relation on substitutions

The general form of the semantics preservation theorem requires a notion of equivalence between substitutions. We designate the constant $\texttt{subst\_equiv\_cps} : \texttt{olist} \to \texttt{nat} \to \texttt{list}\ (\texttt{map tm tm}) \to \texttt{list}\ (\texttt{map tm tm}) \to \texttt{prop}$ to represent this equivalence relation such that $\texttt{subst\_equiv\_cps}\ L\ I\ \mathit{ML}\ \mathit{ML}'$ holds if and only if $\theta \approx_{\Gamma;i} \theta'$ holds where L, I, ML and ML' are respectively encodings of Γ, i, θ and θ'. It is defined as follows:

$$\begin{array}{l}
\texttt{subst\_equiv\_cps}\ \texttt{nil}\ I\ \texttt{nil}\ \texttt{nil} \quad \triangleq \quad \top \\
\nabla x.\texttt{subst\_equiv\_cps}\ (\texttt{of}\ x\ T :: L)\ I\ (\texttt{map}\ x\ V :: \mathit{ML})\ (\texttt{map}\ x\ V' :: \mathit{ML}') \quad \triangleq \\
\quad \texttt{equiv\_cps}\ T\ I\ V\ V' \wedge \texttt{subst\_equiv\_cps}\ L\ I\ \mathit{ML}\ \mathit{ML}'.
\end{array}$$

Formalizing the semantics preservation theorems

To formalize the semantics preservation theorems, we will need another auxiliary predicate constant $\texttt{vars\_of\_cctx} : \texttt{olist} \to \texttt{list tm} \to \texttt{prop}$ for collecting variables in the transformation context defined as follows:

$$\begin{array}{lcl}
\texttt{vars\_of\_cctx}\ \texttt{nil}\ \texttt{nil} & \triangleq & \top \\
\nabla x.\texttt{vars\_of\_cctx}\ ((\Pi k.\texttt{cps}\ x\ k\ (k\ x)) :: L)\ (x :: L') & \triangleq & \texttt{vars\_of\_cctx}\ L\ L'.
\end{array}$$

Theorem 3 is now formalized as follows:

$$\begin{array}{l}
\forall \mathit{ML}, \mathit{ML}', \mathit{TL}, \mathit{CL}, \mathit{VL}, M, T, K, K', M', P, P', I. \\
\quad \{\texttt{is\_nat}\ I\} \supset \{\texttt{is\_sty}\ T\} \supset \\
\quad \texttt{sctx}\ \mathit{TL} \supset \texttt{vars\_of\_sctx}\ \mathit{TL}\ \mathit{VL} \supset \\
\quad \texttt{cctx}\ \mathit{CL} \supset \texttt{vars\_of\_cctx}\ \mathit{CL}\ \mathit{VL} \supset \\
\quad \texttt{subst}\ \mathit{ML} \supset \texttt{subst}\ \mathit{ML}' \supset \texttt{subst\_equiv\_cps}\ \mathit{TL}\ I\ \mathit{ML}\ \mathit{ML}' \supset \\
\quad \{\mathit{TL} \vdash \texttt{of}\ M\ T\} \supset \{\mathit{CL} \vdash \texttt{cps}\ M\ K\ M'\} \supset \\
\quad \texttt{app\_subst}\ \mathit{ML}\ M\ P \supset \texttt{app\_subst}\ \mathit{ML}'\ M'\ P' \supset \\
\quad (\nabla x.\texttt{app\_subst}\ \mathit{ML}'\ (K\ x)\ (K'\ x)) \supset \\
\qquad \texttt{sim\_cps}\ T\ I\ P\ K'\ P'.
\end{array}$$

We prove this theorem by induction on $\{CL \vdash \texttt{cps}\ M\ K\ M'\}$, the derivation of the CPS transformation. The proof closely follows the informal proof for Theorem 3. The proof for base cases is obvious. In the remaining cases, other than when M is a let expression or a fixed-point, the proof follows a set pattern: we first apply the distribution lemmas of `app_subst` to M, then apply the inductive hypotheses to sub-expressions of M to get the simulation relation between sub-expressions, and finally conclude by applying the compatibility lemmas of `sim_cps`. When M is a let expression or a fixed-point, we extend the substitutions with equivalent values for new variables introduced by recursion for applying the inductive hypothesis on expressions under the binding operators of M and use the results to conclude the case.

As we have said, many aspects of bindings that are implicit in implementation must be made explicit in the correctness proof, which may blur the essential contents of the proof. Thanks to the uses of the λ-tree syntax approach, reasoning about bindings is greatly simplified, resulting in the closed correspondence between our formal proof and the informal one. The uses and benefits of the λ-tree syntax approach in our formal proof are summarized as follows. First, because the treatments of bindings at the specification level are transparently reflected into the reasoning level via the two-level logic approach, properties of administrative β-redexes are reflected into properties of meta-level β-redexes in Abella and are immediately available in reasoning. As a result, minimum effort is needed to reason about administrative β-redexes. Second, the "boilerplate" properties of substitution and closed terms are captured as lemmas of `app_subst` and `tm` and used to simplify the reasoning about them. For example, when M is a natural number, we apply the lemma that `app_subst` has no effect on closed terms to show that it does not affect M. As another example, when M is a fixed-point, to prove the simulation relation $\texttt{sim\_cps}\ T\ I\ P\ K'\ P'$ we need to show P and P' are closed terms. To show the former, we first apply `sof_to_tm` to $\{TL \vdash \texttt{of}\ M\ T\}$ to show that M is a well-formed term whose variables are contained in the domain of ML and then apply `subst_result_closed_tm` to $\texttt{app\_subst}\ ML\ M\ P$ to show that $\{\texttt{tm}\ P\}$ holds, *i.e.*, P is closed. To show the latter, we first apply the type preservation lemma to $\{TL \vdash \texttt{of}\ M\ T\}$ and $\{CL \vdash \texttt{cps}\ M\ K\ M'\}$ to show that M' is well-typed, at this point we follow the same process for proving P is closed to show that $\{\texttt{tm}\ P'\}$ holds.

Finally, we formalize Corollary 5 as the following theorem in Abella which is easily proved by applying the above theorem:

$$\forall M, K, M', V.\{\texttt{of}\ M\ \texttt{tnat}\} \supset \{\texttt{cps}\ M\ K\ M'\} \supset \\ \{\texttt{eval}\ M\ V\} \supset \texttt{step}*\ M'\ (K\ V).$$

Letting $K = x \setminus x$, it is easy to prove the following formalized version of Corollary 6:

$$\forall M, M', V.\{\texttt{of}\ M\ \texttt{tnat}\} \supset \{\texttt{cps}\ M\ (x \setminus x)\ M'\} \supset \\ \{\texttt{eval}\ M\ V\} \supset \{\texttt{eval}\ M'\ V\}.$$

Chapter 6

The Closure Conversion Transformation

An aspect that complicates the compilation of functional programs is the presence of nested functions: since such functions can use non-local variables, their invocation must be parameterized by the context in which they appear. The closure conversion transformation makes the kind of parameterization that is needed explicit. It does so by transforming every function into a *closure* which consists of a closed function called its code part and an environment part. For every function, closure conversion identifies its free variables and constructs the environment that contains bindings for these free variables in the context of the function. It also abstracts the original function over an extra environment parameter and replaces the occurrences of free variables in the function with appropriate references to the environment parameter to form the closed function. The closed function and the constructed environment constitute the closure for the original function. Because functions become closed after closure conversion, they can be freely moved around, enabling other transformations. One such transformation is code hoisting, which eliminates nested functions.

The closure conversion transformation we develop in this chapter is the second phase of the compiler described in Section 4.6. Following the approach described in Section 4.2, we can describe closure conversion in a rule-based and relational style and prove its semantics preservation property based on logical relations. To get a verified implementation of closure conversion, we follow the approach given in Section 4.3 to implement the rule-based relational descriptions of closure conversion as a λProlog program and prove the correctness of the implementation in Abella. Similar to the CPS transformation described in Chapter 5, we exploit the λ-tree syntax approach to simplify both the implementation and verification.

A major difficulty in formalizing closure conversion is to formally describe the computation of free variables. This involves non-trivial analysis of the binding structure of functional terms. Moreover, in formally proving the correctness of the closure conversion, we need to prove non-trivial properties about such analysis. If not handled properly, the proof of such properties can significantly complicate the main verification effort. We shall see that by using the λ-tree syntax approach we can implement the identification of free variables as a λProlog program and give concise proofs to the desired properties of this program in Abella.

We describe the implementation and verification of closure conversion in the rest of this chapter. Again, our discussion focuses on showing how the λ-tree syntax approach can be effectively exploited in both the implementation and verification of closure conversion. Because the structure and contents of this chapter share a lot of similarity with that in Chapter 5, we shall omit discussion of the similar contents and focus on the distinct aspects of the implementation and verification of closure conversion. We start by giving an overview of closure conversion in Section 6.1. We then present the source and target languages of the transformation and the rule-based relational descriptions of the transformation in Section 6.2. In Section 6.3 we present the implementation of the rule-based description as an λProlog program for closure conversion. In Section 6.4 we describe the informal verification of the closure conversion transformation. In Section 6.5 we present the formalization of the verification.

6.1 An Overview of the Transformation

The formulation of the closure conversion transformation that we use here is based on [85]. To illustrate this transformation concretely, when it is applied to the following pseudo OCaml code segment

$\texttt{let}\ x = 2\ \texttt{in}\ \texttt{let}\ y = 3\ \texttt{in}$

$\quad \texttt{fun}\ z \to z + x + y$

it will yield

$\texttt{let}\ x = 2\ \texttt{in}\ \texttt{let}\ y = 3\ \texttt{in}$

$\quad \langle(\texttt{fun}\ z\ e \to z + e.1 + e.2), (x, y)\rangle$

We write $\langle F, E\rangle$ here to represent a closure whose code part is F and environment part is E, and $e.i$ to represent the i-th projection applied to an "environment parameter" e. A closure can be thought as a partial application of the code part to its environment part, which is suspended until the actual argument of the function is provided at run-time. This transformation makes the function part independent of the context in which it appears, thereby allowing it to be extracted out to the top-level of the program.

Closure conversion is performed recursively on the structure of terms. In the general case when such terms contain nested functions, closure conversion needs to identify free variables and transform occurrences of free variables under the nested binders as well. As an illustration, consider the following pseudo OCaml code:

$\texttt{let}\ x = 3\ \texttt{in}$

$\quad \texttt{fun}\ y \to \texttt{fun}\ z \to x + y + z.$

Closure conversion transforms it into the following code:

$\texttt{let}\ x = 3\ \texttt{in}$

$\quad \langle(\texttt{fun}\ y\ e_1 \to \langle(\texttt{fun}\ z\ e_2 \to e_2.1 + e_2.2 + z), (e_1.1, y)\rangle), (x)\rangle.$

In the first phase, it transforms the outer function. Specifically, it determines that the only free variable at that level is x. It then constructs the environment (x) for the function and parameterizes that function with the environment argument e_1. It must then transform the body of the outer function, which is exactly the inner function. Observe, however, that when it effects this transformation, the variable x that appears free in it must not be referred to directly but as $e_1.1$. The consequence of this observation can be seen in the code that is shown for the transformed version of the inner function.

From the previous example, it is obvious that the essential work of closure conversion is to compute free variables and to replace occurrences of free variables with references to environment parameters under nested binders. We shall discuss how to precisely describe such manipulation and analysis of bindings and how to prove properties about them in the rest of the chapter.

6.2 A Rule-Based Description of the Transformation

We give a rule-based relational description of the closure conversion transformation in this section. We first describe the source and target languages of the transformation, including their typing rules, and then present the transformation rules.

6.2.1 The source and target languages

The source language of closure conversion is the same as the target language of the CPS transformation. Its syntax and typing rules are already given in Figures 5.1 and 5.2. The syntax of the target language is depicted in Figure 6.1. One important difference between the target language and the source language is that the former includes constructs for dealing with closures. Compared to the source language, the target language includes the expressions $\langle M_1, M_2\rangle$ and ($\mathbf{open}\ \langle x_f, x_e\rangle = M_1\ \mathbf{in}\ M_2$) representing the formation and application of closures. Another important difference is that the target language does not have an explicit fixed point constructor. Instead, recursion is realized by parameterizing the function part of a closure with a function component; this treatment should become clear from the rules for typing closures and the transformation that we present below. The usual forms of abstraction and application are included in the target language to simplify the presentation of the transformation. Note that the usual function type is reserved for closures. The abstractions in the target language are given the type $T_1 \Rightarrow T_2$ in the target language. We abbreviate $(M_1, \ldots (M_n, ()))$ by $(M_1, \ldots, M_n)$ and $\mathbf{fst}\ (\mathbf{snd}\ (\ldots (\mathbf{snd}\ M)))$ where $\mathbf{snd}$ is applied $i-1$ times for $i \geq 1$ by $\pi_i(M)$.

Typing judgments for the target language are written as $\Gamma \vdash M : T$, where Γ is a list of type assignments for variables. Note that here we overload the syntax of the typing judgments in Section 5.2.1. The exact meaning of such a judgment is inferred from the context if it is not explained explicitly. Many of the program constructs in the source language are present also in the target language and the rules in Figure 5.2 for typing

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

Figure 6.1: The Syntax of The Target Language of Closure Conversion

them carry over also to the target language. The only exceptions are those for typing abstractions and applications. In addition, we need rules for introducing and eliminating closures that are present only in the target language. Closures and abstractions both have a "function" type but the specific way in which they figure in the target language is different. We use types of two different forms, $T_1 \to T_2$ and $T_1 \Rightarrow T_2$, to distinguish between the roles of these expressions. The typing rules, whose spirit is borrowed from the presentation in [46], should help in explaining the intended roles of the different function-values expressions.

The type used for abstractions has the form $T_1 \Rightarrow T_2$. The rules for typing them and the associated applications are shown below:

$$\frac{\Gamma, x : T_1 \vdash M : T_1}{\Gamma \vdash \lambda x.\, M : T_1 \Rightarrow T_2}\ x \notin dom(\Gamma)\ \text{cof-abs} \qquad \frac{\Gamma \vdash M_1 : T_1 \Rightarrow T \quad \Gamma \vdash M_2 : T_1}{\Gamma \vdash M_1\ M_2 : T}\ \text{cof-app}$$

A closure differs from an abstraction in the sense that it represents a partially applied function: it is packaged with an environment that must be coupled with the "real" argument to complete the application. We use a type of the form $T_1 \to T_2$ to encode this kind of partial application. The specific rules for introducing and eliminating closures are shown below:

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

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

In cof-open, x_f, x_e must be names that are new to Γ. This rule also uses a "type" l that is to be interpreted as a new type constant, different from $\mathbb{N}$ and $()$ and any other type constant used in the typing derivation. Intuitively, the rule cof-clos states that for a closure $\langle M_1, M_2 \rangle$ to have the type $T_1 \rightarrow T_2$ there must exists some type T_e for its environment part and M_1 must be a closed function of type $((T_1 \rightarrow T_2) \times T_1 \times T_e) \Rightarrow T_2$; the requirement that M_1 is closed follows from the fact that it must be typable in an empty context. The type of M_1 indicates how recursion is realized in the target language: the function part of a closure takes the closure itself as an input besides the actual and environment arguments. The rule cof-open conforms to how closure applications work. In an expression of the form **open** $\langle x_f, x_e \rangle = M_1$ **in** M_2, M_1 is must evaluate to a closure whose function and environment parts are extracted into x_f and x_e and then used in M_2 through the occurrences of these variables in that expression. Another interesting aspect to note about this rule is that it enforces an opaqueness criterion on the environment component that is extracted. This follows from the use of the fresh type constant l to represent the type of the environment.

6.2.2 The transformation rules

In the general case, we must transform terms under mappings for their free variables: for a function term, this mapping represents the replacement of the free variables by projections from the environment variable for which a new abstraction will be introduced into the term. Accordingly, we specify the transformation as a 3-place relation written as $\rho \triangleright M \rightsquigarrow_{cc} M'$, where M and M' are source and target language terms and ρ is a mapping from source language variables to target language terms. We write $(\rho, x \mapsto M)$ to denote the extension of ρ with a mapping for x and $(x \mapsto M) \in \rho$ to mean that ρ contains a mapping of x to M. Figure 6.2 defines the $\rho \triangleright M \rightsquigarrow_{cc} M'$ relation in a rule-based fashion; these rules use the auxiliary relation $\rho \triangleright (x_1, \ldots, x_n) \rightsquigarrow_e M_e$ that determines an environment corresponding to a tuple of variables. The cc-let and cc-fix rules have a proviso: the bound variables, x and f, x respectively, should have been renamed to avoid clashes with the domain of ρ. Most of the rules have an obvious structure. We comment only on the ones for transforming fixed point expressions and applications. The former translates into a closure. The function part of the closure is obtained by transforming the body of the abstraction, but under a new mapping ρ' that maps

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

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

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

$$\frac{\rho \triangleright M \rightsquigarrow_{cc} M' \quad \rho \triangleright M_1 \rightsquigarrow_{cc} M_1' \quad \rho \triangleright M_2 \rightsquigarrow_{cc} M_2'}{\rho \triangleright \textbf{if}\ M\ \textbf{then}\ M_1\ \textbf{else}\ M_2 \rightsquigarrow_{cc} \textbf{if}\ M'\ \textbf{then}\ M_1'\ \textbf{else}\ M_2'}\ \text{cc-ifz}$$

$$\frac{}{\rho \triangleright () \rightsquigarrow_{cc} ()}\ \text{cc-unit} \qquad \frac{\rho \triangleright M_1 \rightsquigarrow_{cc} M_1' \quad \rho \triangleright M_2 \rightsquigarrow_{cc} M_2'}{\rho \triangleright (M_1, M_2) \rightsquigarrow_{cc} (M_1', M_2')}\ \text{cc-pair}$$

$$\frac{\rho \triangleright M \rightsquigarrow_{cc} M'}{\rho \triangleright \textbf{fst}\ M \rightsquigarrow_{cc} \textbf{fst}\ M'}\ \text{cc-fst} \qquad \frac{\rho \triangleright M \rightsquigarrow_{cc} M'}{\rho \triangleright \textbf{snd}\ M \rightsquigarrow_{cc} \textbf{snd}\ M'}\ \text{cc-snd}$$

$$\frac{\rho \triangleright M_1 \rightsquigarrow_{cc} M_1' \quad \rho, x \mapsto y \triangleright M_2 \rightsquigarrow_{cc} M_2'}{\rho \triangleright \textbf{let}\ x = M_1\ \textbf{in}\ M_2 \rightsquigarrow_{cc} \textbf{let}\ y = M_1'\ \textbf{in}\ M_2'}\ (y \text{ is fresh})\ \text{cc-let}$$

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

$$\frac{(x_1, \ldots, x_n) \supseteq \textbf{fvars}(\textbf{fix}\ f\, x.M) \quad \rho \triangleright (x_1, \ldots, x_n) \rightsquigarrow_e M_e \quad \rho' \triangleright M \rightsquigarrow_{cc} M'}{\begin{array}{l}\rho \triangleright \textbf{fix}\ f\, x.M \rightsquigarrow_{cc} \\ \qquad \langle \lambda p.\, \textbf{let}\ g = \pi_1(p)\ \textbf{in}\ \textbf{let}\ y = \pi_2(p)\ \textbf{in}\ \textbf{let}\ x_e = \pi_3(p)\ \textbf{in}\ M', M_e \rangle \\ \qquad (\text{where } \rho' = (x \mapsto y, f \mapsto g, x_1 \mapsto \pi_1(x_e), \ldots, x_n \mapsto \pi_n(x_e)) \\ \qquad \text{and } p, g, y, \text{ and } x_e \text{ are fresh variables})\end{array}}\ \text{cc-fix}$$

Figure 6.2: The Rules for Closure Conversion

$(x_1, \ldots, x_n)$ to appropriate projections from the environment parameter; the expression $(x_1, \ldots, x_n) \supseteq \textbf{fvars}(\textbf{fix}\ f\, x.M)$ means that all the free variables of $(\textbf{fix}\ f\, x.M)$ appear in the tuple. The environment part of the closure correspondingly contain mappings for the variables in the tuple that are determined by the enclosing context. Note also that the parameter for the function part of the closure is expected to be a triple, the first item of which corresponds to the function being defined recursively in the source language expression. The transformation of a source language application makes clear how this structure is used to realize recursion: the constructed closure application has the effect of feeding the closure to its function part as the first component of its argument.

6.3 Implementing the Transformation in λProlog

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

6.3.1 Encoding the language

The source language of closure conversion is the target language of the CPS transformation. The encoding of its syntax and typing relations are already given in Section 5.3.1. We are left with the encoding of the target language.

We first consider the encoding of types in the target language. We use the λProlog type `ty`, the same type we used for encoding the source language, for representing the types of the target language. The constants for encoding the natural number, unit and pair types are also the same as those for encoding the source language. We represent the new arrow type constructor $\Rightarrow$ by `arr'`.

We use the λProlog type `tm'` for encodings of target language terms. To represent the constructs the target language shares with the source language, we will use "primed" versions of the λProlog constants seen in Section 5.3.1; *e.g.*, `unit'` of type `tm'` will represent the unit constructor. Of course, there will be no constructor corresponding to `fix`. We will also need the following additional constructors:

$$\texttt{abs'} : (\texttt{tm'} \to \texttt{tm'}) \to \texttt{tm'}$$
$$\texttt{clos'} : \texttt{tm'} \to \texttt{tm'} \to \texttt{tm'}$$
$$\texttt{open'} : \texttt{tm'} \to (\texttt{tm'} \to \texttt{tm'} \to \texttt{tm'}) \to \texttt{tm'}$$

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

We use the predicates `of'` : `tm'` $\to$ `ty` $\to$ `prop` to encode typing in the target language. The clauses defining this predicate are routine and we show only the encoding of typing rules for closures and closure applications. The following clauses encode these rules.

$$\texttt{of'}\ (\texttt{clos'}\ F\ E)\ (\texttt{arr}\ T_1\ T_2) \quad \texttt{:-}$$

$\quad$ `of'` F (`arr'` (`prod` (`arr` T_1 T_2) (`prod` T_1 TL)) T_2) , `of'` E TL.
`of'`(`open'` M R) T `:-`
$\quad$ `of'` M (`arr` T_1 T_2) ,
$\quad$ $\Pi f, e, l.$`of'` f (`arr'` (`prod` (`arr` T_1 T_2) (`prod` T_1 l)) T_2) $\Rightarrow$
$\qquad\quad$ `of'` e l $\Rightarrow$ `of'` (R f e) T.

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

We should note that the clause that encodes the typing of closures is not an entirely accurate rendition of the typing rule presented in Section 6.2.1. In particular, it does not capture the fact that the function part must be typed in an empty typing environment. However, the fact that it *is* typed in an empty typing environment can be encoded in a suitable theorem about the implementation of the transformation and this theorem can be proved using Abella.

6.3.2 Specifying the closure conversion transformation

We first introduce an auxiliary predicate `combine` : `list` $A \rightarrow$ `list` $A \rightarrow$ `list` $A \rightarrow$ `o` that holds between three lists when the last is composed of the elements of the first two. It is defined as follows where `memb` is the membership relation defined in Section 3.2.2:

`combine` `nil` L L.
`combine` (X `::` L_1) L_2 L `:-` `memb` X L_2 , `combine` L_1 L_2 L.
`combine` (X `::` L_1) L_2 (X `::` L) `:-` `combine` L_1 L_2 L.

Observe that the definition of `combine` does not enforce a possible additional requirement that the third argument does not contain repeated elements; indeed, such a requirement, which involves a negative constraint, is impossible to encode using the logical apparatus of HH$^\omega$. However, if the first two arguments to a `combine` goal are specific lists of distinct elements and the third is an unspecified one, then at least one of the values for the last argument that satisfies the goal is a list in which each of the elements is distinct. Indeed, this would be the first "solution" that will be found for the goal by the logic programming style interpretation embedded in λProlog.

The crux in formalizing the definition of closure conversion is capturing the content of the cc-fix rule. A key part of this rule is identifying the free variables in a given source language term. We realize the requirement by defining a predicate constant `fvars` : `tm` $\to$ `list tm` $\to$ `list tm` $\to$ `o` such that if L_1 is a list that includes all the free variables of M and (`fvars` $M\ L_1\ L_2$) holds, then L_2 is another list that contains exactly the free variables of M. We show the definition of this predicate as follows:

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

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

$\texttt{fvars}\ (\texttt{nat}\ N)\ \mathit{Vs}\ \texttt{nil}.$

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

$\quad \texttt{fvars}\ M_1\ \mathit{Vs}\ \mathit{FVs}_1, \texttt{fvars}\ M_2\ \mathit{Vs}\ \mathit{FVs}_2, \texttt{combine}\ \mathit{FVs}_1\ \mathit{FVs}_2\ \mathit{FVs}.$

$\texttt{fvars}\ (\texttt{ifz}\ M\ M_1\ M_2)\ \mathit{Vs}\ \mathit{FVs}'\ \ \texttt{:-}$

$\quad \texttt{fvars}\ M\ \mathit{Vs}\ \mathit{FVs}, \texttt{fvars}\ M_1\ \mathit{Vs}\ \mathit{FVs}_1, \texttt{fvars}\ M_2\ \mathit{Vs}\ \mathit{FVs}_2,$

$\quad \texttt{combine}\ \mathit{FVs}\ \mathit{FVs}_1\ \mathit{FVs}'_1, \texttt{combine}\ \mathit{FVs}'_1\ \mathit{FVs}_2\ \mathit{FVs}'.$

$\texttt{fvars}\ \texttt{unit}\ \_\ \texttt{nil}.$

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

$\quad \texttt{fvars}\ M_1\ \mathit{Vs}\ \mathit{FVs}_1, \texttt{fvars}\ M_2\ \mathit{Vs}\ \mathit{FVs}_2, \texttt{combine}\ \mathit{FVs}_1\ \mathit{FVs}_2\ \mathit{FVs}.$

$\texttt{fvars}\ (\texttt{fst}\ M)\ \mathit{Vs}\ \mathit{FVs}\ \ \texttt{:-}\ \ \texttt{fvars}\ M\ \mathit{Vs}\ \mathit{FVs}.$

$\texttt{fvars}\ (\texttt{snd}\ M)\ \mathit{Vs}\ \mathit{FVs}\ \ \texttt{:-}\ \ \texttt{fvars}\ M\ \mathit{Vs}\ \mathit{FVs}.$

$\texttt{fvars}\ (\texttt{let}\ M\ R)\ \mathit{Vs}\ \mathit{FVs}\ \ \texttt{:-}\ \ \texttt{fvars}\ M\ \mathit{Vs}\ \mathit{FVs}_1,$

$\quad (\Pi x.\texttt{notfree}\ x \Rightarrow \texttt{fvars}\ (R\ x)\ \mathit{Vs}\ \mathit{FVs}_2), \texttt{combine}\ \mathit{FVs}_1\ \mathit{FVs}_2\ \mathit{FVs}.$

$\texttt{fvars}\ (\texttt{fix}\ R)\ \mathit{Vs}\ \mathit{FVs}\ \ \texttt{:-}$

$\quad \Pi f, x.\texttt{notfree}\ f \Rightarrow \texttt{notfree}\ x \Rightarrow \texttt{fvars}\ (R\ f\ x)\ \mathit{Vs}\ \mathit{FVs}.$

The essence of the definition of `fvars` is in the treatment of binding constructs. Viewed operationally, the body of such a construct is descended into after instantiating the binder with a new variable marked `notfree` : `tm` $\to$ `o`. Thus, the variables that are marked in this way correspond to exactly those that are explicitly bound in the term. When the first argument of `fvars` is such a variable it will not be collected as a free variable as indicated by the first clause. Only those that are not marked `notfree` are collected through the second clause and combination of the results of recursive calls to `fvars` on subterms. It is important also to note that the specification of `fvars` has a completely logical structure; this fact can be exploited during verification.

The cc-fix rule requires us to construct an environment representing the mappings for the variables found by `fvars`. The predicate `mapenv` : `list tm` $\rightarrow$ `list` (`map tm tm'`) $\rightarrow$ `tm'` $\rightarrow$ `o`. specified by the following clauses provides this functionality.

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

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

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

We can now specify the closure conversion transformation. We provide clauses below that define the predicate `cc` : `list` (`map tm tm'`) $\rightarrow$ `list tm` $\rightarrow$ `tm` $\rightarrow$ `tm'` $\rightarrow$ `o` such that (`cc` $\mathit{Map}\ \mathit{Vs}\ M\ M'$) holds if M' is a transformed version of M under the mapping Map for the variables in Vs; we assume that Vs contains all the free variables of M.

$$\begin{array}{l}
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{nat}\ N)\ (\texttt{nat'}\ N).\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ X\ M\ \ \texttt{:-}\ \ \texttt{member}\ (\texttt{map}\ X\ M)\ \mathit{Map}.\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{pred}\ M)\ (\texttt{pred'}\ M')\ \ \texttt{:-}\ \ \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M\ M'.\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{plus}\ M_1\ M_2)\ (\texttt{plus'}\ M_1'\ M_2')\ \ \texttt{:-}\\
\quad \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M_1\ M_1'\,, \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M_2\ M_2'.\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{ifz}\ M\ M_1\ M_2)\ (\texttt{ifz'}\ M'\ M_1'\ M_2')\ \ \texttt{:-}\\
\quad \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M\ M'\,, \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M_1\ M_1'\,, \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M_2\ M_2'.\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ \texttt{unit}\ \texttt{unit'}.\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{pair}\ M_1\ M_2)\ (\texttt{pair'}\ M_1'\ M_2')\ \ \texttt{:-}\\
\quad \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M_1\ M_1'\,, \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M_2\ M_2'.\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{fst}\ M)\ (\texttt{fst'}\ M')\ \ \texttt{:-}\ \ \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M\ M'.\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{snd}\ M)\ (\texttt{snd'}\ M')\ \ \texttt{:-}\ \ \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M\ M'.\\
\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ (\texttt{let}\ M\ R)\ (\texttt{let'}\ M'\ R')\ \ \texttt{:-}\ \ \texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M\ M'\,,
\end{array}$$

```
    $\Pi x, y.$cc ((map $x$ $y$) :: $Map$) ($x$ :: $Vs$) ($R$ $x$) ($R'$ $y$).
  cc $Map$ $Vs$ (fix $R$)
    (clos' (abs' ($p$ \ let' (fst' $p$) ($g$ \
      let' (fst' (snd' $p$)) ($y$ \
      let' (snd' (snd' $p$)) ($e$ \ $R'$ $g$ $y$ $e$))))) $E$)  :-
    fvars (fix $R$) $Vs$ $FVs$ , mapenv $FVs$ $Map$ $E$ , mapvar $FVs$ $NMap$ ,
    $\Pi f, x, g, y, e.$
      cc ((map $x$ $y$) :: (map $f$ $g$) :: ($NMap$ $e$)) ($x$ :: $f$ :: $FVs$) ($R$ $f$ $x$) ($R'$ $g$ $y$ $e$).
  cc $Map$ $Vs$ (app' $M_1$ $M_2$)
    (let $M_1'$ ($g$ \ open' $g$ ($f$ \ $e$ \ app' $f$ (pair' $g$ (pair' $M_2'$ $e$)))))  :-
      cc $Map$ $Vs$ $M_1$ $M_1'$ , cc $Map$ $Vs$ $M_2$ $M_2'$.
```

These clauses correspond very closely to the rules in Figure 6.2. Note especially the clause for transforming an expression of the form (`fix` R) that encodes the content of the cc-fix rule. In the body of this clause, `fvars` is used to identify the free variables of the expression, and `mapenv` and `mapvar` are used to create the reified environment and the new mapping. In both this clause and in the one for transforming a `let` expression, the λ-tree representation, universal goals and (meta-language) applications are used to encode freshness and renaming requirements related to bound variables in a concise and logically precise way.

6.4 Informal Verification of the Transformation

We informally describe the verification of closure conversion in this section based on the ideas presented in Section 4.2. Similar to the informal verification of the CPS transformation discussed in Section 5.4, we first discuss the type preservation and then the semantics preservation of closure conversion. The proofs we present in this section is based on the proofs in [85]. In particular, we have adapted the proof of semantics preservation in [85] which is based on regular logical relations to be based on step-indexing logical relations.

6.4.1 Type preservation of the transformation

In proving that closure conversion preserves typing, the most difficult step is to ensure that this property holds at the points of the transformation where we cross a function boundary. For this we need the following strengthening lemma:

Lemma 4. *If* $\Gamma \vdash M : T$*,* $\{x_1, \ldots, x_n\} \supseteq \mathbf{fvars}(M)$ *and* $x_i : T_i \in \Gamma$ *for* $1 \leq i \leq n$*, then* $x_1 : T_1, \ldots, x_n : T_n \vdash M : T$*.*

The type preservation property is then stated as follows:

Theorem 4. *Let* ρ *be the mapping* $(x_1 \mapsto M_1, \ldots, x_n \mapsto M_n)$ *from source variables to target terms,* Γ *be the typing context* $(x_1 : T_1, \ldots, x_n : T_n)$ *in the source language,* Γ' *be a typing context in the target language such that* $\Gamma' \vdash M_i : T_i$ *for* $1 \leq i \leq n$*. If* $\Gamma \vdash M : T$ *and* $\rho \triangleright M \rightsquigarrow_{cc} M'$*, then* $\Gamma' \vdash M' : T$*.*

This theorem states that if ρ is a type preserving mapping and M is transformed into M' under ρ, then the type of M' is preserved. It is proved by induction on the derivation of $\rho \triangleright M \rightsquigarrow_{cc} M'$ and analyzing its last rule. When the last rule is cc-nat, cc-var or cc-unit, the proof is obvious. The rest of the cases, except when the last rule is cc-fix, are proved by following a set pattern: We analyze the derivation of $\Gamma \vdash M : T$, apply the inductive hypothesis on the subterms of M and conclude from the results. When the last rule is cc-fix, we apply Lemma 4 to get a typing judgment that can be used for applying the inductive hypothesis. The rest of the proof follows the set pattern.

Given Theorem 4, it is easy to show the following type preservation property for closed terms:

Corollary 7. *If* $\emptyset \vdash M : T$ *and* $\emptyset \triangleright M \rightsquigarrow_{cc} M'$*, then* $\emptyset \vdash M' : T$*.*

6.4.2 Semantics preservation of the transformation

We give an informal description of semantics preservation for closure conversion in this section. We first describe the operational semantics of the source and target languages of the transformation, then the logical relations for denoting equivalence between the source and target programs and their properties, and finally the semantics preservation theorem and its proof.

Operational semantics of the source and target languages

The operational semantics of the source language is already given in Section 5.4.2. The operational semantics of the target language is based on a left to right, call-by-value evaluation strategy and is presented in small-step form. We overload the syntax of evaluation judgments described in Section 5.4.2 for representing the evaluation judgments for the target language. That is, $M \hookrightarrow_1 M'$ if and only if M evaluates to M' in one-step, $M \hookrightarrow_n M'$ if M evaluates to M' in n steps, $M \hookrightarrow_* M'$ if there exists some n such that $M \hookrightarrow_n M'$, and $M \hookrightarrow V$ if $M \hookrightarrow_* V$ and V is a value. The designations of such judgments are inferred from the context if they are not given explicitly. The one-step evaluation rules for the target language are mostly the same as that for the source language. The only evaluation rules that may be non-obvious are the ones for closures and closure applications. They are the following:

$$\frac{M_1 \hookrightarrow_1 M_1'}{\langle M_1, M_2\rangle \hookrightarrow_1 \langle M_1', M_2\rangle} \qquad \frac{M_2 \hookrightarrow_1 M_2'}{\langle V_1, M_2\rangle \hookrightarrow_1 \langle V_1, M_2'\rangle}$$

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

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

Logical relations and their properties

Following the ideas in Section 4.2, we use step-indexing logical relations to characterize the semantics preservation property of closure conversion. Specifically, we define the mutually recursive simulation relation $\sim$ between closed source and target terms and equivalence relation $\approx$ between closed source and target values, each indexed by a type and a step measure, in Figure 6.3. By definition, the simulation relation $M \sim_{T;i} M'$ holds if and only if M simulates M' within at least i steps of evaluation at the type T. The equivalence relation $V \approx_{T;i} V'$ holds if and only if the values V and V' cannot be distinguished from each other (*i.e.*, considered as equivalent) in any context within at least i steps of evaluation at the type T. The rules in Figure 6.3 except the last one have obvious meaning. The last rule states that a function ($\textbf{fix}\ f\,x.M$) is equivalent to a closure $\langle V', V_e\rangle$ at type $T_1 \to T_2$ for i steps if and only if given any step j smaller than

i and any argument V_1, V_1', V_2, V_2' such that V_1 and V_1' are equivalent at T_1 and V_2 and V_2 are equivalent at $T_1 \to T_2$ for j steps, the application of $(\mathbf{fix}\ f\,x.M)$ to V_1 and V_2 simulates the application of $\langle V', V_e\rangle$ to V_1' and V_2'. Note that the definition of $\approx$ in the last rule uses $\approx$ negatively at the same type. However, it is still a well-defined notion because the index decreases.

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

Figure 6.3: The Logical Relations for Verifying Closure Conversion

A property we will need later is that $\approx$ is closed under decreasing indexes. It is stated as the following lemma:

Lemma 5. *If $V \approx_{T;i} V'$ holds, then for any j such that $j \le i$, $V \approx_{T;j} V'$ holds.*

We prove this lemma by a (nested) induction first on the types and then on the step indexes of $\approx$. The proof itself is obvious.

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

Lemma 6. 1. *If $M \sim_{\mathbb{N};k} M'$ then* **pred** *$M \sim_{\mathbb{N};k}$* **pred** *M'. If also $N \sim_{\mathbb{N};k} N'$ then $M + N \sim_{\mathbb{N};k} M' + N'$.*

2. *If $M \sim_{T_1 \times T_2;k} M'$ then* **fst** *$M \sim_{T_1;k}$* **fst** *M' and* **snd** *$M \sim_{T_2;k}$* **snd** *M'.*

3. *If $M \sim_{T_1;k} M'$ and $N \sim_{T_2;k} N'$ then $(M, N) \sim_{T_1 \times T_2;k} (M', N')$.*

4. *If $M \sim_{\mathbb{N};k} M'$, $M_1 \sim_{T;k} M_1'$ and $M_2 \sim_{T;k} M_2'$, then* **if** *M* **then** *M_1* **else** *$M_2 \sim_{T;k}$* **if** *M'* **then** *M_1'* **else** *M_2'.*

5. *If $M_1 \sim_{T_1 \to T_2;k} M_1'$ and $M_2 \sim_{T_1;k} M_2'$ then $M_1\ M_2 \sim_{T_2;k}$* **let** *$g = M_1'$* **in open** *$\langle x_f, x_e\rangle = g$* **in** *$x_f\ (g, M_2', x_e)$.*

These lemmas are proved by analyzing the simulation relations. Some proofs of these properties need the property that the equivalence relation is closed under decreasing indices, which we have already proved as Lemma 2. The proof of the last of these properties requires us to consider the evaluation of the application of a fixed point expression which involves "feeding" the expression to its own body, albeit at a smaller step measure. We apply Lemma 2 to obtain such expressions.

Informal proof of semantics preservation

Similar to semantics preservation of the CPS transformation described in Section 5.4.2, we consider semantics preservation for possibly open terms under closed substitutions. We overload the notation $(V_1/x_1, \ldots, V_n/x_n)$ for representing substitution of closed values for variables for the target language. In defining a correspondence between source and target language substitutions, we need to consider the possibility that a collection of free variables in the first may be reified into an environment variable in the second. This motivates the following definition in which γ represents a source language substitution:

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

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

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

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

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

Lemma 7. *Let $\delta = (V_1/x_1, \ldots, V_n/x_n);\gamma$ and $\delta' = (V_1'/y_1, \ldots,\ V_n'/y_n, V_e/x_e)$ be source and target language substitutions and let $\Gamma = (x_m' : T_m', \ldots, x_1' : T_1', x_n : T_n, \ldots, x_1 : T_1)$ be a source language typing context such that $\delta \approx_{\Gamma;k} \delta'$. Further, let $\rho = (x_1 \mapsto y_1, \ldots, x_n \mapsto y_n, x_1' \mapsto \pi_1(x_e), \ldots, x_m' \mapsto \pi_m(x_e))$.*

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

The semantic preservation theorem in its most general form can now be stated as follows, which is a realization of Property 3 in the setting of closure conversion:

Theorem 5. *Let $\delta = (V_1/x_1, \ldots, V_n/x_n);\gamma$ and $\delta' = (V_1'/y_1, \ldots,\ V_n'/y_n, V_e/x_e)$ be source and target language substitutions and let $\Gamma = (x_m' : T_m', \ldots, x_1' : T_1', x_n : T_n, \ldots, x_1 : T_1)$ be a source language typing context such that $\delta \approx_{\Gamma;k} \delta'$. Further, let $\rho = (x_1 \mapsto y_1, \ldots, x_n \mapsto y_n, x_1' \mapsto \pi_1(x_e), \ldots, x_m' \mapsto \pi_m(x_e))$. If $\Gamma \vdash M : T$ and $\rho \triangleright M \leadsto_{cc} M'$, then $M[\delta] \sim_{T;k} M'[\delta']$.*

We outline the main steps in the argument for this theorem: these will guide the development of a formal proof in Section 6.5. We proceed by induction on the derivation of $\rho \triangleright M \leadsto_{cc} M'$, analyzing the last step in it. This obviously depends on the structure of M. The cases for a number or the unit constructor are obvious and for a variable we use Lemma 7.1. In the remaining cases, other than when M is of the form (**let** $x = M_1$ **in** M_2) or (**fix** $f\,x.M_1$), the argument follows a set pattern: we observe that substitutions distribute to the sub-components of expressions, we invoke the induction hypothesis over the sub-components and then we use Lemma 6 to conclude. If M is of the form (**let** $x = M_1$ **in** M_2), then M' must be of the form (**let** $y = M_1'$ **in** M_2'). Here again the substitutions distribute to M_1 and M_2 and to M_1' and M_2', respectively. We then apply the induction hypothesis first to M_1 and M_1' and then to M_2 and M_2'; in the latter case, we need to consider extended substitutions with equivalent values from the former case. Finally, if M is of the form (**fix** $f\,x.M_1$), then M' must have the form $\langle M_1', M_2' \rangle$. We can prove that the abstraction M_1' is closed by using the type preservation theorem (Theorem 4) and therefore that $M'[\sigma'] = \langle M_1', M_2'[\sigma'] \rangle$. We then apply the induction hypothesis on M_1. In order to do so, we generate the appropriate typing judgment using Lemma 4 and a new pair of equivalent substitutions (under a suitable step index) using Lemma 7.2. The case is easily concluded from the result of the previous application of the inductive hypothesis.

An immediate corollary of Theorem 5 is the following which is a realization of Property 2 in the setting of closure conversion:

Corollary 8. *If $\emptyset \vdash M : T$ and $\emptyset \triangleright M \rightsquigarrow_{cc} M'$, then $M \sim_{T;i} M'$ for any i.*

From this corollary, it is easy to derive the following correctness property of the CPS transformation for closed programs at atomic types that correspond to Property 4:

Corollary 9. *If $\emptyset \vdash M : \mathbb{N}$, $\emptyset \triangleright M \rightsquigarrow_{cc} M'$ and $M \hookrightarrow V$, then $M' \hookrightarrow V$.*

6.5 Verifying the λProlog Program in Abella

In this section, we formalize the verification of closure conversion described in Section 6.4 in Abella. For this we need to make explicit the proofs of all the binding related properties. We show that the λ-tree syntax approach can be used to significantly alleviate this effort, leading to formal proofs that closely follows the informal ones.

6.5.1 Type preservation of the transformation

We prove that the λProlog implementation of closure conversion preserves typing by following the informal argument given in Section 6.4.1. First we characterize the typing contexts for the source and target languages. The typing context of the source language is defined through the following clauses for $\mathtt{ctx} : \mathtt{olist} \to \mathtt{prop}$:

$$\begin{array}{lcl}
\mathtt{ctx}\,\mathtt{nil} & \triangleq & \top \\
\mathtt{ctx}\,(\mathtt{of}\ X\ T :: L) & \triangleq & \mathtt{ctx}\ L \wedge \mathtt{name}\ X \wedge \{\mathtt{is\_sty}\ T\} \wedge \\
 & & \quad \forall T'.(\mathtt{member}\ (\mathtt{of}\ X\ T')\ L \supset T = T').
\end{array}$$

Typing contexts often encode a constraint that every element in the context pertains to a distinct variable. The definition here does not enforce such a requirement. The reason for not doing so is that we may in fact have to consider contexts in which there are multiple entries for the same variable; the reason for this is that the `combine` predicate that we used in the specification of closure conversion does not force uniqueness in the listing of the free variables. What our definition of `ctx` does ensure, though, is that every assignment for a given variable assigns it the same type.

The typing context of the target language is defined through the following clauses for $\texttt{ctx'} : \texttt{olist} \to \texttt{prop}$:

$$\begin{array}{lcl} \texttt{ctx' nil} & \triangleq & \top \\ \nabla x.\texttt{ctx'}(\texttt{of'}\ x\ T :: L) & \triangleq & \texttt{ctx'}\ L \wedge \{\texttt{is\_cty}\ T\}. \end{array}$$

Here the predicate constant $\texttt{is\_cty} : \texttt{ty} \to \texttt{o}$ defines well-formed types in the target language and is given by the following λProlog clauses:

$$\begin{array}{lcl} \texttt{is\_cty tnat}. & & \\ \texttt{is\_cty tunit}. & & \\ \texttt{is\_cty}\ (\texttt{prod}\ T_1\ T_2) & \texttt{:-} & \texttt{is\_cty}\ T_1\ , \texttt{is\_cty}\ T_2. \\ \texttt{is\_cty}\ (\texttt{arr}\ T_1\ T_2) & \texttt{:-} & \texttt{is\_cty}\ T_1\ , \texttt{is\_cty}\ T_2. \\ \texttt{is\_cty}\ (\texttt{arr'}\ T_1\ T_2) & \texttt{:-} & \texttt{is\_cty}\ T_1\ , \texttt{is\_cty}\ T_2. \end{array}$$

We need to prove the strengthening lemma of typing for the source language (*i.e.*, Lemma 4). We define the strengthening of typing context through the following clauses for the predicate constant $\texttt{prune\_ctx} : \texttt{list tm} \to \texttt{olist} \to \texttt{olist} \to \texttt{prop}$:

$$\begin{array}{l} \texttt{prune\_ctx nil}\ L\ \texttt{nil} \quad \triangleq \quad \top \\ \texttt{prune\_ctx}\ (X :: Vs)\ L\ (\texttt{of}\ X\ T :: L') \quad \triangleq \\ \quad \texttt{member}\ (\texttt{of}\ X\ T)\ L \wedge \texttt{prune\_ctx}\ Vs\ L\ L'. \end{array}$$

By definition, $\texttt{prune\_ctx}\ Vs\ L\ L'$ if and only if Vs is a sequence of variables in the domain of the typing context L and L' is the typing context obtained by strengthening L to the domain Vs. Then Lemma 4 is formalized as follows:

$$\begin{array}{l} \forall L, Vs, M, T, FVs.\texttt{ctx}\ L \supset \texttt{vars\_of\_ctx}\ L\ Vs \supset \{L \vdash \texttt{of}\ M\ T\} \supset \\ \quad \{\texttt{fvars}\ M\ Vs\ FVs\} \supset \exists L'.\texttt{prune\_ctx}\ FVs\ L\ L' \wedge \{L' \vdash \texttt{of}\ M\ T\}. \end{array}$$

We may think of proving it by induction on $\{\texttt{fvars}\ M\ Vs\ FVs\}$. However, this will not work because `fvars` may go under binders in M and introduce new assumptions into the dynamic context for marking bound variables. Thus, we need to generalize the above theorem to accommodate such possibility. We define the predicate

$\texttt{bvars} : \texttt{olist} \rightarrow \texttt{prop}$ through the following clauses for characterizing the assumptions dynamically introduced by `fvars`:

$$\begin{array}{lcl} \texttt{bvars}\,\texttt{nil} & \triangleq & \top \\ \nabla x.\texttt{bvars}\,(\texttt{notfree}\,x :: L) & \triangleq & \texttt{bvars}\,L. \end{array}$$

We designate the constant $\texttt{ctx\_bvars} : \texttt{olist} \rightarrow \texttt{olist} \rightarrow \texttt{prop}$ for relating the type assignments with the `nonfree` assumptions:

$$\begin{array}{lcl} \texttt{ctx\_bvars}\,\texttt{nil}\,\texttt{nil} & \triangleq & \top \\ \texttt{ctx\_bvars}\,(\texttt{of}\ X\ T :: L)\,(\texttt{notfree}\ X :: L') & \triangleq & \texttt{ctx\_bvars}\ L\ L'. \end{array}$$

We now generalize the theorem to the following:

$$\begin{array}{l} \forall L, L_1, L_2, Ps, Vs, FVs, M, T. \\ \quad \texttt{ctx}\ L \supset \texttt{append}\ L_1\ L_2\ L \supset \texttt{ctx\_bvars}\ L_1\ Ps \supset \texttt{vars\_of\_ctx}\ L_2\ Vs \supset \\ \quad \texttt{bvars}\ Ps \supset \{Ps \vdash \texttt{fvars}\ M\ \ Vs\ FVs\} \supset \{L \vdash \texttt{of}\ M\ T\} \supset \\ \quad \exists L', L_2'.\texttt{prune\_ctx}\ FVs\ L_2\ L_2' \wedge \texttt{append}\ L_1\ L_2'\ L' \wedge \texttt{ctx}\ L' \wedge \{L' \vdash \texttt{of}\ M\ T\}. \end{array}$$

Note here that the typing context L consists of two parts: L_1 contains variables that are marked as non-free by `fvars` and L_2 contains the initially free variables. The theorem is proved by induction on $\{Ps \vdash \texttt{fvars}\ M\ \ Vs\ FVs\}$. The proof itself is easy to construct by inspecting the logical structure of `fvars` derivations. It is easy to see that the formalized version of Lemma 4 is just a special case of the above theorem when L_1 is empty.

To formally state the type preservation theorem, we need to characterize the type preserving mappings from source variables to target terms. This is done through defining the predicate $\texttt{good\_map} : \texttt{olist} \rightarrow \texttt{list}\ (\texttt{map}\ \texttt{tm}\ \texttt{tm'}) \rightarrow \texttt{olist} \rightarrow \texttt{prop}$, as follows:

$$\begin{array}{l} \texttt{good\_map}\ CL\ \texttt{nil}\ \texttt{nil} \quad \triangleq \quad \top \\ \texttt{good\_map}\ CL\ (\texttt{map}\ X\ M :: ML)\ (\texttt{of}\ X\ T :: SL) \quad \triangleq \\ \quad \texttt{name}\ X \wedge \{CL \vdash \texttt{of'}\ M\ T\} \wedge \texttt{good\_map}\ CL\ ML\ SL. \end{array}$$

The type preserving theorem is now formally stated as follows which corresponds to Theorem 4:

$$\forall SL, CL, Map, M, Vs, M', T.$$
$$\texttt{ctx}\ SL \supset \texttt{ctx'}\ CL \supset \texttt{good\_map}\ CL\ Map\ SL \supset \{SL \vdash \texttt{of}\ M\ T\} \supset$$
$$\texttt{vars\_of\_ctx}\ SL\ Vs \supset \{\texttt{cc}\ Map\ Vs\ M\ M'\} \supset \{CL \vdash \texttt{of'}\ M'\ T\}.$$

This theorem is proved by induction on $\{\texttt{cc}\ Map\ Vs\ M\ M'\}$. The argument closely follows the informal proof for Theorem 4. Given this theorem, it is easy to prove the following formalized version of Corollary 7:

$$\forall M, M', T.\{\texttt{of}\ M\ T\} \supset \{\texttt{cc}\ \texttt{nil}\ \texttt{nil}\ M\ M'\} \supset \{\texttt{of'}\ M'\ T\}.$$

6.5.2 Semantics preservation of the transformation

In this section, we formalize the semantics preservation proof of closure conversion described in Section 6.4.2 in Abella. Again, we focus on showing the benefits of the λ-tree syntax approach in this formalization. The formal treatments of binding related properties such as closedness and substitution described in Section 5.5.2 carries over to this setting. We avoid a detailed discussion of them and instead focus on showing how reasoning about the unique features of closure conversion such as computation of free variables can benefit from the λ-tree syntax approach.

Formalizing the operational semantics

The operational semantics of the source language has already been formalized in Section 5.5.2. We are left with the operational semantics of the target language. We encode the one step evaluation rules for the target language using the predicate constant $\texttt{step'} : \texttt{tm'} \to \texttt{tm'} \to \texttt{o}$. We again consider only a few interesting cases in their definition. Assuming that $\texttt{val'} : \texttt{tm'} \to \texttt{o}$ recognize values in the target language which has an obvious definition in λProlog, the clauses for evaluating the application of closures and closure applications are the following.

$$\begin{array}{lll}
\texttt{step'}\ (\texttt{clos'}\ F\ E)\ (\texttt{clos'}\ F'\ E) & \texttt{:-} & \texttt{step'}\ F\ F'. \\
\texttt{step'}\ (\texttt{clos'}\ F\ E)\ (\texttt{clos'}\ F\ E') & \texttt{:-} & \texttt{val'}\ F, \texttt{step'}\ E\ E'. \\
\texttt{step'}\ (\texttt{open'}\ M\ R)\ (\texttt{open'}\ M'\ R) & \texttt{:-} & \texttt{step'}\ M\ M'. \\
\texttt{step'}\ (\texttt{open'}\ (\texttt{clos'}\ F\ E)\ R)\ (R\ F\ E) & \texttt{:-} & \texttt{val'}\ (\texttt{clos'}\ F\ E).
\end{array}$$

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

We use the predicates $\texttt{nstep'} : \texttt{nat} \to \texttt{tm'} \to \texttt{tm'} \to \texttt{o}$ and $\texttt{eval} : \texttt{tm'} \to \texttt{tm'} \to \texttt{o}$ to represent the n-step and full evaluation relations for the target language, respectively. These predicates have definitions similar to their counterparts in Section 5.5.2.

Formalizing the closedness property

The closedness property for terms in the source language has already been defined through $\texttt{tm} : \texttt{tm} \to \texttt{o}$ in Section 5.5.2. We further identify the predicate $\texttt{tm'} : \texttt{tm'} \to \texttt{o}$ such that $\{\texttt{tm'}\ M\}$ holds if M is a well-formed term in the target language. It is defined through a λProlog program similar to that defining tm. The only interesting clauses in the definition are the following ones for closures and closure applications:

$$\begin{array}{lll} \texttt{tm'}\ (\texttt{clos'}\ M_1\ M_2) & \texttt{:-} & \texttt{tm'}\ M_1\,, \texttt{tm'}\ M_2. \\ \texttt{tm'}\ (\texttt{open'}\ M\ R) & \texttt{:-} & \texttt{tm'}\ M\,, \Pi f, e.\texttt{tm'}\ f \Rightarrow \texttt{tm'}\ e \Rightarrow \texttt{tm'}\ (R\ f\ e). \end{array}$$

Similar to tm, we can easily prove the properties about well-formed or closed terms in the target language through the definition of tm', including that a well-typed term is also well-formed and that a closed term cannot contain any nominal constant.

Formalizing the logical relations

The following clauses define the simulation relation represented by $\texttt{sim\_cc} : \texttt{ty} \to \texttt{nat} \to \texttt{tm} \to \texttt{tm'} \to \texttt{prop}$ and the equivalence relations represented by $\texttt{equiv\_cc} : \texttt{ty} \to \texttt{nat} \to \texttt{tm} \to \texttt{tm'} \to \texttt{prop}$.

$$\begin{array}{l}
\texttt{sim\_cc}\ T\ K\ M\ M' \quad \triangleq \quad \forall J, V.\texttt{le}\ J\ K \supset \{\texttt{nstep}\ J\ M\ V\} \supset \{\texttt{val}\ V\} \supset \\
\quad \exists V', N.\{\texttt{eval'}\ M'\ V'\} \wedge \{\texttt{add}\ J\ N\ K\} \wedge \texttt{equiv\_cc}\ T\ N\ V\ V' \\
\texttt{equiv\_cc}\ \texttt{tnat}\ K\ (\texttt{nat}\ N)\ (\texttt{nat'}N) \quad \triangleq \quad \top \\
\texttt{equiv\_cc}\ \texttt{tunit}\ K\ \texttt{unit}\ \texttt{unit'} \quad \triangleq \quad \top \\
\texttt{equiv\_cc}\ (\texttt{prod}\ T_1\ T_2)\ K\ (\texttt{pair}\ V_1\ V_2)\ (\texttt{pair'}\ V_1'\ V_2') \quad \triangleq \\
\quad \texttt{equiv\_cc}\ T_1\ K\ V_1\ V_1' \wedge \texttt{equiv\_cc}\ T_2\ K\ V_2\ V_2' \wedge \\
\quad \{\texttt{tm}\ V_1\} \wedge \{\texttt{tm}\ V_2\} \wedge \{\texttt{tm'}\ V_1'\} \wedge \{\texttt{tm'}\ V_2'\} \\
\texttt{equiv\_cc}\ (\texttt{arr}\ T_1\ T_2)\ \texttt{z}\ (\texttt{fix}\ R)\ (\texttt{clos'}\ (\texttt{abs'}\ R')\ \mathit{VE}) \quad \triangleq \\
\quad \{\texttt{val'}\ \mathit{VE}\} \wedge \{\texttt{tm}\ (\texttt{fix}\ R)\} \wedge \{\texttt{tm'}\ (\texttt{clos'}\ (\texttt{abs'}\ R')\ \mathit{VE})\}
\end{array}$$

$$\begin{aligned}&\texttt{equiv\_cc}\ (\texttt{arr}\ T_1\ T_2)\ (\texttt{s}\ K)\ (\texttt{fix}\ R)\ (\texttt{clos'}\ (\texttt{abs'}\ R')\ VE) \quad \triangleq\\ &\quad \texttt{equiv\_cc}\ (\texttt{arr}\ T_1\ T_2)\ K\ (\texttt{fix}\ R)\ (\texttt{clos'}\ (\texttt{abs'}\ R')\ VE) \wedge\\ &\quad \forall V_1, V_1', V_2, V_2'.\texttt{equiv\_cc}\ T_1\ K\ V_1\ V_1' \supset \texttt{equiv\_cc}\ (\texttt{arr}\ T_1\ T_2)\ K\ V_2\ V_2' \supset\\ &\qquad \texttt{sim\_cc}\ T_2\ K\ (R\ V_2\ V_1)\ (R'\ (\texttt{pair'}\ V_2'\ (\texttt{pair'}\ V_1'\ VE))).\end{aligned}$$

The formula $\texttt{sim\_cc}\ T\ K\ M\ M'$ is intended to mean that M simulates M' at type T in at least K steps; $\texttt{equiv\_cc}\ T\ K\ V\ V'$ has a similar interpretation. Note the exploitation of λ-tree syntax, specifically the use of application, to realize substitution in the definition of equiv_cc. It is easily shown that sim_cc holds only between closed source and target terms and similarly equiv_cc holds only between closed source and target values. Similar to the formalization of equivalence in the CPS transformation, the definition of equiv_cc uses itself negatively in the last clause and thereby cannot be treated as a fixed-point definition. However, since the use is at a decreased step-index, we can treat it as a recursive definition inductively defined on the types and the step measures. In fact, this is the reason why we "build" the relation up over the natural numbers rather than mirroring directly the structure of the informal definition.

The property that the equivalence relation is closed under decreasing step-indexes is formalized as follows:

$$\begin{aligned}&\forall T, K, J, V, V'.\{\texttt{is\_sty}\ T\} \supset \{\texttt{is\_nat}\ K\} \supset\\ &\quad \texttt{equiv\_cc}\ T\ K\ V\ V' \supset \texttt{le}\ J\ K \supset \texttt{equiv\_cc}\ T\ J\ V\ V'.\end{aligned}$$

It is proved by induction on $\{\texttt{is\_sty}\ T\}$ and $\{\texttt{is\_nat}\ K\}$.

Compatibility lemmas in the style of Lemma 6 are easily stated for sim_cc. For example, the one for pairs is the following.

$$\begin{aligned}&\forall T_1, T_2, K, M_1, M_2, M_1', M_2'.\{\texttt{is\_nat}\ K\} \supset \{\texttt{is\_sty}\ T_1\} \supset \{\texttt{is\_sty}\ T_2\} \supset\\ &\quad \texttt{sim\_cc}\ T_1\ K\ M_1\ M_1' \supset \texttt{sim\_cc}\ T_2\ K\ M_2\ M_2' \supset\\ &\quad \texttt{sim\_cc}\ (\texttt{prod}\ T_1\ T_2)\ K\ (\texttt{pair}\ M_1\ M_2)\ (\texttt{pair'}\ M_1'\ M_2').\end{aligned}$$

The proofs of these lemmas follows from the informal ones in a straightforward manner.

Representing substitutions

A substitution is represented as a list of mappings from source variables to closed target values. The substitutions for the source language have already been given through `subst` in Section 5.5.2. The substitutions of target values are represented by `subst'` : `list` (`map` `tm'` `tm'`) $\rightarrow$ `prop` which is defined as follows:

$$\begin{array}{lcl} \texttt{subst' nil} & \triangleq & \top \\ \nabla x.\texttt{subst'}\ (\texttt{map}\ x\ V :: ML) & \triangleq & \texttt{subst'}\ ML \wedge \{\texttt{val'}\ V\} \wedge \{\texttt{tm'}\ V\}. \end{array}$$

The application of substitutions has already been given by the polymorphic definition of `app_subst`. As before, we can easily prove properties about substitution application based on this definition such as that such an application distributes over term structures and that closed terms are not affected by substitution.

The equivalence relation on substitutions

We first define the relation `subst_env_equiv_cc` between source substitutions and target environments:

$$\begin{array}{l} \texttt{subst\_env\_equiv\_cc nil}\ K\ ML\ \texttt{unit'} \quad \triangleq \quad \top \\ \texttt{subst\_env\_equiv\_cc}\ (\texttt{of}\ X\ T :: L)\ K\ ML\ (\texttt{pair'}\ V'\ VE) \quad \triangleq \\ \quad \exists V.\texttt{subst\_env\_equiv\_cc}\ L\ K\ ML\ VE \wedge \\ \quad\quad \texttt{member}\ (\texttt{map}\ X\ V)\ ML \wedge \texttt{equiv\_cc}\ T\ K\ V\ V'. \end{array}$$

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

$$\begin{array}{l} \nabla e.\texttt{subst\_equiv\_cc}\ L\ K\ ML\ (\texttt{map}\ e\ VE :: \texttt{nil}) \quad \triangleq \\ \quad \texttt{subst\_env\_equiv\_cc}\ L\ K\ ML\ VE \\ \nabla x, y.\texttt{subst\_equiv\_cc}\ (\texttt{of}\ x\ T :: L)\ K\ (\texttt{map}\ x\ V :: ML)\ (\texttt{map}\ y\ V' :: ML') \quad \triangleq \\ \quad \texttt{equiv\_cc}\ T\ K\ V\ V' \wedge \texttt{subst\_equiv\_cc}\ L\ K\ ML\ ML'. \end{array}$$

Lemmas about `mapvar` and `mapenv`.

A formalization of Lemma 7 is needed for the main theorem. We start with a lemma about `mapvar`.

$$\begin{array}{l} \forall L, Vs, Map, ML, K, VE, X, T, M', V.\nabla e. \\ \quad \{\texttt{is\_nat}\ K\} \supset \texttt{ctx}\ L \supset \texttt{vars\_of\_ctx}\ L\ Vs \supset \\ \quad \texttt{subst}\ ML \supset \texttt{subst\_env\_equiv\_cc}\ L\ K\ ML\ VE \supset \end{array}$$

$$\begin{array}{l}
\quad\texttt{member}\;(\texttt{of}\;X\;T)\;L \supset \{\texttt{mapvar}\;\mathit{Vs}\;\mathit{Map}\} \supset \\
\quad\texttt{app\_subst}\;\mathit{ML}\;X\;V \supset \{\texttt{member}\;(\texttt{map}\;X\;(M'\;e))\;(\mathit{Map}\;e)\} \supset \\
\qquad\exists V'.\{\texttt{eval'}(M'\;\mathit{VE})\;V'\} \wedge \texttt{equiv\_cc}\;T\;K\;V\;V'.
\end{array}$$

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

Lemma 7 is now formalized as follows.

$$\begin{array}{l}
\forall L, \mathit{ML}, \mathit{ML}', K, \mathit{Vs}, \mathit{Vs}', \mathit{Map}.\{\texttt{is\_nat}\;K\} \supset \\
\quad\texttt{ctx}\;L \supset \texttt{vars\_of\_ctx}\;L\;\mathit{Vs} \supset \\
\quad\texttt{subst}\;\mathit{ML} \supset \texttt{subst'}\;\mathit{ML}' \supset \texttt{subst\_equiv\_cc}\;L\;K\;\mathit{ML}\;\mathit{ML}' \supset \\
\quad\texttt{vars\_of\_subst'}\;\mathit{ML}'\;\mathit{Vs}' \supset \texttt{tomapping}\;\mathit{Vs}\;\mathit{Vs}'\;\mathit{Map} \supset \\
\quad(\forall X, T, V, M', M''. \\
\qquad\texttt{member}\;(\texttt{of}\;X\;T)\;L \supset \{\texttt{member}\;(\texttt{map}\;X\;M')\;\mathit{Map}\} \supset \\
\qquad\texttt{app\_subst}\;\mathit{ML}\;X\;V \supset \texttt{app\_subst}\;\mathit{ML}'\;M'\;M'' \supset \\
\qquad\exists V'.\{\texttt{eval'}\;M''\;V'\} \wedge \texttt{equiv\_cc}\;T\;K\;V\;V') \quad \wedge \\
\quad(\forall L', \mathit{NFVs}, E, E'. \\
\qquad\texttt{prune\_ctx}\;\mathit{NFVs}\;L\;L' \supset \{\texttt{mapenv}\;\mathit{NFVs}\;\mathit{Map}\;E\} \supset \\
\qquad\texttt{app\_subst}\;\mathit{ML}'\;E\;E' \supset \\
\qquad\exists \mathit{VE}'.\{\texttt{eval'}\;E'\;\mathit{VE}'\} \wedge \texttt{subst\_env\_equiv\_cc}\;L'\;K\;\mathit{ML}\;\mathit{VE}').
\end{array}$$

Two new predicates are used here. The judgment ($\texttt{vars\_of\_subst'}\;\mathit{ML}'\;\mathit{Vs}'$) "collects" the variables in the target substitution ML' into Vs'. The predicate `tomapping` is defined as follows:

$$\begin{array}{l}
\texttt{tomapping}\;\mathit{Vs}\;(E :: \texttt{nil})\;(\mathit{Map}\;E) \quad \triangleq \quad \{\texttt{mapvar}\;\mathit{Vs}\;\mathit{Map}\} \\
\texttt{tomapping}\;(X :: \mathit{Vs})\;(X' :: \mathit{Vs}')\;(\texttt{map}\;X\;X' :: \mathit{Map}) \quad \triangleq \\
\quad\texttt{tomapping}\;\mathit{Vs}\;\mathit{Vs}'\;\mathit{Map}.
\end{array}$$

By this definition, given source variables $\mathit{Vs} = (x_1, \ldots, x_n, x'_1, \ldots, x'_m)$ and target variables $\mathit{Vs}' = (y_1, \ldots, y_n, x_e)$, the predicate `tomapping` creates in Map the mapping

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

The conclusion of the lemma is a conjunction representing the two parts of Lemma 7. The first part is proved by induction on $\{\texttt{member}\ (\texttt{map}\ X\ M')\ \mathit{Map}\}$, using the lemma for `mapvar` when X is some $x'_i (1 \leq i \leq m)$. The second part is proved by induction on $\{\texttt{mapenv}\ \mathit{NFVs}\ \mathit{Map}\ E\}$ using the first part.

Formalizing the semantics preservation theorems

Theorem 5 is now formalized as follows:

$$\begin{array}{l}
\forall L, \mathit{ML}, \mathit{ML}', K, \mathit{Vs}, \mathit{Vs}', \mathit{Map}, T, P, P', M, M'. \\
\quad \{\texttt{is\_nat}\ K\} \supset \texttt{ctx}\ L \supset \texttt{vars\_of\_ctx}\ L\ \mathit{Vs} \supset \\
\quad \texttt{subst}\ \mathit{ML} \supset \texttt{subst'}\ \mathit{ML}' \supset \texttt{subst\_equiv\_cc}\ L\ K\ \mathit{ML}\ \mathit{ML}' \supset \\
\quad \texttt{vars\_of\_subst'}\ \mathit{ML}'\ \mathit{Vs}' \supset \texttt{tomapping}\ \mathit{Vs}\ \mathit{Vs}'\ \mathit{Map} \supset \\
\quad \{L \vdash \texttt{of}\ M\ T\} \supset \{\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M\ M'\} \supset \\
\quad \texttt{app\_subst}\ \mathit{ML}\ M\ P \supset \texttt{app\_subst}\ \mathit{ML}'\ M'\ P' \supset \\
\qquad \texttt{sim\_cc}\ T\ K\ P\ P'.
\end{array}$$

We use an induction on $\{\texttt{cc}\ \mathit{Map}\ \mathit{Vs}\ M\ M'\}$, the closure conversion derivation, to prove this theorem. As should be evident from the preceding development, the proof in fact closely follows the structure we outlined in Section 6.4.2. A particular point to note is that when M is a function and M' is a closure $\langle F, E\rangle$, we need to show that F is closed and substitution has no effect on it. This is achieved by applying the type preservation lemma to M to show that F is typable in an empty context and hence $\{\texttt{tm'}\ F\}$ holds. By $\{\texttt{tm'}\ F\}$ and the definition of `app_subst`, we can easily show that substitution has no effect on F.

Finally, from the above theorem it is easy to prove the following properties which correspond to Corollaries 8 and 9:

$$\begin{array}{l}
\forall K, T, M, M'. \{\texttt{of}\ M\ T\} \supset \{\texttt{cc}\ \texttt{nil}\ \texttt{nil}\ M\ M'\} \supset \texttt{sim\_cc}\ T\ K\ M\ M'. \\
\forall K, T, M, M'. \{\texttt{of}\ M\ \texttt{tnat}\} \supset \{\texttt{cc}\ \texttt{nil}\ \texttt{nil}\ M\ M'\} \supset \\
\quad \{\texttt{eval}\ M\ (\texttt{nat}\ N)\} \supset \{\texttt{eval'}\ M'\ (\texttt{nat'}\ N)\}
\end{array}$$

Chapter 7

The Code Hoisting Transformation

Code hoisting is a general transformation that has the effect of moving code that appears in a nested context but that is not dependent on that context to a place outside of it. In the compilation process we are considering, this transformation is useful in moving nested functions that have been converted into a closed form by closure conversion to the outermost level in the program. Note, however, that code hoisting has a justification independently of closure conversion and can be used even in situations where the latter transformation has not been applied. Despite this fact, these two transformations have been considered only in combination in many efforts at verified compilation of functional programs; see, for example, [46] and [86]. The reason for this phenomenon is the following: the correctness of code hoisting relies on the expression being hoisted out being closed, an aspect that is easily verified for the function expressions that result from closure conversion.

The situation described above is somewhat unfortunate: it is relatively easy to write code to determine that an expression is independent of the context in which it appears, but we are compelled to combine its movement with another process because of the difficulty in verifying this fact. The λ-tree syntax approach that we are elucidating in this thesis provides a way to overcome this problem and thereby to describe code hoisting as a transformation in its own right. The idea we use is impressive in its simplicity. The independence of a subpart of a particular abstraction can be characterized as a logical property of the binding structure of the abstraction. Then, since independence is given a logical characterization, this information can also be used in the process of proving the correctness of the transformation.

We elaborate on these ideas in this chapter. We start with an an overview of the code hoisting transformation in Section 7.1. We then describe it in a rule-based and relational style in Section 7.2 and discuss its implementation in λProlog in Section 7.3. We finally discuss the informal and formal verification of the transformation in Sections 7.4 and 7.5.

7.1 An Overview of the Transformation

The code hoisting transformation that we will consider is based on [46]. In that context, the transformation is focused on moving nested functions out to the top-level in the program. Its particular content is most easily understood through examples. The following code is the output of closure conversion in the first example in Section 6.1:

$$\begin{array}{l}\texttt{let}\ x = 2\ \texttt{in}\ \texttt{let}\ y = 3\ \texttt{in}\\ \quad \langle(\texttt{fun}\ z\ e \to z + e.1 + e.2), (x, y)\rangle\end{array}$$

As we can see, the function part of the environment is closed. Code hoisting extracts this function to the top-level, resulting in the following code:

$$\begin{array}{l}\texttt{let}\ f = (\texttt{fun}\ z\ e \to z + e.1 + e.2)\ \texttt{in}\\ \quad \texttt{let}\ x = 2\ \texttt{in}\ \texttt{let}\ y = 3\ \texttt{in}\langle f, (x, y)\rangle\end{array}$$

In the general case, the transformation of interest has to treat functions that have an arbitrarily nested structure. Towards this end, it can be implemented as a recursive procedure: given a function $(\lambda x.\, M)$, the procedure is applied to the subterms of M and the extracted functions are then moved out of $(\lambda x.\, M)$. Of course, for this movement to be possible, it must be the case that the variable x does not appear in the functions that are candidates for extraction. This is the case for all the nested functions that are produced by closure conversion. For example, the following code is the output of closure conversion in the second example in Section 6.1:

$\texttt{let}\ x = 3\ \texttt{in}$
$\quad \langle(\texttt{fun}\ y\ e_1 \to \langle(\texttt{fun}\ z\ e_2 \to e_2.1 + e_2.2 + z), (e_1.1, y)\rangle), (x)\rangle.$

Here both functions are closed. Therefore they can be extracted to the top-level. Note also that the outer function depends on the inner one. To break the dependence, code hoisting introduces an extra argument to the outer function which is bound by an application to the inner function at the point where the outer function occurs. The result of code hoisting on the above code is as follows:

$\texttt{let}\ f_1 = \texttt{fun}\ z\ e_2 \to e_2.1 + e_2.2 + z\ \texttt{in}$
$\texttt{let}\ f_2 = \texttt{fun}\ f_1\ y\ e_1 \to \langle f_1, (e_1.1, y)\rangle\ \texttt{in}$
$\quad \texttt{let}\ x = 3\ \texttt{in}\ \langle(f_2\ f_1), (x)\rangle$

From the above example, we can see that the key to specifying code hoisting is to capture the constraint that the functions extracted from M in $(\lambda x. M)$ do not depend on x. If we combine code hoisting with closure conversion, we have an over-arching property that every function is closed and therefore cannot depend on any variable bound by an external abstraction. However, we can also think of checking the constraint that must be satisfied directly as a structural property of the expressions being considered. This is the approach we take in the describing closure conversion in the next section.

7.2 A Rule-Based Description of the Transformation

We give a rule-based relational description of the code hoisting transformation in this section. We first describe the source and target languages of the transformation, including their typing rules, and then present the transformation rules.

7.2.1 The source and target languages

The source language of the transformation is the target language of closure conversion, depicted in Figure 6.1. The target language of the transformation is the same as the source language. The result of code hoisting will be a term of the form

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

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

7.2.2 The transformation rules

We write the judgment of code hoisting as ($\rho \triangleright M \leadsto_{ch} M'$) where ρ has the form $(x_1, \ldots, x_n)$. This judgment asserts that M' is the result of extracting all functions in M to the top level, assuming that ρ contains all the bound variables in the context in which M appears. The transformation rules are summarized in Figure 7.1 where we write $(\vec{f}, \vec{g})$ and $(\vec{F}, \vec{G})$ to represent the concatenation of tuples of variables and terms. The transformation judgment is defined by recursion on the structure of M. In the base case, a (bound) variable transforms to itself. Most of the recursive cases have the following 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 recursion. One of the main rule that deserves further discussion is ch-abs for transforming functions. Intuitively, the term $(\lambda x.\, M)$, is transformed by extracting the functions from within M and then moving them further out of the scope of x. For this, this rule has a side condition that the function argument x must not occur in $\vec{F}$ which are functions extracted from M. The resulting body is made independent of the extracted functions by converting it into a form where it can be applied to these functions. Similarly, the ch-let rule also has a side condition that the extracted functions $\vec{G}$ cannot depend on the binding variable x. Another point to note is that the rule ch-open works for applications of closures with a particular structure, which suffices for extracting functions from applications of closures in our setting of compilation.

7.3 Implementing the Transformation in λProlog

Our presentation of the implementation of code hoisting has two parts: we first show the encoding of the syntax and typing rules of the source and target languages and then present a λProlog implementation of the transformation.

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

$$\frac{}{\rho \triangleright n \rightsquigarrow_{ch} \textbf{letf } () = () \textbf{ in } n}\text{ ch-nat} \qquad \frac{\rho \triangleright M \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M'}{\rho \triangleright \textbf{pred } M \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in pred } M'}\text{ ch-pred}$$

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

$$\frac{\begin{array}{c}\rho \triangleright M_1 \rightsquigarrow_{ch} \textbf{letf } \vec{f} = \vec{F} \textbf{ in } M_1' \\ \rho \triangleright M_2 \rightsquigarrow_{ch} \textbf{letf } \vec{g} = \vec{G} \textbf{ in } M_2' \qquad \rho \triangleright M_3 \rightsquigarrow_{ch} \textbf{letf } \vec{h} = \vec{H} \textbf{ in } M_3'\end{array}}{\begin{array}{l}\rho \triangleright \textbf{if } M_1 \textbf{ then } M_2 \textbf{ else } M_3 \rightsquigarrow_{ch} \\ \qquad \textbf{letf } (\vec{f}, \vec{g}, \vec{h}) = (\vec{F}, \vec{G}, \vec{H}) \textbf{ in if } M_1' \textbf{ then } M_2' \textbf{ else } M_3'\end{array}}\text{ ch-if}$$

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

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

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

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

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

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

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

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

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

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

Figure 7.1: The Rules for Code Hoisting

7.3.1 Encoding the language

The only issue here is to provide a convenient representation for the output of code hoisting, or *hoisted terms*. Towards this end, we identify the constants $\texttt{hbase} : \texttt{tm'} \to \texttt{tm'}$, $\texttt{habs} : (\texttt{tm'} \to \texttt{tm'}) \to \texttt{tm'}$ and $\texttt{htm} : \texttt{list tm'} \to \texttt{tm'} \to \texttt{tm'}$ for representing hoisted terms. Using these constants, a hoisted term of the form

$$\mathbf{letf}\ (f_1, \ldots, f_n) = (M_1, \ldots, M_n)\ \mathbf{in}\ M$$

will be represented by

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

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

We also identify the constant $\texttt{of''} : \texttt{tm'} \to \texttt{ty} \to \texttt{o}$ for typing the output of code hoisting such that $\texttt{of''}\ M\ T$ holds if and only if M has type T. They are defined by the following clauses which simply restate the rules for typing the output of code hoisting:

$\texttt{of''}\ (\texttt{htm}\,\texttt{nil}\ (\texttt{hbase}\ M))\ T\ \texttt{:-}\ \texttt{of'}\ M\ T.$
$\texttt{of''}\ (\texttt{htm}\ (M :: L)\ (\texttt{habs}\ R))\ T\ \texttt{:-}$
  $\texttt{of'}\ M\ T_1\,, \Pi x.\texttt{of'}\ x\ T_1 \Rightarrow \texttt{of''}\ (\texttt{htm}\ L\ (R\ x))\ T.$

7.3.2 Specifying the code hoisting transformation

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 in Figure 7.1 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.

$\texttt{ch}\ (\texttt{nat'}\ N)\ (\texttt{htm}\,\texttt{nil}\ (\texttt{hbase}\ (\texttt{nat'}\ N))).$
$\texttt{ch}\,\texttt{unit'}\ (\texttt{htm}\,\texttt{nil}\ (\texttt{hbase}\,\texttt{unit'})).$
$\texttt{ch}\ (\texttt{pred}\ M)\ (\texttt{htm}\ FE\ M'')\ \texttt{:-}\ \texttt{ch}\ M\ (\texttt{htm}\ FE\ M')\,, \texttt{hconstr}\ M'\ (x \setminus \texttt{pred'}\ x)\ M''.$
$\texttt{ch}\ (\texttt{plus'}\ M_1\ M_2)\ (\texttt{htm}\ FE\ M')\ \texttt{:-}\ \texttt{ch}\ M_1\ (\texttt{htm}\ FE_1\ M_1')\,, \texttt{ch}\ M_2\ (\texttt{htm}\ FE_2\ M_2')\,,$
  $\texttt{append}\ FE_1\ FE_2\ FE\,, \texttt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \texttt{plus'}\ x\ y)\ M'.$
$\texttt{ch}\ (\texttt{ifz'}\ M_1\ M_2\ M_3)\ (\texttt{htm}\ FE\ M')\ \texttt{:-}$
  $\texttt{ch}\ M_1\ (\texttt{htm}\ FE_1\ M_1')\,, \texttt{ch}\ M_2\ (\texttt{htm}\ FE_2\ M_2')\,, \texttt{ch}\ M_3\ (\texttt{htm}\ FE_3\ M_3')\,,$

$\quad\mathtt{append}\ FE_1\ FE_2\ FE_{12}\,,\mathtt{append}\ FE_{12}\ FE_3\ FE\,,$

$\quad\mathtt{hcombine3}\ M_1'\ M_2'\ M_3'\ (x \setminus y \setminus z \setminus \mathtt{ifz'}\ x\ y\ z)\ M'.$

$\mathtt{ch}\ (\mathtt{pair'}\ M_1\ M_2)\ (\mathtt{htm}\ FE\ M')\ \mathtt{:-}\ \mathtt{ch}\ M_1\ (\mathtt{htm}\ FE_1\ M_1')\,,\mathtt{ch}\ M_2\ (\mathtt{htm}\ FE_2\ M_2')\,,$

$\quad\mathtt{append}\ FE_1\ FE_2\ FE\,,\mathtt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \mathtt{pair'}\ x\ y)\ M'.$

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

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

$\mathtt{ch}\ (\mathtt{let'}\ M\ R)\ (\mathtt{htm}\ FE\ M'')\ \mathtt{:-}\ \mathtt{ch}\ M\ (\mathtt{htm}\ FE_1\ M')\,,$

$\quad(\Pi x.\mathtt{ch}\ x\ (\mathtt{htm}\ \mathtt{nil}\ (\mathtt{hbase}\ x)) \Rightarrow \mathtt{ch}\ (R\ x)\ (\mathtt{htm}\ FE_2\ (R'\ x)))\,,$

$\quad\mathtt{append}\ FE_1\ FE_2\ FE\,,\mathtt{hcombine\_abs}\ M'\ R'\ (x \setminus y \setminus \mathtt{let'}\ x\ y)\ M''.$

$\mathtt{ch}\ (\mathtt{abs'}\ R)\ (\mathtt{htm}\ ((\mathtt{abs'}\ F) :: FE)\ (\mathtt{habs}\ R''))\ \mathtt{:-}$

$\quad(\Pi x.\mathtt{ch}\ x\ (\mathtt{htm}\ \mathtt{nil}\ (\mathtt{hbase}\ x)) \Rightarrow \mathtt{ch}\ (R\ x)\ (\mathtt{htm}\ FE\ (R'\ x)))\,,$

$\quad\mathtt{abstract}\ R'\ R''\ F.$

$\mathtt{ch}\ (\mathtt{app'}\ M_1\ M_2)\ (\mathtt{htm}\ FE\ M')\ \mathtt{:-}\ \mathtt{ch}\ M_1\ (\mathtt{htm}\ FE_1\ M_1')\,,\mathtt{ch}\ M_2\ (\mathtt{htm}\ FE_2\ M_2')\,,$

$\quad\mathtt{append}\ FE_1\ FE_2\ FE\,,\mathtt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \mathtt{app'}\ x\ y)\ M'.$

$\mathtt{ch}\ (\mathtt{clos'}\ M_1\ M_2)\ (\mathtt{htm}\ FE\ M')\ \mathtt{:-}\ \mathtt{ch}\ M_1\ (\mathtt{htm}\ FE_1\ M_1')\,,\mathtt{ch}\ M_2\ (\mathtt{htm}\ FE_2\ M_2')\,,$

$\quad\mathtt{append}\ FE_1\ FE_2\ FE\,,\mathtt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \mathtt{clos'}\ x\ y)\ M'.$

$\mathtt{ch}\ (\mathtt{open'}\ M_1\ (f \setminus e \setminus \mathtt{app'}\ f\ (\mathtt{pair'}\ M_1\ (\mathtt{pair'}\ M_2\ e))))\ (\mathtt{htm}\ FE\ M')\ \mathtt{:-}$

$\quad\mathtt{ch}\ M_1\ (\mathtt{htm}\ FE_1\ M_1')\,,\mathtt{ch}\ M_2\ (\mathtt{htm}\ FE_2\ M_2')\,,\mathtt{append}\ FE_1\ FE_2\ FE\,,$

$\quad\mathtt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus (\mathtt{open'}\ x\ (f \setminus e \setminus \mathtt{app'}\ f\ (\mathtt{pair'}\ x\ (\mathtt{pair'}\ y\ e)))))\ M'.$

These clauses are transparently translated from the rules in Figure 7.1. The predicates `hconstr`, `hcombine`, `hcombine3`, `hcombine_abs`, `hcombine_abs2` are defined to combine the results of code hoisting on subterms to form the outputs. For instance, `hcombine` is defined through the following clauses:

$\mathtt{hcombine}\ (\mathtt{hbase}\ M_1)\ (\mathtt{hbase}\ M_2)\ C\ (\mathtt{hbase}\ (C\ M_1\ M_2)).$

$\mathtt{hcombine}\ (\mathtt{habs}\ R)\ M\ C\ (\mathtt{habs}\ R')\ \mathtt{:-}\ \Pi f.\mathtt{hcombine}\ (R\ f)\ M\ C\ (R'\ f).$

$\mathtt{hcombine}\ (\mathtt{hbase}\ M)\ (\mathtt{habs}\ R)\ C\ (\mathtt{habs}\ R')\ \mathtt{:-}$

$\quad\Pi f.\mathtt{hcombine}\ (\mathtt{hbase}\ M)\ (R\ f)\ C\ (R'\ f).$

By this definition, if M_1' is $(\mathtt{habs}\ (f_1,\ldots,f_n) \setminus M_1)$ and M_2' is $(\mathtt{habs}\ (g_1,\ldots,g_m) \setminus M_2)$, then $(\mathtt{hcombine}\ M_1'\ M_2'\ (x \setminus y \setminus \mathtt{pair}'\ x\ y)\ M')$ holds when M' is

$$(\mathtt{habs}\ (f_1,\ldots,f_n,g_1,\ldots,g_m) \setminus \mathtt{pair}'\ M_1\ M_2).$$

Again, we use universal goals to perform recursion over binding operators and hypothetical goals to introduce the rules for transforming binding variables. As a result, the context of code hoisting is represented implicitly by the dynamic context in HH$^\omega$ sequents. Note that the side condition that the extracted functions in the rules ch-abs do not contain free occurrences of the binding variable which they are hoisted over has a completely logical encoding: it is statically captured by the ordering of quantifiers and dynamically via unification. For instance, by the fourth last clause above which encodes ch-abs, to perform code hoisting on a function (`abs'` R), we need to derive the following goal:

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

Here the variable FE which represents the functions extracted from R cannot depend on x because FE is bound out side of x. Thus, this goal is derivable only if FE matches with some term not containing x as a free variable. The same observation can be made for the clause encoding the rules ch-let.

We have used the predicate `abstract` to build the final result of the transformation on functions. Intuitively, `abstract` eliminates the dependence of a function on the functions nested in its body by abstracting it over a tuple that binds these functions and generates an application expression for representing the original function. Specifically, let (`htm` FE $(R'\ x)$) be the result of recursive code hoisting on the body of a function where x is the function argument. Here R' is a meta-level abstraction of the form

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

where $f_1,\ldots,f_n$ are binders for the functions in FE. Then (`abstract` R' R'' F) is derivable if and only if F is

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

and R'' is

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

As a result, (`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

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

the function $(\texttt{abs'}\ F)$ is hoisted to the top-level and bound by f in R'' and the application $(f\ (f_1, \ldots, f_n))$ in R'' represents the original function. The definition of `abstract` is easy to construct and is not provided here.

7.4 Informal Verification of the Transformation

We informally describe the verification of code hoisting in this section based on the ideas presented in Section 4.2. Similar to the informal verification of the CPS transformation and closure conversion, we first discuss the type preservation and then the semantics preservation.

7.4.1 Type preservation of the transformation

The type preservation property of code hoisting is stated as follows:

Theorem 6. *Let* $\rho = (x_1, \ldots, x_n)$ *and* $\Gamma = (x_1 : T_1, \ldots, x_n : T_n)$*. If* $\Gamma \vdash M : T$ *and* $\rho \triangleright M \leadsto_{ch} M'$*, then* $\Gamma \vdash M' : T$*.*

This theorem is easily proved by induction on the derivation of $\rho \triangleright M \leadsto_{ch} M'$, from which we can derive the following corollary:

Corollary 10. *If* $\emptyset \vdash M : T$ *and* $\emptyset \triangleright M \leadsto_{ch} M'$*, then* $\emptyset \vdash M' : T'$*.*

7.4.2 Semantics preservation of the transformation

We give an informal description of semantics preservation for code hoisting in this section. The operational semantics of the source and target languages is already known. In the rest of this section, we present the logical relations for denoting equivalence between the source and target programs, their properties, the semantics preservation theorem and its proof.

An alternative typing rule for closure applications

The rule cof-open for typing the closure applications described in Section 6.2.1 introduces new type constants as the types of the environments of the closures. Since those type constants stand for "unknown" types, logical relations cannot be easily defined by recursion on their structures. To circumvent this problem, notice that the code hoisting transformation only deals with closure applications of the following form

$$\textbf{open } \langle x_f, x_e \rangle = M_1 \textbf{ in } x_f\ (M_1, M_2, x_e).$$

In this case, we can use the following typing rule equivalent to cof-open that does not introduce new constants:

$$\frac{\Gamma \vdash M_1 : T_1 \to T_2 \quad \Gamma \vdash M_2 : T_1}{\Gamma \vdash \textbf{open } \langle x_f, x_e \rangle = M_1 \textbf{ in } x_f\ (M_1, M_2, x_e) : T_2}$$

We shall assume that this typing rule is used instead of cof-open in the following discussion of the semantics preservation proof for code hoisting.

Logical relations and their properties

The step-indexing logical relations for code hoisting are depicted in Figure 7.2. The second to last rule in this figure defines the equivalence relation between abstraction values and the last rule defines the equivalence relation between closure values which function like recursive functions. Similar to closure conversion, we can prove the property that $\approx$ is closed under decreasing indexes and compatibility lemmas for program constructs in the source language.

Informal proof of semantics preservation

Similar to the CPS transformation and closure conversion, we consider semantics preservation for possibly open terms under closed substitutions. A substitution has the form $(V_1/x_1, \ldots, V_n/x_n)$ where V_i are closed values. The equivalence relation between source and target substitutions is based on step-indexing logical relation and given as follows:

$$(V_1/x_1, \ldots, V_n/x_n) \approx_{(x_1:T_1,\ldots,x_n:T_n);k} (V_1'/x_1, \ldots, V_n'/x_n) \iff (\forall 1 \leq i \leq n. V_i \approx_{T_i;i} V_i')$$

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

Figure 7.2: The Logical Relations for Verifying Code Hoisting

The semantics preservation theorem for the CPS transformation can now be stated as follows:

Theorem 7. *Let* $\Gamma = (x_1 : T_1, \ldots, x_n : T_n)$*,* $\rho = (x_1, \ldots, x_n)$*,* $\theta \approx_{\Gamma;k} \theta'$ *and* $\Gamma \vdash M : T$*. If* $\rho \rhd M \rightsquigarrow_{ch} M'$*, then* $M[\theta] \sim_{T;k} M'[\theta']$*.*

We outline the main steps in the argument for this theorem. We proceed by induction on the derivation of $\rho \rhd M \rightsquigarrow_{ch} M'$, analyzing the last step in it. This obviously depends on the structure of M. The cases for a natural number, the unit constructor or a variable are obvious. In the remaining case, other than when M is a let expression or a function, the argument follows a set pattern: we observe that substitutions distribute to the sub-components of expressions, we invoke the induction hypothesis over the sub-components and then we use the compatibility lemmas to conclude. When M is a let expression or a function, we need to extend the substitutions with equivalent values for the binding variable for applying the inductive hypothesis over the body of M. Note that to conclude the proof, we need to make use of the side condition that the binding variable does not occur in the functions extracted from the body of M to show that the extensions to substitutions have no effect on these functions.

From Theorem 7, it is easy to derive the following corollaries of semantic preservation for closed terms and closed terms at atomic types:

Corollary 11. *If* $\emptyset \vdash M : T$ *and* $\emptyset \rhd M \rightsquigarrow_{ch} M'$*, then* $M \sim_{T;i} M'$ *for any* i*.*

Corollary 12. *If* $\emptyset \vdash M : \mathbb{N}$*,* $\emptyset \rhd M \rightsquigarrow_{ch} M'$ *and* $M \hookrightarrow V$*, then* $M' \hookrightarrow V$*.*

7.5 Verifying the λProlog Program in Abella

In this section, we formalize the verification of code hoisting described in Section 7.4 in Abella. In this discussion, we pay particular attention to showing how the logical treatment of the side condition that describes the independence of extracted functions on the arguments of enclosing functions in the λProlog program is exploited to simplify the proofs.

7.5.1 Type preservation of the transformation

In the λProlog program, the contexts of code hoisting is represented by the dynamic contexts of HH$^\omega$ sequents that encode the rule for transforming free variables. Such contexts are represented by the predicate `ch_ctx` : `olist` $\to$ `prop` given by the following clauses:

$$\begin{array}{lcl} \texttt{ch\_ctx}\,\texttt{nil} & \triangleq & \top \\ \nabla x.\texttt{ch\_ctx}\ ((\texttt{ch}\ x\ (\texttt{htm}\ \texttt{nil}\ (\texttt{hbase}\ x)))::L) & \triangleq & \texttt{ch\_ctx}\ L. \end{array}$$

Theorem 6 is then formalized as follows where `vars_of_ch_ctx` and `vars_of_ctx'` are respectively used to collect variables in the contexts of code hoisting and variables in the typing contexts:

$$\begin{array}{l} \forall L, CL, Vs, M, M', T. \\ \quad \texttt{ctx'}\ L \supset \texttt{ch\_ctx}\ CL \supset \texttt{vars\_of\_ctx'}\ L\ Vs \supset \texttt{vars\_of\_ch\_ctx}\ CL\ Vs \supset \\ \quad \{L \vdash \texttt{of'}\ M\ T\} \supset \{CL \vdash \texttt{ch}\ M\ M'\} \supset \{L \vdash \texttt{of''}\ M'\ T\}. \end{array}$$

It is proved by induction on $\{CL \vdash \texttt{ch}\ M\ M'\}$ in an obvious manner. From this theorem, it is easy to prove the following theorem corresponding to Corollary 10:

$$\forall M, M', T.\{\texttt{of'}\ M\ T\} \supset \{\texttt{ch}\ M\ M'\} \supset \{\texttt{of''}\ M'\ T\}.$$

7.5.2 Semantics preservation of the transformation

In this section, we formalize the semantics preservation proof of closure conversion described in Section 6.4.2 in Abella.

Formalizing the operational semantics

The operational semantics of the source and target languages has already been given in Section 6.5.2. The only issue here is to encode the operational semantics for the output of code hoisting. Towards this end, we introduce the constant $\mathtt{eval''} : \mathtt{tm'} \to \mathtt{tm'} \to \mathtt{o}$ such that $\mathtt{eval''}\ (\mathtt{htm}\ FE\ M)\ V$ if and only if $(\mathtt{htm}\ FE\ M)$ evaluates to the value V. Its definition is given as follows which simply restates the evaluation rules for the term represented by $(\mathtt{htm}\ FE\ M)$:

$$\mathtt{eval''}\ (\mathtt{htm}\ \mathtt{nil}\ (\mathtt{hbase}\ M))\ V\ \ \mathtt{:-}\ \mathtt{eval'}\ M\ V.$$
$$\mathtt{eval''}\ (\mathtt{htm}\ (F :: FE)\ (\mathtt{habs}\ R))\ V\ \ \mathtt{:-}\ \mathtt{eval'}\ (\mathtt{htm}\ FE\ (R\ F))\ V.$$

Formalizing the logical relations

The simulation and equivalence relations are represented by the predicate constants $\mathtt{sim\_ch} : \mathtt{ty} \to \mathtt{nat} \to \mathtt{tm'} \to \mathtt{tm'} \to \mathtt{prop}$ and $\mathtt{equiv\_ch} : \mathtt{ty} \to \mathtt{nat} \to \mathtt{tm'} \to \mathtt{tm'} \to \mathtt{prop}$, which are defined as follows:

$$\mathtt{sim\_ch}\ T\ K\ M\ M' \ \triangleq\ \forall J, V. \mathtt{le}\ J\ K \supset \{\mathtt{nstep'}\ J\ M\ V\} \supset \{\mathtt{val'}\ V\} \supset$$
$$\quad \exists V', N. \{\mathtt{eval''}\ M'\ V'\} \wedge \{\mathtt{add}\ J\ N\ K\} \wedge \mathtt{equiv\_ch}\ T\ N\ V\ V'$$
$$\mathtt{equiv\_ch}\ \mathtt{tnat}\ K\ (\mathtt{nat'}\ N)\ (\mathtt{nat'} N) \ \triangleq\ \top$$
$$\mathtt{equiv\_ch}\ \mathtt{tunit}\ K\ \mathtt{unit'}\ \mathtt{unit'} \ \triangleq\ \top$$
$$\mathtt{equiv\_ch}\ (\mathtt{prod}\ T_1\ T_2)\ K\ (\mathtt{pair'}\ V_1\ V_2)\ (\mathtt{pair'}\ V_1'\ V_2') \ \triangleq$$
$$\quad \mathtt{equiv\_ch}\ T_1\ K\ V_1\ V_1' \wedge \mathtt{equiv\_ch}\ T_2\ K\ V_2\ V_2'$$
$$\mathtt{equiv\_ch}\ (\mathtt{arr'}\ T_1\ T_2)\ \mathtt{z}\ (\mathtt{abs'}\ R)\ (\mathtt{abs'}\ R') \ \triangleq$$
$$\quad \{\mathtt{tm'}\ (\mathtt{abs'}\ R)\} \wedge \{\mathtt{tm'}\ (\mathtt{abs'}\ R')\}$$
$$\mathtt{equiv\_ch}\ (\mathtt{arr'}\ T_1\ T_2)\ (\mathtt{s}\ K)\ (\mathtt{abs'}\ R)\ (\mathtt{abs'}\ R') \ \triangleq$$
$$\quad \mathtt{equiv\_ch}\ (\mathtt{arr'}\ T_1\ T_2)\ K\ (\mathtt{abs'}\ R)\ (\mathtt{abs'}\ R') \wedge$$
$$\quad \forall V, V'. \mathtt{equiv\_ch}\ T_1\ K\ V\ V' \supset \mathtt{sim\_ch}\ T_2\ K\ (R\ V)\ (\mathtt{htm}\ \mathtt{nil}\ (\mathtt{hbase}\ (R'\ V')))$$
$$\mathtt{equiv\_ch}\ (\mathtt{arr}\ T_1\ T_2)\ \mathtt{z}\ (\mathtt{clos'}(\mathtt{abs'}\ R)\ VE)\ (\mathtt{clos'}(\mathtt{abs'}\ R')\ VE') \ \triangleq$$
$$\quad \{\mathtt{tm'}\ (\mathtt{clos'}(\mathtt{abs'}\ R)\ VE)\} \wedge \{\mathtt{tm'}\ (\mathtt{clos'}(\mathtt{abs'}\ R')\ VE')\} \wedge$$
$$\quad \{\mathtt{val'}\ VE\} \wedge \{\mathtt{val'}\ VE'\}$$
$$\mathtt{equiv\_ch}\ (\mathtt{arr}\ T_1\ T_2)\ (\mathtt{s}\ K)\ (\mathtt{clos'}(\mathtt{abs'}\ R)\ VE)\ (\mathtt{clos'}(\mathtt{abs'}\ R')\ VE') \ \triangleq$$
$$\quad \mathtt{equiv\_ch}\ (\mathtt{arr}\ T_1\ T_2)\ K\ (\mathtt{clos'}(\mathtt{abs'}\ R)\ VE)\ (\mathtt{clos'}(\mathtt{abs'}\ R')\ VE') \wedge$$
$$\quad \forall V_1, V_1', V_2, V_2'. \mathtt{equiv\_ch}\ T_1\ K\ V_1\ V_1' \supset \mathtt{equiv\_ch}\ (\mathtt{arr}\ T_1\ T_2)\ K\ V_2\ V_2' \supset$$

$$\texttt{sim\_ch}\, T_2\ K\ (R\ (\texttt{pair'}\ V_2\ (\texttt{pair'}\ V_1\ \mathit{VE})))$$
$$(\texttt{htm}\,\texttt{nil}\ (\texttt{hbase}\ (R'\ (\texttt{pair'}\ V_2'\ (\texttt{pair'}\ V_1'\ \mathit{VE}'))))).$$

The formula $\texttt{sim\_ch}\, T\ K\ M\ M'$ is intended to mean that M simulates M' at type T in at least K steps; $\texttt{equiv\_ch}\ T\ K\ V\ V'$ has a similar interpretation. Note that the last argument of `sim_ch` must be an output of code hoisting and therefore must have the form $(\texttt{htm}\ \mathit{FE}\ M'')$; this is also manifested in the uses of `sim_ch` in the above definition.

Using the above definition, it is easy to formally prove the property that the equivalence relation is closed under decreasing step measures and the compatibility lemmas, as we have done for the CPS transformation and closure conversion. The structures of those theorems and proofs are easy to predict and we omit these details here.

The equivalence relation on substitutions

We designate the constant $\texttt{subst\_equiv\_ch} : \texttt{olist} \to \texttt{nat} \to \texttt{list}\ (\texttt{map}\ \texttt{tm'}\ \texttt{tm'}) \to \texttt{list}\ (\texttt{map}\ \texttt{tm'}\ \texttt{tm'}) \to \texttt{prop}$ to represent this equivalence relation between substitutions. It is defined as follows:

$$\texttt{subst\_equiv\_ch}\,\texttt{nil}\ K\ \texttt{nil}\,\texttt{nil} \quad \triangleq \quad \top$$
$$\nabla x.\texttt{subst\_equiv\_ch}\ (\texttt{of'}\ x\ T :: L)\ K\ (\texttt{map}\ x\ V :: \mathit{ML})\ (\texttt{map}\ x\ V' :: \mathit{ML}') \quad \triangleq$$
$$\texttt{equiv\_ch}\ T\ K\ V\ V' \wedge \texttt{subst\_equiv\_ch}\ L\ K\ \mathit{ML}\ \mathit{ML}'.$$

Formalizing the semantics preservation theorems

Theorem 7 is now formalized as follows:

$$\begin{array}{l}
\forall L, K, \mathit{CL}, \mathit{ML}, \mathit{ML}', M, M', T, \mathit{FE}, \mathit{FE}', P, P', \mathit{Vs}.\{\texttt{is\_nat}\ K\} \supset \\
\quad \texttt{ctx'}\ L \supset \texttt{vars\_of\_ctx'}\ L\ \mathit{Vs} \supset \\
\quad \texttt{ch\_ctx}\ \mathit{CL} \supset \texttt{vars\_of\_ch\_ctx}\ \mathit{CL}\ \mathit{Vs} \supset \\
\quad \texttt{subst'}\ \mathit{ML} \supset \texttt{subst'}\ \mathit{ML}' \supset \texttt{subst\_equiv\_ch}\ L\ K\ \mathit{ML}\ \mathit{ML}' \supset \\
\quad \{L \vdash \texttt{of'}\ M\ T\} \supset \{\mathit{CL} \vdash \texttt{ch}\ M\ (\texttt{htm}\ \mathit{FE}\ M')\} \supset \\
\quad \texttt{app\_subst}\ \mathit{ML}\ M\ P \supset \texttt{app\_subst}\ \mathit{ML}'\ (\texttt{htm}\ \mathit{FE}\ M')\ (\texttt{htm}\ \mathit{FE}'\ P') \supset \\
\quad\quad \texttt{sim\_ch}\ T\ K\ P\ (\texttt{htm}\ \mathit{FE}'\ P').
\end{array}$$

This theorem is proved by induction on $\{\mathit{CL} \vdash \texttt{ch}\ M\ (\texttt{htm}\ \mathit{FE}\ M')\}$. The proof closely follows the informal argument we provided for Theorem 7. Note that when M is a function ($\texttt{abs'}\ R$), we need to apply the inductive hypothesis on $(R\ x)$ where x is the function argument with substitutions ($\texttt{map}\ x\ V\ \texttt{::}\ ML$) and ($\texttt{map}\ x\ V'\ \texttt{::}\ ML'$) where V and V' are equivalent values for x. A critical step for proving this case is to show the extension for x in the second substitution has no effect on the functions FE extracted from $(R\ x)$, *i.e.*, $\texttt{app\_subst}$ ($\texttt{map}\ x\ V'\ \texttt{::}\ ML'$) $FE\ FE'$ if and only if $\texttt{app\_subst}\ ML'\ FE\ FE'$ for some FE'. By observing the logical structure of the clause of $\texttt{ch}$ for this case, Abella knows that FE cannot depend on x. By the definition of $\texttt{app\_subst}$, it is immediate that the above relation holds.

From the above theorem, we can easily prove the following theorems that formalize Corollaries 11 and 12:

$$\forall K, T, M, M'. \{\texttt{of'}\ M\ T\} \supset \{\texttt{ch}\ M\ M'\} \supset \texttt{sim\_ch}\ T\ K\ M\ M'.$$

$$\forall M, K, M', V. \{\texttt{of'}\ M\ \texttt{tnat}\} \supset \{\texttt{ch}\ M\ (\texttt{htm}\ FE\ M')\} \supset$$
$$\{\texttt{eval'}\ M\ (\texttt{nat'}\ N)\} \supset \{\texttt{eval''}\ (\texttt{htm}\ FE\ M')\ (\texttt{nat'}\ N)\}$$

Chapter 8

Completing the Compilation Process

After the CPS transformation, closure conversion and code hoisting, the higher-order functional programs have been transformed into a first-order form in which all the functions are closed and in a flat space at the top-level. It is now easy to transform programs in this form into an intermediate language to which conventional techniques for compiling imperative languages can be applied. We call this transformation the *code generation* phase. The intermediate language we choose is closely related to Cminor [43], the input language for the back-end of the well-known CompCert compiler. Our investigation of verified compilation of functional languages stops after code generation because for implementation and verification of the transformations after code generation: 1) the higher-order abstract syntax does not have obvious benefits since they only deal with programs in first order forms and do not perform complicated manipulation of bindings anymore; 2) they have been studied extensively in the area of verifying compilers for imperative languages.

In Section 8.1, we give a rule-based relational description of the code generation phase and discuss its implementation and verification using our framework. In Section 8.2, we discuss how the semantics preservation proofs for the individual phases are composed to form the correctness proof for the full compiler, which concludes our exercise on verified compilation of functional languages.

8.1 The Code Generation Transformation

The code generation transformation takes the output of code hoisting and generates programs in a Cminor-like language in which every operation is applied to variables or constants and the allocation of pairs and closures and accesses to their elements are made explicit. In the following successive subsections, we shall introduce the target language of code generation, the memory model of the target language, the rule-based description of the transformation, its implementation in λProlog, its informal verification and its verification in Abella.

8.1.1 The target language of the transformation

The target language of the code generation transformation, which is also the target language of our compiler, is a typeless language. Its syntax is given in Figure 8.1. We use x to denote variables, n to denote natural numbers, C to denote constants or variables, E to denote expressions, S to denote statements which represent a sequence of operations ending with an expression, F to denote functions and P to denote programs. Note that P has the form of programs that result from code hoisting. That is, every function resides at the top-level and takes two arguments where the first argument refers to a sequence of functions it depends on and the second argument is the actual argument of the function. Note also that we use nested let expressions to represent a sequence of instructions in statements.

$$
\begin{array}{lcl}
C & ::= & n \mid x \mid () \\
E & ::= & C \mid \mathbf{pred}\ C \mid C_1 + C_2 \mid \mathbf{if}\ C\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \mid \\
 & & (C_1\ C_2) \mid \mathbf{alloc}\ n \mid \mathbf{move}\ C_1\ n\ C_2 \mid \mathbf{load}\ C\ n \\
S & ::= & E \mid \mathbf{let}\ x = E\ \mathbf{in}\ S \\
\\
F & ::= & \lambda \vec{f}.\, \lambda x.\, S \\
P & ::= & \mathbf{letf}\ \vec{f} = \vec{F}\ \mathbf{in}\ S
\end{array}
$$

Figure 8.1: The Syntax of the Target Language of Code Generation

8.1.2 The memory model of the target language

The only interesting components of the syntax in Figure 8.1 are expressions for manipulating memory, *i.e.*, (**alloc** n) for memory allocation, (**move** C_1 n C_2) for writing to memory and (**load** C n) for reading from memory. To understand them, we need to understand the memory model of the language. We assume a very simple model of memory in which there is a single heap consisting of an infinite number of memory cells indexed by natural numbers. The memory cells are allocated from lower to higher indexes[1]. We use i_{max} to represent the index such that the cells indexed by natural numbers up to (but not including) i_{max} are allocated and accessible. We use the expression (**loc** i) to represent a reference to the memory cell indexed by i. The instruction (**alloc** n) is used to allocate the n cells indexed from i_{max} to $(i_{max} + n - 1)$. Letting C_1 be (**loc** i) for some i and C_2 be a value, (**move** C_1 n C_2) is used to assign C_2 to the memory cell at $i + n$ and (**load** C_1 n) is used to read the value in that cell.

8.1.3 A rule-based description of the transformation

We first describe how terms not containing functions in the source language are translated into statements in the target language. We identify the relation $\rho \rhd_s M; K \leadsto_{cg} S'$ such that it holds if M is a source term whose free variables are contained in the set of variables ρ and that does not contain functions, K is a context or continuation of the form $\hat{\lambda} x.\, S$, and S' is a statement resulting from applying code generation to M in the context K. The transformation rules of $\rho \rhd_s M; K \leadsto_{cg} S'$ are given in Figure 8.2. The rules defining this relation are very similar to those of the CPS transformation (described in Figure 5.3): they recursively translate the sub-expressions of the source program and accumulates the results in K in an order that reflects the control flow of the source program; the substitution operations in the transformation are represented by administrative β-redexes like in the CPS transformation. The rules cg-pair, cg-plus and cg-app have the side condition that x_1 does not occur in ρ, M_2 and K. The rule cg-open has the side condition that x_f,x_e and x_1 do not occur in ρ, M_2 and K. The cg-pair rule translates a pair expression into a program fragment for explicitly allocating two consecutive memory cells and assigning the elements of the pair to the cells. The

[1]We do not consider garbage collection in this thesis. It is left for future work

$$\frac{}{\rho \rhd_s n; K \rightsquigarrow_{cg} @\ K\ n}\ \text{cg-nat} \qquad \frac{x \in \rho}{\rho \rhd_s x; K \rightsquigarrow_{cg} @\ K\ x}\ \text{cg-var}$$

$$\frac{}{\rho \rhd_s (); K \rightsquigarrow_{cg} @\ K\ ()}\ \text{cg-unit}$$

$$\frac{\rho \rhd_s M; \hat{\lambda} x.\, \textbf{let}\ v = \textbf{pred}\ x\ \textbf{in}\ (@\ K\ v) \rightsquigarrow_{cg} S}{\rho \rhd_s \textbf{pred}\ M; K \rightsquigarrow_{cg} S}\ \text{cg-pred}$$

$$\frac{\rho \rhd_s M_2; \hat{\lambda} x_2.\, \textbf{let}\ v = x_1 + x_2\ \textbf{in}\ (@\ K\ v) \rightsquigarrow_{cg} S \quad \rho \rhd_s M_1; \hat{\lambda} x_1.\, S \rightsquigarrow_{cg} S'}{\rho \rhd_s M_1 + M_2; K \rightsquigarrow_{cg} S'}\ \text{cg-plus}$$

$$\frac{\rho \rhd_s M_2; K' \rightsquigarrow_{cg} S \quad \rho \rhd_s M_1; \hat{\lambda} x_1.\, S \rightsquigarrow_{cg} S'}{\rho \rhd_s (M_1, M_2); K \rightsquigarrow_{cg} S'}\ \text{cg-pair}$$

where $K' = (\hat{\lambda} x_2.\, \textbf{let}\ p = \textbf{alloc}\ 2\ \textbf{in}\ \textbf{let}\ v_1 = \textbf{move}\ p\ 0\ x_1\ \textbf{in}\ \textbf{let}\ v_2 = \textbf{move}\ p\ 1\ x_2\ \textbf{in}\ (@\ K\ p))$

$$\frac{\rho \rhd_s M; \hat{\lambda} x.\, \textbf{let}\ v = \textbf{load}\ x\ 0\ \textbf{in}\ (@\ K\ v) \rightsquigarrow_{cg} S}{\rho \rhd_s \textbf{fst}\ M; K \rightsquigarrow_{cg} S}\ \text{cg-fst}$$

$$\frac{\rho \rhd_s M; \hat{\lambda} x.\, \textbf{let}\ v = \textbf{load}\ x\ 1\ \textbf{in}\ (@\ K\ v) \rightsquigarrow_{cg} S}{\rho \rhd_s \textbf{snd}\ M; K \rightsquigarrow_{cg} S}\ \text{cg-snd}$$

$$\frac{\begin{array}{c}\rho \rhd_s M_2; \hat{\lambda} x.\, x \rightsquigarrow_{cg} S_2 \qquad\qquad \rho \rhd_s M_3; \hat{\lambda} x.\, x \rightsquigarrow_{cg} S_3 \\ \rho \rhd_s M_1; \hat{\lambda} x_1.\, \textbf{let}\ a = (\textbf{if}\ x_1\ \textbf{then}\ S_2\ \textbf{else}\ S_3)\ \textbf{in}\ (@\ K\ a) \rightsquigarrow_{cg} S\end{array}}{\rho \rhd_s \textbf{if}\ M_1\ \textbf{then}\ M_2\ \textbf{else}\ M_3; K \rightsquigarrow_{cg} S}\ \text{cg-if}$$

(provided k does not occur in ρ, M_2 and M_3)

$$\frac{\rho, x \rhd_s M_2; K \rightsquigarrow_{cg} S \quad \rho \rhd_s M_1; \hat{\lambda} x.\, S \rightsquigarrow_{cg} S'}{\rho \rhd_s \textbf{let}\ x = M_1\ \textbf{in}\ M_2; K \rightsquigarrow_{cg} S'}\ \text{cg-let}$$

(provided x does not occur in ρ and K)

$$\frac{\rho \rhd_s M_2; \hat{\lambda} x_2.\, \textbf{let}\ v = x_1\ x_2\ \textbf{in}\ (@\ K\ v) \rightsquigarrow_{cg} S \quad \rho \rhd_s M_1; \hat{\lambda} x_1.\, S \rightsquigarrow_{cg} S'}{\rho \rhd_s M_1\ M_2; K \rightsquigarrow_{cg} S'}\ \text{cg-app}$$

$$\frac{\rho \rhd_s (M_1, M_2); K \rightsquigarrow_{cg} S'}{\rho \rhd_s \langle M_1, M_2 \rangle; K \rightsquigarrow_{cg} S'}\ \text{cg-clos}$$

$$\frac{\rho \rhd_s M_2; K_1 \rightsquigarrow_{cg} S \quad \rho \rhd_s M_1; K_2 \rightsquigarrow_{cg} S'}{\rho \rhd_s \textbf{open}\ \langle x_f, x_e \rangle = M_1\ \textbf{in}\ x_f\ (M_1, M_2, x_e); K \rightsquigarrow_{cg} S'}\ \text{cg-open}$$

where $K_1 = (\hat{\lambda} x_2.\, \textbf{let}\ p_1 = \textbf{alloc}\ 2\ \textbf{in}\ \textbf{let}\ v_1 = \textbf{move}\ p_1\ 0\ x_2\ \textbf{in}\ \textbf{let}\ v_2 = \textbf{move}\ p_1\ 1\ x_e\ \textbf{in}$
$\textbf{let}\ p_2 = \textbf{alloc}\ 2\ \textbf{in}\ \textbf{let}\ v_3 = \textbf{move}\ p_2\ 0\ x_1\ \textbf{in}\ \textbf{let}\ v_4 = \textbf{move}\ p_2\ 1\ p_1\ \textbf{in}$
$\textbf{let}\ v = x_f\ p_2\ \textbf{in}\ @\ K\ v)$
and $K_2 = (\hat{\lambda} x_1.\, \textbf{let}\ x_f = \textbf{load}\ x_1\ 0\ \textbf{in}\ \textbf{let}\ x_e = \textbf{load}\ x_1\ 1\ \textbf{in}\ S)$

Figure 8.2: The Rules for Generating Statements

cg-fst and cg-snd rules translate the access of elements of a pair into instructions for loading values from memory cells allocated for the pair. The cg-clos rule transforms the closures like pairs. The cg-open rule makes explicit the selection of function and environment parts from closures and the application of the function part to its arguments by using the memory operators.

We then describe the transformation of programs and functions in the source language. Let M be a program resulting from code hoisting, *i.e.*, M has the form $(\textbf{letf}\ \vec{f} = \vec{F}\ \textbf{in}\ M')$ where M' does not contain any function and $\vec{F}$ is a sequence of functions of the form $\lambda\vec{f}.\,\lambda x.\,M''$ where M'' does not contain any function. We identify the relation $\rho \rhd_p M \rightsquigarrow_{cg} P$ such that it holds if P is the result of applying code generation to M. We also identify the relation $\rho \rhd_f F \rightsquigarrow_{cg} F'$ such that it holds if F is a function extracted by code hoisting (*i.e.*, F has the form $\lambda\vec{f}.\,\lambda x.\,M''$ where M'' does not contain any function) whose free variables are contained in ρ and F' is the result of applying code generation to F. The rules defining $\rho \rhd_p M \rightsquigarrow_{cg} P$ and $\rho \rhd_f F \rightsquigarrow_{cg} F'$ are given in Figure 8.3. Note that the two rules make use of $\rho \rhd_s M; K \rightsquigarrow_{cg} S'$ where K is an identity context for generating statements from the bodies of functions and programs.

$$\frac{\rho, \vec{f}, x \rhd_s M; \hat{\lambda}x.\,x \rightsquigarrow_{cg} S}{\rho \rhd_f (\lambda\vec{f}.\,\lambda x.\,M) \rightsquigarrow_{cg} (\lambda\vec{f}.\,\lambda x.\,S)}\ \text{cg-abs}$$

$$\text{(provided } x \text{ do not occur in } \rho \text{ and } M)$$

$$\frac{\{\rho \rhd_f F_i \rightsquigarrow_{cg} F'_i \mid 1 \le i \le n\} \quad \rho, \vec{f} \rhd_s M; \hat{\lambda}x.\,x \rightsquigarrow_{cg} S}{\rho \rhd_p (\textbf{letf}\ \vec{f} = \vec{F}\ \textbf{in}\ M) \rightsquigarrow_{cg} (\textbf{letf}\ \vec{f} = \vec{F'}\ \textbf{in}\ S)}\ \text{cg-prog}$$

$$\text{(where } f = (f_1, \ldots, f_n),\ \vec{F} = (F_1, \ldots, F_n) \text{ and } \vec{F'} = (F'_1, \ldots, F'_n))$$

Figure 8.3: The Rules for Generating Programs and Functions

8.1.4 Implementing the transformation in λProlog

We first consider encoding terms in the target language. We use `tm'` to represent the type of terms in the target language. For the constructors of the target language that also occur in the source language, we reuse the constants for encode them in the source language. For example, `pred'` is used to encode **pred** C and `pair'` is used to encode

(C_1, C_2). We further introduce the constants `loc'` : `nat` $\to$ `tm'`, `alloc'` : `nat` $\to$ `tm'`, `move'` : `tm'` $\to$ `nat` $\to$ `tm'` $\to$ `tm'` and `load'` : `tm'` $\to$ `nat` $\to$ `tm'` for encoding the expressions for manipulating the memory.

We then describe the encoding of the transformation relation rules. We identify the following predicate constant to represent the transformation relation $\rho \triangleright_s M; K \rightsquigarrow_{cg} S'$:

$$\texttt{cgen} : \texttt{tm'} \to (\texttt{tm'} \to \texttt{tm'}) \to \texttt{tm'} \to \texttt{o}.$$

Similar to the encoding of the CPS transformation, we use meta-level β-redexes to represent administrative β-redexes. The transformation rules in Figure 8.2 transparently translate into clauses for `cgen`. Most of the clauses have similar structures as those for the CPS transformation. We only talk about the interesting ones, including the clauses representing cg-pair, cg-fst, cg-snd, cg-clos and cg-open listed as follows:

```
cgen (pair' M1 M2) K M' :-
  (Πx1.cgen M2
          (x2 \ let' (alloc' (s (s z)))
            (p \ let' (move' p z x1)
            (u1 \ let' (move' p (s z) x2)
            (u2 \ K p))))
          (M2' x1)) ,
  cgen M1 M2' M'.
cgen (fst' M) K M' :- cgen M (x \ let' (load' x z) (v \ K v)) M'.
cgen (snd' M) K M' :- cgen M (x \ let' (load' x (s z)) (v \ K v)) M'.
cgen (clos' M1 M2) K M' :- cgen (pair' M1 M2) K M'.
cgen (open' M1 (f \ e \ app' f (pair' M1 (pair' M2 e)))) K M' :-
  (Πf, e, x1.cgen M2
            (x2 \ let' (alloc' (s (s z)))
              (p1 \ let' (move' p1 z x2)
              (v1 \ let' (move' p1 (s z) e)
              (v2 \ let' (alloc' (s (s z)))
              (p2 \ let' (move' p2 z x1)
              (v3 \ let' (move' p2 (s z) p1)
              (v4 \ let' (app' f p2) (v \ K v))))))))
```

$(M_3\ f\ e\ x_1))\,,$

$\texttt{cgen}\ M_1\ (x_1 \setminus \texttt{let'}\ (\texttt{load'}\ x_1\ \texttt{z})\ (f \setminus \texttt{let'}\ (\texttt{load'}\ x_1\ (\texttt{s}\ \texttt{z}))\ (e \setminus M_3\ f\ e\ x_1))).$

In the clause encoding cg-pair, we use the terms of `alloc'`, `move'` embedded in let expressions to represent the sequence of instructions for allocating memory cells for a pair and initializing the cells with elements of the pair. Similarly in the clause encoding cg-fst and cg-snd, we use terms of `load'` to represent the instructions for reading values from the cells for the pairs. The clauses encoding cg-clos and cg-open are direct translations from the corresponding rules. As usual, the freshness side conditions of these rules are captured via universal goals.

We identify the following predicate constants to represent the transformation relations $\rho \rhd_f F \leadsto_{cg} F'$ and $\rho \rhd_p M \leadsto_{cg} P$:

$\texttt{ecgen} : \texttt{list tm'} \to \texttt{list tm'} \to \texttt{o}.$

$\texttt{bcgen} : \texttt{tm'} \to \texttt{tm'} \to \texttt{o}.$

$\texttt{hcgen} : \texttt{tm'} \to \texttt{tm'} \to \texttt{o}.$

Their definitions are given as follows:

$\texttt{ecgen nil nil}.$

$\texttt{ecgen}\ ((\texttt{abs'}\ l \setminus \texttt{abs'}\ x \setminus F\ l\ x)\ \texttt{::}\ FE)\ ((\texttt{abs'}\ l \setminus \texttt{abs'}\ x \setminus F'\ l\ x)\ \texttt{::}\ FE')\ \texttt{:-}$
$\quad (\Pi l, x.(\Pi k.\texttt{cgen}\ l\ k\ (k\ l)) \Rightarrow (\Pi k.\texttt{cgen}\ x\ k\ (k\ x)) \Rightarrow$
$\qquad \texttt{cgen}\ (F\ l\ x)\ (x \setminus x)\ (F'\ l\ x))\,, \texttt{ecgen}\ FE\ FE'.$

$\texttt{bcgen}\ (\texttt{hbase}\ M)\ (\texttt{hbase}\ M')\ \texttt{:-}\ \texttt{cgen}\ M\ (x \setminus x)\ M'.$

$\texttt{bcgen}\ (\texttt{habs}\ R)\ (\texttt{habs}\ R')\ \texttt{:-}\ \Pi x.(\Pi k.\texttt{cgen}\ x\ k\ (k\ x)) \Rightarrow \texttt{bcgen}\ (R\ x)\ (R'\ x).$

$\texttt{hcgen}\ (\texttt{htm}\ FE\ M)\ (\texttt{htm}\ FE'\ M')\ \texttt{:-}\ \texttt{ecgen}\ FE\ FE'\,, \texttt{bcgen}\ M\ M'.$

As usual, we use universal and hypothetical goals to preform recursion over binding operators and to introduce the rules for transforming variables. By definition `hcgen` M P holds if M encodes code that has the form of outputs of code hoisting and P encodes the result of apply code generation to M. The constant `ecgen` represents the transformation of the top-level functions and `bcgen` represents the transformation of the body of M. Note that in the base case of `bcgen` we use `cgen` with the initial context $(x \setminus x)$ to generate the target code from the body of the program. Similarly, we use `cgen` in the definition of `ecgen` to generate code from the bodies of functions.

8.1.5 Informal verification of the transformation

Because code generation closely corresponds to the CPS transformation, its verification is carried out in a way similar to the CPS transformation. The only interesting part of the verification is to identify an operational semantics for the target language of code generation. In the rest of this section, we give the rule-based description of this semantics and its encoding and briefly talk about the verification of code generation using this semantics.

We first describe the representation of and the operations on heaps. We use H to denote heaps. A heap with no allocated cells is written as $\emptyset$. A heap with cells indexed from 0 to n allocated such that V_i is the content of the i-th cell is written as $\{0 \to V_0, 1 \to V_1, \ldots, n \to V_n\}$. We define the following operations for allocation, modification and query of the contents of a heap:

- The function *allocate* for allocating memory such that $H' = \mathit{allocate}(H, i)$ if n is the smallest index of the unallocated cells in H and H' is H extended with the mappings $\{n \to (), \ldots, n + i - 1 \to ()\}$. That is, $\mathit{allocate}(H, i)$ allocates i fresh memory cells in H and set their default values to $()$.

- The function *update* for updating the content of a memory cell such that $H' = \mathit{update}(H, i, V)$ if H contains a mapping for i and H' is obtained from H by replacing this mapping with $(i \to V)$.

- The function *lookup* for reading the content of a memory cell such that $V = \mathit{lookup}(H, i)$ if H contains the mapping $(i \to V)$.

We identify the following classes of terms for describing evaluation where V denotes values, B denotes values or variables and D and U respectively denote the intermediate forms of expressions and statements in the evaluation process.

$$
\begin{array}{lcl}
V & ::= & n \mid () \mid \mathbf{loc}\, n \mid \lambda x.\, U \\
B & ::= & V \mid x \\
D & ::= & B \mid \mathbf{pred}\, B \mid B_1 + B_2 \mid \mathbf{if}\ B\ \mathbf{then}\ U_1\ \mathbf{else}\ U_2 \mid \\
 & & (B_1\ B_2) \mid \mathbf{alloc}\, n \mid \mathbf{move}\, B_1\, n\, B_2 \mid \mathbf{load}\, B\, n \\
U & ::= & D \mid \mathbf{let}\ x = U\ \mathbf{in}\ U
\end{array}
$$

We now describe an operational semantics based on a left-to-right, call-by-value evaluation strategy and in a small-step form. One step of evaluation takes a statement U and a heap H as input, evaluates the first instruction in U based on the content of H and outputs the modified statement and heap. The changes to the heap reflects the side effects incurred by the evaluation. We write $H, M \hookrightarrow_1 H', M'$ where M is either an intermediate expression or statement to denote that M evaluates to M' in one step and the evaluation changes the heap H to H'. We write $H, M \hookrightarrow_n H', M'$ where $n > 0$ to denote the M evaluates to M' in n steps and H is changed to H' by the evaluation and $P \hookrightarrow H, V$ to denote that a program P evaluates to the value V starting with an empty heap and the final memory state is represented by H. Figure 8.4 depicts the rules defining these relations.

$$\frac{n' \text{ is the predecessor of } n}{H, \textbf{pred}\ n \hookrightarrow_1 H, n'} \qquad \frac{n_3 \text{ is the sum of } n_1 \text{ and } n_2}{H, n_1 + n_2 \hookrightarrow_1 H, n_3}$$

$$\frac{}{H, \textbf{if}\ 0\ \textbf{then}\ M_1\ \textbf{else}\ M_2 \hookrightarrow_1 H, M_1} \qquad \frac{n > 0}{H, \textbf{if}\ n\ \textbf{then}\ M_1\ \textbf{else}\ M_2 \hookrightarrow_1 H, M_2}$$

$$\frac{i = \text{the number of allocated cells in } H \quad H' = \mathit{allocate}(H, n)}{H, \textbf{alloc}\ n, H \hookrightarrow_1 H', \textbf{loc}\ i}$$

$$\frac{H' = \mathit{update}(H, (n+i), V)}{H, \textbf{move}\ (\textbf{loc}\ n)\ i\ V \hookrightarrow_1 H', ()} \qquad \frac{V = \mathit{lookup}(H, (n+i))}{H, \textbf{load}\ (\textbf{loc}\ n)\ i \hookrightarrow_1 H, V}$$

$$\frac{M_1, H \hookrightarrow_1 M_1', H'}{H, \textbf{let}\ x = M_1\ \textbf{in}\ M_2 \hookrightarrow_1 H', \textbf{let}\ x = M_1'\ \textbf{in}\ M_2}$$

$$\frac{}{H, \textbf{let}\ x = V\ \textbf{in}\ M \hookrightarrow_1 H, M[V/x]} \qquad \frac{}{H, (\lambda x. U)\ V \hookrightarrow_1 H, U[V/x]}$$

$$\frac{}{H, M \hookrightarrow_0 H, M} \qquad \frac{H, M \hookrightarrow_1 H', M' \quad H', M' \hookrightarrow_n H'', M''}{H, M \hookrightarrow_{n+1} H'', M''}$$

$$\frac{\emptyset, S[\vec{F}/\vec{f}] \hookrightarrow_n H, V}{\textbf{letf}\ \vec{f} = \vec{F}\ \textbf{in}\ S \hookrightarrow H, V}$$

Figure 8.4: Evaluation Rules for the Target Language of Code Generation

Given the evaluation semantics, we can prove that code generation preserves semantics by following a development similar to that for the CPS transformation. In the end, we can prove the following theorem that code generation on statements preserves semantics:

Theorem 8. *If* $\vdash M : \mathbb{N}$*,* $\emptyset \triangleright_s M; K \rightsquigarrow_{cg} M'$ *and* $M \hookrightarrow V$*, then* $\emptyset, M' \hookrightarrow_n H, (@\ K\ V)$ *for some* n *and* H*.*

From this theorem, it is easy to derive the following theorem which states code generation on programs preserves semantics:

Theorem 9. *If* $\vdash P : \mathbb{N}$*,* $\emptyset \triangleright_p P \rightsquigarrow_{cg} P'$ *and* $P \hookrightarrow V$*, then* $P' \hookrightarrow H, V$ *for some* H*.*

8.1.6 Verifying the λProlog implementation in Abella

We first formalize the representation of and operations on memories in λProlog. We represent a heap as a list of the type (`list` (`map nat tm'`)). It contains the mappings from indexes to the values for the allocated cells. We use the type `state` to represent the memory state. The sole constructor of `state` is `st` : `nat` → (`list` (`map nat tm'`)) → `state` which takes a heap H and the smallest index of the allocated cells in H to form a memory state. We then identify the following predicate constants to represent the allocation, update and look-up operations on heaps:

> `allocate` : `nat` → `nat` → (`list` (`map nat tm'`)) → (`list` (`map nat tm'`)) → `o`.
> `update_heap` : (`list` (`map nat tm'`)) → `nat` → `tm'` → (`list` (`map nat tm'`)) → `o`.
> `lookup_heap` : (`list` (`map nat tm'`)) → `nat` → `tm'` → `o`.

They are defined as follows:

> `allocate` N `z` H H.
> `allocate` N (`s` S) H H' `:-` `allocate` (`s` N) S (`map` N `unit` `::` H) H'.
> `update_heap` (`map` L V `::` H) L V' (`map` L V' `::` H).
> `update_heap` (M `::` H) L V' (M `::` H') `:-` `update_heap` H L V' H'.
> `lookup_heap` (`map` L V `::` H) L V.
> `lookup_heap` (M `::` H) L V `:-` `lookup_heap` H L V.

We then formalize the evaluation of expressions and statements. We identify the following constants to represent the evaluation relations:

$\texttt{step''} : \texttt{state} \to \texttt{tm'} \to \texttt{state} \to \texttt{tm'} \to \texttt{o}$

$\texttt{nstep''} : \texttt{nat} \to \texttt{state} \to \texttt{tm'} \to \texttt{state} \to \texttt{tm'} \to \texttt{o}.$

$\texttt{eval'''} : \texttt{tm} \to \texttt{state} \to \texttt{tm'} \to \texttt{o}.$

The constant `step''` is used to represent the one-step evaluation relation such that `step''` $St\ M\ St'\ M'$ if and only if the term M evaluates to M' in the state St and St is changed to St' by the evaluation, `nstep''` is used to represent the n-step evaluation relation, and `eval'''` is used to represent the evaluation of programs such that `eval'''` $P\ H\ V$ holds if the program P in an empty heap evaluates to V and the final heap is H. We summarize the clauses defining these constants which are transparently translated from the rules in Figure 8.4 follows:

$\texttt{step''}\ St\ (\texttt{pred'}\ (\texttt{nat'}\ N)) St\ (\texttt{nat'}\ N')\ \texttt{:-}\ \texttt{npred}\ N\ N'.$

$\texttt{step''}\ St\ (\texttt{plus'}\ (\texttt{nat'}\ N_1)\ (\texttt{nat'}\ N_2))\ St\ (\texttt{nat'}\ N)\ \texttt{:-}\ \texttt{add}\ N_1\ N_2\ N.$

$\texttt{step''}\ St\ (\texttt{ifz'}\ (\texttt{nat'}\ \texttt{z})\ M_1\ M_2)\ St\ M_1.$

$\texttt{step''}\ St\ (\texttt{ifz'}\ (\texttt{nat'}\ (\texttt{s}\ N))\ M_1\ M_2) St\ M_2.$

$\texttt{step''}\ (\texttt{st}\ N\ H)\ (\texttt{alloc'}\ S)\ (\texttt{st}\ N'\ H')\ (\texttt{loc'}\ N)\ \texttt{:-}$
  $\texttt{add}\ N\ S\ N', \texttt{allocate}\ N\ S\ H\ H'.$

$\texttt{step''}\ (\texttt{st}\ N\ H)\ (\texttt{move'}\ (\texttt{loc'}\ L)\ S\ V)\ (\texttt{st}\ N\ H')\ \texttt{unit}\ \texttt{:-}$
  $\texttt{add}\ L\ S\ L', \texttt{update\_heap}\ H\ L'\ V\ H'.$

$\texttt{step''}\ (\texttt{st}\ N\ H)\ (\texttt{load'}\ (\texttt{loc'}\ L)\ S)\ (\texttt{st}\ N\ H)\ V\ \texttt{:-}$
  $\texttt{add}\ L\ S\ L', \texttt{lookup\_heap}\ H\ L'\ V.$

$\texttt{step''}\ St\ (\texttt{let'}\ M\ R)\ St'(\texttt{let'}\ M'\ R)\ \texttt{:-}\ \texttt{step''}\ St\ M St'M'.$

$\texttt{step''}\ St\ (\texttt{let'}\ V\ R)\ St\ (R\ V)\ \texttt{:-}\ \texttt{val''}\ V.$

$\texttt{step''}\ St\ (\texttt{app'}\ (\texttt{abs'}\ R)\ V)\ St\ (R\ V)\ \texttt{:-}\ \texttt{val''}\ V.$

$\texttt{nstep''}\ \texttt{z}\ St\ M\ St\ M.$

$\texttt{nstep''}\ (\texttt{s}\ N)\ St\ M\ St''\ M''\ \texttt{:-}$
  $\texttt{step''} St\ M\ St'\ M', \texttt{nstep''}\ N\ St'M'\ St''M''.$

$\texttt{eval'''}\ (\texttt{htm}\ \texttt{nil}\ (\texttt{hbase}\ M))\ St\ V\ \texttt{:-}\ \texttt{nstep''}\ N\ (\texttt{st}\ \texttt{z}\ \texttt{nil})\ M\ St\ V, \texttt{val''}\ V.$

$\texttt{eval'''}\ (\texttt{htm}\ (F :: FE)\ (\texttt{habs}\ R))\ St\ V\ \texttt{:-}\ \texttt{eval'''}\ (\texttt{htm}\ FE\ (R\ F))\ St\ V.$

Here the predicate constant `val''` : `tm'` $\to$ `o` is used to identify values in the target language.

We formalize the informal development of semantics preservation proofs in Abella by following a way similar to that in which we develop the formal proof of the CPS transformation. In the end, Theorem 8 is proved as the following theorem in Abella:

$$\forall M, K, M', V. \{\texttt{of'}\ M\ \texttt{tnat}\} \supset \{\texttt{cgen}\ M\ K\ M'\} \supset \{\texttt{eval'}\ M\ V\} \supset$$
$$\exists N, St. \{\texttt{step''}\ N\ (\texttt{st}\ \texttt{z}\ \texttt{nil})\ M'\ St\ (K\ V)\}.$$

From this theorem it is easy to prove the following formalized version of Theorem 9:

$$\forall P, P', V. \{\texttt{of''}\ P\ \texttt{tnat}\} \supset \{\texttt{hcgen}\ P\ P'\} \supset \{\texttt{eval''}\ P\ V\} \supset$$
$$\exists St. \{\texttt{eval'''}\ P'\ St\ V\}.$$

8.2 The Correctness Proof for the Full Compiler

We designate the constant $\texttt{compile} : \texttt{tm} \to \texttt{tm'} \to \texttt{o}$ to represent the full compilation process. It is defined by the following program clause in λProlog:

$$\texttt{compile}\ M\ M'\ \texttt{:-}$$
$$\texttt{cps}\ M\ (x \setminus x)\ M_1\,, \texttt{cc}\ \texttt{nil}\ \texttt{nil}\ M_1\ M_2\,,$$
$$\texttt{ch}\ M_2\ (\texttt{htm}\ FE\ M_3)\,, \texttt{hcgen}\ (\texttt{htm}\ FE\ M_3)\ M'.$$

To prove the correctness of the full compiler, we need the type and semantics preservation theorems of its individual transformations. We summarize the type preservation theorems corresponding to Corollaries 3, 7 and 10 as follows:

$$\forall M, M'. \{\texttt{of}\ M\ \texttt{tnat}\} \supset \{\texttt{cps}\ M\ (x \setminus x)\ M'\} \supset \{\texttt{of}\ M'\ \texttt{tnat}\}.$$
$$\forall M, M', T. \{\texttt{of}\ M\ T\} \supset \{\texttt{cc}\ \texttt{nil}\ \texttt{nil}\ M\ M'\} \supset \{\texttt{of'}\ M'\ T\}.$$
$$\forall M, M', T. \{\texttt{of'}\ M\ T\} \supset \{\texttt{ch}\ M\ M'\} \supset \{\texttt{of''}\ M'\ T\}.$$

Note that the code generation transformation does not preserve types because its target language is typeless. We summarize the semantics preservation theorems corresponding to Corollaries 6, 9, 12 and 9 as follows:

$$\forall M, M', V. \{\texttt{of}\ M\ \texttt{tnat}\} \supset \{\texttt{cps}\ M\ (x \setminus x)\ M'\} \supset$$
$$\{\texttt{eval}\ M\ V\} \supset \{\texttt{eval}\ M'\ V\}.$$
$$\forall K, T, M, M'. \{\texttt{of}\ M\ \texttt{tnat}\} \supset \{\texttt{cc}\ \texttt{nil}\ \texttt{nil}\ M\ M'\} \supset$$
$$\{\texttt{eval}\ M\ (\texttt{nat}\ N)\} \supset \{\texttt{eval'}\ M'\ (\texttt{nat'}\ N)\}$$

$$\forall M, K, M', V.\{\texttt{of'}\ M\ \texttt{tnat}\} \supset \{\texttt{ch}\ M\ (\texttt{htm}\ \mathit{FE}\ M')\} \supset$$
$$\{\texttt{eval'}\ M\ (\texttt{nat'}\ N)\} \supset \{\texttt{eval''}\ (\texttt{htm}\ \mathit{FE}\ M')\ (\texttt{nat'}\ N)\}$$
$$\forall P, P', V.\{\texttt{of''}\ P\ \texttt{tnat}\} \supset \{\texttt{hcgen}\ P\ P'\} \supset \{\texttt{eval''}\ P\ V\} \supset$$
$$\exists \mathit{St}.\{\texttt{eval'''}\ P'\ \mathit{St}\ V\}.$$

The correctness theorem of the full compiler is stated as follows:

$$\forall M, M', N.\{\texttt{of}\ M\ \texttt{tnat}\} \supset \{\texttt{compile}\ M\ M'\} \supset \{\texttt{eval}\ M\ (\texttt{nat}\ N)\} \supset$$
$$\exists \mathit{St}.\{\texttt{eval'''}\ M'\ \mathit{St}\ (\texttt{nat'}\ N)\}$$

It is proved by analyzing $\{\texttt{compile}\ M\ M'\}$ and interleaving the applications of the above type and semantics preservation theorems in an obvious way.

Chapter 9

Related Work

Compiler verification is an old topic, interest in which can be traced back to 1960s [1]. The past decade has witnessed impressive developments on mechanizing compiler verification, due partly to the maturation of formal verification tools. Many of these developments have focused on implementing and verifying compilers for imperative programming languages such as C, C++ and Java; see [2] for a catalog of these developments. Among these efforts, the most notable and influential has been that of the CompCert project that has developed a verified multi-pass compiler for a subset of C using the Coq theorem prover [20]. The correctness of the CompCert compiler has been proved by establishing the permutability of evaluation and the compilation process as we have described in Section 4.5. As such, it only makes sense for compilation of full programs. Recently, researchers have also begun investigating the separate compilation of program modules and a composible way to verify such compilation [78]. The CompCert project has shown that verification of non-trivial compilers for realistic programming languages is feasible with the state-of-art verification tools. This project has also provided the impetus for other efforts such as the Verified Software Toolchain project [21] related to overall program verification.

Our focus in this thesis has been on the implementation and formal verification of compilers for functional programming languages. In contrast to the compilation of imperative languages, this task requires the transformation of more abstract programs into low-level executable code and brings with it the need to represent, manipulate, analyze and reason about binding structure in compiler transformations. While there are new difficulties in this area, there have also been efforts targeted at overcoming them. These efforts have ranged from verifying individual compiler transformations to ones with a much more ambitious scope such as the verification of complete compilers for realistic functional languages. We discuss these efforts in the rest of this chapter, paying particular attention to how they deal with binding structure and how this impacts on the development of proofs. Because there are significant differences in scope and focus and in the theories and tools used in implementation and verification, a direct comparison of our work with many of the projects described below is neither feasible nor sensible. However, we do try to make comparative assessments where this seems possible and sensible.

A large part of the existing work on verified compilation of functional languages makes use of general theorem provers such as Coq [87], Isabelle [18] and HOL [88]. In [89] Dargaye describes a verified compiler for a subset of ML, which extends the simply typed λ-calculus with n-ary functions, recursive functions, data types and pattern matching. The compilation process for this language is similar to ours: it transforms source programs through multiple passes, including the CPS transformation, closure conversion and code generation, into the Cminor intermediate language of CompCert. The compiler transformations are verified by following the approach adopted by CompCert, *i.e.*, by showing the permutability between evaluation and transformation. The verification of the full compiler is obtained by utilizing the result previously established within CompCert of the correctness of the translation from Cminor to low-level code. In [22], Chlipala develops a verified multi-pass compiler for the simply typed λ-calculus in Coq by using logical relations as the notion of semantics preservation. In a manner similar to what we did in Section 8.2, Chlipala composed the correctness proofs of individual transformations to prove that closed programs at atomic types have the same behavior before and after compilation. In [66], Benton and Hur have implemented a verified single-pass compiler that takes programs in an extension of PCF as input and outputs virtual code executable an SECD machine [67]. Their notion of semantics preservation is based on a step-indexing logical relation. Because logical relations satisfy the modularity property described in Section 4.5, the semantics preservation property of this compiler makes sense for program modules with external references. Moreover, Benton and Hur have not only proved semantics preservation, they have also proved that the compiler is *fully abstract*, namely two pieces of source programs are contextual equivalent if and only if their compiled versions are. As a result, properties proved about program equivalence at the source level are still valid after the compilation.

Because general theorem provers do not have built-in support for bindings, users have to implement it explicitly or use a library based on some first-order or higher-order approach to treating bindings. All of the above work uses standard de Bruijn indexes or its variant for representing bindings, bearing the burden of explicitly representing, manipulating and reasoning about binding related notions such as substitution and renaming. These difficulties are best illustrated by considering the treatment of the CPS transformation in the Danvy and Filinski style. Chlipala has used standard de Bruijn indexes to represent both the administrative and dynamic abstractions in the transformation and explicitly implemented all the binding related operations and proved their properties [22]. To alleviate the effort, Dargaye has used two kinds of de Bruijn indexes, one for representing administrative abstractions, another for dynamic abstractions [89, 90]. Still she needs to explicitly implement and reason about such binding representations. In [91], Minamide and Okuma have also formally verified a Danvy Filinski style CPS transformation using Isabelle/HOL. They argue that de Bruijn indexes are not effective for formalizing the CPS transformation because they are sensitive to the changes to the context. For example, the reduction of an administrative β-redex will cause the shifting of de Bruijn indexes. To avoid this problem they use named variables to represent abstractions and realize α-equivalence via explicit renaming. In contrast, by adopting an HOAS approach we are able to use meta-level abstractions to represent administrative abstractions, which eliminates the necessity to explicitly encode renaming and substitution and to reason about such encoding. This is also observed in [45] by Tian who has used Twelf to encode a CPS transformation and has provided a simple correctness proof for the transformation via the HOAS approach (We shall discuss compiler verification work done using Twelf more thoroughly later in the chapter.)

The most comprehensive work on verified compilers for functional languages we know of is the CakeML compiler which supports a substantial subset of Standard ML [24]. CakeML is designed to serve as a platform for running verified software with a reduced trusted computing base. The compiler is implemented as an interactive read-eval-print loop that compiles Standard ML programs into x86-64 machine code through a sequence of transformations. Its verification is done by using HOL4 and essentially based on showing permutability of evaluation and compilation. A distinguish characteristic of CakeML is that it is bootstrapped: the compiler is fed as a source program into itself to generate an x86-64 implementation of it. Moreover, it is proved that the correctness of the source compiler is preserved by the compilation. Therefore we get a verified x86-64 implementation of CakeML. CakeML represents bindings through named variables in its source language and through de Bruijn indexes in its intermediate language. As a result, all the properties about bindings must be proved explicitly based on those representations. For instance, the closure conversion transformation in CakeML represents an environment for a function as a list of de Bruijn indexes pointing to the abstractions binding the free variables in the function body. The encoding of closure conversion replaces the de Bruijn indexes of the free variables in the function body with pointers to the elements in the environment. Such manipulation of de Bruijn indexes must be reasoned about explicitly in the correctness proof of closure conversion. All this takes significant effort that is obviously orthogonal to the actual task that is of interest.

In [86] Chlipala has tried to alleviate the difficulties in dealing with bindings in general theorem provers by introducing the *Parametric Higher Order Abstract Syntax* or *PHOAS* and applied PHOAS to re-implement the verified compiler for the STLC introduced in [22]. PHOAS is a further development of an approach for treating bindings known as the *weak higher-order abstract syntax* or *weak HOAS*[92, 93]. In weak HOAS, we designate a type `var` for representing variables which is separate from the type `tm` for representing terms and we use abstractions over `var` in the meta-language to encode object-level abstractions over variables. For instance, to encode the STLC we can introduce a constructor $\mathtt{var} : \mathtt{var} \to \mathtt{tm}$ for encoding variables, $\mathtt{app} : \mathtt{tm} \to \mathtt{tm} \to \mathtt{tm}$ for encoding applications and $\mathtt{abs} : (\mathtt{var} \to \mathtt{tm}) \to \mathtt{tm}$ for encoding object-level abstractions. By embedding those constructors in an inductive definition for the type `tm`, we partially derive the benefits of the HOAS approach, such as that renaming is modeled by α-conversion. However, because variables are separated from ordinary terms, substitutions in weak HOAS cannot be represented elegantly through β-conversion like in HOAS; more specifically, the term $(M\ N)$ where M has the type $(\mathtt{var} \to \mathtt{tm})$ and N has the type `tm` is not well-typed. Instead, we must explicitly define substitutions as a relation or function that traverses a term and replaces variables with terms. Moreover, properties about substitutions must be proved explicitly using their representation. PHOAS inherits those characteristics of weak HOAS and further *parameterizes* the type `tm` and its constructors with the type of variables instead of using the fixed type `var`. As a result, it is possible to choose different representations for variables to suit the need of users. Chlipala demonstrates the usefulness of this parameterization in implementing and verifying a compiler for the STLC in [22]. For verifying compiler transformations that need to check identity of variables such as closure conversion, he instantiates the variable type with the type of natural numbers for facilitating the identity check. For this, he needs to define the well-formedness of the terms of the instantiated types and to prove that the every parametric term is well-formed under this instantiation. This incurs an extra level of complexity in reasoning about terms with bindings.

Some comparison of our compiler is possible with the above compiler for the STLC by Chlipala. Chlipala's implementation of closure conversion comprises about 400 lines of Coq code, in contrast to about 70 lines of λProlog code that are needed in our implementation. Chlipala's proof of correctness comprises about 270 lines but it benefits significantly from the automation framework that was the focus of his work; that framework is built on top of the already existing Coq libraries and consists of about 1900 lines of code. The Abella proof script runs about 1600 lines. We note that Abella has virtually no automation and no library. We also note that, in contrast to Chlipala's work, our development treats a version of the STLC that includes recursion. This necessitates the use of a step-indexed logical relation which makes the overall proof more complex.

There has also been some work on implementing and verifying compiler transformations using the functional higher-order approach; it is to be noted that the "verification" part of such work has typically been quite weak or even non-existent. Guillemette has encoded a CPS transformation in Haskell using this style [35]. 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 functional higher-order approach. Instead, he falls back to de Bruijn indexes for encoding closure conversion and proves that the transformation is type preserving [94]. Because of the inability to analyze higher-order objects in the functional higher-order approach, none of the work above has proved semantics preservation of the encoded transformations. Hickey and Nogin have proposed an implementation of closure conversion in the MetaPRL logical framework that is based on the functional higher-order approach [36]. Rather than analyzing the body of a function to compute what variables appear free in it and thereby to form a suitable environment, they take the simplistic approach of constructing the environment based on all the abstractions the function term appears under. They also do not prove the correctness of their implementation because the MetaPRL framework offers no capabilities for verification.

The earliest work related to using the HOAS approach in compiler verification seems to be that of Hannan and Pfenning in which they use Twelf (called Elf at that time) as the specification and reasoning vehicle [44]. In that paper they have implemented and verified a compiler from the STLC to a variant of the Categorical Abstract Machine [68]. The compiler consists of some very simple transformations such as conversion of λ-terms into their de Bruijn forms. Since this work, there have been investigations of more complicated compiler transformations for functional languages using Twelf. In [95] Murphy has used Twelf to verify the type preservation of the CPS transformation and closure conversion for a programming language for distributed computing based on a modal logic. In [45] Tian implemented and verified a Danvy-Filinski style CPS transformation for a slight extension of the STLC. All of the described efforts exploit the support for the HOAS approach that Twelf provides to simplify the implementation and verification of binding related operations in the transformations.

The studies of compiler transformations on functional languages using Twelf have all been based on the permutability of evaluation and transformation. As we have discussed in Section 4.5, permutability is not very flexible as a notion of semantics preservation. Recently, researchers have proposed many different notions of semantics preservation that possess most or all of the desired properties such as modularity, flexibility and transitivity as described in Section 4.5 (*e.g.*, see [25, 66, 96]). These notions are either more complex forms of logical relations or have similar characteristics as logical relations. This suggests that the ability to effectively treat logical relations in a verification framework that supports HOAS may greatly simplify the verification of compilers for functional languages based on those more powerful notions of semantic preservation. However, this idea has not been explored using Twelf, perhaps because it is not easy to encode a logical relation-style definition within it (see [97] for discussions about a way to achieve this.)

The Beluga system [6], which implements a functional programming language based on contextual modal type theory and also supports HOAS [98], overcomes some of the shortcomings of Twelf. Reasoning in Beluga is based on the idea that by type checking programs and by ensuring that they satisfy certain coverage and termination conditions, we can obtain a guarantee of their correctness with respect to the properties expressed by the types. Coupling this approach with an expressive type system that is able to express rich properties of programs results in a framework that has the potential for proving deep properties, such as semantics preservation, for the programs we write. Belanger *et al.* have shown how this idea can be used to ensure type preservation for implementations of CPS transformation, closure conversion and code hoisting in Beluga [46]. There are many similarities between the use made of HOAS in the cited work and that described in this thesis, although the property considered there is weaker than the semantics preservation that we have been concerned with here. Recently, Cave and Pientka have shown that stronger properties, such as those based on logical relations, can be reflected into the types in Beluga and shown to hold of programs through the methods it provides [99]. However, these ideas have not yet been applied to developing semantics preservation proofs of the kind discussed in this thesis in Beluga.

There has been recent work towards developing approaches to verified compilation that satisfy modularity, flexibility and transitivity at the same time. One such approach was proposed by Perconti and Ahmed in [96] based on what is called *multi-language semantics*. In particular, a “big-tent” language that encompasses all the source, target and intermediate languages of the compiler is defined and the program equivalence between different languages is defined as contextual equivalence of terms in the big-tent language. They then use logical relations to prove that the source program module which may not be closed (*i.e.*, it is a module containing external references) is contextually equivalent to its compiled code when embedded into the big-tent language. The correctness argument using this approach currently only exists on paper. Another promising approach, which we have already discussed in Section 4.5, is that based on a notion of semantics preservation called *Parametric Inter-Languages Simulation* or *PILS* [25]. Using PILS, Neis *et al.* have developed a verified multi-pass compiler called Pilsner that satisfies all the desired properties. Pilsner and its correctness proof are formalized in Coq where explicit names are used to represent bindings. At the beginning of the compilation, α-renaming is applied to establish the invariant that every variable is bound at most once. This invariant is carefully maintain by the rest of the transformations, thereby avoiding the problem of variable capturing. We conjecture that using a HOAS approach may enable a more flexible approach to compilation and may also simplify the verification effort even with a formulation of semantic correctness in the style of PILS.

Formalizing binding related notions is not unique to verified compilation of functional programs. It has been long recognized that it is an important part of mechanizing the meta-theories of formal systems involving types, programs, formulas and proofs. The POPLMark challenge [100] proposed a set of problems for measuring the strength of formalization systems in dealing with bindings. The posted solutions to those problems [101] make use of various kinds of approaches to representing binding structure. We have discussed some of these approaches in Section 1.3 in the introductory chapter.

Chapter 10

Conclusion and Future Work

This thesis has demonstrated the effectiveness of the Higher-Order Abstract Syntax approach in implementing and verifying compilers for functional programming languages. In particular, it has shown how this approach simplifies the treatment of binding related notions which is an essential part of the task. In the demonstration, we have used a framework consisting of λProlog, a language suitable for specifying formal systems in a rule-based and relational style such that the specifications are also executable programs, and Abella, a theorem proving system that provides rich mechanisms for reasoning about λProlog programs. Both λProlog and Abella support a realization of the HOAS approach known as the λ-tree syntax approach. We have proposed the following methodology for implementing and verifying compilers for functional programming languages: we implement the compiler transformations as λProlog programs and we formulate and prove the correctness of these implementations using Abella. Using this methodology we have developed a verified compiler for a representative functional programming language. In doing so we have demonstrated that the HOAS approach provided by λProlog and Abella significantly simplifies the treatment of bindings in both the implementation and verification of the compilation of functional programs, leading to formal correctness proofs that closely follow the informal ones.

In the course of the above work, we have also addressed some shortcomings of the Abella system. First, we have extended Abella so that it can be used to reason about the full class of specifications in λProlog. Second, we have built support for a schematic form of polymorphism into Abella, thereby allowing us to use more modular λProlog implementations and also to modularize proofs of correctness.

The work in this thesis leaves several interesting questions unanswered that could provide the topics for further investigations. We discuss some of these in the sections below.

10.1 Verified Compilers for Realistic Functional Languages

One natural continuation of our work is to apply our methodology to verify compiler transformations for richer functional programming languages. Such transformations often involve more complicated manipulation of bindings. For example, in [30], Minamide *et al.* have presented a description of the closure conversion transformation for a polymorphic functional language and provided a pencil-and-paper correctness proof based on logical relations. To separate the code of a polymorphic function from its context, they abstract the function not only over an environment binding the free term variables of the function but also an environment binding the free type variables. Moreover, to make the code shareable across different sites where the function is called, they require the type environment to have *translucent types* [102]. It is interesting to investigate if HOAS can be used to simplify the implementation and formal verification of such a transformation.

A more ambitious project would be to construct a verified compiler for a realistic functional programming language, such as a substantial subset of Standard ML or OCaml that supports features like data structures, polymorphism, exceptions and modules. A possible and interesting experiment would be to formalize the development of the TIL compiler in Morrisett's Ph.D. thesis [103] using our framework. The TIL compiler transforms a subset of Standard ML to Alpha assembly code through a sequence of transformations. Its key property is that it uses typed intermediate languages at all but the lowest levels of compilation. TIL is able to generate efficient code by exploiting the type information which is difficult to do if untyped languages were used instead. For example, a common technique used by compilers with untyped intermediate language for representing data structures of types instantiated from polymorphic types is to tag the data objects together with the instantiating types. To use such tagged values, they must be unpacked first. The tagging and unpacking operations are quite expensive and have significant impact on the performance of the generated code. By using a technique called *dynamic type dispatch* that exploits the type information of intermediate languages, TIL avoids tagging and unpacking data structures. It would be interesting to see how our framework can be applied to formally develop the implementation and verification of a compiler like TIL and if the HOAS approach can benefit this formal development.

10.2 Verified Implementation of Realistic Compilers

The work of this thesis is just a starting point towards showing the effectiveness of HOAS in verified implementation of realistic compilers. We have implemented and verified the core transformations for compiling functional languages to bring out the effectiveness argument. However, our compiler does not perform any kind of optimization. Consequently, the code generated by it is not very efficient. A realistic compiler should contain optimization phases besides the core transformations for generating efficient code. In the past programming language researchers have developed various optimization transformations for functional programs. Such transformations often involve complicated analysis of the structure of functional programs, especially their binding structure. An example would be *lambda shrinking* proposed by Appel and Jim [104]. Lambda shrinking inlines functions or values bound by let expressions when this operation does not increase the code size, *i.e.*, when the variable binding the value or function occurs at most once in the program (excluding the binding site) and the arguments the function applies to are variables. Such an inlining operation may expose further opportunities for inlining because it may remove occurrences of variables bound at other places. Lambda shrinking works roughly as follows: For every binding variable it computes the number of times the variable occurs in the program; It maintain and update this information as the inlining operations go until no more inlining is possible. The analysis and update carried out in lambda shrinking is quite complicated when a first-order representation of bindings is used, as indicated in [104]. As we have shown in the thesis, our framework is able to carry out non-trivial analysis on binding structure. An interesting research topic would be to investigate how our methodology could help simplify the mechanization of the optimization transformations for compiling functional languages like lambda shrinking.

10.3 Stronger Notions of Semantics Preservation

In Section 4.5 we have discussed three criteria proposed by Neis *et al.* in [25] for measuring the strength of notions of semantics preservation: modularity, flexibility and transitivity. We have also discussed why logical relations do not satisfy all three properties in that section. We would like to base our compiler verification on richer notions of semantics preservation that satisfy all these properties so that we get a stronger safety guarantee. One such notion is the Parametric Inter-Languages Simulation (PILS) proposed in [25]. The formulation of PILS in [25] has a lot of similarities to logical relations: PILS is initially defined on closed programs using their operational semantics and then extended to open terms such that they are related by PILS if and only if the results of applying any related and closed substitutions to them are related. Neis *et al.* have developed two compilers, Pilsner and Zwickel, to demonstrate the effectiveness of PILS, which according to their paper "has been a significant undertaking, involving several person-years of work and over 55,000 lines of Coq". We suspect that part of the difficulties come from how the binding related notions are handled in their formal development. For example, they maintain an invariant that every variable is bound at most once throughout the compilation and they have used operational semantics which maintain explicit environments containing bindings for free variables in evaluation instead of the more natural operational semantics that base evaluation on substitutions. It would be interesting to see how our methodology can be used to alleviate their effort. Eventually, we would like to leverage our understanding of this kind of rich notions of semantics preservation to benefit the formal verification of compilers for functional languages in realistic settings.

10.4 Further Extensions to the Framework

While the framework comprising λProlog and Abella has significant benefits in the verified implementation of compiler transformations for functional languages, its current realization has some limitations that make it insufficient for compiler verification in the more realistic settings. By removing those limitations we can get a more flexible system suitable for carrying out the tasks we described in the previous sections.

One limitation of Abella is that it lacks a module system for managing proofs in large scales. This causes extra effort in managing proofs, and more importantly, results in proofs that are not modular or portable. We can already see this problem manifested in the verification work in this thesis. First, we have to carefully assign distinct names to definitions and theorems for individual compiler transformations to avoid clashing of names. Second, even though some of the definitions and theorems are only used locally for the type or semantics preservation proofs for a particular transformation, they are exposed to proofs for other transformations because there is no mechanism for hiding such definitions and theorems; this leads to pollution of name space and less abstracted proofs. We are interested in solving those problems by borrowing ideas from other established interactive theorem provers such as Isabelle and Coq to develop a module system for Abella.

Abella also has some practical limitations that lead to a larger proof development effort than seems necessary. One such limitation is the need to make explicit every step in the process of constructing interactive proofs. The effect of this requirement is especially felt with respect to lemmas about contexts that arise routinely in the λ-tree syntax approach: such lemmas have fairly obvious proofs but, currently, the user must provide them to complete the overall verification task. In the Twelf and Beluga systems, such lemmas are obviated by absorbing them into the meta-theoretic framework. There are reasons related to the validation of verification that lead us to prefer explicit proofs (*e.g.*, it is easier to generate proof certificates from explicit proofs). However, as shown in [105], it is often possible to generate these proofs automatically, thereby allowing the user to focus on the less obvious aspects. In ongoing work, we are exploring the impact of using such ideas on reducing the overall proof effort.

References

[1] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science, Volume 19 of Proceedings of Symposia in Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.

[2] Maulik A. Dave. Compiler verification: A bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2–2, November 2003.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.

[10] Andrew Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.

[11] 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.

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

[13] Yaron Minsky. OCaml for the masses. *ACM Queue*, 9(9):44:40–44:49, September 2011.

[14] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The TeachScheme! Project: Computing and programming for every student. *Computer Science Education*, 14(1):55–77, 2004.

[15] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, KTH Royal Institute of Technology, 2003.

[16] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. F#3.0 — Strongly-typed language

support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

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

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

[19] 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/`.

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

[21] 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.

[22] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 54–65. ACM Press, 2007.

[23] 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*, pages 93–106, New York, NY, USA, 2010. ACM Press.

[24] 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*, pages 179–191. ACM Press, 2014.

[25] 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*, pages 166–178. ACM Press, 2015.

[26] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[27] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[28] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 219–233. ACM, 1986.

[29] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 435–445. ACM, 1994.

[30] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, New York, NY, USA, 1996. ACM.

[31] 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.

[32] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2011.

[33] 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.

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

[35] Louis-Julien Guillemette and Stefan Monnier. Type-safe code transformations in Haskell. *Electronic Notes in Theoretical Computer Science*, 174(7):23–39, 2007.

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

[37] Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *J. of Automated Reasoning*, 48:43–105, 2012.

[38] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1 — A common infrastructure for benchmarks. *CoRR*, abs/1503.06095, 2015.

[39] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations : Part 2 — A survey. *Journal of Automated Reasoning*, 55(4):307–372, 2015.

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

[41] 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.

[42] G. D. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[43] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[44] 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.

[45] Ye Henry Tian. Mechanically verifying correctness of CPS compilation. In *Proceedings of Twelfth Computing: The Australasian Theory Symposium*, volume 51 of *CRPIT*, pages 41–51. ACS, 2006.

[46] Olivier Savary Bélanger, Stefan Monnier, and Brigitte Pientka. Programming type-safe transformations using higher-order abstract syntax. In *Third International Conference on Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2013.

[47] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

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

[49] 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.

[50] 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.

[51] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

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

[53] 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.

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

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

[56] Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.

[57] Alwen Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, volume 173 of *ENTCS*, pages 3–18, 2006.

[58] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

[59] David Baelde and Gopalan Nadathur. Combining deduction modulo and logics of fixed-point definitions. In *27th Symp. on Logic in Computer Science*, pages 105–114. IEEE Computer Society Press, June 2012.

[60] Alwen Tiu. Stratification in logics of definitions. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2012.

[61] 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.

[62] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.

[63] 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.

[64] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.

[65] Gordon Plotkin. A structural approach to operational semantics. DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[66] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 97–108. ACM, 2009.

[67] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(5):308–320, 1964.

[68] G. Cousineau, P-L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.

[69] Mitchell Wand. The register-closure abstract machine: A machine model to support CPS compiling. Technical Report NU-CCS-89-24, Northeastern University, College of Computer Science, Boston, MA, July 1989.

[70] Simon Peyton Jones and David Lester. *Implementing functional languages: a tutorial.* Prentice Hall, 1992.

[71] L. Cardelli. Compiling a functional language. In *1984 Symposium on LISP and Functional Programming*, pages 208–217. ACM, 1984.

[72] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[73] David Tarditi, J. Gregory Morrisett, Perry Cheng, Christopher A. Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192. ACM, 1996.

[74] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[75] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[76] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Proceedings of the 15th European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[77] Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 157–168. ACM Press, 2008.

[78] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 275–287. ACM, 2015.

[79] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3):233–247, 1993.

[80] Gerald Jay Sussman and Guy L. Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[81] Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 177–190, New York, NY, USA, 2007. ACM Press.

[82] Michael J. Fischer. Lambda calculus schemata. *ACM SIGPLAN Notices*, 7(1):104–109, January 1972.

[83] Gordin Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1976.

[84] Oliver Danvy and Andrzej Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[85] 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.

[86] 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.

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

[88] Michael J. C. Gordon. Introduction to the HOL system. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 2–3. IEEE Computer Society, 1991.

[89] 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.

[90] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In Nachum Dershowitz and Andrei Voronkov, editors, *Proceedings of the 14th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 211–225. Springer, 2007.

[91] Yasuhiko Minamide and Koji Okuma. Verifying CPS transformations in Isabelle/HOL. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning About Languages with Variable Binding*, pages 1–8. ACM, 2003.

[92] Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.

[93] Furio Honsell, Marino Miculan, and Ivan Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, number 2076 in LNCS, pages 963–978. Springer, 2001.

[94] 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.

[95] Tom Murphy, VII. *Modal Types for Mobile Code.* PhD thesis, Carnegie-Mellon University, January 2008. Available as technical report CMU-CS-08-126.

[96] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In Zhong Shao, editor, *Proceedings of the 23rd European Symposium on Programming*, pages 128–148. Springer Berlin Heidelberg, 2014.

[97] 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.

[98] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual model type theory. *ACM Trans. on Computational Logic*, 9(3):1–49, 2008.

[99] Andrew Cave and Brigitte Pientka. A case study on logical relations using contextual types. In *Proceedings of the 10th International Workshop on Logical Frameworks and Meta Languages: Theory and Practice*, volume 185 of *EPTCS*, pages 33–45. ACM, 2015.

[100] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer, 2005.

[101] The POPLmark Challenge webpage. `http://www.seas.upenn.edu/~plclub/poplmark/`, 2015.

[102] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137. ACM, 1994.

[103] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie-Mellon University, December 1995.

[104] Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, September 1997.

[105] Olivier Savary-Bélanger and Kaustuv Chaudhuri. Automatically deriving schematic theorems for dynamic contexts. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2014)*. ACM, July 2014.

Appendix A

Tables

The two tables below contain references to the notions related to the source, intermediate and target languages, the transformations in our compiler and the type and semantics preservation proofs for the transformations. In these tables, "CPS" stands for "the CPS transformation", "CC" stands for "closure conversion", "CH" stands for "code hoisting" and "CG" stands for "code generation".

Language	Syntax		Typing		Evaluation	
	On Paper	λProlog	Rule-Based	λProlog	Rule-Based	λProlog
Source	Figure 5.1	Page 142	Figure 5.2	Page 142	Page 148	Page 156
Target of CPS	(Same as Source)					
Target of CC	Figure 6.1	Page 174	Page 170	Page 174	Page 180	Page 205
Target of CH	Page 195	Page 198	Page 170	Page 198	Page 180	Page 205
Target of CG	Figure 8.1	Pages 212	(untyped)		Page 215	Page 217

Table A.1: Summary of the Source, Intermediate and Target Languages

	Transformation		Type Preservation		Simulation		Semantics Preservation	
	Rule-Based	λProlog	On Paper	Abella	On Paper	Abella	On Paper	Abella
CPS	Figure 5.3	Page 143	Theorem 2	Page 155	Figure 5.6	Page 160	Theorem 3	Page 164
CC	Figure 6.2	Page 177	Theorem 4	Page 186	Figure 6.3	Page 188	Theorem 5	Page 192
CH	Figure 7.1	Page 198	Theorem 6	Page 204	Figure 7.2	Page 205	Theorem 7	Page 192
CG	Figure 8.2	Page 213	—	—	—	—	Theorem 9	Page 218

Table A.2: The Transformations and Their Type and Semantics Preservation Proofs