Abstract

VeriML: A dependently-typed, user-extensible and language-centric approach to proof assistants

Antonios Michael Stampoulis

2013

Software certification is a promising approach to producing programs which are virtually free of bugs. It requires the construction of a formal proof which establishes that the code in question will behave according to its specification – a higher-level description of its functionality. The construction of such formal proofs is carried out in tools called proof assistants. Advances in the current state-of-the-art proof assistants have enabled the certification of a number of complex and realistic systems software.

Despite such success stories, large-scale proof development is an arcane art that requires significant manual effort and is extremely time-consuming. The widely accepted best practice for limiting this effort is to develop domain-specific automation procedures to handle all but the most essential steps of proofs. Yet this practice is rarely followed or needs comparable development effort as well. This is due to a profound architectural shortcoming of existing proof assistants: developing automation procedures is currently overly complicated and error-prone. It involves the use of an amalgam of extension languages, each with a different programming model and a set of limitations, and with significant interfacing problems between them.

This thesis posits that this situation can be significantly improved by designing a proof assistant with extensibility as the central focus. Towards that effect, I have designed a novel programming language called VeriML, which combines the benefits of the different extension languages used in current proof assistants while eschewing their limitations. The key insight of the VeriML design is to combine a rich programming model with a rich type system, which retains at the level of types information about the proofs manipulated inside automation procedures. The effort required for writing new automation procedures is significantly reduced by leveraging this typing information accordingly.

I show that generalizations of the traditional features of proof assistants are a direct consequence of the VeriML design. Therefore the language itself can be seen as the proof assistant in its entirety and also as the single language the user has to master. Also, I show how traditional automation mechanisms offered by current proof assistants can be programmed directly within the same language; users are thus free to extend them with domain-specific sophistication of arbitrary complexity.

In this dissertation I present all aspects of the VeriML language: the formal definition of the language; an extensive study of its metatheoretic properties; the details of a complete prototype implementation; and a number of examples implemented and tested in the language.

VeriML: A dependently-typed, user-extensible and language-centric approach to proof assistants

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Antonios Michael Stampoulis

Dissertation Director: Zhong Shao

May 2013

Copyright © 2013 by Antonios Michael Stampoulis
All rights reserved.

Contents

List of Figures

List of Tables

List of Code Listings

Acknowledgements

This dissertation would have been impossible without the help, advice and support of a great number of people. They all have my deepest gratitude.

First of all, I would like to thank my advisor, Zhong Shao, for giving me the chance to work on an exciting research topic, for his constant help and his input on this work and for supporting me in every way possible throughout these years. His insights and close supervision were of critical importance in the first stages of this research; his eye for intellectual clarity and simplicity were tremendously influential in the later stages of the development of the VeriML theory and implementation. He gave me the space to work out things on my own when I needed it and when I wanted to concentrate on the development of VeriML, yet he was always available when I needed his opinion. I am indebted to him for all of his advice and suggestions, for being patient with me and for instilling the true qualities of a researcher in me. I am leaving Yale with fond memories of our endless discussions in his office; I hope these will continue in the years to follow.

I would also like to express my gratitude to the rest of my committee, Paul Hudak, Drew McDermott and Xavier Leroy, for their advice during these years and for taking the time to read this dissertation. They have all influenced this work, Paul through his research and his profound, insightful lectures on Haskell and formal semantics; Drew by his suggestion to look at resolution-based provers, starting a path which eventually led to reading and implementing the equality with uninterpreted functions procedure; and Xavier by showing that software certification is possible and exciting through the development of CompCert, by creating OCaml which was a pleasure to use in order to develop VeriML, and by his excitement and encouragement when I first asked him to serve in my committee

and described the VeriML idea to him.

I would also like to thank the rest of my professors at Yale, especially Bryan Ford, Richard Yang, Gaja Jarosz, Yorgos Makris and Andreas Savvides. Nikolaos Papaspyrou, my undegraduate thesis advisor, has influenced me greatly through his supervision at NTUA; my initial interest in programming languages research in general and dependent types in particular is due to him. Ella Bounimova gave me the chance to work at Microsoft Research Redmond as a summer intern during my first year at Yale; I am grateful to her as well as Tom Ball and Vlad Levin, my two other mentors at Microsoft at the time.

During my time at Yale, I have had the pleasure to collaborate and discuss research and non-research with many colleagues. I would especially like to thank Andi Voellmy, my office-mate, for our discussions and his input during the six years of our Ph.D.; all the past, present and newly-arrived members of the FLINT group, especially Xinyu Feng, Alex Vaynberg, Rodrigo Ferreira, David Constanzo, Shu-Chun Weng, Jan Hoffman and Mike Marmar. Zhong Zhuang deserves special mention for using the VeriML prototype for his own research. I would also like to thank the rest of my friends in the CS department, especially Nikhil, Eli, Dan, Edo, Lev, Aaron, Pradipta and Justin. A number of researchers and colleagues whom I have discussed this work with in conferences and beyond have also been profoundly influential; I would like to especially thank Herman Geuvers, Brigitte Pientka, Carsten Schürmann, Andreas Abel, Randy Pollack, Adam Chlipala, Viktor Vafeiadis, Dimitrios Vytiniotis, Christos Dimoulas, Daniel Licata and Gregory Malecha.

This work was supported in part by the NSF grants CCF-0811665, CNS-0915888, CNS-0910670 and CNS-1065451 and by the DARPA CRASH grant FA8750-10-2-0254. I would like to thank both agencies, as well as Yale University and the Department of Computer Science, for supporting me during these years and making this work possible.

During my time in New Haven I was extremely lucky to meet many amazing people that made my Ph.D. years unforgettable. Argyro and Mihalis have been next to me in the most fun and happy moments and in the hardest times as well; they have been infinitely patient with me and very supportive. During these years they became my second family and I thank them a lot. I would also like to thank Dimitris 'the Teacher' and Maria, for showing me the ropes in my first year; my roomie Thanassis for his friendship and patience, for our

time in Mansfield St. as well as for his excellent cooking; Maria for our endless walks and talks in the HGS courtyard; Marisa and the Zattas brothers, especially for being around during this last year; Petros, Tasos, Nikos, Roula, Yiorgos and their wonderful daughter Zoi, Palmyra, Sotiria and the rest of the Greeks at Yale. The members of the legendary Sixth Flour Lounge: my roomates Dominik and Eugene, Yao, Isaak, Nana, Philip; also all my other friends during my time in HGS, especially Yan and Serdar. The graphics and the organics for our times in New York, New Haven and Boston: Yiannis, Kostas, Rimi, Dan and Cagil – and Nikoletta, even though technically she was in Europe. Elli, for the fun times in Brooklyn and for her profound words and support; and the Yiorgides as well. The band: Randy (especially for convincing me to join again), Tarek, Cecily, Tomas, Anne, Ragy and Lucas. All of my other friends who supported me throughout these years, especially Areti, Danai, Ioanna, Themis, Enrico, Yiannis, Mairi, Alvertos, Melina and Panagiotis.

I am deeply grateful to all of my family for their love and their unwavering support in this long process. My father instilled in me his love of education and my mother her sense of perserverance, both of which were absolutely essential for completing this work. I would like to thank my siblings Nikos and Eva for always believing in me; my siblings-in-law Chin Szu and Thodoris and my three adorable nieces who were born during my Ph.D., Natalie, Matina and Eirini; my uncles, aunts and cousins, especially uncle Thanos and cousin Antonis for the kind words of support they offered and their appreciation of the hard work that goes into a doctoral dissertation.

Last, I would like to thank Erato, for being next to me during the good moments and the bad moments of this Ph.D., for her wisdom and advice, and for her constant love and support.

Chapter 1

Introduction

Computer software is ubiquitous in our society. The repercussions of software errors are becoming increasingly more severe, leading to huge financial losses every year [e.g. Zhivich and Cunningham, 2009] and even resulting in the loss of human lives [e.g. Blair et al., 1992, Lions et al., 1996]. A promising research direction towards more robust software is the idea of *certified software* [Shao, 2010]: software that comes together with a high-level description of its behavior – its *specification*. The software itself is related to its specification through an unforgeable formal proof that includes all possible details, down to some basic mathematical axioms. Recent success stories in this field include the certified optimizing C compiler [Leroy, 2009] and the certified operating system kernel seL4 [Klein et al., 2009].

The benefit of formal proofs is that they can be mechanically checked for validity using a small computer program, owing to the high level of detail that they include. Their drawback is that they are hard to write, even when we utilize *proof assistants* – specialized tools that are designed to help in formal proof development. We argue that this is due to a profound architectural shortcoming of current proof assistants: though extending a proof assistant with domain-specific sophistication is of paramount importance for large-scale proof development [e.g. Chlipala et al., 2009, Morrisett et al., 2012], developing such extensions (in the form of *proof-producing procedures*) is currently overly complex and error-prone.

This dissertation is an attempt to address this shortcoming of formal proof development in current systems. Towards that end, I have designed and developed a new program-

ming language called VeriML, which serves as a novel proof assistant. The main benefit of VeriML is that it includes a rich *type system* which provides helpful information and robustness guarantees when developing new proof-producing procedures for specialized domains. Furthermore, users can register such procedures to be utilized transparently by the proof assistant, so that the development of proofs and further procedures can be significantly simplified. Though an increasing number of trivial details can be omitted in this way from proofs and proof procedures, the type system ensures that this does not eschew logical soundness – the resulting proofs are as trustworthy as formal proofs with full details. This leads to a truly extensible proof assistant where domain-specific sophistication can be built up in layers.

1.1 Problem description

Formal proof development is a complicated endeavor. Formal proofs need to contain all possible details, down to some basic logical axioms. Contrast this with the informal proofs of normal mathematical practice: one needs to only point out the essential steps of the argument at hand. The person reading the proof needs to convince themselves that these steps are sufficient and valid, relying on their intuition or even manually reconstructing some missing parts of the proof (e.g. when the proof mentions that a certain statement is trivial, or that 'other cases follow similarly'). This is only possible if they are sufficiently familiar with the specific domain of the proof. Thus the distinguishing characteristic is that the receiver of an informal proof is expected to possess certain sophistication, whereas the receiver of formal proof is an absolutely skeptical agent that only knows of certain basic reasoning principles.

Proof assistants are environments that provide users with mechanisms for formal proof development, which bring this process closer to writing down an informal proof. One such mechanism is *tactics*: functions that produce proofs under certain circumstances. Alluding to a tactic is similar to suggesting a methodology in informal practice, such as 'use induction on x'. Tactics range from very simple, corresponding to basic reasoning principles, to very sophisticated, corresponding to automated proof discovery for large classes of problems.

When working on a large proof development, users are faced with the choice of either using the already existing tactics to write their proofs, or to develop their own domain-specific tactics. The latter choice is often suggested as advantageous [e.g. Morrisett et al., 2012], as it leads to more concise proofs that omit details, which are also more robust to changes in the specifications – similar to informal proofs that expect a certain level of domain sophistication on the part of the reader. But developing new tactics comes with its own set of problems, as tactics in current proof assistants are hard to write. The reason is that the workflow that modern proof assistants have been designed to optimize is developing new proofs, not developing new tactics. For example, when a partial proof is given, the proof assistant informs the user about what remains to be proved; thus proofs are usually developed in an interactive dialog with the proof assistant. Similar support is not available for tactics and would be impossible to support using current tactic development languages. Tactics do not specify under which circumstances they are supposed to work, the type of arguments they expect, what proofs they are supposed to produce and whether the proofs they produce are actually valid. Formally, we say that tactic programming is *untyped*. This hurts composability of tactics and also allows a large potential for bugs that occur only on particular invocations of tactics. This lack of information is also what makes interactive development of tactics impossible.

Other than tactics, proof assistants also provide mechanisms that are used implicitly and ubiquitously throughout the proof. These aim to handle trivial details that are purely artifacts of the high level of rigor needed in a formal proof, but would not be mentioned in an informal proof. Examples are the *conversion rule*, which automates purely computational arguments (e.g. determining that $5! = 120$), and *unification algorithms*, which aim to infer details of the terms used in a proof that can be easily determined from the context (e.g. determining that the generic addition symbol $+$ refers to natural number addition when used as $5 + x$). The mechanism of the conversion rule is even integrated with the base proof checker, in order to keep formal proofs feasible in terms of size. We refer to these mechanisms as *small-scale automation* in order to differentiate them from *large-scale automation* which is offered by explicit reference to specific automation tactics and development of further ones, following Asperti and Sacerdoti Coen [2010].

Small-scale automation mechanisms usually provide their own ways for adding domain-specific extensions to them; this allows users to provide transparent automation for the trivial details of the domain at hand. For example, unification mechanisms can be extended through first-order lemmas; and the conversion rule can support the development of automation procedures that are correct by construction. Still, the programming model that is supported for these extensions is inherrently quite limited. In the case of the conversion rule this is because of its tight coupling with the core of the proof assistant; in the case of unification, this because of the very way the automation mechanism work. On the other hand, the main cost in developing such extensions is in proving associated lemmas; therefore it significantly benefits from the main workflow of current proof assistants for writing proofs.

Contrasting this distinction between small-scale and large-scale automation with the practice of informal proof reveals a key insight: the distinction between what constitutes a trivial detail that is best left implicit and what constitutes an essential detail of a proof is very thin; and it is arbitrary at best. Furthermore, as we progress towards more complex proofs in a certain domain, or from one domain A to a more complicated one B, which presupposes domain A (e.g. algebra and calculus), it is crucial to omit more details. Therefore, the fact that small-scale automation and large-scale automation are offered through very different means is a liability in as far as it precludes moving automation from one side to the other. Users can develop the exact automation algorithms they have in mind as a large-scale automation tactic; but there is significant re-engineering required if they want this to become part of the 'background reasoning' offered by the small-scale automation mechanisms. They will have to completely rewrite those algorithms and even change the data structures they use, in order to match the limitations of the programming model for small-scale automation. In cases where such limitations would make the algorithms suboptimal, the only alternative is to extend the very implementation of the small-scale automation mechanisms in the internals of the proof assistant. While this has been attempted in the past, it is obviously associated with a large development cost and raises the question whether the overall system is still logically sound.

Overall, users are faced with a choice between multiple mechanisms when trying to

develop automation for a new domain. Each mechanism has its own programming model and a set of benefits, issues and limitations. In many cases, users have to be proficient in multiple of these mechanisms in order to achieve the desired result in terms of verbosity in the resulting proof, development cost, robustness and efficiency. The fact that significant issues exist in interfacing between the various mechanisms and languages involved, further limits the extensibility and reusability of the automation that users develop.

In this dissertation, we propose a novel architecture for proof assistants, guided from the following insights. First, development of proof-producing procedures such as tactics should be a central focal point for proof assistants, just as development of proofs is of central importance in current proof assistants. Second, the distinction between large-scale automation and small-scale automation should be de-emphasized in light of their similarities: the involved mechanisms are in essence proof-producing procedures, which work under certain circumstances and leverage different algorithms. Third, that writing such proof-producing procedures in a single general-purpose programming model coupled with a rich *type system*, directly offers the benefits of the traditional proof assistant architecture and generalizes them.

Based on these insights, we present a novel *language-based architecture* for proof assistants. We propose a new programming language, called VeriML, which focuses on the development of *typed proof-producing procedures*. Proofs, as well as other logic-related terms such as propositions, are explicitly represented and manipulated in the language; their types precisely capture the relationships between such terms (e.g. a proof which proves a specific proposition). We show how this language enables us to support the traditional workflows of developing proofs and tactics in current proof assistants, utilizing the information present in the type system to recover and extend the benefits associated with these workflows. We also show that small-scale automation mechanisms can be implemented within the language, rather than be hardcoded within its internal implementation. We demonstrate this fact through an implementation of the conversion rule within the language; we show how it can be made to behave identically to a hardcoded conversion rule. In this way we solve the long-standing problem of enabling arbitrary user extensions to conversion while maintaining logical soundness, by leveraging the rich type system of the language. Last, we show that

once domain-specific automation is developed it can transparently benefit the development of not only proofs, but further tactics and automation procedures as well. Overall, this results in a style of formal proof that comes one step closer to informal proof, by increasing the potential for omitting details, while maintaining the same guarantees with respect to logical soundness.

1.2 Thesis statement

The thesis statement that this dissertation establishes follows.

> A type-safe programming language combining typed manipulation of logical terms with a general-purpose side-effectful programming model is theoretically possible, practically implementable, viable as an alternative architecture for a proof assistant and offers improvements over the current proof assistant architectures.

By 'logical terms' we refer to the terms of a specific higher-order logic, such as propositions and proofs. The logic that we use is called λHOL and is specified as a type system in Chapter 3. By 'manipulation' we mean the ability to introduce logical terms, pass them as arguments to functions, emit them as results from functions and also pattern match on their structure programmatically. By 'typed manipulation' we mean that the logical-level typing information of logical terms is retained during such manipulation. By 'type-safe' we mean that subject reduction holds for the programming language. By 'general-purpose side-effectful programming model' we mean a programming model that includes at the very least datatypes, general recursion and mutable references. We choose the core ML calculus to satisfy this requirement. We demonstrate theoretical possibility by designing a type system and operational semantics for this programming language and establishing its metatheory. We demonstrate practical implementability through an extensive prototype implementation of the language, which is sufficiently efficient in order to test various examples in the language. We demonstrate viability as a proof assistant by implementing a set of examples tactics and proofs in the language. We demonstrate the fact that this language offers improvements over current proof assistant architectures by showing that it

enables user-extensible static checking of proofs and tactics and utilizing such support in our examples. We demonstrate how this support simplifies complex implementations of similar examples in traditional proof assistants.

1.3 Summary of results

In this section I present the technical results of my dissertation research.

1. **Formal design of VeriML.** I have developed a type system and operational semantics supporting a combination of general-purpose, side-effectful programming with first-class typed support for logical term manipulation. The support for logical terms allows specifying the input-output behavior of functions that manipulate logical terms. The language includes support for pattern matching over open logical terms, as well as over variable environments. The type system works by leveraging *contextual type theory* for representing open logical terms as well as the variable environments they depend on. I also show a number of extensions to the core language, such as a simple staging construct and a proof erasure procedure.

2. **Metatheory.** I have proved a number of metatheoretic results for the above language.

   Type safety ensures that a well-typed expression of the language evaluates to a well-typed value of the same type, in the case where evaluation is successful and terminates. This is crucial in ensuring that the type that is statically assigned to an expression can be trusted. For example, it follows that expressions whose static type claims that a proof for a specific proposition will be created indeed produce such a proof upon successful evaluation.

   Type safety for static evaluation ensures that the extension of the language with the staging construct for static evaluation of expressions obeys the same safety principle.

   Compatibility of normal and proof-erasure semantics establishes that every step in the evaluation of a source expression that has all proof objects erased corresponds to a step in the evaluation of the original source expression. This guar-

antees that even if we choose not to generate proof objects while evaluating expressions of the language, valid proof objects of the right type always exist. Thus, if users are willing to trust the type checker and runtime system of VeriML, they can use the optimized semantics where no proof objects get created.

Collapsing transformation of contextual terms establishes that under reasonable limitations with respect to the definition and use of contextual variables, a contextual term can be transformed into one that does not mention such contextual variables. This proof provides a way to transform proof expressions inside tactics into *static proof expressions* evaluated statically, at the time that the tactic is defined, using staging.

3. **Prototype implementation.** I have developed a prototype implementation of VeriML in OCaml, which is used to type check and evaluate a wide range of examples. The prototype has an extensive feature set over the pure language as described formally, supporting type inference, surface syntax for logical terms and VeriML code, special tactic syntax and translation to simply-typed OCaml code.

4. **Extensible conversion rule.** We showcase how VeriML can be used to solve the long-standing problem of a *user-extensible conversion rule*. The conversion rule that we describe combines the following characteristics: it is safe, meaning that it preserves logical soundness; it is user-extensible, using a familiar, generic programming model; and, it does not require metatheoretic additions to the logic, but can be used to simplify the logic instead. Also, we show how this conversion rule enables receivers of a formal proof to choose the exact size of the trusted core of formal proof checking; this choice is traditionally only available at the time that a logic is designed. We believe this is the first technique that combines these characteristics leading to a safe and user-extensible static checking technique for proofs.

5. **Static proof expressions.** I describe a method that enables discovery and proof of required lemmas within tactics. This method requires minimal programmer annotation and increases the static guarantees provided by the tactics, by removing potential

sources of errors. I showcase how the same method can be used to separate the proof-related aspect of writing a new tactic from the programming-related aspect, leading to increased separation of concerns.

6. **Dependently typed programming.** I also show how static proof expressions can be useful in traditional dependently-typed programming (in the style of Dependent ML [Xi and Pfenning, 1999] and Agda [Norell, 2007]), exactly because of the increased separation of concerns between the programming and the proof-related aspects. By combining this with the support for writing automation tactics within VeriML, we show how our language enables a style of dependently-typed programming where the extra proof obligations are generated and resolved within the same language.

7. **Technical advances in metatheoretic techniques.** The metatheory for VeriML that I have developed includes a number of technical advances over developments for similar languages such as Delphin or Beluga. These are:

   - *Orthogonality between logic and computation.* The theorems that we prove about the computational language do not depend on specifics of the logic language, save for a number of standard theorems about it. These theorems establish certain substitution and weakening lemmas for the logic language, as well as the existence of an effective pattern matching procedure for terms of this language. This makes the metatheory of the computational language modular with respect to the logic language, enabling us to extend or even replace the logic language in the future.
   - *Hybrid deBruijn variable representation.* We use a concrete variable representation for variables of the logic language instead of using the named approach for variables. This elucidates the definitions of the various substitution forms, especially wih respect to substitution of a polymorphic context with a concrete one; it also makes for precise substitution lemma statements. It is also an efficient implementation technique for variables and enables subtyping based on context subsumption. Using the same technique for variable representation in the metatheory and in the actual implementation of the language decreases the trust one needs to place in their correspondence. Last, this representation opens up

possibilities for further extensions to the language, such as multi-level contextual types and explicit substitutions.

- *Pattern matching.* We use a novel technique for type assignment to patterns that couples our hybrid variable representation technique with a way to identify relevant variables of a pattern, based on our notion of *partial variable contexts*. This leads to a simple and precise induction principle, used in order to prove the existence of deterministic pattern matching – our key theorem. The computational content of this theorem leads to a simple pattern matching algorithm.

8. **Implementations of extended conversion rules.** We describe our implementation of the traditional conversion rule in VeriML as well as various extensions to it, such as congruence closure and arithmetic simplification. Supporting such extensions has prompted significant metatheoretic and implementation additions to the logic in past work. The implementation of such algorithms in itself in existing proof assistants has required a mix of techniques and languages. We show how the same extensions can be achieved without any metatheoretic additions to the logic or special implementation techniques by using our computational language. Furthermore, our implementations are concise, utilize language features such as mutable references and require a minimal amount of manual proof from the programmer.

Chapter 2

Informal overview

In this chapter I am going to present a high-level picture of the architecture of modern proof assistants, covering the basic notions involved and identifying a set of issues with this architecture. I will then present the high-level ideas behind VeriML and how they address the issues. I will show the correspondences between the traditional notions in a proof assistant and their counterpart in VeriML, as well as the new possibilities that are offered. Last, I will give a brief overview of the constructs of the language.

2.1 Proof assistant architecture

2.1.1 Preliminary notions

Formal logic. A *formal logic* is a mathematical system that defines a collection of objects; a collection of statements describing properties of these objects, called *propositions*; as well as the means to establish the validity of such statements (under certain hypotheses), called *proofs* or *derivations*. Derivations are composed by using *axioms* and *logical rules*. *Axioms* are a collection of propositions that are assumed to be always valid; therefore we always possess valid proofs for them. *Logical rules* describe under what circumstances a collection of proofs for certain propositions (the premises) can be used to establish a proof for another proposition (the consequence). A derivation is thus a recording of a number of logical steps, claiming to establish a proposition; it is *valid* if every step is an axiom or a valid application

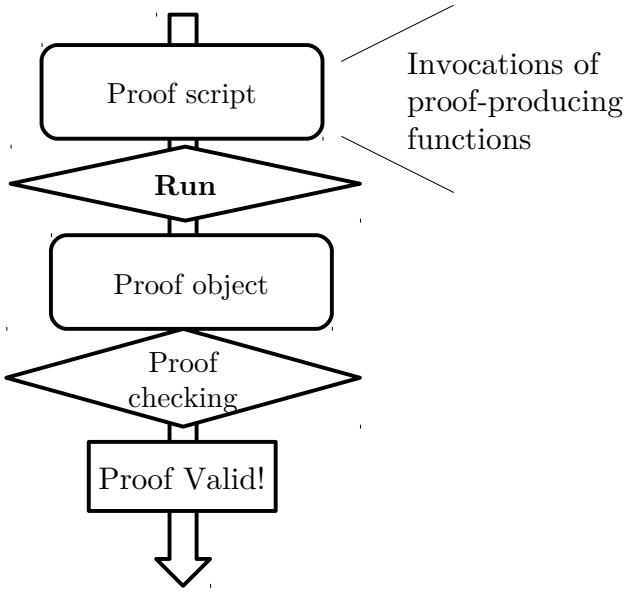

Figure 2.1: Basic architecture of proof assistants

of a logical rule[1].

Proof object; proof checker. A *mechanized logic* is an implementation of a specific formal logic as a computer system. This implementation at the very least consists of a way to represent the objects, the propositions and the derivations of the logic as computer data, and also of a procedure that checks whether a claimed proof is indeed a valid proof for a specific proposition, according to the logical rules. We refer to the machine representation of a specific logical derivation as the *proof object*; the computer code that decides the validity of proof objects is called the *proof checker*. We use this latter term in order to signify the trusted core of the implementation of the logic: bugs in this part of the implementation might lead to invalid proofs being admitted, destroying the logical soundness of the overall system.

Proof assistant. A mechanized logic automates the process of validating formal proofs, but does not help with actually developing such proofs. This is what a proof assistant is for. Examples of proof assistants include Coq [Barras et al., 2012], Isabelle [Paulson, 1994],

1. Note that these definitions are not normative and can be generalized in various ways, but they will suffice for the purposes of our discussion.

HOL4 [Slind and Norrish, 2008], HOL-Light [Harrison, 1996], Matita [Asperti et al., 2007], Twelf [Pfenning and Schürmann, 1999], NuPRL [Constable et al., 1986], PVS [Owre et al., 1996] and ACL2 [Kaufmann and Strother Moore, 1996]. Each proof assistant supports a different logic, follows different design principles and offers different mechanisms for developing proofs. For the purposes of our discussion, the following general architecture for a proof assistant will suffice and corresponds to many of the above-mentioned systems. A proof assistant consists of a logic implementation as described above, a library of *proof-producing functions* and a language for writing *proof scripts* and proof-producing functions. We will give a basic description of the latter two terms below and present some notable examples of such proof assistants as well as important variations in the next section.

Proof-producing functions. The central idea behind proof assistants is that instead of writing proof objects directly, we can make use of specialized functions that produce such proof objects. By choosing the right functions and composing their results, we can significantly cut down on the development effort for large proofs. We refer to such functions as *proof-producing functions* and we characterize their definition as follows: functions that manipulate data structures involving logical terms, such as propositions and proofs, and produce other such data structures. This definition is deliberately very general and can refer to a very large class of functions. They range from simple functions that correspond to logical reasoning principles to sophisticated functions that perform automated proof search for a large set of problems utilizing complicated data structures and algorithms. Users are not limited to a fixed set of proof-producing functions, as most proof assistants allow users to write new proof-producing functions using a suitable language.

A conceptually simple class of proof-producing functions are *decision procedures*: functions that can decide the provability of *all* the propositions of a particular, well-defined theory. For example, Gaussian elimination corresponds to a decision procedure for systems of linear equations[2]. *Automated theorem provers* are functions that attempt to discover proofs for arbitrary propositions, taking into account new definitions and already-proved theorems; the main difference with decision procedures being that the class of propositions

2. More accurately: for propositions that correspond to systems of linear equations.

they can handle is not defined a priori but is of an ad hoc nature. Automated provers might employ sophisticated algorithms and data structures in order to discover such proofs. Another example is *rewriters*, functions that simplify terms and propositions into equivalent ones, while also producing a proof witnessing this equivalence.

A last important class of proof-producing functions are *tactics*. A tactic is a proof-producing function that works on incomplete proofs: a specific data structure which corresponds to a 'proof-in-development' where some parts might still be missing. A tactic accepts an incomplete proof and transforms it into another potentially still incomplete proof; furthermore, it provides some justification why the transformation is valid. The resulting incomplete proof is expected to be simpler to complete than the original one. Every missing part is characterized by a set of hypotheses and by its goal – the proposition we need to prove in order to fill it in; we refer to the overall description of the missing parts as the current *proof state*. Examples of tactics are: induction principles – where a specific goal is changed into a set of goals corresponding to each case of an inductive type with extra induction hypotheses; decision procedures and automated provers as described above; and higher-order tactics for applying a tactic everywhere – e.g., given an automated prover, try to finish an incomplete proof by applying the prover to all open goals. The notion of tactics is a very general one and can encompass most proof-producing functions that are of interest to users[3]. Tactics thus play a central role in many current proof assistants.

Proof scripts. If we think of proof objects as a low-level version of a formal proof, then proof scripts are their high-level equivalent. A proof script is a program which composes several proof-producing functions together. When executed, a proof script emits a proof object for a specific proposition; the resulting proof object can then be validated through the proof checker, as shown in Figure 2.1. This approach to proof scripts and their validation procedure is the heart of many modern proof assistants.

Proof scripts are written in a language that is provided as part of the proof assistant and follows one of two styles: the imperative style or the declarative style. In the imperative

3. In fact, the term 'tactic' is more established than 'proof-producing function' and is often used even in cases where the latter term would be more accurate. We follow this tradition in later chapters of this dissertation, using the two terms interchangeably.

style, the focus is placed on which proof-producing functions are used: the language is essentially special syntax for directly calling proof-producing functions and composing their results. In the declarative style, the focus is placed on the intermediate steps of the proof: the language consists of human-readable reasoning steps (e.g. split by these cases, know that a certain proposition holds, etc.). A default proof-producing function is used to justify each such step, but users can explicitly specify which proof-producing function is used for each step. In general, the imperative style is significantly more concise than the declarative style, at the expense of being 'write-only' – meaning that it is usually hard for a human to follow the logical argument made in an imperative proof script after it has been written.

In both cases, the amount of detail that needs to be included in a proof script directly depends on the proof-producing functions that are available. This is why it is important to be able to write new proof-producing functions. When we work in a specific domain, we might need functions that better correspond to the reasoning principles for that domain; we can even automate proof search for a large class of propositions of that domain through a specialized algorithm. In this way we can cover cases where the generic proof search strategies (offered by the existing proof procedures) do not perform well. One example would be deciding whether two polynomials are equivalent by attempting to use normal arithmetic properties in a fixed order, until the two polynomials take the same form – instead of a specialized equivalence checking procedure.

In summary, a proof assistant enables users to develop valid formal proofs for certain propositions, by writing proof scripts, utilizing an extensible set of proof-producing functions and validating the resulting proof objects through a proof checker. In the next section we will see examples of such proof assistants as well as additions to this basic architecture. As we will argue in detail below, the main drawback of current proof assistants is that the language support available for writing proof-producing functions is poor. This hurts the extensibility of current proof assistants considerably. The main object of the language presented in this dissertation is to address exactly this fact.

2.1.2 Variations

LCF family

The Edinburgh LCF system [Gordon et al., 1979] was the first proof assistant that introduced the idea of a meta-language for programming proof-producing functions and proof scripts. A number of modern proof assistants, most notably HOL4 [Slind and Norrish, 2008] and HOL-Light [Harrison, 1996], follow the same design ideas as the original LCF system other than the specific logic used (higher-order logic instead of Scott's logic of computable functions). Furthermore, ML, the meta-language designed for the original LCF system became important independently of this system; modern dialects of this language, such as Standard ML, OCaml and F# enjoy wide use today. An interesting historical account of the evolution of LCF is provided by Gordon [2000].

The unique characteristic of proof assistants in the LCF family is that the same language, namely ML, is used both for the implementation of the proof assistant and by the user. The design of the programming constructs available in ML was very much guided by the needs of writing proof-producing functions and are thus very well-suited to this task. For example, a notion of algebraic data types is supported so that we can encode sophisticated data structures such as the logical propositions and the 'incomplete proofs' that tactics manipulate; general recursion and pattern matching provide a good way to work with these data structures; higher-order functions are used to define tacticals – functions that combine tactics together; exceptions are used in proof-producing functions that might fail; and mutable references are crucial in order to implement various efficient algorithms that depend on an imperative programming model.

Proof scripts are normal ML expressions that return values of a specific proof data type. We can use decision procedures, tactics and tacticals provided by the proof assistant as part of those proof scripts or we can write new functions more suitable to the domain we are interested in. We can also define further classes of proof-producing functions along with their associated data structures: for example, we can define an alternate notion of incomplete proofs and an associated notion of tactics that are better suited to the declarative proof style, if we prefer that style instead of the imperative style offered by the proof assistant.

The key technical breakthrough of the original Edinburgh LCF system is its approach to ensuring that only valid proofs are admitted by the system. Having been developed in a time where memory space was limited, the approach of storing large proof objects in memory and validating them post-hoc as we described in the previous section was infeasible. The solution was to introduce an *abstract data type* of *valid proofs*. Values of this type can only be introduced through a fixed set of constructor functions: each function consumes zero or more valid proofs, produces another valid proof and is in direct correspondence to an axiom or rule of the original logic[4]. Expressions having the type of valid proofs are guaranteed to correspond to a derivation in the original logic when evaluated – save for failing to evaluate successfully (i.e. if they go into an infinite loop or throw an exception). This correspondence is direct in the case where such expressions are formed by combining calls to the constructor functions. The careful design of the type system of ML guarantees that such a correspondence continues to hold even when using all the sophisticated programming constructs available. Formally, this guarantee is a direct corollary of the *type-safety* theorem of the ML language, which establishes that any expression of a certain type evaluates to a value of that same type, or fails as described above.

The ML type system is expressive enough to describe interesting data structures as well as the signatures of functions such as decision procedures and tactics, describing the number and type of arguments they expect. For example, the type of a decision procedure can specify that it expects a proposition as an input argument and produces a valid proof as output. Yet, we cannot specify that the resulting proof actually proves the given proposition. The reason is that all valid proofs are identified at the level of types. We cannot capture finer distinctions of proofs in the type system, specifying for example that a proof proves a specific proposition, or a proposition that has a specific structure. All other logical terms (e.g. propositions, natural numbers, etc.) are also identified. Thus the signatures that we give to proof-producing functions do not fully reflect the input-output relationships of

4. This is the main departure compared to the picture given in the previous section, as the logic implementation does not include explicit proof objects. Instead, proof object generation is essentially combined with proof checking, rendering the full details of the proof object irrelevant. We only need to retain information about the proposition that it proves in order to be able to form further valid proofs. Based on this correspondence, it is easy to see that the core of the proof checker is essentially the set of valid proof constructor functions.

the logical terms involved. Because of the identification of the types of logical terms, we say that proof-producing functions are programmed in an essentially *untyped* manner. By extension, the same is true of proof scripts.

This is a severe limitation. It means that no information about logical terms is available at the time when the user is writing functions or proof scripts, information which would be useful to the user and could also be used to prevent errors at the time of function or proof script definition – *statically*. Instead, all logic-related errors, such as proving the wrong proposition or using a tactic in a case where it does not apply, will only be caught when (and if) evaluation reaches that point – *dynamically*. This is a profound issue especially in the case of proof-producing functions, where logic-related errors might be revealed unexpectedly at a particular invocation of the function. Also, in the case of imperative proof scripts, this limitation precludes us from knowing how the proof state evolves – as no information about the input and output proof state of the tactics used is available statically. The user has to keep a mental model of the evolution of proof state in order to write proof scripts that evaluate successfully.

Overall, the architecture of proof assistants in this family is remarkably simple yet leads to a very extensible and flexible proof development style. Users have to master a single language where they can mix development of proofs and development of specialized proof-producing functions, employing a general-purpose programming model. The main shortcoming is that all logic-related programming (whether it be proof scripts or proof-producing functions) is essentially done in an untyped manner.

Isabelle

Isabelle [Paulson, 1994] is a widely used proof assistant which evolved from the LCF tradition. The main departure from the LCF design is that instead of being based on a specific logic, its logical core is a *logical framework* instead. Different logics can then be implemented on top of this basic logical framework. The proof checker is then understood as the combination of the implementation of the base logical framework and of the definition of the particular logic we work with. Though this design allows for flexibility in choosing the

logic to work with, in practice the vast majority of developments in Isabelle are done using an encoding of higher-order logic, similar to the one used in HOL4 and HOL-Light. This combination is referred to as Isabelle/HOL [Nipkow et al., 2002].

The main practical benefit that Isabelle offers over proof assistants following the LCF tradition is increased interactivity when developing proof scripts. Though interactivity support is available in some LCF proof assistants, it is emphasized more in the case of Isabelle. The proof assistant is able to run a proof script up to a specified point and inform the user of the proof state at that point – the remaining goals that need to be proved and the hypotheses at hand. The user can then continue developing the proof script based on that information. Proof scripts are therefore developed in a dialog with the proof assistant where the evolution of proof state is clearly shown to the user: the user starts with the goal they want to prove, chooses an applicable tactic and gets immediate feedback about the new subgoals they need to prove. This process continues until a full proof script is developed and no more subgoals remain.

This interactivity support is made possible in Isabelle by following a two-layer architecture. The first layer consists of the implementation of the proof assistant in the classic LCF style, including the library of tactics and other proof-producing functions, using a dialect of ML. It also incorporates the implementation of a higher-level language which is specifically designed for writing proof scripts; and of a user interface providing interactive development support for this language as described above. The second layer consists of proof developments done using the higher-level language. Unless otherwise necessary, users work at this layer.

The split into two layers de-emphasizes the support for writing new proof-producing functions using ML. Writing a new tactic comes with the extra overhead of having to learn the internal layer of the proof assistant. This difficulty is added on top of the issues with the development of proof-producing functions as in the normal LCF case. Rather than writing their own specialized tactics to automate domain-specific details, users are thus more likely to use already-existing tactics for proving these details. This leads to longer proof scripts that are less robust to small changes in the definitions involved.

In Isabelle, this issue is minimized through the use of a powerful *simplifier* – an automa-

tion mechanism that rewrites propositions into simpler forms. It makes use of a rule for *higher-order unification* which is part of the core logical theory of Isabelle. We can think of the simplifier as a specific proof-producing function which is utilized by most other tactics and decision procedures. Users can extend the simplifier by registering rewriting first-order lemmas with it. The lemmas are proved normally using the interactive proof development support. This is an effective way to add domain-specific automation with a very low cost to the user, while making use of the primary workload that Isabelle was designed to optimize. We refer to the simplifier as a *small-scale automation* mechanism: it is used ubiquitously and implicitly throughout the proof development process. We thus differentiate similar mechanisms from the *large-scale automation* offered through the use of tactics.

Still, the automation support that can be added through the simplifier is quite limited when compared to the full ML programming model available when writing new tactics. As mentioned above, we can only register first-order lemmas with the simplifier. These correspond roughly to the Horn clauses composing a program in a logic programming language such as Prolog. The simplifier can then be viewed as an interpreter for that language, performing proof search over such rules using a fixed strategy. In the cases where this programming model cannot capture the automation we want to support (e.g. when we want to use an efficient algorithm that uses imperative data structures), we have to fall back to developing a new tactic.

Coming back to proof scripts, there is one subtle issue that we did not discuss above. As we mentioned, being able to evaluate proof scripts up to some point is a considerable benefit. Even with this support, proof scripts still leave something to be desired, which becomes more apparent by juxtaposing them with proof objects. Every sub-part of a proof object corresponds to a derivation in itself and has a clearly defined set of prerequisites and conclusions, based on the logical rules. Proof scripts, on the other hand, do not have a similar property. Every sub-part of a proof script critically depends on the proof state precisely before that sub-part begins. This proof state cannot be determined save for evaluating the proof script up to that point; there is no way to identify the general circumstances under which that sub-part of the proof script applies. This hurts the *composability* of proof scripts, precluding isolation of interesting parts to reuse in other situations.

Coq

Coq [Bertot et al., 2004] is considered to be the current state-of-the-art proof assistant stemming from the LCF tradition. It has a wide variety of features over proof assistants such as HOL and Isabelle: it uses CIC [Coquand and Huet, 1988, Werner, 1994] as its logical theory, which is a version of Martin-Löf type theory [Martin-Löf, 1984] and enables the encoding of mathematical results that are not easily expressible in HOL; it supports the extraction of computational content out of proofs, resulting in programs that are correct by construction, by exploiting the Curry-Howard isomorphism; and it supports a rich notion of inductive and coinductive datatypes. Since our main focus is the extensibility of proof assistants with respect to automation for new domains, we will not focus on these features, as they do not directly affect the extensibility. We will instead focus on the support for a specialized tactic development language, called LTac; the inclusion of a conversion rule in the core logical theory; and the combined use of these features for a proof technique called proof-by-reflection. As we will see, Coq allows various languages for developing tactics and other automation procedures; Figure 2.2 summarizes these, shows how they are combined into the basic architecture of Coq, and compares between them. The comparison is based around three factors:

- *convenience*, based on the presence of pattern matching and other specialized constructs specifically tailored to writing proof-producing procedures vs. the requirement to use general-purpose constructs through encodings
- *the presence of types* that can capture logical errors inside tactics statically, and
- *expressivity*, depending on whether a full programming model with side-effects and data types can be used.

The basic architecture of Coq follows the basic description of the previous section, as well as the two-layer architecture described for Isabelle. The implementation layer of the proof assistant is programmed in ML. It includes an explicit notion of proof objects along with a proof checker for validating them; the implementation of basic proof-producing functions and tactics; and the implementation of the user-level language along with an interface for

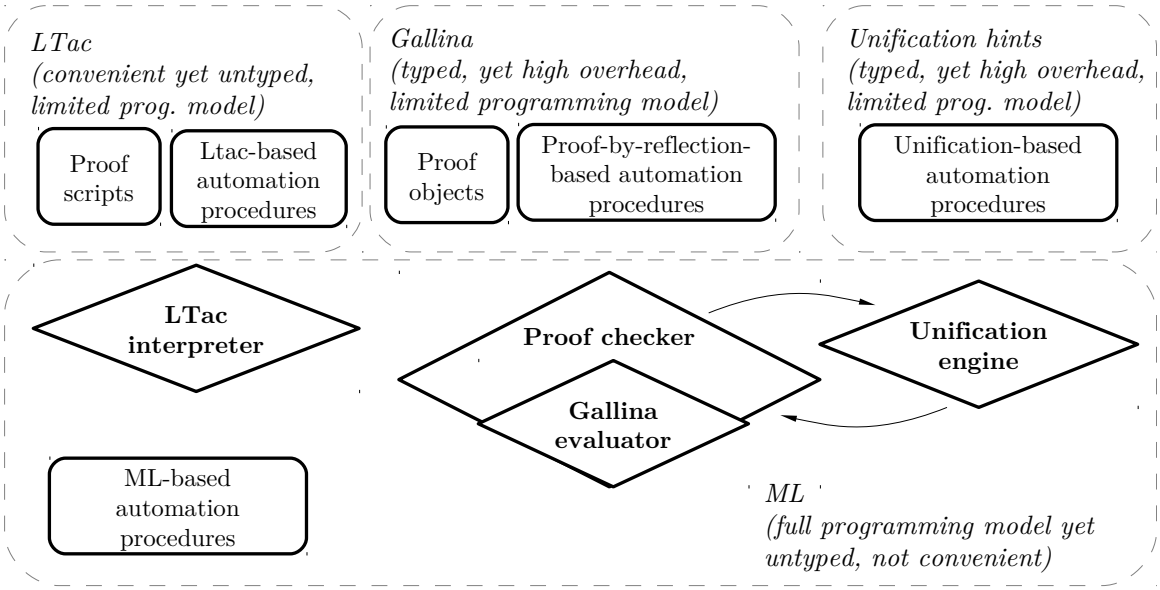

Figure 2.2: Programming languages available for writing automation procedures in the Coq architecture

it. Proof scripts written in this language make calls to tactics and ultimately produce proof objects which are then validated. Interactive support is available for developing the proof scripts.

The user-level language for proof development is called LTac [Delahaye, 2000, 2002]. It includes support not only for writing proof scripts, but also for writing new tactics at a higher level than writing them directly in ML. LTac provides constructs for pattern matching on propositions (in general on logical terms) and on the current proof state. Defining recursive functions and calling already defined tactics is also allowed. In this way, the overhead of writing new tactics is lowered significantly. Thus developing new domain-specific tactics as part of a proof development effort is re-emphasized and is widely regarded to be good practice [Chlipala, 2007, 2011, Morrisett et al., 2012]. Yet LTac has its fair share of problems. First of all, the programming model supported is still limited compared to writing tactics in ML: there is no provision for user-defined data structures or for mutable references. Also, LTac is completely untyped – both with respect to the logic (just as writing tactics in ML is), but also with respect to the limited data structure support that there is. There is no support for interactive development of tactics in the style supported

for proof scripts; such support would be impossible to add without having access to some typing information. Therefore programming tactics using LTac is at least as error-prone as developing them using ML. The untyped nature of the language makes LTac tactics brittle with respect to small changes in the involved logical definitions, posing significant problems in the maintenance and adaptation of tactics. Furthermore, LTac is interpreted. The interpretation overhead considerably limits the performance of tactics written using it. Because of these reasons, developing LTac tactics is often avoided despite the potential benefits that it can offer [Nanevski et al., 2010].

Another variation of the basic architecture that is part of the Coq proof assistant is the inclusion of a *conversion rule* in its logical core. Arguments that are based solely on computation of functions defined in the logic, such as the fact that $1+1=2$, are ubiquitous throughout formal proofs. If we record all the steps required to show that such arguments are valid as part of the proof objects, their size soon becomes prohibitively large. It has been argued that such arguments should not be recorded in formal proofs but rather should be automatically decided through computation – a principle that has been called the *Poincaré principle* [Barendregt and Geuvers, 1999]. Coq follows this idea through the conversion rule – a small-scale automation mechanism that implicitly decides exactly such computational equivalences. The implementation of the conversion rule needs to be part of the trusted proof checker. Also, because of the tight integration into the logical core, it is important to make sure that the equivalences decided by the conversion rule cannot jeopardize the soundness of the logic. Proving this fact is one of the main difficulties in establishing the soundness of the CIC logic that Coq is based on.

At the same time, extensions to the equivalences that can be decided through the conversion rule are desirable. The reason is straightforward: since these equivalences are decided automatically and implicitly, users do not have to explicitly allude to a tactic in order to prove them and thus the proof scripts can be more concise[5]. If we view these equivalences as trivial steps of a proof, deciding them implicitly corresponds accurately to the informal

5. A further benefit of an extended conversion rule is that it allows more terms to be typed. This is true in CIC because of the inclusion of dependent types in the logical theory. We will focus on a logic without dependent types, so we will not explore the consequences of this in this section. We refer the reader to Section 3.2 and 7.1 for more details.

practice of omitting them from our proofs. Therefore we would like the conversion rule to implicitly decide other trivial details, such as simple arithmetic simplifications, equational reasoning steps, rearrangement of logical connectives, etc. Such extensions to the conversion rule supported by Coq have been proposed [Blanqui et al., 1999, 2010] and implemented as part of the CoqMT project [Strub, 2010]. Furthermore, the NuPRL proof assistant Constable et al. [1986] is based on a different variant of Martin-Löf type theory which includes an extensional conversion rule. This enables arbitrary decision procedures to be added to conversion, at the cost of having to include all of them in the trusted core of the system.

In general, extensions to the conversion rule come at a significant cost: considerable engineering effort is required in order to change the internals of the proof assistant; the trusted core of the system needs to be expanded by the new conversion rule; and the already complex metatheoretic proofs about the soundness of the CIC logic need to be adapted. It is especially telling that the relatively small extension of η-conversion, a form of functional extensionality, took several years before it was considered logically safe to add to the base Coq system [Barras et al., 2012]. Because of these costs, user extensions to the conversion rule are effectively impossible.

Still, the conversion rule included in Coq is powerful enough to support another way in which automation procedures can be written, through the technique called *proof-by-reflection* [Boutin, 1997]. The main idea is to program automation procedures directly within the logic and then utilize the conversion rule in order to evaluate them. As described earlier, the conversion rule can decide whether two logical terms are equivalent up to evaluation – in fact, it is accurate to consider the implementation of the conversion rule as a *partial evaluator* for the functional language contained within the logic. We refer to this language as Gallina, from the name of the language used to write out proof objects and other logical terms in Coq. The technique works as follows: we reflect the set of propositions we want to decide as an inductive datatype in the logic; we develop the decision procedure in Gallina, as a function that works over terms of that datatype; and we prove that the decision procedure is sound – that is, when the decision procedure says that such an 'inductive proposition' is true, the original proposition it corresponds to is also provable. Now, a simple proof object which just calls the Gallina decision procedure on an inductive

proposition can serve as a proof for the original proposition, because the conversion rule will implicitly evaluate the procedure call. Usually, a wrapper LTac tactic is written that finds the inductive proposition corresponding to the current goal and returns such a proof object for it.

The proof-by-reflection technique leads to decision procedures that are correct by construction. There is a clear specification of the proof that the decision procedure needs to provide in each case – the proposition corresponding to the inductive proposition at hand. Thus programming using proof-by-reflection is typed with respect to logical terms, unlike programming decision procedures directly in ML. For this reason decision procedures developed in this style have much better maintainability characteristics. The bulk of the development of a decision procedure in this style is proving its soundness; this has a clear statement and can be proved through normal proof scripts, making use of the interactive proof support. Last, it can lead to fairly efficient decision procedures. This is true both because large proof objects do not need to be generated thanks to the conversion rule, and also because we can use a fast bytecode- or compilation-based backend as the evaluation engine of the conversion rule [Grégoire and Leroy, 2002, Grégoire, 2003]. This latter choice of course adds considerable complexity to the trusted core.

On the other hand, the technique is quite involved and is associated with a high overhead, as is apparent from its brief description above. Reflecting the propositions of the logic as an inductive type within the logic itself, as well as the encoding and decoding functions required, is a costly and tedious process. This becomes a problem when we want to reflect a large class of propositions, especially if this class includes propositions with binding structure, such as quantified formulas. The biggest problem is that the programming model supported for writing functions in Gallina is inherrently limited. Non-termination, as arising from general recursion, and any other kind of side-effectful operation is not allowed as part of Gallina programs. Including such side-effects would destroy the logical soundness of the system because the conversion rule would be able to prove invalid equivalences. Thus when we want to encode an algorithm that uses imperative data structures we are left with two choices: we either re-engineer the algorithm to work with potentially inefficient functional data structures; or we implement the algorithm in ML and use a reflective procedure to

validate the results of the algorithm – adding a third language into the mix. Various tactics included in Coq utilize this latter option.

A last Coq feature of note is the inclusion of a unification engine as part of the type inference algorithm for logical terms. It has recently been shown that this engine can be utilized to provide transparent automation support, through a mechanism called *canonical structures* [Gonthier et al., 2011]. Matita [Asperti et al., 2007], a proof assistant similar to Coq, offers the related mechanism of *unification hints* [Asperti et al., 2009]. This technique requires writing the automation procedure as a logic program. Therefore the issues we argued above for the Isabelle simplifier and for proof-by-reflection also hold for this technique.

2.1.3 Issues

We will briefly summarize the issues with the current architecture of proof assistants that we have identified in the discussion above. As we mentioned in the introduction, we perceive the support for extending the available automation to be lacking in current proof assistants – both with respect to small-scale and to large-scale automation. This ultimately leads to long proof scripts that include more details than necessary and limit the scalability of the overall proof development process. More precisely, the issues we have seen are the following.

1. Proof-producing functions such as tactics and decision procedures are programmed in an essentially untyped manner. We cannot specify the circumstances under which they apply and what proofs they are supposed to return. Programming such functions is thus error prone and limits their maintainability and composability.

2. No helpful information is offered to the user while developing such functions, for example in a form similar to the interactive development of proof scripts.

3. Though proof scripts can be evaluated partially in modern proof assistants, we cannot isolate sub-parts of proof scripts in order to reuse them in other situations. This is due to their implicit dependence on the current proof state which is only revealed upon evaluation. There is no way to identify their general requirements on what the proof state should be like.

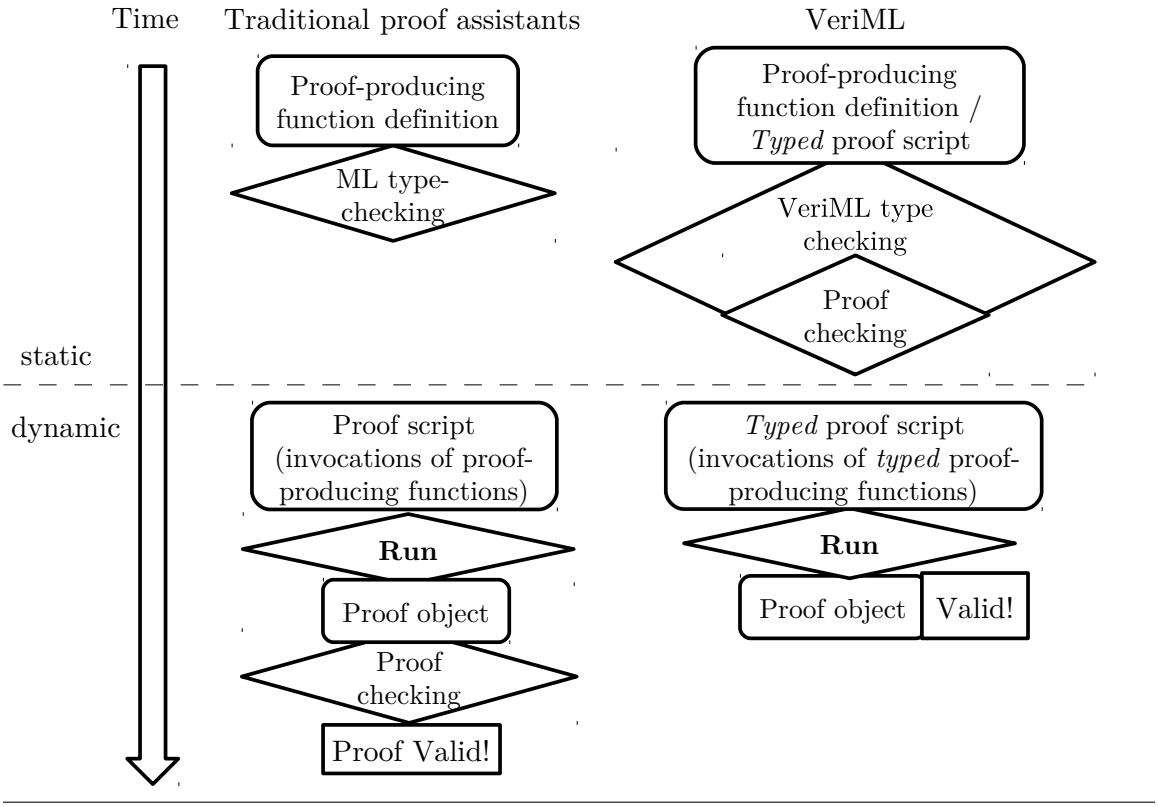

Figure 2.3: Comparison of architecture between current proof assistants and VeriML

4. The programming model offered through small-scale automation mechanisms is limited and is not a good target for many automation algorithms that employ mutable data structures.

5. The small-scale automation mechanisms such as the conversion rule are part of the core implementation of the proof assistant. This renders user extensions to their functionality practically impossible.

6. Users have to master a large set of techniques and different programming models in order to provide the domain-specific automation they desire.

2.2 A new architecture: VeriML

In this dissertation we propose a new architecture for proof assistants that tries to address the problems identified above. The new architecture is based on a new programming language called VeriML, which is used as a proof assistant. The essence behind the new

architecture is to *move the proof checker inside the type system of the meta-language, so that proof-producing functions are verified once and forall, at the time of their definition – instead of validating the proof objects that they produce every time they are invoked.* We present the basic idea of this architecture in juxtaposition to the traditional proof assistant architecture in Figure 2.3. VeriML addresses the problems identified above as follows:

1. VeriML allows type-safe programming of proof-producing functions. This is achieved through a rich type system, allowing us to specify the relationships between input and output logical terms accurately.

2. VeriML offers rich information while developing proof-producing functions in the form of assumptions and current proof goals, similar to what is offered in interactive proof assistants for proof scripts.

3. Proof scripts are typed, because the functions they use are typed. Therefore they can be decomposed just like proof objects; their type makes evident the situations under which they apply and the proposition that they prove.

4. The extra typing allows us to support small-scale automation mechanisms that are extensible through the full VeriML programming model. We demonstrate this through a new approach to the conversion rule, where arbitrary functions of a certain type can be added to it. The type of these functions specifies that they return a valid proof of the equivalence they claim, thus making sure that logical soundness is not jeopardized.

5. Small-scale automation mechanisms are implemented as normal VeriML code; new mechanisms can be implemented by the user. We regain the benefits of including them in the core of the proof assistant through two simple language constructs: proof-erasure and staging support. Proof-erasure enables us to regain and generalize the reduction in proof object size; staging enables us to use such mechanisms ubiquitously and implicitly, not only for checking proof scripts, but tactics as well.

6. There is a single programming model that users have to master which is general-purpose and allows side-effectful programming constructs such as non-termination, mutable references and I/O.

Furthermore, the language can potentially be used for other applications that mix computation and proof, which cannot easily be supported by current proof assistants. One such example is certifying compilation or certifying static analysis.

VeriML is a computational language that manipulates logical terms of a specific logic. It includes a core language inspired by ML (featuring algebraic datatypes, polymorphism, general recursion and mutable references), as well as constructs to introduce and manipulate logical terms. Rich typing information about the involved logical terms is maintained at all times. The computational language is kept separate from the logical language: logical terms are part of the computational language but computational terms never become part of the logical terms directly. This separation ensures that soundness of the logic is guaranteed and is independent of the specifics of the computational language.

Let us now focus on the rich typing information available for logical terms. First, we can view the rules for formation of propositions and domain objects, as well as the logical rules utilized for constructing proof objects as a type system. Each logical term can thus be assigned a type, which is a logical term in itself – for example, a proof object has the proposition it proves as its type. Viewing this type system as an 'object' language, the computational language provides constructs to introduce and manipulate terms of this object language; their computational-level type includes their logical-level type. An example would be an automated theorem prover that accepts a proposition P as an argument (its type specifying that P needs to be a well-formed proposition) and returns a proof object which proves that specific proposition (its type needs to be exactly P). Since the type of one term of the object language might depend on another term of the same language, this is a form of *dependently typed programming*.

The computational language treats these logical terms as actual runtime data: it provides a way to look into their structure through a pattern matching construct. In this way, we can implement the automated theorem prover by looking into the possible forms of the proposition P and building an appropriate proof object for each case. In each branch, type checking ensures that the type of the resulting proof object matches the expected type, after taking into account the extra information coming from that branch. Thus pattern matching is dependently typed as well.

The pattern-matching constructs, along with the core ML computational constructs, allow users to write *typed proof-producing functions*. Based on their type, we clearly know under which situations they apply and what their consequences are, at the point of their definition. Furthermore, while writing such functions, the user can issue queries to the VeriML type checker in order to make use of the available type information. In this way, they will learn of the hypotheses at hand and the goals that need to be proved at yet-unwritten points of the function. This extends the benefit of interactive proof development offered by traditional proof assistants to the case of proof-producing functions.

Typed proof scripts arise naturally by composing such functions together. They are normal VeriML expressions whose type specifies that they yield a proof of a specific proposition upon successful evaluation; for this reason we also refer to them as *proof expressions*. The evolution of proof state in these scripts is captured explicitly in their types. Thus, we know the hypotheses and goals that their sub-parts prove, enabling us to reuse and compose proof scripts. Interactive development of proof scripts is offered by leaving parts of proof scripts unspecified, and querying the VeriML type checker. The returned type information corresponds exactly to the information given by a traditional interactive proof assistant.

An important point is that despite the types assigned to tactics and proof scripts, the presence of side-effects such as non-termination means that their evaluation can still fail. Thus, proof scripts need to be evaluated in order to ensure that they evaluate successfully to a proof object. Type safety ensures that if evaluation is indeed successful, the resulting proof object will be a valid proof, proving the proposition that is claimed from the type system.

Small-scale automation. We view the type checker as a 'behind-the-scenes' process that gives helpful information to the user and prevents a large class of errors when writing proofs and proof-producing functions. This mitigates the problems (1), (2), (3) identified above that current proof assistants have. But what about support for small-scale automation? Our main intuition is to view small-scale automation mechanisms as *normal proof-producing functions*, with two extra concerns – which are traditionally mixed together. The first concern is that these mechanisms apply ubiquitously, transparently to the user, taking care

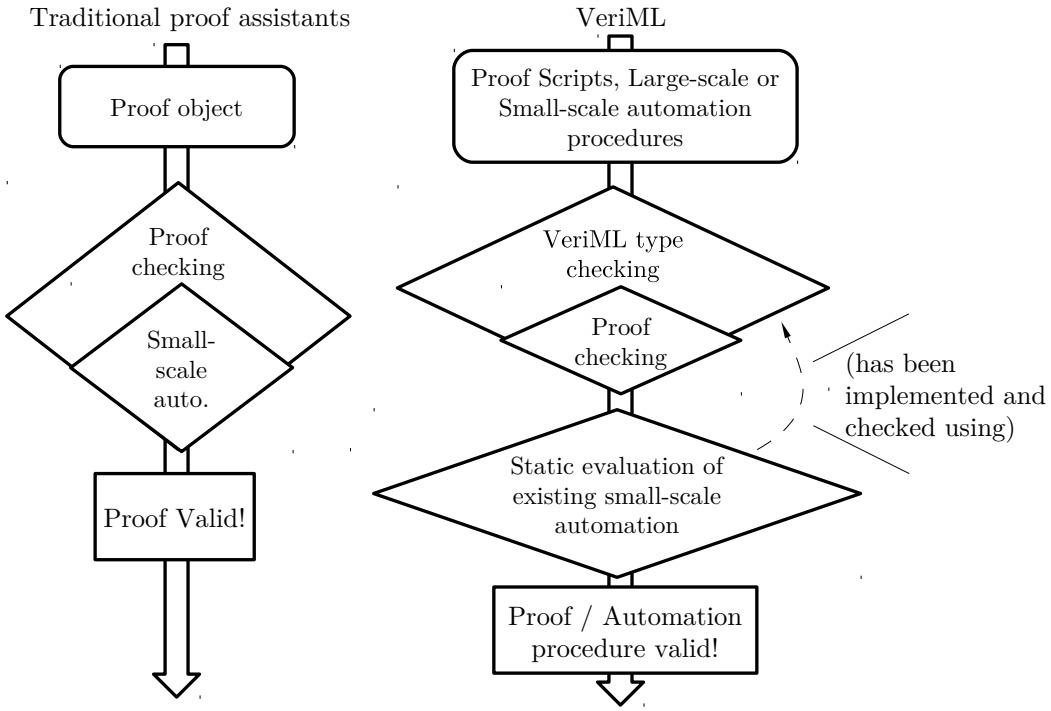

Figure 2.4: Comparison of small-scale automation approach between current proof assistants and VeriML

of details that the user should not have to think about. The second concern is that in some cases, as in the case of the conversion rule supported in Coq, these mechanisms are used for optimization purposes: by including them in the trusted core of the proof assistant, we speed up proof object generation and checking.

With respect to the first concern, we view small-scale automation as another phase of checking, similar to typing, as presented in Figure 2.4. Just like typing, it should happen 'behind-the-scenes' and offer helpful information to the user: when we allude to a fact that holds trivially, small-scale automation should check if that fact indeed holds and provide sufficient proof. This is similar to a second phase of type checking, save for the fact that it should be extensible. This checking should apply both in the case of developing a proof and in the case of developing a proof-producing function – otherwise the function could fail at exactly those points that were deemed too trivial to explicitly prove. The user should be able to provide their own small-scale automation procedures, tailored to their domain of interest. This view generalizes the notion of what small-scale automation should be about in proof assistants; we believe it better reflects the informal practice of omitting details.

Based on this idea, we can say that the main distinction between small-scale automation and large-scale automation is only a matter of the phase when they are evaluated. Small-scale automation should be evaluated during a phase similar to type-checking, so that we know its results at the definition time of a proof script or a proof-producing function. Other than that, both should be implemented through exactly the same code. We introduce such a phase distinction by adding a *static evaluation* phase. This phase occurs after type checking but before normal runtime evaluation. We annotate calls to the small-scale automation mechanisms to happen during this phase through a simple staging construct. Other than this annotation, these calls are entirely normal calls to ordinary proof-producing functions. We can even hide such calls from the user by employing some simple syntactic sugar. Still the user is able to use the same annotations for new automation mechanisms that they develop.

The second concern that small-scale automation addresses, at least in the case of the conversion rule, is keeping the proof object size tractable. We view the conversion rule as a trusted procedure that is embedded within the proof checker of the logic, one that decides equivalences as mentioned. The fact that we trust its implementation is exactly why we can omit the proof of those equivalences from the proof objects. We can therefore say that the conversion rule is a special proof-producing procedure that does not produce a proof – because we trust that such a proof exists.

Consider now the implementation of the conversion rule as a function in VeriML. We can ascribe a strong type to this function, where we specify that if this function says that two propositions are equivalent, it should in fact return a proof object of that equivalence. Because of the type safety of VeriML, we know that if a call to this function evaluates successfully, such a proof object will indeed be returned. But we can take this one step further: the proof object does not even need to be generated yet is guaranteed to exist because of type safety. We say that VeriML can be evaluated under *proof-erasure semantics* to formally describe this property. Therefore, we can choose to evaluate the conversion rule under proof-erasure, resulting in the same space savings as in the traditional approach. Still the same semantics can be used for further extensions for better space and time savings. By this account, it becomes apparent that the conversion rule can be removed from the

trusted core of the system and from the core logical theory as shown in Figure 2.4, without losing any of the benefits that the tight integration provides.

Overall, we can view VeriML as an evolution of the tradition of the LCF family, informed by the features of the current state-of-the-art proof assistants. In the LCF family, both the logic and the logic-manipulating functions are implemented within the same language, ML. In VeriML, the implementation of the logic becomes part of the base computational language. Logic-manipulating functions can thus be given types that include the information gathered from the type system of the logic, in order to reflect the properties of the logical terms involved. This information enables interactive development of proof scripts and proof-producing functions and also leads to a truly extensible approach to small-scale automation, generalizing the benefits offered by current proof assistants in both cases.

2.3 Brief overview of the language

In this section, we will give a brief informal overview of the main features of VeriML. We will present them mostly through the development of a procedure that proves propositional tautologies automatically. We assume some familiarity with functional programming languages in the style of ML or Haskell.

A simple propositional logic

In order to present the programming constructs available in VeriML in detail, we will first need to be more concrete with respect to what our logic actually is – what propositions and proofs we will manipulate and produce. For the purposes of our discussion, we will choose a very simple propositional logic which we briefly present here. Through this logic, we can state and prove simple arguments of the form:

1. If it is raining and I want to go outside, I need to get my umbrella.
2. It is raining.
3. I want to go outside.

4. Therefore, I need to get my umbrella.

This logic consists of two classes of terms: propositions and proofs. Propositions are either atomic propositions (e.g. "it is raining"), denoted as A, which state a fact and are not further analyzable, or are combinations of other propositions through logical connectives: conjunction, denoted as $\wedge$; disjunction, denoted as $\vee$; and implication, denoted as $\supset$. We use capital letters P, Q, R to refer to propositions. The propositions of our logic are formed through the following grammar:

$$P ::= P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \supset P_2 \mid A$$

Proofs are evidence that certain propositions are valid. We will give precise rules for a specific form that such evidence might take; we will therefore describe a *formal proof system*. Every rule has some premises and one consequence. The premises describe what proofs are required. By filling in a proof for each required premise, we get a proof for the consequence. We write rules in a vertical style, where premises are described in the top and the consequence in the bottom, divided by an horizontal line. Proofs therefore have a bottom-up tree-like structure, with the final consequence as the root of the tree at the very bottom of the proof. Proofs can also make use of hypotheses, which are the set of leaves of this tree. A hypothesis corresponds to an assumption that a proof of a certain proposition exists.

Every connective is associated with two sets of logical rules: a set of *introduction* rules, which describe how we can form a proof of a proposition involving that connective, and a set of *elimination* rules, which describe what further proofs we can construct when we have a proof about that connective at hand. The logical rules for conjuction are the simplest ones:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ P & Q\end{array}}{P \wedge Q}\ \wedge\textsc{Intro} \qquad \frac{\begin{array}{c}\vdots \\ P \wedge Q\end{array}}{P}\ \wedge\textsc{Elim1} \qquad \frac{\begin{array}{c}\vdots \\ P \wedge Q\end{array}}{Q}\ \wedge\textsc{Elim2}$$

The content of these rules is as follows. The introduction rules says that to form a proof of a conjunction, we need two proofs as premises, whose consequences are the two propositions

involved. Inversely, the elimination rules say that out of a proof of a conjunction, we can extract a proof of either proposition, using the corresponding rule.

$$\dfrac{\begin{array}{c}\overline{}\\ P\\ \vdots\\ Q\end{array}}{P \supset Q}\ \supset\textsc{Intro} \qquad\qquad \dfrac{\begin{array}{cc}\vdots & \vdots\\ P \supset Q & P\end{array}}{Q}\ \supset\textsc{Elim}$$

Introduction of implication is a little bit more complicated: it is proved through a proof of the proposition Q, where we can make use of an extra hypothesis for P – that is, an assumption that we have a proof of proposition P. This is what the notation in the introduction rule stands for. Elimination of implication corresponds to the well-known *modus ponens* reasoning principle.

The rules about disjunction follow. The introduction rules are straightforward; the elimination rule says that given a proof of $P \vee Q$, if we can prove a third proposition R starting with either a proof of P or a proof of Q as hypotheses, then we have a proof of R without any extra hypotheses.

$$\dfrac{\begin{array}{c}\vdots\\ P\end{array}}{P \vee Q}\ \vee\textsc{Intro1} \qquad \dfrac{\begin{array}{c}\vdots\\ Q\end{array}}{P \vee Q}\ \vee\textsc{Intro2} \qquad \dfrac{\begin{array}{ccc}\overline{P} & \overline{Q} & \\ \vdots & \vdots & \vdots\\ R & R & P \vee Q\end{array}}{R}\ \vee\textsc{Elim}$$

Based on these rules, we can form simple proofs of propositional tautologies. For example, we can prove that $P \supset (P \wedge P)$ as follows.

$$\dfrac{\dfrac{\overline{P} \qquad \overline{P}}{P \wedge P}}{P \supset (P \wedge P)}$$

Logical terms as data

In VeriML, logical terms such as propositions and proofs are a type of data – similar to integers, strings and lists. We therefore have constant expressions in order to introduce them, just as we have constant expressions such as 1, 2, and `"hello world"` for normal data types. We use $\langle P \rangle$ as the constant expression for a proposition P, thus differentiating the computational expression from the normal logical term.

Unlike normal types like integers, the type of these constant expressions is potentially different based on the logical term we introduce. In the case of proofs, we assign its consequence –the proposition that it proves– as its type. Therefore logical terms have types that involve further logical terms. We denote the *type* corresponding to the logical term P as (P). Therefore, the constant expression corresponding to the proof given in the previous section will be written and typed as follows. Note that we use a shaded box for types; this is meant as an annotation which users of the language do not have to write, but is determined through the type checker.

$$\left\langle \dfrac{\dfrac{\overline{P} \quad \overline{P}}{P \wedge P}}{P \supset P \wedge P} \right\rangle : \boxed{(P \supset P \wedge P)}$$

Propositions get assigned a special logical term, the sort *Prop*. Other than this, they should be perceived as normal data. For example, we can form a list of propositions, as follows:

$$[\langle P \supset Q \rangle ; \langle P \wedge Q \vee R \rangle] : \boxed{(\mathit{Prop})\ \mathsf{list}}$$

Dependent functions and tuples

Let us now consider our motivating example: writing a procedure that automatically proves propositional tautologies – propositions that are always valid, without any hypotheses. What should the type of this procedure be? We want to specify the fact that it accepts a proposition and produces a proof of that proposition. The type of the result is thus *dependent* on the input of the function. We introduce names in the types so as to be able to track such dependencies, allowing us to give the following type to our tautology prover:

$$\mathsf{tautology} : \boxed{(P : \mathit{Prop}) \rightarrow (P)}$$

Similarly, we can have *dependent tuples* – tuples where the type of the second component might depend on the first component. In this way we can form a pair of a proposition together with a proof for this proposition and use such pairs in order to create lists of proofs:

$$\left[\left\langle P \supset P, \frac{\overline{P}}{P \supset P}\right\rangle ; \left\langle P \supset P \wedge P, \frac{\dfrac{\overline{P} \quad \overline{P}}{P \wedge P}}{P \supset P \wedge P}\right\rangle\right] : (P : \mathit{Prop}, X : P)\ \mathsf{list}$$

Pattern matching on terms

Our automated prover is supposed to prove a proposition – but in order to do that, it needs to know what that proposition actually is. In order to be able to write such functions, VeriML includes a construct for pattern matching over logical terms. Let us consider the following sketch of our tautology prover:

$$\begin{array}{lcl}
\mathsf{tautology} & : & (P : \mathit{Prop}) \rightarrow (P) \\
\mathsf{tautology}\ P & = & \mathsf{match}\ P\ \mathsf{with} \\
 & & \quad Q \wedge R \ \mapsto\ \mathsf{let}\ X\ =\ \mathsf{tautology}\ \langle\ Q\ \rangle\ \mathsf{in} \\
 & & \qquad\qquad\quad \mathsf{let}\ Y\ =\ \mathsf{tautology}\ \langle\ R\ \rangle\ \mathsf{in} \\
 & & \qquad\qquad\quad \cdots \\
 & & \mid Q \vee R \ \mapsto\ \cdots
\end{array}$$

We pattern match on the input proposition P in order to see its topmost connective. In the conjunction case, we recursively try to find a proof for the two propositions involved. In the disjunction case, we need to try finding a proof for either of them. Our prover will not always be successful in finding a proof, since not all propositions are provable. Therefore we can refine the type of our prover so that it optionally returns a proof – through the use of the familiar ML option type. The new sketch looks as follows, where we use monadic syntax for presentation purposes:

```
tautology    :  (P : Prop) → (P) option
tautology P  =  match P with
                  Q ∧ R  ↦  do  X ← tautology ⟨ Q ⟩ ;
                                Y ← tautology ⟨ R ⟩ ;
                                return ···1
                | Q ∨ R  ↦  (do X ← tautology ⟨ Q ⟩ ;
                               return ···2) ||
                            (do Y ← tautology ⟨ R ⟩ ;
                               return ···3)
```

What do the variables X and Y stand for? The evaluation of the recursive calls such as tautology Q will result in a proof which will have Q as the consequence and an arbitrary set of hypotheses. Thus X is a variable that stands for a proof of Q; it will be replaced by an actual proof once the call to tautology is evaluated[6]. Though we can write the type of X as (Q), another way to write it is $X : \begin{array}{c}\vdots\\ Q\end{array}$, reflecting the fact that X can be viewed as a proof of Q. The above description of the nature of X is quite similar to a hypotheses of Q used inside a proof. Yet there is a difference: hypotheses stand for *closed* proofs, that is, proofs without additional hypotheses; whereas X stands for an *open* proof, which can have arbitrary additional hypotheses. This is why we refer to X as a *meta-variable*. We can use such meta-variables inside proofs, in which case we can perceive them as holes that are later filled in with the actual proofs.

Let us now focus on the missing parts. By issuing a query to the VeriML typechecker for this incomplete program, we will get back the following information:

6. The astute reader will notice that we are being a bit imprecise here in the interest of presentation. More accurately, the call to tautology will result in a constant expression enclosing a proof with Q as a consequence, and an arbitrary set of hypotheses. As written in the code example, X is a variable that stands for such a constant. Instead, we should write $\langle\, X \,\rangle \leftarrow$ tautology Q, making the fact that X refers to the proof enclosed within the returned constant apparent.

$$\frac{\begin{array}{l}P, Q, R : (\mathit{Prop}) \\ X : (Q) \\ Y : (R)\end{array}}{\cdots_1 : (Q \wedge R)} \qquad \frac{\begin{array}{l}P, Q, R : (\mathit{Prop}) \\ X : (Q)\end{array}}{\cdots_2 : (Q \vee R)} \qquad \frac{\begin{array}{l}P, Q, R : (\mathit{Prop}) \\ Y : (R)\end{array}}{\cdots_3 : (Q \vee R)}$$

The typechecker informs us of the proofs that we have at hand in each case and of the type of the VeriML expression that we need to fill in. When we fill them in, the typechecker will also make sure that the proofs indeed follow the rules of the logic and have the right consequences, based on their required type. In this way many errors can be caught early. The rich information contained in these types and the ability to use them to check the proofs enclosed in functions such as tautology is one of the main departures of VeriML compared to traditional proof assistants, where all the proofs inside such functions show up as having the same type.

We note that the typechecker has taken into account the information for each particular branch in order to tell what proof needs to be filled in. We thus say that pattern matching is dependent. Based on the provided information, it is easy to fill in the missing expressions:

$$\cdots_1 = \left\langle \frac{X : \begin{array}{c}\vdots \\ Q\end{array} \qquad Y : \begin{array}{c}\vdots \\ R\end{array}}{Q \vee R} \right\rangle \qquad \cdots_2 = \left\langle \frac{X : \begin{array}{c}\vdots \\ Q\end{array}}{Q \vee R} \right\rangle \qquad \cdots_3 = \left\langle \frac{Y : \begin{array}{c}\vdots \\ R\end{array}}{Q \vee R} \right\rangle$$

If we made a typo when filling in $\cdots_1$, such as:

$$\cdots_1 = \left\langle \frac{X \qquad X}{Q \vee R} \right\rangle$$

the type checker would be able to inform us of the error and decline the definition of tautology.

Contextual terms

The previous sketch of the automated prover still lacks two things: support for logical implication as well as (perhaps more importantly) a base case. In fact, both of these are impossible to implement in a meaningful way based on what we have presented so far, because of a detail that we have been glossing over: hypotheses contexts.

Consider the implication introduction rule given above.

$$\dfrac{\begin{array}{c}\overline{}\\ P\\ \vdots\\ Q\end{array}}{P \supset Q}\ \supset\textsc{Intro}$$

The premise of the rule calls for a proof of Q under the assumption that P holds. So far, we have been characterizing proofs solely based on the proposition that they prove, so we cannot capture the requirement for an extra assumption. Essentially, we have been identifying all proofs of the same proposition that depend on different hypotheses. This is clearly inadequate: consider the difference between a proof of P with no hypotheses and a proof of P with a hypothesis of P.

In order to fix this issue, we will introduce a context of hypotheses Φ. We can reformulate our logical rules so that they correspond to *hypothetical judgements*, where the dependence on hypotheses is clearly recorded. We write $\Phi \vdash P$ for a proof where the context of hypotheses is Φ and the consequence is P. The new formulation of the implication rules is as follows; the other rules are re-formulated similarly.

$$\dfrac{\Phi,\ P \vdash Q}{\Phi \vdash Q}\ \supset\textsc{Intro} \qquad\qquad \dfrac{\Phi \vdash P \supset Q \qquad \Phi \vdash P}{\Phi \vdash Q}\ \supset\textsc{Elim}$$

Furthermore, we will make two changes to the computational constructs we have seen. First, the types of the proofs will not simply be their consequence, but will record the hypotheses context as well, reflecting the new judgement form $\Phi \vdash P$. We can also say that the type of proofs are now *contextual terms* instead of simple logical terms – that is, they are

logical terms with their context (the hypothesis context they depend on) explicitly recorded. Second, we will add *dependent functions over contexts* which are similar to the dependent functions over logical terms that we have seen so far. We will use this latter feature in order to write code that works under any context Φ, just like our tautology prover.

Based on these, we give the new sketch of our prover with the implication case added:

$$
\begin{array}{lcl}
\mathsf{tautology} & : & (\Phi : \mathit{context}) \to (P : \mathit{Prop}) \to (\Phi \vdash P)\ \mathsf{option} \\
\mathsf{tautology}\ \Phi\ P & = & \mathsf{match}\ P\ \mathsf{with} \\
 & & \quad Q \wedge R \ \mapsto\ \mathsf{do}\ X \leftarrow \mathsf{tautology}\ \Phi\ \langle\, Q \,\rangle\ ; \\
 & & \qquad\qquad\qquad\quad Y \leftarrow \mathsf{tautology}\ \Phi\ \langle\, R \,\rangle\ ; \\
 & & \qquad\qquad\qquad\quad \mathsf{return}\ \cdots_1 \\
 & & \quad \mid Q \vee R \ \mapsto\ (\mathsf{do}\ X \leftarrow \mathsf{tautology}\ \Phi\ \langle\, Q \,\rangle\ ; \\
 & & \qquad\qquad\qquad\quad \mathsf{return}\ \cdots_2)\ \|\| \\
 & & \qquad\qquad\qquad\quad (\mathsf{do}\ Y \leftarrow \mathsf{tautology}\ \Phi\ \langle\, R \,\rangle\ ; \\
 & & \qquad\qquad\qquad\quad \mathsf{return}\ \cdots_3) \\
 & & \quad \mid Q \supset R \ \mapsto\ \mathsf{do}\ X \leftarrow \mathsf{tautology}\ (\Phi,\ Q)\ \langle\, R \,\rangle\ ; \\
 & & \qquad\qquad\qquad\quad \mathsf{return}\ \left\langle \dfrac{X : \Phi,\ Q \vdash R}{\Phi \vdash Q \supset R} \right\rangle
\end{array}
$$

A similar problem becomes apparent if we inspect the propositions that we have been constructing in VeriML under closer scrutiny: we have been using the names P, Q, etc. for atomic propositions – but where do those names come from? The answer is that those come from a variables context, similar to how proofs for assumptions come from a hypotheses context. In the logic that VeriML is based on, these contexts are mixed together. In order to be able to precisely track the contexts that propositions and proofs depend on, all logical terms in VeriML are contextual terms, as are their types.

Pattern matching on contexts

The prover described above is still incomplete, since it has no base case for atomic propositions. For such propositions, the only thing that we can do is to search our hypotheses to see if we already possess a proof for them. Now that the hypotheses context is explicitly represented, we can view it as data that we can look into, just as we did for logical terms. VeriML therefore has a further pattern matching construct for matching over contexts.

We will write a function findHyp that tries to find whether a specific hypothesis exists in the current context, by looking through the context for an exact match. Note the use of the variable H in the first inner branch: the pattern will match if the current element of the context is a hypothesis which matches the desired proposition H exactly. Contrast this with the last branch, where we use a wildcard pattern to match any other hypothesis.

$$
\begin{array}{lcl}
\mathsf{findHyp} & : & (\Phi : \mathit{context}) \to (H : \mathit{Prop}) \to (\Phi \vdash H)\ \mathsf{option} \\
\mathsf{findHyp}\ \Phi\ H & = & \mathsf{match}\ \Phi\ \mathsf{with} \\
 & & \quad \emptyset \mapsto \mathsf{None} \\
 & & \quad \mid (\Phi',\ H') \mapsto \mathsf{match}\ H'\ \mathsf{with} \\
 & & \qquad H \mapsto \mathsf{return}\ \left\langle \dfrac{}{\Phi',\ H \vdash H} \right\rangle \\
 & & \qquad \mid \_ \mapsto \mathsf{findHyp}\ \Phi'\ \langle\ H\ \rangle
\end{array}
$$

Furthermore, we add a case at the end of our prover in order to handle atomic propositions using this function.

$$
\begin{array}{lcl}
\mathsf{tautology}\ \Phi\ P & = & \mathsf{match}\ P\ \mathsf{with} \\
 & & \quad \ldots \\
 & & \quad \mid P \mapsto \mathsf{findHyp}\ \Phi\ \langle\ P\ \rangle
\end{array}
$$

With this, the first working version of our prover is complete. We can now write a simple proof script in order to prove the tautology $P \supset P \wedge P$, as follows:

$$\mathsf{let\ tauto1}\ :\ (\vdash P \supset P \wedge P) = \mathsf{tautology}\ \emptyset\ \langle P \supset P \wedge P \rangle$$

This will return the proof object for the proposition we gave earlier. Through type inference the VeriML type checker can tell what the arguments to tautology should be, so users would leave them as implicit arguments in an actual proof script.

It is easy to modify our tautology prover to be more open-ended, by allowing the use of functions other than findHyp to handle the base cases. We can do this thanks to the support for higher-order functions of the ML core of our language. Consider the following auto function. Most of the branches are identical to tautology, save for the base case which makes a call to another proving function, supplied as an input argument.

```
auto : ((Φ : context) → (P : Prop) → (Φ ⊢ P) option)
         → (Φ : context) → (P : Prop) → (Φ ⊢ P) option
auto baseprover Φ P = match P with
                        ··· (same as tautology) ···
                      | P ↦ baseprover Φ ⟨ P ⟩
```

We can now use different base provers in combination with auto. For example, we can combine auto with an arithmetic theorem prover: a function arith that can handle goals of the form $\Phi \vdash P$ where all hypothesis in Φ and the desired consequence in P are atomic propositions involving arithmetic. In this way, the arithmetic prover is extended into a prover that can handle propositions involving both arithmetic and logical connectives – with the logical reasoning handled by auto. By combining functions in this way, we construct proof expressions that compute proofs for the desired propositions.

Mixing imperative features

Our prover so far has been purely functional. Even though we are manipulating terms such as propositions and contexts, we still have access to the full universe of data structures that exist in a normal ML implementation – such as integers, algebraic data types (e.g. lists that we saw above), mutable references and arrays. We can include logical terms as parts of such data structures without any special provision. One simple example would be a memoized version of our prover, which keeps a cache of the propositions that it has already proved, along with their proofs.

```
tautologyCache      : (Φ : context, P : Prop, Φ ⊢ P) option array
memoizedTautology   : (Φ : context) → (P : Prop) → (Φ ⊢ P) option
memoizedTautology Φ P =
    let entry = hash ⟨Φ, P⟩ in
    match tautologyCache[entry] with
      Some ⟨ Φ, P, X ⟩ ↦ Some ⟨ X ⟩
    | _                ↦ do X ← tautology Φ ⟨ P ⟩ ;
                            tautologyCache[entry] := ⟨Φ, P, X⟩ ;
                            return ⟨ X ⟩
```

Staging

Let us now assume that we have developed our prover further, to be able to prove more tautologies compared to the rudimentary prover given above. Suppose we want to write a function that performs a certain simplification on a given proposition and returns a proof that the transformed proposition is equivalent to the original one. It will have the following type:

$$\mathsf{simplify} \ : \ \big(P : \mathit{Prop}\big) \to \big(Q : \mathit{Prop},\ \emptyset \vdash P \supset Q \wedge Q \supset P\big)$$

Consider a simple case of this function:

$$\begin{array}{lcl} \mathsf{simplify}\ P & = & \mathsf{match}\ P\ \mathsf{with} \\ & & \quad Q \wedge R \supset O \ \mapsto\ \langle Q \supset R \supset O, \cdots \rangle \end{array}$$

If we query the VeriML typechecker about the type of the missing part, we will learn the proposition that we need to prove:

$$\frac{P, Q, R, O : \ \big(\mathit{Prop}\big)}{\cdots : \ ((Q \wedge R \supset O) \supset (Q \supset R \supset O)) \wedge ((Q \supset R \supset O) \supset (Q \wedge R \supset O))}$$

How do we fill in this missing part? One option is to replace it by a complete proof. But we already have a tautology prover that is able to handle this proposition, so we would rather avoid the manual effort. The second option would be to replace this part with a call to the tautology prover: $\cdots = \mathsf{tautology}\ \_\ \_$, where we have left both arguments implicit as they can be easily inferred from the typing context. Still, this would mean that every time the tactic is called and that branch is reached, the same call to the prover would need to be evaluated. Furthermore, we would not know whether the prover was indeed successful in proving this proposition until we reach that branch – which might never happen if it is a branch not exercised by a test case! The third option is to separate this proposition into a lemma, similar to what we did for tauto1, and prove it prior to writing the simplify function. This option is undesirable too, because the statement of the lemma itself is not

interesting; furthermore it is much more verbose than the call to the prover itself. Having to separate such uninteresting proof obligations generated by functions such as simplify as lemmas would soon become tedious.

In VeriML we can solve this issue through the use of staging. We can annotate expressions so that they are staged – they are evaluated during a phase of evaluation that happens after typing but before runtime. These expressions can be inside functions as well, yet they are still evaluated at the time that these functions are defined. Through this approach, we get the benefits of the separation into lemmas approach but with the conciseness of the direct call to the prover. The example code would look as follows. We use brackets as the staging annotation for expressions.

$$\begin{array}{lcl} \mathsf{simplify}\ P & = & \mathsf{match}\ P\ \mathsf{with} \\ & & \quad Q \wedge R \supset O \ \mapsto\ \langle Q \supset R \supset O, \{\mathsf{tautology}\ \_\ \_\}\rangle \end{array}$$

After type checking, this expression is made equivalent to the following, where the missing arguments are filled in based on the typing information.

$$\begin{array}{lcl} \mathsf{simplify}\ P & = & \mathsf{match}\ P\ \mathsf{with} \\ & & \quad Q \wedge R \supset O \ \mapsto\ \langle Q \supset R \supset O, \{\mathsf{tautology}\ \emptyset\ S\}\rangle \\ \mathsf{where}\ S = ((Q \wedge R \supset O) \supset (Q \supset R \supset O)) \wedge ((Q \supset R \supset O) \supset (Q \wedge R \supset O)) & & \end{array}$$

After the static evaluation phase, the staged expression gets replaced by the proof that is returned by the tautology prover.

$$\begin{array}{lcl} \mathsf{simplify}\ P & = & \mathsf{match}\ P\ \mathsf{with} \\ & & \quad Q \wedge R \supset O \ \mapsto\ \langle Q \supset R \supset O, X\rangle \\ \mathsf{where}\ X = \dfrac{\dfrac{}{\vdots}}{\emptyset \vdash S} & & \\ \mathsf{where}\ S = ((Q \wedge R \supset O) \supset (Q \supset R \supset O)) \wedge ((Q \supset R \supset O) \supset (Q \wedge R \supset O)) & & \end{array}$$

Proof erasure

The last feature of VeriML that we will consider is proof erasure. Based on the type safety of the language, we know that expressions of a certain type will indeed evaluate to values of the same type, if successful. For example, consider an expression that uses functions such as tautology in order to prove a proposition P: if the expression does evaluate to a result (instead of diverging or throwing an exception), that result will be a valid proof of P.

We have said that we are able to pattern match on logical terms such as propositions and proofs. If we relax this so that we cannot pattern match on proof terms, type safety can give us the further property of proof-erasure: if we delete all proofs from our VeriML expressions and proceed to evaluate these expressions normally, their runtime behavior is exactly the same as before the deletion – other than the space savings of not having to realize the proofs as data in the computer memory. Therefore, even if we do not generate these proofs, valid proof objects are still guaranteed to exist. Relaxing the pattern matching construct as such is not a serious limitation, as we are not interested in the particular structure of a proof but only on its existence.

For purposes of our discussion, we will show what the proof-erased version of the prover we have written above would look like.

```
tautology        :  (Φ : context) → (P : Prop) → unit option
tautology Φ P    =  match P with
                      Q ∧ R  ↦  do tautology Φ Q ;
                                   tautology Φ R ;
                                   return ()
                    | Q ∨ R  ↦  (do tautology Φ Q ;
                                    return ()) ||
                                (do tautology Φ R ;
                                    return ())
                    | Q ⊃ R  ↦  do tautology (Φ, Q) R ;
                                   return ()
```

This code looks very similar to what we would write in a normal ML function that did not need to produce proofs. This equivalence is especially evident if we keep in mind that

unit option is essentially the boolean type, monadic composition corresponds to boolean AND, monadic return to boolean true and the || operation to boolean OR.

The space savings associated with proof erasure are considerable, especially in the case of functions that are used ubiquitously in order to produce proofs, such as the *small-scale automation* mechanisms that we mentioned earlier in this chapter. Still, full proof erasure means that we have to completely trust the VeriML implementation. Instead we will use proof erasure selectively, e.g. in order to evaluate the functions comprising the conversion rule, enlarging the trusted base only by the proof-erased versions of those functions. Yet a receiver of a VeriML proof can choose to re-evaluate these functions without proof erasure and get complete proofs without missing parts. Therefore the trusted base of our system is not decided at the design time of the logic we are going to use, but at the sites of the receivers of the proofs. This allows a wide range of possibilities in choosing between the level of rigor required and the cost associated with proof checking, and is a significant advancement from the traditional way of handling proofs as certificates sent to a third party.

Last, we would like to briefly comment on the fact that the combination of proof erasure with the staging annotation is very powerful, as it allows us to *extend the notion of what constitutes a valid proof*. Take the following sketch of a proof as an example:

$$\left\langle \frac{\dfrac{}{\vdots}\ (x+y)^2 > 0 \qquad \{\mathsf{arith}\ \_\ \_\} : (x+y)^2 = x^2 + 2xy + y^2}{x^2 + 2xy + y^2 > 0} \right\rangle$$

In this proof, we use the automated arithmetic prover arith to validate a simple algebraic identity. By using proof erasure for this call, no proof object for this identity is ever generated – yet we are still guaranteed that a proof that uses only the initial logical rules (plus axioms of arithmetic) exists. Furthermore, this identity is validated statically: if we use the above proof inside a function such as tautology to fill in the proof obligations (as we did for $\cdots_1$ earlier), we will still know that the whole proof is valid at the time that the function is defined. This is due to staging the call to the arith function. This process roughly corresponds to extending the available logical rules with calls to proof-producing functions; yet we maintain the property that we can statically validate these "extended"

proofs. The fact that no evidence needs to be produced, means that we can effectively hide these trivial details – both in terms of user effort required, and of the cost of validating them.

Yet the automation functions handling these trivial details are developed in VeriML, which means that we can develop better versions of them in the same language to handle further details or further domains. Thus our approach to small-scale automation departs from the traditional approach of hardcoding fixed small-scale automation mechanisms in the core implementation of the proof assistant; it enables instead layered development of increasingly more sophisticated small-scale automation, in a manner resembling the informal practice of acquiring further intuition.

Chapter 3

The logic λHOL: Basic framework

In this chapter we will present the details of the λHOL logic that VeriML uses.

3.1 The base λHOL logic

The logic that we will use is a simple type-theoretic higher-order logic with explicit proof objects. This logic was first introduced in Barendregt and Geuvers [1999] and can be seen as a shared logical core between CIC [Coquand and Huet, 1988, Bertot et al., 2004] and the HOL family of logics as used in proof assistants such as HOL4 [Slind and Norrish, 2008], HOL-Light [Harrison, 1996] and Isabelle/HOL [Nipkow et al., 2002]. It is a constructive logic yet admits classical axioms.

The logic is composed of the following classes:

- The **objects of the domain of discourse**, denoted d: these are the objects that the logic reasons about, such as natural numbers, lists and functions between such objects.

- The **propositions**, denoted as P: these are the statements of the logic. As this is a higher-order logic, propositions themselves are objects of the domain of discourse d, having a special *Prop* type. We use P instead of d when it is clear from the context that a domain object is a proposition.

- The **kinds**, denoted $\mathcal{K}$: these are the types that classify the objects of the domain of

$$
\begin{array}{rrl}
\text{(sorts)} & s & ::= \mathit{Type} \mid \mathit{Type}' \\
\text{(kinds)} & \mathcal{K} & ::= \mathit{Prop} \mid c_{\mathcal{K}} \mid \mathcal{K}_1 \to \mathcal{K}_2 \\
\text{(dom.obj. and prop.)} & d, P & ::= P_1 \to P_2 \mid \forall x : \mathcal{K}.P \mid \lambda x : \mathcal{K}.d \mid d_1\ d_2 \mid x \mid c_d \\
\text{(proof objects)} & \pi & ::= \lambda x : P.\pi \mid \pi\ \pi' \mid \lambda x : \mathcal{K}.\pi \mid \pi\ d \mid x \mid c_\pi \\
\text{(variable contexts)} & \Phi & ::= \bullet \mid \Phi,\ x : \mathit{Type} \mid \Phi,\ x : \mathcal{K} \mid \Phi,\ x : P \\
\text{(constant signature)} & \Sigma & ::= \bullet \mid \Sigma,\ c_{\mathcal{K}} : \mathit{Type} \mid \Sigma,\ c_d : \mathcal{K} \mid \Sigma,\ c_\pi : P
\end{array}
$$

Figure 3.1: The base syntax of the logic λHOL

discourse, including the *Prop* type for propositions.

- The **sorts**, denoted s, which are the classifiers of kinds.
- The **proof objects**, denoted π, which correspond to a linear representation of valid derivations of propositions.
- The **variables context**, denoted Φ, which is a list of variables along with their type.
- The **signature**, denoted Σ, which is a list of axioms, each one corresponding to a constant and its type.

The syntax of these classes is given in Figure 3.1, while the typing judgements of the logic are given in Figures 3.2 and 3.3. We will now comment on the term formers and their associated typing rules.

The proposition $P_1 \to P_2$ denotes logical implication whereas the kind former $\mathcal{K}_1 \to \mathcal{K}_2$ is inhabited by *total* functions between domain objects. Both use the standard forms of lambda abstraction and function application as their introduction and elimination rules. In the case of the proposition $P_1 \to P_2$, the typing rules for the proof objects $\lambda x : P_1.\pi$ and $\pi_1\ \pi_2$ correspond exactly to the introduction and elimination rules for logical implication as we gave them in Section 2.3; the proof objects are simply a way to record the derivation as a term instead of a tree. The proposition $\forall x : \mathcal{K}.P$ denotes universal quantification. As $\mathcal{K}$ includes a domain former for propositions and functions yielding propositions, we can quantify over propositions and predicates. This is why λHOL is a higher-order logic – for example, it is capable of representing natural number induction as the proposition:

$$\forall P : \mathit{Nat} \to \mathit{Prop}.P\ 0 \to (\forall n : \mathit{Nat}, P\ n \to P\ (\mathit{succ}\ n)) \to \forall n : \mathit{Nat}, P\ n$$

$\vdash \Sigma$ wf	well-formedness of signatures
$\vdash_\Sigma \Phi$ wf	well-formedness of contexts
$\Phi \vdash_\Sigma \mathcal{K} : \mathit{Type}$	well-formedness for kinds
$\Phi \vdash_\Sigma d : \mathcal{K}$	kinding of domain objects
$\Phi \vdash_\Sigma \pi : P$	valid logical derivations of proof objects proving a proposition

$\boxed{\vdash \Sigma \text{ wf}}$

$$\vdash \bullet \text{ wf} \qquad \frac{\vdash \Sigma \text{ wf} \qquad c_\mathcal{K} : \mathit{Type} \notin \Sigma}{\vdash \Sigma,\ c_\mathcal{K} : \mathit{Type} \text{ wf}} \qquad \frac{\vdash \Sigma \text{ wf} \qquad \bullet \vdash_\Sigma \mathcal{K} : \mathit{Type} \qquad c_d : \_ \notin \Sigma}{\vdash \Sigma,\ c_d : \mathcal{K} \text{ wf}}$$

$$\frac{\vdash \Sigma \text{ wf} \qquad \bullet \vdash_\Sigma P : \mathit{Prop} \qquad c_\pi : \_ \notin \Sigma}{\vdash \Sigma,\ c_\pi : P \text{ wf}}$$

$\boxed{\vdash_\Sigma \Phi \text{ wf}}$

$$\frac{}{\vdash \bullet \text{ wf}} \qquad \frac{\vdash \Phi \text{ wf}}{\vdash \Phi,\ x : \mathit{Type} \text{ wf}} \qquad \frac{\vdash \Phi \text{ wf} \qquad \Phi \vdash \mathcal{K} : \mathit{Type}}{\vdash \Phi,\ x : \mathcal{K} \text{ wf}} \qquad \frac{\vdash \Phi \text{ wf} \qquad \Phi \vdash P : \mathit{Prop}}{\vdash \Phi,\ x : P \text{ wf}}$$

$\boxed{\Phi \vdash_\Sigma \mathcal{K} : \mathit{Type}}$

$$\frac{}{\Phi \vdash \mathit{Prop} : \mathit{Type}} \qquad \frac{c_\mathcal{K} : \mathit{Type} \in \Sigma}{\Phi \vdash_\Sigma c_\mathcal{K} : \mathit{Type}} \qquad \frac{\Phi \vdash \mathcal{K}_1 : \mathit{Type} \qquad \Phi \vdash \mathcal{K}_2 : \mathit{Type}}{\Phi \vdash \mathcal{K}_1 \to \mathcal{K}_2 : \mathit{Type}}$$

$\boxed{\Phi \vdash_\Sigma d : \mathcal{K}}$

$$\frac{\Phi \vdash P_1 : \mathit{Prop} \qquad \Phi \vdash P_2 : \mathit{Prop}}{\Phi \vdash P_1 \to P_2 : \mathit{Prop}} \qquad \frac{\Phi \vdash \mathcal{K} : \mathit{Type} \qquad \Phi,\ x : \mathcal{K} \vdash P : \mathit{Prop}}{\Phi \vdash \forall x : \mathcal{K}.P : \mathit{Prop}}$$

$$\frac{\Phi \vdash \mathcal{K} : \mathit{Type} \qquad \Phi,\ x : \mathcal{K} \vdash d : \mathcal{K}'}{\Phi \vdash \lambda x : \mathcal{K}.d : \mathcal{K} \to \mathcal{K}'} \qquad \frac{\Phi \vdash d_1 : \mathcal{K}' \to \mathcal{K} \qquad \Phi \vdash d_2 : \mathcal{K}'}{\Phi \vdash d_1\ d_2 : \mathcal{K}}$$

$$\frac{x : \mathcal{K} \in \Phi}{\Phi \vdash x : \mathcal{K}} \qquad \frac{c_d : \mathcal{K} \in \Sigma}{\Phi \vdash_\Sigma c_d : \mathcal{K}}$$

Figure 3.2: The typing rules of the logic λHOL

$\boxed{\Phi \vdash_\Sigma \pi : P}$

$$\frac{\Phi \vdash P : \mathit{Prop} \qquad \Phi,\ x : P \vdash \pi : P'}{\Phi \vdash \lambda x : P.\pi : P \to P'}\ {\to}\textsc{Intro} \qquad \frac{\Phi \vdash \pi_1 : P' \to P \qquad \Phi \vdash \pi_2 : P'}{\Phi \vdash \pi_1\ \pi_2 : P}\ {\to}\textsc{Elim}$$

$$\frac{\Phi \vdash \mathcal{K} : \mathit{Type} \qquad \Phi,\ x : \mathcal{K} \vdash \pi : P}{\Phi \vdash \lambda x : \mathcal{K}.\pi : \forall x : \mathcal{K}.P}\ \forall\textsc{Intro} \qquad \frac{\Phi \vdash \pi : \forall x : \mathcal{K}.P \qquad \Phi \vdash d : \mathcal{K}}{\Phi \vdash \pi\ d : P[d/x]}\ \forall\textsc{Elim}$$

$$\frac{x : P \in \Phi}{\Phi \vdash x : P}\ \textsc{Var} \qquad \frac{c_\pi : P \in \Sigma}{\Phi \vdash_\Sigma c_\pi : P}\ \textsc{Constant}$$

Figure 3.3: The typing rules of the logic λHOL (continued)

$\boxed{\mathit{fv}(d)}$

$$\begin{array}{lcl}
\mathit{fv}(P_1 \to P_2) & = & \mathit{fv}(P_1) \cup \mathit{fv}(P_2) \\
\mathit{fv}(\forall x : \mathcal{K}.P) & = & \mathit{fv}(P) \setminus \{x\} \\
\mathit{fv}(\lambda x : \mathcal{K}.d) & = & \mathit{fv}(d) \setminus \{x\} \\
\mathit{fv}(d_1\ d_2) & = & \mathit{fv}(d_1) \cup \mathit{fv}(d_2) \\
\mathit{fv}(x) & = & \{x\} \\
\mathit{fv}(c_d) & = & \emptyset
\end{array}$$

$\boxed{d[d'/x]}$

$$\begin{array}{lcl}
(P_1 \to P_2)[d/x] & = & P_1[d/x] \to P_2[d/x] \\
(\forall y : \mathcal{K}.P)[d/x] & = & \forall y : \mathcal{K}.P[d/x] \text{ when } \mathit{fv}(P) \cap \mathit{fv}(d) = \emptyset \text{ and } x \neq y \\
(\lambda y : \mathcal{K}.d')[d/x] & = & \lambda y : \mathcal{K}.d'[d/x] \text{ when } \mathit{fv}(d') \cap \mathit{fv}(d) = \emptyset \text{ and } x \neq y \\
(d_1\ d_2)[d/x] & = & d_1[d/x]\ d_2[d/x] \\
(x)[d/x] & = & d \\
(y)[d/x] & = & y \\
(c_d)[d/x] & = & c_d
\end{array}$$

Figure 3.4: Capture avoiding substitution

When a quantified formula $\forall x.P$ is instantiated with an object d using the rule $\forall$ELIM, we get a proof of $P[d/x]$. The notation $[d/x]$ represents standard capture-avoiding substitution, defined in Figure 3.4.

There are three different types of constants; their types are determined from the signature context. Constants denoted as $c_{\mathcal{K}}$ correspond to the domains of discourse such as natural numbers (written as *Nat*), ordinals (written as *Ord*) and lists (written as *List*). Constant domain objects, denoted as c_d are used for constructors of these domains (for example, $zero : Nat$ and $succ : Nat \to Nat$ for natural numbers); for functions between objects (for example, $plus : Nat \to Nat \to Nat$ for the addition function of natural numbers); for predicates (e.g. $le : Nat \to Nat \to Prop$ for representing when one natural number is less-than-or-equal to another); and for logical connectives, which are understood as functions between propositions (e.g. logical conjunction, represented as $and : Prop \to Prop \to Prop$). Constant proof objects, denoted as c_π, correspond to axioms. The constants above get their expected meaning by adding their corresponding set of axioms to the signature – for example, logical conjunction has the following axioms:

$$andIntro : \forall P_1, P_2 : Prop.P_1 \to P_2 \to and\ P_1\ P_2$$

$$andElim1 : \forall P_1, P_2 : Prop.and\ P_1\ P_2 \to P_1$$

$$andElim2 : \forall P_1, P_2 : Prop.and\ P_1\ P_2 \to P_2$$

Our logic supports inductive definitions of domains and predicates, in a style similar to CIC. We do not represent them explicitly in the logic, opting instead for viewing such definitions as generators of a number of 'safe to add' constants. The logic we have presented – along with the additions made below – is a subset of CIC. It has been shown in Werner [1994] that CIC is a consistent logic; thus it is understood that the logic we present is also consistent, when the signature Σ is generated only through such inductive definitions. We will present more details about inductive definitions after we make some extensions to the core of the logic that we have presented so far.

A simple example of a proof object for commutativity of conjunction follows.

$$\lambda P : Prop.\lambda Q : Prop.\lambda H : \wedge\ P\ Q.$$

$$andIntro\ Q\ P\ (andElim2\ P\ Q\ H)\ (andElim1\ P\ Q\ H)$$

Based on the typing rules, it is simple to show that it proves the proposition:

$$\forall P : Prop.\forall Q : Prop.and\ P\ Q \to and\ Q\ P$$

3.2 Adding equality

The logic we have presented so far is rather weak when it comes to reasoning about the total functions inhabiting $\mathcal{K}_1 \to \mathcal{K}_2$. For example, we cannot prove that the standard β-reduction rule:

$$(\lambda x : \mathcal{K}.d)\ d' = d[d'/x]$$

Moreover, consider the case where we have a proof of a proposition such as $P(1+1)$. We need to be able to construct a proof of $P(2)$ out of this, yet the logic we have presented so far does not give us this ability.

There are multiple approaches for adding rules to logics such as λHOL in order to make equality behave in the expected way. In order to understand the differences between these approaches, it is important to distinguish two separate notions of equality in a logic: definitional equality and propositional equality. Definitional equality relates terms that are indistinguishable from each other in the logic. Proving that two definitionally equal terms are equal is trivial through reflexivity, since they are viewed as exactly the same terms. We will use $d_1 \equiv d_2$ to represent definitional logic. Propositional equality is an actual predicate of the logic. Terms are proved to be propositionally equal using the standard axioms and rules of the logic. In λHOL we would represent propositional equality through propositions of the form $d_1 = d_2$. With this distinction in place, we will give a brief summary of the different approaches below.

Explicit equality. In the HOL family of logics, as used in systems like HOL4 [Slind and Norrish, 2008] and Isabelle/HOL [Nipkow et al., 2002], definitional equality is just syntactic equality. Any equality further than that is propositional equality and needs to be proved using the logic. The logic includes a rule where propositions that can be proved equal can replace each other as the type of proof objects. Any such rewriting needs to be witnessed in the resulting proof objects. A sketch of the typing rule for this axiom is the following:

$$\frac{\Phi \vdash \pi : P \qquad \Phi \vdash \pi' : P = P'}{\Phi \vdash \mathit{conv}\ \pi\ \pi' : P'}$$

The advantage of this approach is that a type checker for it only has to check proofs that contain all the required details about why terms are equal. It is therefore extremely simple, keeping the trusted base of our logic as simple as possible. Still, it has a big disadvantage: the proof objects are very large, as even trivial equalities need to be painstakingly proved. It is possible that this is one of the reasons why systems based on this logic do not generate proof objects by default, even though they all support them.

Intensional type theories. Systems based on Martin-Löf intensional type theory, such as Coq [Barras et al., 2012] and Agda [Norell, 2007], have an extended notion of definitional equality. In these systems, definitional equality includes terms that are identical up to evaluation of the total functions used inside the logical terms. In order to make sure that the soundness of the logic is not compromised, this notion of evaluation needs to be decidable. As we have said, terms that are definitionally equal are indistinguishable, so their equality does not need to be witnessed inside proof objects, resulting in a big reduction in their size.

The way that this is supported in such type theories is to define an equality relation R based on some fixed, confluent rewriting system. In Coq and Agda this includes β-conversion and ι-conversion (evaluation of total recursive functions), pattern matching and unfolding of definitions. This equality relation is then understood as the definitional equality, by adding the following *conversion rule* to the logic:

$$\frac{\Phi \vdash \pi : P_1 \qquad P_1 \equiv_R P_2}{\Phi \vdash \pi : P_2}$$

The benefit of this approach is that trivial arguments based on computation, which would normally not be "proved" in standard mathematical discourse, actually do not need to be proved and witnessed in the proof objects. Therefore the logic follows what is called the Poincaré principle (e.g. in Barendregt and Geuvers [1999]). One disadvantage of this approach is that our trusted base needs to be bigger, since a type-checker for such a logic needs to include a decision procedure for the relation R.

Extensional type theories. In extensional type theories like NuPRL [Constable et al., 1986], the notion of definitional equality and propositional equality are identified. That means that anything that can be proved equal in the logic is seen implicitly as equal. A typing rule that sketches this idea would be the following:

$$\frac{\Phi \vdash \pi : P_1 \qquad \Phi \vdash \pi' : P_1 = P_2}{\Phi \vdash \pi : P_2}$$

The advantage of this approach is that the size of proof objects is further reduced, bringing them even closer to normal mathematical practice. Still, theories with this approach have a big disadvantage: type-checking proof objects becomes undecidable. It is easy to see why this is so from the above typing rule, as the π' proof object needs to be reconstructed from scratch during type checking time – that is, type checking depends on being able to decide whether two arbitrary terms are provably equal or not, an undecidable problem in any meaningful logic. Implementations of such theories deal with this by only implementing a set of decision procedures that can decide on such equalities, but all of these procedures become part of the trusted base of the system.

We should note here that type theories such as CIC and NuPRL support a much richer notion of domain objects and kinds, as well as a countable hierarchy of universes above the *Type* sort that we support. For example, kinds themselves can be computed through total functions. In these cases, the chosen approach with respect to equality also affects which terms are typable. Our logic is much simpler and thus we will only examine equality at the level of domain objects. Still, we believe that the techniques we develop are applicable to these richer logics and to cases where equality at higher universes such as the level of kinds $\mathcal{K}$ is also available.

We want λHOL to have the simplest possible type checker, so we choose the explicit equality version. We will see how to recover and extend the benefits of the conversion rule through VeriML in Section 7.1. We therefore extend our logic with a built-in propositional equality predicate, as well as a set of equality axioms. The extensions required to the logic

Syntax extensions

$$\begin{array}{rrll} \text{(dom.obj. and prop.)} & d, P & ::= & \cdots \mid d_1 = d_2 \\ \text{(proof objects)} & \pi & ::= & \mathit{refl}\ d \mid \mathit{conv}\ \pi\ \pi' \mid \mathit{subst}\ (x : \mathcal{K}.P)\ \pi \\ & & & \mid \mathit{congLam}\ (x : \mathcal{K}.\pi) \mid \mathit{congForall}\ (x : \mathcal{K}.\pi) \\ & & & \mid \mathit{beta}\ (x : \mathcal{K}.d)\ d' \end{array}$$

$\boxed{\Phi \vdash_\Sigma d : \mathcal{K}}$

$$\frac{\Phi \vdash d_1 : \mathcal{K} \qquad \Phi \vdash d_2 : \mathcal{K}}{\Phi \vdash d_1 = d_2 : \mathit{Prop}}$$

$\boxed{\Phi \vdash_\Sigma \pi : P}$

$$\frac{\Phi \vdash d : \mathcal{K}}{\Phi \vdash \mathit{refl}\ d : d = d} \qquad \frac{\Phi \vdash \pi : P \qquad \Phi \vdash P : \mathit{Prop} \qquad \Phi \vdash \pi' : P = P'}{\Phi \vdash \mathit{conv}\ \pi\ \pi' : P'}$$

$$\frac{\Phi,\ x : \mathcal{K} \vdash P : \mathit{Prop} \qquad \Phi \vdash \pi : d_1 = d_2 \qquad \Phi \vdash d_1 : \mathcal{K}}{\Phi \vdash \mathit{subst}\ (x : \mathcal{K}.P)\ \pi : P[d_1/x] = P[d_2/x]}$$

$$\frac{\Phi,\ x : \mathcal{K} \vdash \pi : d_1 = d_2}{\Phi \vdash \mathit{congLam}\ (x : \mathcal{K}.\pi) : (\lambda x : \mathcal{K}.d_1) = (\lambda x : \mathcal{K}.d_2)}$$

$$\frac{\Phi,\ x : \mathcal{K} \vdash \pi : P_1 = P_2 \qquad \Phi,\ x : \mathcal{K} \vdash P_1 : \mathit{Prop}}{\Phi \vdash \mathit{congForall}\ (x : \mathcal{K}.\pi) : (\forall x : \mathcal{K}.P_1) = (\forall x : \mathcal{K}.P_2)}$$

$$\frac{\Phi \vdash (\lambda x : \mathcal{K}.d) : \mathcal{K} \to \mathcal{K}' \qquad \Phi \vdash d' : \mathcal{K}}{\Phi \vdash \mathit{beta}\ (x : \mathcal{K}.d)\ d' : ((\lambda x : \mathcal{K}.d)\ d') = (d[d'/x])}$$

Figure 3.5: The logic $\lambda\mathrm{HOL}_E$: syntax and typing extensions for equality

$$
\begin{array}{rl}
\text{(sorts)} & s ::= \mathit{Prop} \mid \mathit{Type} \mid \mathit{Type}' \\
\text{(logical terms)} & t ::= s \mid x \mid c \mid \forall x : t_1.t_2 \mid \lambda x : t_1.t_2 \mid t_1\ t_2 \mid t_1 = t_2 \mid \mathit{refl}\ t \\
& \quad \mid \mathit{conv}\ t_1\ t_2 \mid \mathit{subst}\ (x : t_k.t_P)\ t \mid \mathit{congLam}\ (x : t_k.t) \\
& \quad \mid \mathit{congForall}\ (x : t_k.t) \mid \mathit{beta}\ (x : t_k.t_1)\ t_2 \\
\text{(variable contexts)} & \Phi ::= \bullet \mid \Phi,\ x : t \\
\text{(constant signature)} & \Sigma ::= \bullet \mid \Sigma,\ c : t
\end{array}
$$

Figure 3.6: The syntax of the logic λHOL given as a PTS

presented so far are given in Figure 3.5. We sometimes refer to λHOL together with these extensions as λHOL_E in order to differentiate with a version that we will present later which includes the conversion rule; that version is called λHOL_C.

Most of the typing rules of the new axioms are standard. We have an explicit axiom for β-reduction through the proof object constructor *beta*. α-equivalence is handled informally at this level by implicitly identifying terms that are equal up to renaming of bound variables; this will be changed in the next section. Note that some rules seem to omit certain premises, such as the fact that P_2 is a proposition in the case of *congForall*. The reason is that these premises are always true based on the existing ones; in the example at hand, the fact that $P_1 = P_2$ implies that their type is identical, thus P_2 is a proposition just like P_1 is.

The given set of axioms is enough to prove other expected properties of equality, such as symmetricity:

$$
\begin{array}{lcl}
\mathit{symm}_{\mathcal{K}} & : & \forall a : \mathcal{K}.\forall b : \mathcal{K}.a = b \to b = a \\
\mathit{symm}_{\mathcal{K}} & = & \lambda a : \mathcal{K}.\lambda b : \mathcal{K}.\lambda H : a = b.\quad \mathit{conv}(\mathit{refl}\ a)\ (\mathit{subst}\ (x : \mathcal{K}.x = a)\ H)
\end{array}
$$

3.3 λHOL as a Pure Type System

Our presentation of λHOL above separates the various different types of logical terms into distinct classes. This separation results in duplicating various typing rules – for example, quantification, logical implication and function kinds share essentially the same type and term formers. This needlessly complicates the rest of our development. We will now present λHOL as a Pure Type System [Barendregt, 1992] which is the standard way to describe typed lambda calculi. Essentially the syntactic classes of all logical terms given above are merged into one single class, denoted as t. We give the new syntax in Figure 3.6 and the

$\vdash \Sigma$ wf	well-formedness of signatures
$\vdash_\Sigma \Phi$ wf	well-formedness of contexts
$\Phi \vdash_\Sigma t : t'$	typing of logical terms

$\boxed{\vdash \Sigma \text{ wf}}$

$$\vdash \bullet \text{ wf} \qquad \frac{\vdash \Sigma \text{ wf} \qquad \bullet \vdash_\Sigma t : s}{\vdash \Sigma,\ x : t \text{ wf}}$$

$\boxed{\vdash_\Sigma \Phi \text{ wf}}$

$$\vdash \bullet \text{ wf} \qquad \frac{\vdash \Phi \text{ wf} \qquad \Phi \vdash t : s}{\vdash \Phi,\ x : t \text{ wf}}$$

$\boxed{\Phi \vdash_\Sigma t : t'}$

$$\frac{(s, s') \in \mathcal{A}}{\Phi \vdash s : s'} \qquad \frac{\Phi \vdash t_1 : s \qquad \Phi,\ x : t_1 \vdash t_2 : s' \qquad (s, s', s'') \in \mathcal{R}}{\Phi \vdash \forall x : t_1.t_2 : s''}$$

$$\frac{\Phi \vdash t_1 : \forall x : t.t' \qquad \Phi \vdash t_2 : t}{\Phi \vdash t_1\ t_2 : t'[t_2/x]} \qquad \frac{\Phi,\ x : t_1 \vdash t_2 : t' \qquad \Phi \vdash \forall x : t_1.t' : s}{\Phi \vdash \lambda x : t_1.t_2 : \forall x : t_1.t'} \qquad \frac{x : t \in \Phi}{\Phi \vdash x : t}$$

$$\frac{c : t \in \Sigma}{\Phi \vdash_\Sigma c : t} \qquad \frac{\Phi \vdash t_1 : t \qquad \Phi \vdash t_2 : t \qquad \Phi \vdash t : \mathit{Type}}{\Phi \vdash t_1 = t_2 : \mathit{Prop}} \qquad \frac{\Phi \vdash t : t' \qquad \Phi \vdash t' : \mathit{Type}}{\Phi \vdash \mathit{refl}\ t : t = t}$$

$$\frac{\Phi \vdash t_1 : t \qquad \Phi \vdash t : \mathit{Prop} \qquad \Phi \vdash t_2 : t = t'}{\Phi \vdash \mathit{conv}\ t_1\ t_2 : t'}$$

$$\frac{\Phi \vdash t_k : \mathit{Type} \qquad \Phi,\ x : t_k \vdash t_P : \mathit{Prop} \qquad \Phi \vdash t : t_a = t_b \qquad \Phi \vdash t_a : t_k}{\Phi \vdash \mathit{subst}\ (x : t_k.t_P)\ t : t_P[t_a/x] = t_P[t_b/x]}$$

$$\frac{\Phi \vdash t_k : \mathit{Type} \qquad \Phi,\ x : t_k \vdash t : t_1 = t_2}{\Phi \vdash \mathit{congLam}\ (x : t_k.t) : (\lambda x : t_k.t_1) = (\lambda x : t_k.t_2)}$$

$$\frac{\Phi \vdash t_k : \mathit{Type} \qquad \Phi,\ x : t_k \vdash t : t_1 = t_2 \qquad \Phi,\ x : t_k \vdash t_1 : \mathit{Prop}}{\Phi \vdash \mathit{congForall}\ (x : t_k.t) : (\forall x : t_k.t_1) = (\forall x : t_k.t_2)}$$

$$\frac{\Phi \vdash (\lambda x : t_a.t_1) : t_a \to t_b \qquad \Phi \vdash t_a \to t_b : \mathit{Type} \qquad \Phi \vdash t_2 : t_a}{\Phi \vdash \mathit{beta}\ (x : t_a.t_1)\ t_2 : ((\lambda x : t_a.t_1)\ t_2) = (t_1[t_2/x])}$$

Figure 3.7: The typing rules of the logic λHOL in PTS style

new typing rules in Figure 3.7. Note that we use $t_1 \to t_2$ as syntactic sugar for $\forall x : t_1.t_2$ when x does not appear free in t_2.

A PTS is characterized by three parameters: the set of *sorts*, the set of *axioms* and the set of *rules*. These parameters are instantiated as follows for λHOL:

$$\begin{aligned}
\mathcal{S} &= \{\mathit{Prop},\ \mathit{Type},\ \mathit{Type}'\} \\
\mathcal{A} &= \{(\mathit{Prop}, \mathit{Type}),\ (\mathit{Type}, \mathit{Type}')\} \\
\mathcal{R} &= \{(\mathit{Prop}, \mathit{Prop}, \mathit{Prop}),\ (\mathit{Type}, \mathit{Type}, \mathit{Type}),\ (\mathit{Type}, \mathit{Prop}, \mathit{Prop})\}
\end{aligned}$$

Terms sorted[1] under *Prop* represent proofs, under *Type* represent objects (including propositions) and under *Type′* represent the domains of discourse (e.g. natural numbers). The axiom (*Prop*, *Type*) corresponds to the fact that propositions are objects of the domain of discourse, while the axiom (*Type*, *Type′*) is essentially what allows us to introduce constants and variables for domains. The rules are understood as logical implication, function formation between objects and logical quantification, respectively. Based on these parameters, it is evident that λHOL is similar to System Fω, which is usually formulated as $\mathcal{S} = \{\star, \Box\}$, $\mathcal{A} = \{(\star, \Box)\}$, $\mathcal{R} = \{(\star, \star, \star),\ (\Box, \Box, \Box),\ (\Box, \star, \star)\}$. λHOL is a straightforward extension with an extra sort above $\Box$ – that is, *Type′* above *Type* – allowing us to have domains other than the universe of propositions. Furthermore, λHOL extends the basic PTS system with the built-in explicit equality as presented above, as well as with the signature of constants Σ.

The substitution lemma

The computational language of VeriML assumes a simple set of properties that need to hold for the type system of the logic language. The most important of these is the substitution lemma, stating that substituting terms for variables of the same type yields well-typed terms. We will prove a number of such substitution lemmas for different notions of 'term' and 'variable'. As an example, let us state the substitution lemma for logical terms of λHOL in the PTS version of the logic.

Lemma 3.3.1 (Substitution) *If* $\Phi,\ x : t'_T,\ \Phi' \vdash t : t_T$ *and* $\Phi \vdash t' : t'_T$ *then* $\Phi,\ \Phi'[t'/x] \vdash$

1. That is, terms typed under a term whose sort is the one we specify.

$t[t'/x] : t_T[t'/x]$.

The proof of the lemma is a straightforward induction on the typing derivation of the term t. The most important auxiliary lemma it depends on is that typing obeys the weakening structural rule, that is:

Lemma 3.3.2 (Weakening) *If* $\Phi \vdash t : t_T$ *then* $\Phi,\ x : t'_T \vdash t : t_T$.

The substitution lemma can alternatively be stated for a list of substitutions of variables to terms, covering the whole context. This alternative statement is technically advantageous in terms of how the proof is carried through and also how the lemma can be used. The lemma is stated as follows:

Lemma 3.3.3 (Simultaneous substitution) *If* $x_1 : t_1,\ x_2, t_2,\ \cdots,\ x_n : t_n \vdash t : t_T$ *and for all* i *with* $1 \leq i \leq n$, $\Phi' \vdash x_i : t_i$, *then* $\Phi' \vdash t[t_1, t_2, \cdots, t_n/x_1, x_2, \cdots, x_n]$.

We can view the list of (x_i, t_i) variable-term pairs as a well-typed substitution covering the context Φ. We will give a more precise definition of this notion in the following section.

For the rest of our development, we will switch to a different variable representation, which will allow us to state these lemmas more precisely, by removing the dependence on variable names and the requirement for implicit α-renaming in the definition of substitution. Thus we will not go into more details of these proofs as stated here.

3.4 λHOL using hybrid deBruijn variables

So far, we have used names to represent variables in λHOL. Terms that are equivalent up to α-renaming are deemed to be exactly the same and are used interchangeably. An example of this shows up in the definition of capture-avoiding substitution in Figure 3.4: constructs that include binders, such as $\lambda y : \mathcal{K}.d$, are implicitly α-renamed so that no clashes with the variables of the subsitutend are introduced. We are essentially relying on viewing terms as representatives of their α-equivalence classes; our theorems would also reason up to this equivalence. It is a well-known fact [Aydemir et al., 2008] that when formally proving such theorems about languages with binding in proof assistants, identifying terms up to

α-equivalence poses a large number of challenges. This is usually addressed by choosing a *concrete variable representation* technique where the non-determinism of choosing variable names is removed and thus all α-equivalent terms are represented in the same way.

We have found that using a similar concrete variable representation technique is advantageous for our development, even though our proofs are done on paper. The main reason is that the addition of contextual terms, contextual variables and polymorphic contexts introduces a number of operations which are hard to define in the presence of named variables, as they trigger complex sets of α-renamings. Using a concrete representation, these operations and the renamings that are happening are made explicit. Furthermore, we use the same concrete representation in our implementation of VeriML itself, so the gap between the formal development and the actually implemented code is smaller, increasing our confidence in the overall soundness of the system.

The concrete representation technique that we use is a novel, to the best of our knowledge, combination of the techniques of de Bruijn indices, de Bruijn levels [De Bruijn, 1972] and the locally nameless approach [McKinna and Pollack, 1993]. The former two techniques replace all variables by numbers, whereas the locally nameless approach introduces a distinction between bound and free variables and handles them differently. Let us briefly summarize these approaches before presenting our hybrid representation. A table comparison is given in Table 3.1. DeBruijn indices correspond to the number of binders above the current variable reference where the variable was bound. DeBruijn levels are the "inverse" number, corresponding to how many binders we have to cross until we reach the one binding the variable we are referring to, if we start at the top of the term. Both treat free variables by considering the context as a list of variable binders. In both cases, when performing the substitution $(\lambda y.t)[t'/x]$, we need to *shift* the variable numbers in t' in order for the resulting term to be well-formed – the free variables in the indices case or the bound variables in the levels case. In the indices case, the introduction of the new binder alters the way to refer to the free variables (the free variable represented by index n in t' will be represented by $n + 1$ under the λ-abstraction), as the identities of free variables directly depend on the number of bound variables at a point. In the levels case, what is altered is the way we refer to bound variables within t' – the 'allocation policy' for bound variables in

Example of performing the substitution: $(\lambda y : x.f\ y\ a)[(\lambda z : x.z)\ y/a]$	Free variables context: $x : \mathit{Type},\ g : x \to x \to x,\ y : x,\ a : x$	
Named representation	$(\lambda y : x.g\ y\ a)[(\lambda z : x.z)\ y/a]$	
	$(\lambda w : x.g\ w\ a)[(\lambda z : x.z)\ y/a]$	$=_\alpha$
	$\lambda w : x.g\ w\ ((\lambda z : x.z)\ y)$	$\Rightarrow$
deBruijn indices	$(\lambda(3).3\ 0\ 1)[(\lambda(3).0)\ 1/0]$	
	$\lambda(3).3\ 0\ (\mathit{shiftFree}\ ((\lambda(3).0)\ 1))$	$\Rightarrow$
	$\lambda(3).3\ 0\ ((\lambda(4).0)\ 2)$	$\Rightarrow$
deBruijn levels	$(\lambda(0).1\ 4\ 3)[(\lambda(0).4)\ 2/3]$	
	$\lambda(0).1\ 4\ (\mathit{shiftBound}\ ((\lambda(0).4)\ 2))$	$\Rightarrow$
	$\lambda(0).1\ 4\ ((\lambda(0).5)\ 2)$	$\Rightarrow$
Locally nameless	$(\lambda(x).g\ 0\ a)[(\lambda(x).0)\ y/a]$	
	$\lambda(x).g\ 0\ ((\lambda(x).0)\ y)$	$\Rightarrow$
Hybrid deBruijn	$(\lambda(v_0).v_1\ b_0\ v_3)[(\lambda(v_0).b_0)\ v_2/v_3]$	
	$\lambda(v_0).v_1\ b_0\ ((\lambda(v_0).b_0)\ v_2)$	$\Rightarrow$

Table 3.1: Comparison of variable representation techniques

a way. For example the lambda term $t' = \lambda x.x$ that was represented as $\lambda.n$ outside the new binder is now represented as $\lambda.n + 1$. This is problematic: substitution is not structurally recursive and needs to be reasoned about together with shifting. This complicates both the statements and proofs of related lemmas.

The locally nameless approach addresses this problem by explicitly separating free from bound variables. Free variables are represented normally using names whereas deBruijn indices are reserved for representing bound variables. In this way the identities of free variables are preserved when a new binder is introduced and no shifting needs to happen. We still need two auxiliary operations: freshening, in order to open up a term under a binder into a term that has an extra free variable by replacing the dangling bound variable, and binding, which is the inverse operation, turning the last-introduced free variable into a bound variable, so as to close a term under a binder. Under this approach, not all α-equivalent terms are identified, as using different names for free variables in the context still yields different terms.

We merge these approaches in a new technique that we call *hybrid deBruijn variables*: following the locally nameless approach, we separate free variables from bound variables

$$
\begin{array}{rl}
s ::= & \mathit{Prop} \mid \mathit{Type} \mid \mathit{Type}' \\
t ::= & s \mid c \mid v_i \mid b_i \mid \lambda(t_1).t_2 \mid t_1\ t_2 \mid \Pi(t_1).t_2 \mid t_1 = t_2 \\
& \mid \mathit{conv}\ t_1\ t_2 \mid \mathit{subst}\ ((t_k).t_P)\ t \mid \mathit{refl}\ t \\
& \mid \mathit{congLam}\ ((t_k).t) \mid \mathit{congForall}\ ((t_k).t) \\
& \mid \mathit{beta}\ ((t_k).t_1)\ t_2 \\
\Sigma ::= & \bullet \mid \Sigma,\ c : t \\
\Phi ::= & \bullet \mid \Phi,\ t \\
\sigma ::= & \bullet \mid \sigma,\ t
\end{array}
$$

Figure 3.8: The logic λHOL with hybrid deBruijn variable representation: Syntax

and use deBruijn indices for bound variables (denoted as b_i in Table 3.1); but also, we use deBruijn levels for free variables instead of names (denoted as v_i). In this way, all α-equivalent terms are identified. Furthermore, terms preserve their identity under additions to the end of the free variables context. Such additions are a frequent operation in VeriML, so this representation is advantageous from an implementation point of view. A similar representation technique where terms up to any weakening of the context are identified (not just additions to the end of the context) by introducing a distinction between used and unused variable names has been discovered independently by Pollack et al. [2011] and Pouillard [2011].

Let us now proceed to the presentation of λHOL using our hybrid representation technique. We give the new syntax of the language in Figure 3.8, the new typing rules in Figures 3.9 and 3.10 and the definition of syntactic operations in Figures 3.11, 3.12 and 3.13.

As expected, the main change to the syntax of terms and of the context Φ is that binding structures do not carry variable names any longer. For example, instead of writing $\lambda x : t_1.t_2$ we now write $\lambda(t_1).t_2$. In the typing rules, the main change is that extra care needs to be taken when going into a binder or when enclosing a term with a binder. This is evident for example in the rules ΠTYPE and ΠINTRO. When we cross a binder, the just-bound variable b_0 needs to be converted into a fresh free variable. This is what the freshening operation $\lceil t \rceil_m^n$ does, defined in Figure 3.11. The two arguments are as follows: n represents which bound variable to replace and is initially 0 when we leave it unspecified; m is the current number of free variables (the length of the current context), so that the next fresh free variable v_m is used as the replacement of the bound variable. Binding is the inverse operation and is

$\boxed{\vdash \Sigma \text{ wf}}$

$$\frac{}{\vdash \bullet \text{ wf}}\ \textsc{SigEmpty} \qquad \frac{\vdash \Sigma \text{ wf} \qquad \bullet \vdash_\Sigma t : s \qquad (c : \_) \notin \Sigma}{\vdash \Sigma,\ c : t \text{ wf}}\ \textsc{SigConst}$$

$\boxed{\vdash_\Sigma \Phi \text{ wf}}$

$$\frac{}{\vdash \bullet \text{ wf}}\ \textsc{CtxEmpty} \qquad \frac{\vdash \Phi \text{ wf} \qquad \Phi \vdash t : s}{\vdash \Phi,\ t \text{ wf}}\ \textsc{CtxVar}$$

$\boxed{\Phi \vdash \sigma : \Phi'}$

$$\frac{\vdash \Phi \text{ wf}}{\Phi \vdash \bullet : \bullet}\ \textsc{SubstEmpty} \qquad \frac{\Phi \vdash \sigma : \Phi' \qquad \Phi \vdash t : t' \cdot \sigma}{\Phi \vdash \sigma,\ t : (\Phi',\ t')}\ \textsc{SubstVar}$$

$\boxed{\Phi \vdash_\Sigma t : t'}$

$$\frac{c : t \in \Sigma}{\Phi \vdash_\Sigma c : t}\ \textsc{Constant} \qquad \frac{\Phi.i = t}{\Phi \vdash v_i : t}\ \textsc{Var} \qquad \frac{(s, s') \in \mathcal{A}}{\Phi \vdash s : s'}\ \textsc{Sort}$$

$$\frac{\Phi \vdash t_1 : s \qquad \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Phi \vdash \Pi(t_1).t_2 : s''}\ \Pi\textsc{Type}$$

$$\frac{\Phi \vdash t_1 : s \qquad \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Phi \vdash \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1} : s'}{\Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}}\ \Pi\textsc{Intro}$$

$$\frac{\Phi \vdash t_1 : \Pi(t).t' \qquad \Phi \vdash t_2 : t}{\Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (id_\Phi, t_2)}\ \Pi\textsc{Elim} \qquad \frac{\Phi \vdash t_1 : t \qquad \Phi \vdash t_2 : t \qquad \Phi \vdash t : \mathit{Type}}{\Phi \vdash t_1 = t_2 : \mathit{Prop}}\ \textsc{EqType}$$

Figure 3.9: The logic λHOL with hybrid deBruijn variable representation: Typing Judgements

$$\frac{\Phi \vdash t_1 : t \qquad \Phi \vdash t_1 = t_1 : \mathit{Prop}}{\Phi \vdash \mathit{refl}\ t_1 : t_1 = t_1}\ \textsc{EqRefl}$$

$$\frac{\Phi \vdash t : t_1 \qquad \Phi \vdash t_1 : \mathit{Prop} \qquad \Phi \vdash t' : t_1 = t_2}{\Phi \vdash \mathit{conv}\ t\ t' : t_2}\ \textsc{EqElim}$$

$$\frac{\Phi \vdash t_k : \mathit{Type} \qquad \Phi,\ t_k \vdash \lceil t_P \rceil_{|\Phi|} : \mathit{Prop} \qquad \Phi \vdash t : t_a = t_b \qquad \Phi \vdash t_a : t_k}{\Phi \vdash \mathit{subst}\ ((t_k).t_P)\ t : t_P \cdot (id_\Phi,\ t_a) = t_P \cdot (id_\Phi,\ t_b)}\ \textsc{EqSubst}$$

$$\frac{\Phi \vdash t_k : \mathit{Type} \qquad \Phi,\ t_k \vdash \lceil t \rceil_{|\Phi|} : t_1 = t_2}{\Phi \vdash \mathit{congLam}\ ((t_k).t) : (\lambda(t_k).\ \lfloor t_1 \rfloor_{|\Phi|+1}) = (\lambda(t_k).\ \lfloor t_2 \rfloor_{|\Phi|+1})}\ \textsc{EqLam}$$

$$\frac{\Phi \vdash t_k : \mathit{Type} \qquad \Phi,\ t_k \vdash \lceil t \rceil_{|\Phi|} : t_1 = t_2 \qquad \Phi,\ t_k \vdash t_1 : \mathit{Prop}}{\Phi \vdash \mathit{congForall}\ ((t_k).t) : (\forall(t_k).\ \lfloor t_1 \rfloor_{|\Phi|+1}) = (\forall(t_k).\ \lfloor t_2 \rfloor_{|\Phi|+1})}\ \textsc{EqForall}$$

$$\frac{\Phi \vdash (\lambda(t_a).t_1) : t_a \to t_b \qquad \Phi \vdash t_a \to t_b : \mathit{Type} \qquad \Phi \vdash t_2 : t_a}{\Phi \vdash \mathit{beta}\ ((t_a).t_1)\ t_2 : ((\lambda(t_a).t_1)\ t_2) = \lceil t_1 \rceil_{|\Phi|} \cdot (id_\Phi,\ t_2)}\ \textsc{EqBeta}$$

Figure 3.10: The logic λHOL with hybrid deBruijn variable representation: Typing Judgements (continued)

used when we are re-introducing a binder, capturing the last free variable and making it a bound variable. The arguments are similar: n represents which bound variable to replace with and m represents the level of the last free variable.

As is evident in the above discussion, the operations of freshening and binding have particular assumptions about the terms they are applied to. For example, freshening needs to be applied to a term with one "dangling" bound variable, whereas binding needs to be applied to a closed term. In order to state such assumptions precisely, we define the notion of free and bound variable limits in Figure 3.12. We say that a term t obeys the limit of n free variables, and write $t <^f n$ when t does not contain any free variable higher than v_{n-1}. Bound variable limits are similar.

We have also changed the way we denote and perform substitutions. We have added simultaneous substitutions as a new syntactic class. We denote them as σ and refer to them simply as substitutions from now on. They represent lists of terms to substitute for the variables in a Φ context. Notice that the lack of variable names requires that both Φ and σ

Freshening: $\lceil t \rceil_m^n$

$$
\begin{array}{lll}
 & \lceil s \rceil_m^n & = s \\
 & \lceil c \rceil_m^n & = c \\
* & \lceil v_i \rceil_m^n & = v_i \\
* & \lceil b_n \rceil_m^n & = v_m \\
 & \lceil b_i \rceil_m^n & = b_i \\
 & \lceil (\lambda(t_1).t_2) \rceil_m^n & = \lambda(\lceil t_1 \rceil_m^n).\, \lceil t_2 \rceil_m^{n+1} \\
 & \lceil t_1\ t_2 \rceil_m^n & = \lceil t_1 \rceil_m^n\ \lceil t_2 \rceil_m^n \\
 & \lceil \Pi(t_1).t_2) \rceil_m^n & = \Pi(\lceil t_1 \rceil_m^n).(\lceil t_2 \rceil_m^{n+1}) \\
 & \lceil t_1 = t_2 \rceil_m^n & = \lceil t_1 \rceil_m^n = \lceil t_2 \rceil_m^n \\
 & \lceil conv\ t_1\ t_2 \rceil_m^n & = conv\ \lceil t_1 \rceil_m^n\ \lceil t_2 \rceil_m^n \\
 & \lceil refl\ t \rceil_m^n & = refl\ \lceil t \rceil \\
 & \lceil subst\ (t_k.t_P)\ t \rceil_m^n & = subst\ (\lceil t_k \rceil_m^n . \lceil t_P \rceil_m^{n+1})\ \lceil t \rceil_m^n \\
 & \lceil congLam\ (t_k.t) \rceil_m^n & = congLam\ (\lceil t_k \rceil_m^n . \lceil t \rceil_m^{n+1}) \\
 & \lceil congForall\ (t_k.t) \rceil_m^n & = congForall\ (\lceil t_k \rceil_m^n . \lceil t \rceil_m^{n+1}) \\
 & \lceil beta\ (t_k.t_1)\ t_2 \rceil_m^n & = beta\ (\lceil t_k \rceil_m^n . \lceil t_1 \rceil_m^{n+1})\ \lceil t_2 \rceil_m^n
\end{array}
$$

Binding: $\lfloor t \rfloor_m^n$

$$
\begin{array}{lll}
 & \lfloor s \rfloor_m^n & = s \\
 & \lfloor c \rfloor_m^n & = c \\
* & \lfloor v_{m-1} \rfloor_m^n & = b_n \\
* & \lfloor v_i \rfloor_m^n & = v_i \\
 & \lfloor b_i \rfloor_m^n & = b_i \\
 & \lfloor (\lambda(t_1).t_2) \rfloor_m^n & = \lambda(\lfloor t_1 \rfloor_m^n).\, \lfloor t_2 \rfloor_m^{n+1} \\
 & \lfloor t_1\ t_2 \rfloor_m^n & = \lfloor t_1 \rfloor_m^n\ \lfloor t_2 \rfloor_m^n \\
 & \lfloor \Pi(t_1).t_2) \rfloor_m^n & = \Pi(\lfloor t_1 \rfloor_m^n).\, \lfloor t_2 \rfloor_m^{n+1} \\
 & \lfloor t_1 = t_2 \rfloor_m^n & = \lfloor t_1 \rfloor_m^n = \lfloor t_2 \rfloor_m^n \\
 & \lfloor conv\ t_1\ t_2 \rfloor_m^n & = conv\ \lfloor t_1 \rfloor_m^n\ \lfloor t_2 \rfloor_m^n \\
 & \lfloor refl\ t \rfloor_m^n & = refl\ \lfloor t \rfloor_m^n \\
 & \lfloor subst\ (t_k.t_P)\ t \rfloor_m^n & = subst\ (\lfloor t_k \rfloor_m^n . \lfloor t_P \rfloor_m^{n+1})\ \lfloor t \rfloor_m^n \\
 & \lfloor congLam\ (t_k.t) \rfloor_m^n & = congLam\ (\lfloor t_k \rfloor_m^n . \lfloor t \rfloor_m^{n+1}) \\
 & \lfloor congForall\ (t_k.t) \rfloor_m^n & = congForall\ (\lfloor t_k \rfloor_m^n . \lfloor t \rfloor_m^{n+1}) \\
 & \lfloor beta\ (t_k.t_1)\ t_2 \rfloor_m^n & = beta\ (\lfloor t_k \rfloor_m^n . \lfloor t_1 \rfloor_m^{n+1})\ \lfloor t_2 \rfloor_m^n
\end{array}
$$

Figure 3.11: The logic λHOL with hybrid deBruijn variable representation: Freshening and Binding

Free variable limits for terms:

$$t <^f n$$

$$\begin{array}{lcl} s <^f n & & \\ c <^f n & & \\ v_i <^f n & \Leftarrow & n > i \\ b_i <^f n & & \\ (\lambda(t_1).t_2) <^f n & \Leftarrow & t_1 <^f n \wedge t_2 <^f n \\ t_1\ t_2 <^f n & \Leftarrow & t_1 <^f n \wedge t_2 <^f n \\ & \cdots & \end{array}$$

Free var. limits for substitutions:

$$\sigma <^f n$$

$$\begin{array}{lcl} \bullet <^f n & & \\ (\sigma, t) <^f n & \Leftarrow & \sigma <^f n \wedge t <^f n \end{array}$$

Bound var. limits for terms:

$$t <^b n$$

$$\begin{array}{lcl} s <^b n & & \\ c <^b n & & \\ v_i <^b n & & \\ b_i <^b n & \Leftarrow & n > i \\ (\lambda(t_1).t_2) <^b n & \Leftarrow & t_1 <^b n \wedge t_2 <^b n + 1 \\ t_1\ t_2 <^b n & \Leftarrow & t_1 <^b n \wedge t_2 <^b n \\ & \cdots & \end{array}$$

Bound var. limits for substitutions:

$$\sigma <^b n$$

$$\begin{array}{lcl} \bullet <^b n & & \\ (\sigma, t) <^b n & \Leftarrow & \sigma <^b n \wedge t <^b n \end{array}$$

Figure 3.12: The logic λHOL with hybrid deBruijn variable representation: Variable limits

are ordered lists rather than sets. The order of a substitution σ needs to match the order of the context that it corresponds to. This is made clear in the definition of the operation of applying a substitution to a term, written as $t \cdot \sigma$ and given in Figure 3.13. In essence, substitution application is a simple structural recursion, the only interesting case[2] being the replacement of the i-th free variable with the i-th term of the substitution. It is evident that substitution cannot be applied to terms that use more free variables than there are in the substitution. It is simple to extend substitution application to work on substitutions as well. This operation is written as $\sigma \cdot \sigma'$.

When using names, typing rules such as ΠElim made use of substitution of a single term for the last variable in the context. Since we only have full substitutions, we use the auxiliary definition of the identity substitution id_Φ for a context Φ given in Figure 3.13 to create substitutions changing just the last variable of the context. Therefore where we previously used $t[t'/x]$ we now use $t \cdot (id_\Phi,\ t')$.

2. As a notational convenience, we mark such interesting cases with a star.

Substitution application: $t \cdot \sigma$

$$
\begin{array}{cll}
 & s \cdot \sigma & = s \\
 & c \cdot \sigma & = c \\
* & v_i \cdot \sigma & = \sigma.i \\
 & b_i \cdot \sigma & = b_i \\
 & (\lambda(t_1).t_2) \cdot \sigma & = \lambda(t_1 \cdot \sigma).(t_2 \cdot \sigma) \\
 & (t_1\ t_2) \cdot \sigma & = (t_1 \cdot \sigma)\ (t_2 \cdot \sigma) \\
 & (\Pi(t_1).t_2) \cdot \sigma & = \Pi(t_1 \cdot \sigma).(t_2 \cdot \sigma) \\
 & (t_1 = t_2) \cdot \sigma & = (t_1 \cdot \sigma) = (t_2 \cdot \sigma) \\
 & (\mathit{conv}\ t_1\ t_2) \cdot \sigma & = \mathit{conv}\ (t_1 \cdot \sigma)\ (t_2 \cdot \sigma) \\
 & (\mathit{refl}\ t) \cdot \sigma & = \mathit{refl}\ (t \cdot \sigma) \\
 & (\mathit{subst}\ (t_k.t_P)\ t) \cdot \sigma & = \mathit{subst}\ (t_k \cdot \sigma.t_P \cdot \sigma)\ (t \cdot \sigma) \\
 & (\mathit{congLam}\ (t_k.t)) \cdot \sigma & = \mathit{congLam}\ (t_k \cdot \sigma.t \cdot \sigma) \\
 & (\mathit{congForall}\ (t_k.t)) \cdot \sigma & = \mathit{congForall}\ (t_k \cdot \sigma.t \cdot \sigma) \\
 & (\mathit{beta}\ (t_k.t_1)\ t_2) \cdot \sigma & = \mathit{beta}\ (t_k \cdot \sigma.t_1 \cdot \sigma)\ (t_2 \cdot \sigma)
\end{array}
$$

Substitution application: $\sigma \cdot \sigma'$

$$
\begin{array}{lcl}
\bullet \cdot \sigma & = & \bullet \\
(\sigma',\ t) \cdot \sigma & = & (\sigma' \cdot \sigma),\ (t \cdot \sigma)
\end{array}
$$

Identity substitution: id_Φ

$$
\begin{array}{lcl}
id_\bullet & = & \bullet \\
id_{\Phi,\,t} & = & id_\Phi,\ v_{|\Phi|}
\end{array}
$$

Figure 3.13: The logic λHOL with hybrid deBruijn variable representation: Substitution Application and Identity Substitution

Also in Figure 3.9 we have added a new typing judgement in order to classify substitutions, denoted as $\Phi' \vdash \sigma : \Phi$ and stating that the substitution σ covers the context Φ whereas the free variables of the terms in σ come from the context Φ'. The interesting case is the SubstVar rule. An initial attempt to this rule could be:

$$\frac{\Phi' \vdash \sigma : \Phi \qquad \Phi' \vdash t : t'}{\Phi' \vdash (\sigma,\ t) : (\Phi,\ t')}\ \textsc{SubstVar?}$$

However, this version of the rule is wrong in cases where the type of a variable in the context Φ mentions a previous variable, as in the case: $\Phi = x : \mathit{Type},\ y : x$. If the substitution instantiates x to a concrete kind like Nat then a term substituting y should have the same type. Thus the actual SubstVar rule expects a term whose type matches the type of the variable in the context after the substitution has been applied. Indeed it matches the wrong version given above when no variable dependence exists in the type t'.

Metatheory

We are now ready to state and prove the substitution lemma, along with a number of important auxiliary lemmas and corollaries[3].

Lemma 3.4.1 *(Identity substitutions are well-typed)*

$$\frac{\vdash \Phi\ \mathit{wf} \qquad \Phi' = \Phi,\ t_1,\ t_2,\ \cdots,\ t_n \qquad \vdash \Phi'\ \mathit{wf}}{\Phi' \vdash \mathit{id}_\Phi : \Phi}$$

Proof. By structural induction on Φ. □

Lemma 3.4.2 *(Interaction between freshening and substitution application)*

$$\frac{t <^f m \qquad \sigma <^f m' \qquad |\sigma| = m}{\lceil t \cdot \sigma \rceil_{m'} = \lceil t \rceil_m \cdot (\sigma,\ v_{m'})}$$

Proof. By structural induction on t. □

3. The interested reader can find more detailed versions of these proofs in our published Technical Report [Stampoulis and Shao, 2012b].

Lemma 3.4.3 *(Interaction between binding and substitution application)*

$$\frac{t <^f m+1 \qquad \sigma <^f m' \qquad |\sigma| = m}{\lfloor t \cdot (\sigma,\ v_{m'}) \rfloor_{m'+1} = \lfloor t \rfloor_{m+1} \cdot \sigma}$$

Proof. By structural induction on t. □

Lemma 3.4.4 *(Substitutions are associative)* $(t \cdot \sigma) \cdot \sigma' = t \cdot (\sigma \cdot \sigma')$

Proof. By structural induction on t. □

Theorem 3.4.5 *(Substitution)*

$$\frac{\Phi \vdash t : t' \qquad \Phi' \vdash \sigma : \Phi}{\Phi' \vdash t \cdot \sigma : t' \cdot \sigma}$$

Proof. By structural induction on the typing derivation for t. We prove three representative cases.

Case ΠType**.**

$$\left(\frac{\Phi \vdash t_1 : s \qquad \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Phi \vdash \Pi(t_1).t_2 : s''} \right)$$

By induction hypothesis for t_1 we get:

$\Phi' \vdash t_1 \cdot \sigma : s$.

By induction hypothesis for $\Phi, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s'$ and $\Phi', t_1 \cdot \sigma \vdash (\sigma, v_{|\Phi'|}) : (\Phi, t_1)$ we get:

$\Phi', t_1 \cdot \sigma \vdash \lceil t_2 \rceil_{|\Phi|} \cdot (\sigma, v_{|\Phi'|}) : s' \cdot (\sigma, v_{|\Phi'|})$.

We have $s' = s' \cdot (\sigma, v_{|\Phi'|})$ trivially.

Also, we have that $\lceil t_2 \rceil_{|\Phi|} \cdot (\sigma, v_{|\Phi'|}) = \lceil t_2 \cdot \sigma \rceil_{|\Phi'|}$.

Thus by application of the same typing rule we get:

$\Phi' \vdash \Pi(t_1 \cdot \sigma).(t_2 \cdot \sigma) : s''$.

This is the desired, since $(\Pi(t_1).t_2) \cdot \sigma = \Pi(t_1 \cdot \sigma).(t_2 \cdot \sigma)$.

Case ΠIntro**.**

$$\left(\frac{\Phi \vdash t_1 : s \qquad \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Phi \vdash \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1} : s'}{\Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}}\right)$$

Similarly to the above.

From the induction hypothesis for t_1 and t_2 we get:

$\Phi' \vdash t_1 \cdot \sigma : s$ and $\Phi', t_1 \cdot \sigma \vdash \lceil t_2 \cdot \sigma \rceil_{|\Phi'|} : t' \cdot (\sigma, v_{|\Phi'|})$

From the induction hypothesis for $\Pi(t_1).\ \lfloor t' \rfloor$ we get:

$\Phi' \vdash (\Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma : s'$, or equivalently

$\Phi' \vdash \Pi(t_1 \cdot \sigma).(\lfloor t' \rfloor_{|\Phi+1|} \cdot \sigma) : s'$, which further rewrites to

$\Phi' \vdash \Pi(t_1 \cdot \sigma).\ \left\lfloor t' \cdot (\sigma, v_{|\Phi'|}) \right\rfloor_{|\Phi'|+1} : s'$.

We can now apply the same typing rule to get:

$\Phi' \vdash \lambda(t_1 \cdot \sigma).(t_2 \cdot \sigma) : \Pi(t_1 \cdot \sigma).\ \left\lfloor t' \cdot (\sigma, v_{|\Phi'|}) \right\rfloor_{|\Phi'|+1}$.

We have $\Pi(t_1 \cdot \sigma).\ \left\lfloor t' \cdot (\sigma, v_{|\Phi'|}) \right\rfloor_{|\Phi'|+1} = \Pi(t_1 \cdot \sigma).((\lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma) = (\Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma$, thus this is the desired result.

Case ΠElim**.**

$$\left(\frac{\Phi \vdash t_1 : \Pi(t).t' \qquad \Phi \vdash t_2 : t}{\Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (id_\Phi, t_2)}\right)$$

By induction hypothesis for t_1 and t_2 we get:

$\Phi' \vdash t_1 \cdot \sigma : \Pi(t \cdot \sigma).(t' \cdot \sigma)$.

$\Phi' \vdash t_2 \cdot \sigma : t \cdot \sigma$.

By application of the same typing rule we get:

$\Phi' \vdash (t_1\ t_2) \cdot \sigma : \lceil t' \cdot \sigma \rceil_{|\Phi'|} \cdot (id_{\Phi'}, t_2 \cdot \sigma)$.

We have that $\lceil t' \cdot \sigma \rceil_{|\Phi'|} \cdot (id_{\Phi'}, t_2 \cdot \sigma) = (\lceil t' \rceil_{|\Phi|} \cdot (\sigma, v_{|\Phi'|})) \cdot (id_{\Phi'},\ t_2 \cdot \sigma) = \lceil t' \rceil_{|\Phi|} \cdot ((\sigma, v_{|\Phi'|}) \cdot (id_{\Phi'}, t_2 \cdot \sigma))$.

But $(\sigma,\ v_{|\Phi'|}) \cdot (id_{\Phi'},\ t_2 \cdot \sigma) = \sigma,\ (t_2 \cdot \sigma)$.

Thus we only need to show that $\lceil t' \rceil_{|\Phi|} \cdot (\sigma,\ (t_2 \cdot \sigma))$ is equal to $(\lceil t' \rceil_{|\Phi|} \cdot (id_\Phi, t_2)) \cdot \sigma$.

This follows directly from the fact that $id_\Phi \cdot \sigma = \sigma$. Thus we have the desired result. □

Lemma 3.4.6 *(Substitution lemma for substitutions)*

$$\frac{\Phi' \vdash \sigma : \Phi \qquad \Phi'' \vdash \sigma' : \Phi'}{\Phi'' \vdash \sigma \cdot \sigma' : \Phi}$$

Proof. By structural induction on the typing derivation of σ and using the main substitution theorem for terms. □

Lemma 3.4.7 *(Types are well-typed)* *If* $\Phi \vdash t : t'$ *then either* $t' = Type'$ *or* $\Phi \vdash t' : s$.

Proof. By structural induction on the typing derivation for t and use of the substitution lemma in cases where t' involves a substitution, as for example in the ΠElim case. □

Lemma 3.4.8 *(Weakening)*

$$\frac{\Phi \vdash t : t' \qquad \Phi' = \Phi,\ t_1,\ t_2,\ \cdots,\ t_n \qquad \vdash \Phi'\ wf}{\Phi' \vdash t : t'}$$

Proof. Simple application of the substitution theorem using $\sigma = id_\Phi$. □

Chapter 4

The logic λHOL: Extension variables

In this chapter we will present an extension to the λHOL logic which is necessary in order to support manipulation of logical terms through the computational layer of VeriML. We will add two new kinds of variables in λHOL: meta-variables, which are used so that VeriML can manipulate *open* logical terms inhabiting specific contexts; and context variables, which are necessary so that we can manipulate contexts themselves, as well as open terms that inhabit *any* context – through combination with the meta-variables support. We refer to both of these new kinds of variables as *extension variables*. We will describe this addition to λHOL in two steps in the interests of presentation, introducing meta-variables first in Section 4.1 and then adding context variables in Section 4.2. In both cases we will present the required metatheoretic proofs in detail.

4.1 λHOL with metavariables

We have mentioned in Chapter 2 that VeriML works over *contextual terms* instead of normal logical terms, because we want to be able to work with open terms in different contexts. The need for open terms naturally arises even if we ultimately want to produce closed proofs for closed propositions. For example, we want to be able to pattern match against the body of quantified propositions like $\forall x : Nat.P$. In this case, P is an open term even if

the original proposition was closed. Similarly, if we want to produce a proof of the closed proposition $P_1 \to P_2$, we need an open proof object of the form $P_1 \vdash ? : P_2$. In a similar case in Section 2.3, we used the following notation for producing the proof of $P_1 \to P_2$:

$$\frac{X}{\Phi \vdash P_1 \to P_2}$$

Using the more precise notation which we use in this chapter, that includes proof objects, we would write the same derivation as:

$$\frac{X : (\Phi,\ P_1 \vdash P_2)}{\Phi \vdash \lambda(P_1).X : P_1 \to P_2}$$

The variable X that we use here is not a simple variable. It stands for a proof object that only makes sense in the $\Phi,\ P_1$ environment – we should not be allowed to use it in an environment that does not include P_1. It is a *meta-variable*: a special kind of variable which is able to 'capture' variables from the current context in a safe way, when it is substituted by a term t. The type of this meta-variable needs to contain both the information about the expected environment where t will make sense in addition to the type of the term. Types of metavariables are therefore *contextual terms*: a package of a context Φ together with a term t, which we will write as $[\Phi]\, t$ and denote as T in the rest of our development. The variable X above will therefore be typed as:

$$X : [\Phi,\ P_1]\, P_2$$

The notational convention that we use is that the $[\cdot]$ bracket notation associates as far to the right as possible, taking precedence over constructors of logical terms. An instantiation for the variable X would be a term t with the following typing derivation:

$$\Phi,\ P_1 \vdash t : P_2$$

If we define this derivation as the typing judgement for the contextual term $[\Phi,\ P_1]\, t$, we can use contextual terms in order to represent instantiations of meta-variables too (instead of just their types)[1]. Thus an instantiation for the variable X is the contextual term $[\Phi,\ P_1]\, t$ with the following typing:

1. It would be enough to instantiate variables with logical terms t, as the context they depend on will theoretically be evident from the type of the meta-variable they are supposed to instantiate. This choice poses unnecessary technical challenges so we opt to instantiate metavariables with contextual terms instead.

Syntax extensions

$$
\begin{array}{rrcl}
(\textit{Logical terms}) & t & ::= & \cdots \mid X_i/\sigma \\
(\textit{Contextual terms}) & T & ::= & [\Phi]\, t \\
(\textit{Meta-contexts}) & \Psi & ::= & \bullet \mid \Psi,\ T \\
(\textit{Meta-substitutions}) & \sigma_\Psi & ::= & \bullet \mid \sigma_\Psi,\ T
\end{array}
$$

Substitution application:

$$t \cdot \sigma = t'$$

$$(X_i/\sigma') \cdot \sigma \quad = \quad X_i/(\sigma' \cdot \sigma)$$

Freshening: $\lceil t \rceil_m^n$ and $\lceil \sigma \rceil_m^n$

$$
\begin{array}{lcl}
\lceil X_i/\sigma \rceil_m^n & = & X_i/(\lceil \sigma_m^n \rceil) \\
\lceil \bullet \rceil_m^n & = & \bullet \\
\lceil \sigma,\ t \rceil_m^n & = & (\lceil \sigma \rceil_m^n),\ \lceil t \rceil_m^n
\end{array}
$$

Binding: $\lfloor t \rfloor_m^n$ and $\lfloor \sigma \rfloor_m^n$

$$
\begin{array}{lcl}
\lfloor X_i/\sigma \rfloor_m^n & = & X_i/(\lfloor \sigma_m^n \rfloor) \\
\lfloor \bullet \rfloor_m^n & = & \bullet \\
\lfloor \sigma,\ t \rfloor_m^n & = & (\lfloor \sigma \rfloor_m^n),\ \lfloor t \rfloor_m^n
\end{array}
$$

Figure 4.1: Extension of λHOL with meta-variables: Syntax and Syntactic Operations

$$\vdash [\Phi,\ P_1]\, t : [\Phi,\ P_1]\, P_2$$

We use capital letters for metavariables in order to differentiate them from normal variables. Since they stand for contextual terms, we also occasionally refer to them as *contextual variables*.

Let us now see how the λHOL logic can be extended in order to properly account for meta-variables, enabling us to use them inside logical terms. The extensions required to the language we have seen so far are given in Figures 4.1 (syntax and syntactic operations) and 4.2 (typing judgements). The main addition to the logical terms is a way to refer to metavariables X_i. These come from a new context of meta-variables represented as Ψ and are instantiated with contextual terms T through meta-substitutions σ_Ψ. All metavariables are free, as we do not have binding constructs for metavariables within the logic. We represent them using deBruijn levels, similar to how we represent normal logical variables. We have made this choice so that future extensions to our proofs are simpler[2]; we will use the name-based representation instead when it simplifies our presentation.

To use a meta-variable X_i inside a logical term t, we need to make sure that when it gets substituted with a contextual term $T = [\Phi']\, t'$, the resulting term $t[T/X_i]$ will still be

2. Specifically, we are interested in adding meta-n-variables to the logic language. We can view normal logical variables as meta-0-variables and the meta-variables we add here as meta-1-variables; similarly, we can view logical terms t as meta-0-terms and contextual terms T as meta-1-terms; and so on. We carry out the same proofs for both levels, leading us to believe that an extension to n levels by induction would be a simple next step.

$\boxed{\Psi;\ \Phi \vdash t : t'}$

$$\frac{\Psi.i = T \qquad T = [\Phi']\, t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\ \textsc{MetaVar}$$

$\boxed{\vdash \Psi \text{ wf}}$

$$\frac{}{\vdash \bullet \text{ wf}}\ \textsc{MetaCEmpty} \qquad \frac{\vdash \Psi \text{ wf} \qquad \Psi \vdash [\Phi]\, t : [\Phi]\, s}{\vdash (\Psi,\ [\Phi]\, t) \text{ wf}}\ \textsc{MetaCVar}$$

$\boxed{\Psi \vdash T : T'}$

$$\frac{\Psi;\ \Phi \vdash t : t'}{\Psi \vdash [\Phi]\, t : [\Phi]\, t'}\ \textsc{CtxTerm}$$

$\boxed{\Psi \vdash \sigma_\Psi : \Psi'}$

$$\frac{}{\Psi \vdash \bullet : \bullet}\ \textsc{MetaSEmpty} \qquad \frac{\Psi \vdash \sigma_\Psi : \Psi' \qquad \Psi \vdash T : T' \cdot \sigma_\Psi}{\Psi \vdash (\sigma_\Psi,\ T) : (\Psi',\ T')}\ \textsc{MetaSVar}$$

Figure 4.2: Extension of λHOL with meta-variables: Typing rules

properly typed. The variables that t' contains need to make sense in the same context Φ that is used to type t. We could therefore require that the meta-variable is only used in contexts Φ that exactly match Φ', but this limits the cases where we can actually use meta-variables. We will opt for a more general choice instead: we require a mapping between the variables that t' refers to and terms that make sense in the context Φ where the metavariable is used. This mapping is provided by giving an explicit substitution when using the variable X_i, leading to the new form X_i/σ included in the logical terms shown in Figure 4.1; the requirement that σ maps the variables of context Φ' to terms in the current context is captured in the corresponding typing rule MetaVar of Figure 4.2. The most common case for σ will in fact be the identity substitution (where the contexts Φ and Φ' match, or Φ' is a prefix of Φ), in which case we usually write X_i instead of X_i/id_Φ.

Extending the operations of substitution application, freshening and binding is straightforward; in the latter two cases, we need to extend those operations to work on substitutions as well. The original typing judgements of λHOL are adapted by adding the new Ψ context. This is ignored save for the new MetaVar rule. Well-formedness conditions for the Ψ

$\boxed{t \cdot \sigma_\Psi = t'}$

$$
\begin{array}{lll}
s \cdot \sigma_\Psi & = & s \\
c \cdot \sigma_\Psi & = & c \\
v_i \cdot \sigma_\Psi & = & v_i \\
b_i \cdot \sigma_\Psi & = & b_i \\
(\lambda(t_1).t_2) \cdot \sigma_\Psi & = & \lambda(t_1 \cdot \sigma_\Psi).(t_2 \cdot \sigma_\Psi) \\
(t_1\ t_2) \cdot \sigma_\Psi & = & (t_1 \cdot \sigma_\Psi)\ (t_2 \cdot \sigma_\Psi) \\
(\Pi(t_1).t_2) \cdot \sigma_\Psi & = & \Pi(t_1 \cdot \sigma_\Psi).(t_2 \cdot \sigma_\Psi) \\
(t_1 = t_2) \cdot \sigma_\Psi & = & (t_1 \cdot \sigma_\Psi) = (t_2 \cdot \sigma_\Psi) \\
(\mathit{conv}\ t_1\ t_2) \cdot \sigma_\Psi & = & \mathit{conv}\ (t_1 \cdot \sigma_\Psi)\ (t_2 \cdot \sigma_\Psi) \\
(\mathit{refl}\ t) \cdot \sigma_\Psi & = & \mathit{refl}\ (t \cdot \sigma_\Psi) \\
(\mathit{subst}\ (t_k.t_P)\ t) \cdot \sigma_\Psi & = & \mathit{subst}\ (t_k \cdot \sigma_\Psi.t_P \cdot \sigma_\Psi)\ (t \cdot \sigma_\Psi) \\
(\mathit{congLam}\ (t_k.t)) \cdot \sigma_\Psi & = & \mathit{congLam}\ (t_k \cdot \sigma_\Psi.t \cdot \sigma_\Psi) \\
(\mathit{congForall}\ (t_k.t)) \cdot \sigma_\Psi & = & \mathit{congForall}\ (t_k \cdot \sigma_\Psi.t \cdot \sigma_\Psi) \\
(\mathit{beta}\ (t_k.t_1)\ t_2) \cdot \sigma_\Psi & = & \mathit{beta}\ (t_k \cdot \sigma_\Psi.t_1 \cdot \sigma_\Psi)\ (t_2 \cdot \sigma_\Psi) \\
* \quad (X_i/\sigma) \cdot \sigma_\Psi & = & t' \cdot (\sigma \cdot \sigma_\Psi) \text{ when } \sigma_\Psi.i = [\_]\, t'
\end{array}
$$

$\boxed{\sigma \cdot \sigma_\Psi = \sigma'}$

$$
\begin{array}{lll}
\bullet \cdot \sigma_\Psi & = & \bullet \\
(\sigma,\ t) \cdot \sigma_\Psi & = & \sigma \cdot \sigma_\Psi,\ t \cdot \sigma_\Psi
\end{array}
$$

$\boxed{\Phi \cdot \sigma_\Psi = \Phi'}$

$$
\begin{array}{lll}
\bullet \cdot \sigma_\Psi & = & \bullet \\
(\Phi,\ t) \cdot \sigma_\Psi & = & \Phi \cdot \sigma_\Psi,\ t \cdot \sigma_\Psi
\end{array}
$$

$\boxed{T \cdot \sigma_\Psi = T'}$

$$
* \quad ([\Phi]\, t) \cdot \sigma_\Psi = [\Phi \cdot \sigma_\Psi]\, (t \cdot \sigma_\Psi)
$$

Figure 4.3: Extension of λHOL with meta-variables: Meta-substitution application

context are similar to the ones for Φ, whereas typing for contextual terms $\Psi \vdash T : T'$ is as suggested above.

Substitutions for metavariables are lists of contextual terms and are denoted as σ_Ψ. We refer to them as metasubstitutions. Their typing rules in Figure 4.2 are similar to normal substitutions. Of interest is the definition of meta-substitution application $t \cdot \sigma_\Psi$, given in Figure 4.3. Just as application of normal substitutions, this is mostly a structural recursion. The interesting case is when $t = X_i/\sigma$. In this case, we extract the i-th contextual term $T = [\Phi]\, t'$ out of the metasubstitution σ_Ψ. t' is then applied to a substitution $\sigma' = \sigma \cdot \sigma_\Psi$. It is important to note that σ' could not simply be equal to σ, as the substitution σ itself could contain further uses of metavariables. Also, the Φ context is not needed in the operation itself, so we have elided it in the definition.

Metatheory

Since we extended the logical terms with the new construct X_i/σ, we need to make sure that the lemmas that we have proved so far still continue to hold and were not rendered invalid because of the extension. Furthermore, we will state and prove a new substitution lemma, capturing the fact that metasubstitution application to logical terms still yields well-typed terms in the new meta-variables environment.

In order to avoid redoing the proofs for the lemmas we have already proved, we follow an approach towards extending our existing proofs that resembles the open recursion solution to the expression problem. All our proofs, in their full detail as published in our Technical Report [Stampoulis and Shao, 2012b], are based on structural induction on terms of a syntactical class or on typing derivations. Through the extension, such induction principles are generalized, by adding some new cases and possibly by adding mutual induction. For example, structural induction on typing derivations of logical terms $\Phi \vdash t : t'$ is now generalized to mutual induction on typing derivations of logical terms $\Psi;\ \Phi \vdash t : t'$ and of substitutions $\Psi;\ \Phi \vdash \sigma : \Phi'$. The first part of the mutual induction matches exactly the existing proof, save for the addition of the new case for $t = X_i/\sigma$; this case actually is what introduces the requirement to do mutual induction on substitutions too. We did our proofs to a level of detail with enough individual lemmas so that all our proofs do not perform nested induction but utilize existing lemmas instead. Therefore, we only need to prove the new cases of the generalized induction principles in order to conclude that the lemmas still hold for the extension. Of course, lemma statements need to be slightly altered to account for the extra environment.

Lemma 4.1.1 (Extension of Lemma 3.4.1) *(Identity substitutions are well-typed)*

$$\frac{\Psi \vdash \Phi\ wf \qquad \Phi' = \Phi,\ t_1,\ t_2,\ \cdots,\ t_n \qquad \Psi \vdash \Phi'\ wf}{\Psi;\ \Phi' \vdash id_\Phi : \Phi}$$

Proof. The original proof was by structural induction on Φ, where no new cases were added. The new proof is thus identical as before. □

Lemma 4.1.2 (Extension of Lemma 3.4.2) *(Interaction between freshening and substitution application)*

$$1.\ \frac{t <^f m \qquad \sigma <^f m' \qquad |\sigma| = m}{\lceil t \cdot \sigma \rceil_{m'} = \lceil t \rceil_m \cdot (\sigma,\ v_{m'})} \qquad 2.\ \frac{\sigma' <^f m \qquad \sigma <^f m' \qquad |\sigma| = m}{\lceil \sigma' \cdot \sigma \rceil^n_{m'} = \lceil \sigma' \rceil^n_m \cdot (\sigma, v_{m'})}$$

Proof. Because of the extension, structural induction on t needs to be generalized to mutual induction on the structure of t and of σ'. Part 1 of the lemma is proved as before, with the addition of the following extra case for $t = X_i/\sigma'$. We have:

$$\begin{aligned}
\lceil (X_i/\sigma') \cdot \sigma \rceil^n_{m'} &= \lceil X_i/(\sigma' \cdot \sigma) \rceil^n_{m'} && \text{(by definition)} \\
&= X_i/\lceil \sigma' \cdot \sigma \rceil^n_{m'} && \text{(by definition)} \\
&= X_i/(\lceil \sigma' \rceil^n_m \cdot (\sigma,\ v_{m'})) && \text{(using mutual induction hypothesis for part 2)} \\
&= (\lceil X_i/\sigma' \rceil^n_m) \cdot (\sigma,\ v_{m'}) && \text{(by definition)}
\end{aligned}$$

The second part of the proof is proved similarly, by structural induction on σ'. □

Lemma 4.1.3 (Extension of Lemma 3.4.3) *(Interaction between binding and substitution application)*

$$1.\ \frac{t <^f m+1 \qquad \sigma <^f m' \qquad |\sigma| = m}{\lfloor t \cdot (\sigma,\ v'_m) \rfloor_{m'+1} = \lfloor t \rfloor_{m+1} \cdot \sigma} \qquad 2.\ \frac{\sigma' <^f m+1 \qquad \sigma <^f m' \qquad |\sigma| = m}{\lfloor \sigma' \cdot (\sigma, v_{m'}) \rfloor^n_{m'+1} = \lfloor \sigma' \rfloor^n_{m+1} \cdot \sigma}$$

Proof. Similar to the above. □

Lemma 4.1.4 (Extension of Lemma 3.4.4) *(Substitutions are associative)*

1. $(t \cdot \sigma) \cdot \sigma' = t \cdot (\sigma \cdot \sigma')$
2. $(\sigma_1 \cdot \sigma) \cdot \sigma' = \sigma_1 \cdot (\sigma \cdot \sigma')$

Proof. Similar to the above. □

Theorem 4.1.5 (Extension of Theorem 3.4.5) *(Substitution)*

$$1.\ \frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi;\ \Phi' \vdash \sigma : \Phi}{\Psi;\ \Phi' \vdash t \cdot \sigma : t' \cdot \sigma} \qquad 2.\ \frac{\Psi;\ \Phi' \vdash \sigma : \Phi \qquad \Psi;\ \Phi'' \vdash \sigma' : \Phi'}{\Psi;\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi}$$

Proof. We have proved the second part of the lemma already as Lemma 3.4.6. For the first part, we need to account for the new typing rule.

Case MetaVar**.**

$$\left(\frac{\Psi.i = T \qquad T = [\Phi_0]\, t_0 \qquad \Psi;\ \Phi \vdash \sigma_0 : \Phi_0}{\Psi;\ \Phi \vdash X_i/\sigma_0 : t_0 \cdot \sigma_0} \right)$$

Applying the second part of the lemma for $\sigma = \sigma_0$ and $\sigma' = \sigma$ we get:

$\Psi;\ \Phi' \vdash \sigma_0 \cdot \sigma' : \Phi_0$.

By applying the same typing rule for $t = X_i/(\sigma_0 \cdot \sigma)$ we get:

$\Psi;\ \Phi' \vdash X_i/(\sigma_0 \cdot \sigma') : t_0 \cdot (\sigma_0 \cdot \sigma')$.

By associativity of substitution application, this is the desired result. □

Lemma 4.1.6 *(Meta-context weakening)*

$$1.\ \frac{\Psi; \Phi \vdash t : t'}{\Psi, T_1, \cdots, T_n;\ \Phi \vdash t : t'} \qquad 2.\ \frac{\Psi; \Phi \vdash \sigma : \Phi'}{\Psi, T_1, \cdots, T_n;\ \Phi \vdash \sigma : \Phi'} \qquad 3.\ \frac{\Psi \vdash \Phi\ wf}{\Psi, T_1, \cdots, T_n \vdash \Phi\ wf}$$

$$4.\ \frac{\Psi \vdash T : T'}{\Psi, T_1, \cdots, T_n \vdash T : T'}$$

Proof. Trivial by structural induction on the typing derivations. □

Lemma 4.1.7 (Extension of Lemma 3.4.7) *(Types are well-typed)*

If $\Psi;\ \Phi \vdash t : t'$ *then either* $t' = \mathit{Type}'$ *or* $\Psi;\ \Phi \vdash t' : s$.

Proof. By induction on typing for t, as before.

Case MetaVar**.**

$$\left(\frac{\Psi.i = T \qquad T = [\Phi']\, t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\right)$$

By inversion of well-formedness for Ψ and meta-context weakening, we get:

$\Psi \vdash \Psi.i : [\Phi']\, s$

By typing inversion we get:

$\Psi;\ \Phi' \vdash t' : s$.

By application of the substitution theorem for t' and σ we get $\Psi;\ \Phi \vdash t' \cdot \sigma : s$, which is the desired result. □

Lemma 4.1.8 (Extension of Lemma 3.4.8) *(Weakening)*

$$1.\ \frac{\Psi;\ \Phi \vdash t : t' \qquad \Phi' = \Phi,\ t_1,\ t_2,\ \cdots,\ t_n \qquad \Psi \vdash \Phi'\ wf}{\Psi;\ \Phi' \vdash t : t'}$$

$$2.\ \frac{\Psi;\ \Phi \vdash \sigma : \Phi'' \qquad \Phi' = \Phi,\ t_1,\ t_2,\ \cdots,\ t_n \qquad \Psi \vdash \Phi'\ wf}{\Psi;\ \Phi' \vdash \sigma : \Phi''}$$

Proof. The new case for the first part is proved similarly to the above. The proof of the second part is entirely similar to the first part (use substitution theorem for the identity substitution). □

We have now extended all the lemmas that we had proved for the original version of λHOL. We will now proceed to prove a new theorem about application of meta-substitutions, similar to the normal substitution theorem 4.1.5. We first need some important auxiliary lemmas.

Lemma 4.1.9 *(Interaction of freshen and metasubstitution application)*

$$1.\ \frac{\Psi \vdash \sigma_\Psi : \Psi'}{\lceil t \rceil_m^n \cdot \sigma_\Psi = \lceil t \cdot \sigma_\Psi \rceil_m^n} \qquad 2.\ \frac{\Psi \vdash \sigma_\Psi : \Psi'}{\lceil \sigma \rceil_m^n \cdot \sigma_\Psi = \lceil \sigma \cdot \sigma_\Psi \rceil_m^n}$$

Proof. The first part is proved by induction on t. The interesting case is the $t = X_i/\sigma$ case, where we have the following.

$$
\begin{aligned}
\lceil X_i/\sigma \rceil_m^n \cdot \sigma_\Psi &= (X_i/\lceil \sigma \rceil_m^n) \cdot \sigma_\Psi \\
&= \sigma_\Psi.i \cdot (\lceil \sigma \rceil_m^n \cdot \sigma_\Psi) \\
&= \sigma_\Psi.i \cdot \lceil \sigma \cdot \sigma_\Psi \rceil_m^n && \text{(based on part 2)} \\
&= t' \cdot \lceil \sigma \cdot \sigma_\Psi \rceil_m^n && \text{(assuming } \sigma_\Psi.i = [\Phi]\, t') \\
&= \left\lceil t' \cdot (\sigma \cdot \sigma_\Psi) \right\rceil_m^n && \text{(since } t' \text{ does not include bound variables)} \\
&= \lceil \sigma_\Psi.i \cdot (\sigma \cdot \sigma_\Psi) \rceil_m^n \\
&= \lceil X_i/\sigma \cdot \sigma_\Psi \rceil_m^n
\end{aligned}
$$

The second part is proved trivially using induction on σ. □

Lemma 4.1.10 *(Interaction of bind and metasubstitution application)*

$$1.\ \frac{\Psi \vdash \sigma_\Psi : \Psi'}{\lfloor t \rfloor_m^n \cdot \sigma_\Psi = \lfloor t \cdot \sigma_\Psi \rfloor_m^n} \qquad 2.\ \frac{\Psi \vdash \sigma_\Psi : \Psi'}{\lfloor \sigma \rfloor_m^n \cdot \sigma_\Psi = \lfloor \sigma \cdot \sigma_\Psi \rfloor_m^n}$$

Proof. Similar to the above. □

Lemma 4.1.11 *(Substitution and metasubstitution application distribute)*

1. $(t \cdot \sigma) \cdot \sigma_\Psi = (t \cdot \sigma_\Psi) \cdot (\sigma \cdot \sigma_\Psi)$
2. $(\sigma \cdot \sigma') \cdot \sigma_\Psi = (\sigma \cdot \sigma_\Psi) \cdot (\sigma' \cdot \sigma_\Psi)$

Proof. Similar to the above. □

Lemma 4.1.12 *(Application of metasubstitution to identity substitution)*

$id_\Phi \cdot \sigma_\Psi = id_{\Phi \cdot \sigma_\Psi}$

Proof. By induction on Φ. □

Theorem 4.1.13 *(Meta-substitution application)*

$$1.\ \frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot \sigma_\Psi : t' \cdot \sigma_\Psi} \qquad 2.\ \frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi}$$

$$3.\ \frac{\Psi \vdash \Phi\ wf \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash \Phi \cdot \sigma_\Psi\ wf} \qquad 4.\ \frac{\Psi \vdash T : T' \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash T \cdot \sigma_\Psi : T' \cdot \sigma_\Psi}$$

Proof. All parts proceed by structural induction on the typing derivations. Here we will give some representative cases.

Case ΠElim**.**

$$\left(\frac{\Phi \vdash t_1 : \Pi(t).t' \qquad \Phi \vdash t_2 : t}{\Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (id_\Phi, t_2)}\right)$$

By induction hypothesis for t_1 we get:

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash t_1 \cdot \sigma_\Psi : \Pi(t \cdot \sigma_\Psi).(t' \cdot \sigma_\Psi)$.

By induction hypothesis for t_2 we get:

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash t_2 \cdot \sigma_\Psi : t \cdot \sigma_\Psi$.

By application of the same typing rule we get:

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash (t_1\ t_2) \cdot \sigma_\Psi : \lceil t' \cdot \sigma_\Psi \rceil_{|\Phi|} \cdot (id_\Phi, t_2 \cdot \sigma_\Psi)$.

We need to prove that $(\lceil t' \rceil_{|\Phi|} \cdot (id_\Phi, t_2)) \cdot \sigma_\Psi = \lceil t' \cdot \sigma_\Psi \rceil_{|\Phi|} \cdot (id_{\Phi \cdot \sigma_\Psi}, t_2 \cdot \sigma_\Psi)$.

$$\begin{aligned}
(\lceil t' \rceil_{|\Phi|} \cdot (id_\Phi, t_2)) \cdot \sigma_\Psi &= (\lceil t' \rceil_{|\Phi|} \cdot \sigma_\Psi) \cdot ((id_\Phi, t_2) \cdot \sigma_\Psi) && \text{(by Lemma 4.1.11)} \\
&= (\lceil t' \cdot \sigma_\Psi \rceil_{|\Phi|}) \cdot ((id_\Phi, t_2) \cdot \sigma_\Psi) && \text{(by Lemma 4.1.9)} \\
&= (\lceil t' \cdot \sigma_\Psi \rceil_{|\Phi|}) \cdot (id_\Phi \cdot \sigma_\Psi,\ t_2 \cdot \sigma_\Psi) && \text{(by definition)} \\
&= \lceil t' \cdot \sigma_\Psi \rceil_{|\Phi|} \cdot (id_{\Phi \cdot \sigma_\Psi}, t_2 \cdot \sigma_\Psi) && \text{(by Lemma 4.1.12)}
\end{aligned}$$

Case MetaVar**.**

$$\left(\frac{\Psi.i = T \qquad T = [\Phi']\, t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\right)$$

Assuming that $\sigma_\Psi.i = [\Phi'']\, t$, we need to show that $\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot (\sigma \cdot \sigma_\Psi) : (t' \cdot \sigma) \cdot \sigma_\Psi$

Equivalently from Lemma 4.1.11 we can instead show

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot (\sigma \cdot \sigma_\Psi) : (t' \cdot \sigma_\Psi) \cdot (\sigma \cdot \sigma_\Psi)$

Using the second part of the lemma for σ we get: $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi$

Furthermore we have that $\Psi' \vdash \sigma_\Psi.i : \Psi.i \cdot \sigma_\Psi$.

From hypothesis we have that $\Psi.i = [\Phi']\, t'$

Thus the above typing judgement is rewritten as $\Psi' \vdash \sigma_\Psi.i : [\Phi' \cdot \sigma_\Psi]\, t' \cdot \sigma_\Psi$

By inversion we get that $\Phi'' = \Phi' \cdot \sigma_\Psi$, thus $\sigma_\Psi.i = [\Phi' \cdot \sigma_\Psi]\, t$; and that

$\Psi';\ \Phi' \cdot \sigma_\Psi \vdash t : t' \cdot \sigma_\Psi$.

Now we use the main substitution theorem 4.1.5 for t and $\sigma \cdot \sigma_\Psi$ and get:

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot (\sigma \cdot \sigma_\Psi) : (t' \cdot \sigma_\Psi) \cdot (\sigma \cdot \sigma_\Psi)$

Case SubstVar**.**

$$\left(\frac{\Phi \vdash \sigma : \Phi' \qquad \Phi \vdash t : t' \cdot \sigma}{\Phi \vdash \sigma,\ t : (\Phi',\ t')}\right)$$

By induction hypothesis and use of part 1 we get:

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi$

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot \sigma_\Psi : (t' \cdot \sigma) \cdot \sigma_\Psi$

By use of Lemma 4.1.11 in the typing for $t \cdot \sigma_\Psi$ we get that:

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot \sigma_\Psi : (t' \cdot \sigma_\Psi) \cdot (\sigma \cdot \sigma_\Psi)$

By use of the same typing rule we get:

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash (\sigma \cdot \sigma_\Psi,\ t \cdot \sigma_\Psi) : (\Phi' \cdot \sigma_\Psi,\ t' \cdot \sigma_\Psi)$ □

4.2 λHOL with extension variables

Most functions in VeriML work with logical terms that live in an arbitrary context. This is for example true for automated provers such as the tautology prover presented in Section 2.3, which need to use recursion to prove propositions in extended contexts. Thus, even if the

original context is closed, recursive calls work over terms living in extensions of the context; and the prover needs to handle all such extensions – it needs to be able to handle all possible contexts. The metavariables that we have seen so far do not allow this possibility, as they depend on a fully-specified Φ context. In this section we will see a further extension of λHOL with *parametric contexts*. Contexts can include context variables that can be instantiated with an arbitrary context. We can see context variables as placeholders for applying the weakening lemma. Thus this extension internalizes the weakening lemma within the λHOL calculus, just as the extension with metavariables internalizes the substitution theorem. We denote context variables with lowercase letters, e.g. ϕ, in order to differentiate from the Φ context.

Let us sketch an example of the use of parametric contexts. Consider the case where we want to produce a proof of $x > 0 \to x \geq 1$ in some unspecified context Φ. This context needs to include a definition of x as a natural number in order for the proposition to make sense. We can assume that we have a metavariable X standing for a proof with the following context and type:

$$X : [x : Nat,\ \phi,\ h : x > 0]\, x \geq 1$$

we can produce a proof for $[x : Nat,\ \phi]\, x > 0 \to x \geq 1$ with:

$$\frac{X : [x : Nat,\ \phi,\ h : x > 0]\, x \geq 1}{x : Nat,\ \phi \vdash (\lambda h : x > 0.X) : x > 0 \to x \geq 1}$$

The context variable ϕ can be instantiated with an arbitrary context, which must be well-formed under the initial $x : Nat$, context. We capture this information into the type of ϕ as follows:

$$\phi : [x : Nat]\ ctx$$

A possible instantiation of ϕ is $\phi = [h : x > 5]$. By applying the substitution of ϕ for this instantiation we get the following derivation:

$$\frac{X : [x : Nat,\ h : x > 5,\ h' : x > 0]\, x \geq 1}{x : Nat,\ h : x > 5 \vdash (\lambda h' : x > 0.X) : x > 0 \to x \geq 1}$$

As is evident, the substitution of a context variable for a specific context might involve a series of α-renamings in the context and terms, as we did here so that the newly-introduced h variable does not clash with the already existing h variable.

We will now cover the extensions required to λHOL as presented so far in order to account properly for parametric contexts. The main ideas of the extension are the following:

1. Both context variables and meta-variables should be typed through the same environment and should be instantiated through the same substitutions, instead of having separate environments and substitutions for context- and meta-variables.

2. The deBruijn levels used for normal free variables should be generalized to *parametric deBruijn levels*: instead of being constant numbers they should be sums involving variables $|\phi|$ standing for the length of as-yet-unspecified contexts.

We have arrived at an extension of λHOL that follows these ideas after working out the proofs for a variety of alternative ways to support parametric contexts. The solution that we present here is characterized both by clarity in the definitions and the lemma statements as well as by conceptually simple proofs that can be carried out using structural recursion. Intuitively, the main reason is that parametric deBruijn levels allow for a clean way to perform the necessary α-renamings when instantiating a parametric context with a concrete context, by simply substituting the length variables for the concrete length. Other solutions – e.g. keeping free variables as normal deBruijn levels and shifting them up when a parametric context is instantiated – require extra knowledge about the initial parametric context upon instantiation, destroying the property that meta-substitution application is structurally recursive. The justification for the first key idea noted above is technically more subtle: by separating meta-variables from context variables into different contexts, the relative order of their introduction is lost. This leads to complications when a meta-variable depends on a parametric context which depends on another meta-variable itself and requires various side-conditions to the statements of lemmas that add significant technical complexity.

We present the extension of λHOL in a series of figures: Figure 4.4 gives the extension to the syntax; Figures 4.5 through 4.7 give the syntactic operations; Figure 4.8 defines the new relation of subsumption for contexts and substitutions; Figures 4.9 and 4.10 give the new typing rules; and Figure 4.11 gives the new syntactic operation of extension substitution application. We will now describe these in more detail.

$$
\begin{array}{rlll}
(\textit{Logical terms}) & t & ::= & s \mid c \mid v_L \mid b_i \mid \lambda(t_1).t_2 \mid t_1\ t_2 \mid \Pi(t_1).t_2 \mid t_1 = t_2 \\
 & & & \mid \textit{conv}\ t_1\ t_2 \mid \textit{subst}\ ((t_k).t_P)\ t \mid \textit{refl}\ t \\
 & & & \mid \textit{congLam}\ ((t_k).t) \mid \textit{congForall}\ ((t_k).t) \\
 & & & \mid \textit{beta}\ ((t_k).t_1)\ t_2 \\
 & & * & \mid X_i/\sigma \\
* (\textit{Parametric deBruijn levels}) & L & ::= & 0 \mid L \dot{+} 1 \mid L \dot{+} |\phi_i| \\
(\textit{Contexts}) & \Phi & ::= & \cdots \mid \Phi,\ \phi_i \\
(\textit{Substitutions}) & \sigma & ::= & \cdots \mid \sigma,\ \mathbf{id}(\phi_i) \\
(\textit{Extension contexts}) & \Psi & ::= & \bullet \mid \Psi,\ K \\
* (\textit{Extension terms}) & T & ::= & [\Phi]\,t \mid [\Phi]\,\Phi' \\
(\textit{Extension types}) & K & ::= & [\Phi]\,t \mid [\Phi]\,\textit{ctx} \\
(\textit{Extension substitutions}) & \sigma_\Psi & ::= & \bullet \mid \sigma_\Psi,\ T
\end{array}
$$

Figure 4.4: Extension of λHOL with parametric contexts: Syntax

Substitution length: $|\sigma| = L$

$$
\begin{array}{lcl}
|\bullet| & = & 0 \\
|\sigma,\ t| & = & |\sigma| \dot{+} 1 \\
|\sigma,\ \mathbf{id}(\phi_i)| & = & |\sigma| \dot{+} |\phi_i|
\end{array}
$$

Context length: $|\sigma| = L$

$$
\begin{array}{lcl}
|\bullet| & = & 0 \\
|\Phi,\ t| & = & |\Phi| \dot{+} 1 \\
|\Phi,\ \phi_i| & = & |\Phi| \dot{+} |\phi_i|
\end{array}
$$

Substitution access: $\sigma.L = t$

$$
\begin{array}{lcl}
(\sigma,\ t).L & = & t \text{ when } |\sigma| = L \\
(\sigma,\ t).L & = & \sigma.L \text{ otherwise} \\
(\sigma,\ \mathbf{id}(\phi_i)).L & = & \sigma.L \text{ when } |\sigma| < L
\end{array}
$$

Context access: $\Phi.L = t$

$$
\begin{array}{lcl}
(\Phi,\ t).L & = & t \text{ when } |\Phi| = L \\
(\Phi,\ t).L & = & \Phi.L \text{ otherwise} \\
(\Phi,\ \phi_i).L & = & \Phi.L \text{ when } L < |\Phi|
\end{array}
$$

Substitution application: $t \cdot \sigma = t'$

$$
v_L \cdot \sigma \ = \ \sigma.L
$$

Substitution application: $\sigma' \cdot \sigma = \sigma''$

$$
(\sigma',\ \mathbf{id}(\phi_i)) \cdot \sigma \ = \ \sigma' \cdot \sigma,\ \mathbf{id}(\phi_i)
$$

Identity substitution: $id_\Phi = \sigma$

$$
\begin{array}{lcl}
id_\bullet & = & \bullet \\
id_{\Phi,\ t} & = & id_\Phi,\ v_{|\Phi|} \\
id_{\Phi,\ \phi_i} & = & id_\Phi,\ \mathbf{id}(\phi_i)
\end{array}
$$

Partial identity substitution: $id_{[\Phi]\,\Phi'} = \sigma$

$$
\begin{array}{lcl}
id_{[\Phi]\bullet} & = & \bullet \\
id_{[\Phi]\,\Phi',\ t} & = & id_{[\Phi]\,\Phi'},\ v_{|\Phi| \dot{+} |\Phi'|} \\
id_{[\Phi]\,\Phi',\ \phi_i} & = & id_{[\Phi]\,\Phi'},\ \mathbf{id}(\phi_i)
\end{array}
$$

Figure 4.5: Extension of λHOL with parametric contexts: Syntactic Operations (Length and access of substitutions and contexts; Substitution application; Identity substitution and partial identity substitution)

Level comparison: $L < L'$

$$
\begin{array}{lcl}
L & < & L' \dot{+} 1 \text{ when } L = L' \text{ or } L < L' \\
L & < & L' \dot{+} |\phi_i| \text{ when } L = L' \text{ or } L < L'
\end{array}
$$

Variable limits: $t <^f L$

$$
\begin{array}{lcl}
s <^f L & & \\
c <^f L & & \\
v_L <^f L' & \Leftarrow & L < L' \\
b_i <^f L & & \\
(\lambda(t_1).t_2) <^f L & \Leftarrow & t_1 <^f L \wedge t_2 <^f L \\
t_1\ t_2 <^f L & \Leftarrow & t_1 <^f L \wedge t_2 <^f L \\
 & \Leftarrow & \cdots
\end{array}
$$

Variable limits: $\sigma <^f L$

$$
\begin{array}{lcl}
\bullet <^f L & & \\
\sigma,\ t <^f L & \Leftarrow & \sigma <^f L \wedge t <^f L \\
\sigma,\ \mathbf{id}(\phi_i) <^f L & \Leftarrow & \sigma <^f L \wedge L = L' \dot{+} |\phi_i| \dot{+} \cdots
\end{array}
$$

Figure 4.6: Extension of λHOL with parametric contexts: Variable limits

Freshening (terms): $\lceil t \rceil_L^n = t'$

$$
\begin{array}{lcl}
\lceil b_n \rceil_L^n & = & v_L \\
\lceil b_i \rceil_L^n & = & b_i
\end{array}
$$

Freshening (subst.): $\lceil \sigma \rceil_L^n = \sigma'$

$$
\begin{array}{lcl}
\lceil \bullet \rceil_L^n & = & \bullet \\
\lceil \sigma,\ t \rceil_L^n & = & \lceil \sigma \rceil_L^n,\ \lceil t \rceil_L^n \\
\lceil \sigma,\ \mathbf{id}(\phi_i) \rceil_L^n & = & \lceil \sigma \rceil_L^n,\ \mathbf{id}(\phi_i)
\end{array}
$$

Binding (terms): $\lfloor t \rfloor_L^n = t'$

$$
\begin{array}{lcl}
\lfloor v_{L'} \rfloor_L^n & = & b_n \text{ when } L = L' \dot{+} 1 \\
\lfloor v_{L'} \rfloor_L^n & = & v_{L'} \text{ otherwise}
\end{array}
$$

Binding (subst.): $\lfloor \sigma \rfloor_L^n = \sigma'$

$$
\begin{array}{lcl}
\lfloor \bullet \rfloor_L^n & = & \bullet \\
\lfloor \sigma,\ t \rfloor_L^n & = & \lfloor \sigma \rfloor_L^n,\ \lfloor t \rfloor_L^n \\
\lfloor \sigma,\ \mathbf{id}(\phi_i) \rfloor_L^n & = & \lfloor \sigma \rfloor_L^n,\ \mathbf{id}(\phi_i)
\end{array}
$$

Figure 4.7: Extension of λHOL with parametric contexts: Syntactic Operations (Freshening and binding)

Environment subsumption: $\Phi \subseteq \Phi'$

$$
\begin{array}{lcl}
\Phi \subseteq \Phi & & \\
\Phi \subseteq \Phi',\ t & \Leftarrow & \Phi \subseteq \Phi' \\
\Phi \subseteq \Phi',\ \phi_i & \Leftarrow & \Phi \subseteq \Phi'
\end{array}
$$

Substitution subsumption: $\sigma \subseteq \sigma'$

$$
\begin{array}{lcl}
\sigma \subseteq \sigma & & \\
\sigma \subseteq \sigma',\ t & \Leftarrow & \sigma \subseteq \sigma' \\
\sigma \subseteq \sigma',\ \mathbf{id}(\phi_i) & \Leftarrow & \sigma \subseteq \sigma'
\end{array}
$$

Figure 4.8: Extension of λHOL with parametric contexts: Subsumption

$\boxed{\Psi \vdash_\Sigma \Phi \text{ wf}}$

$$\frac{}{\Psi \vdash \bullet \text{ wf}}\ \textsc{CtxEmpty} \qquad \frac{\Psi \vdash \Phi \text{ wf} \qquad \Psi;\ \Phi \vdash t : s}{\Psi \vdash (\Phi,\ t) \text{ wf}}\ \textsc{CtxVar}$$

$$\frac{\Psi \vdash \Phi \text{ wf} \qquad \Psi.i = [\Phi]\ ctx}{\Psi \vdash (\Phi,\ \phi_i) \text{ wf}}\ \textsc{CtxCVar}$$

$\boxed{\Psi;\ \Phi \vdash \sigma : \Phi'}$

$$\frac{}{\Psi;\ \Phi \vdash \bullet : \bullet}\ \textsc{SubstEmpty} \qquad \frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi;\ \Phi \vdash t : t' \cdot \sigma}{\Psi;\ \Phi \vdash (\sigma,\ t) : (\Phi',\ t')}\ \textsc{SubstVar}$$

$$\frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi.i = [\Phi']\ ctx \qquad (\Phi',\ \phi_i) \subseteq \Phi}{\Psi;\ \Phi \vdash (\sigma,\ \mathbf{id}(\phi_i)) : (\Phi',\ \phi_i)}\ \textsc{SubstCVar}$$

Figure 4.9: Extension of λHOL with parametric contexts: Typing

First of all, instead of having just metavariables X_i as in the previous section, we now also have context variables ϕ_i. These are free variables coming from the same context as metavariables Ψ. We refer to both metavariables and context variables as *extension variables* and to the context Ψ as the *extension context*. Note that even though we use different notation for the two kinds of variables this is mostly a presentation convenience; we use V_i to mean either kind. Extension variables are typed through *extension types* – which in the case of metavariables are normal contextual terms. Context variables stand for contexts that assume a given context prefix Φ, as in the case of the informal example of ϕ above. We write their type as $[\Phi]\ ctx$, specifying the prefix they assume Φ and the fact that they stand for contexts. The instantiations of extension variables are *extension terms* T, lists of which form extension substitutions denoted as σ_Ψ. In the case of metavariables, an instantiation is a contextual term $[\Phi]\,t$ as we saw it in the previous section, its type matching the contextual type $[\Phi]\,t'$ (rule ExtCtxTerm in in Figure 4.10). In the case of a context variable $\phi_i : [\Phi]\ ctx$ an instantiation a context Φ' that extends the prefix Φ (rule ExtCtxInst in Figure 4.10). We denote *partial contexts* such as Φ' as $[\Phi]\,\Phi'$ repeating the information about Φ just as we did for instantiations of metavariables.

In order to be able to use context parameters, we extend the contexts Φ in Figure 4.4

$\boxed{\Psi;\ \Phi \vdash t : t'}$

$$\frac{c : t \in \Sigma}{\Psi;\ \Phi \vdash_\Sigma c : t}\ \textsc{Constant} \qquad \frac{\Phi.L = t}{\Psi;\ \Phi \vdash v_L : t}\ \textsc{Var} \qquad \frac{(s, s') \in \mathcal{A}}{\Psi;\ \Phi \vdash s : s'}\ \textsc{Sort}$$

$$\frac{\Psi;\ \Phi \vdash t_1 : s \qquad \Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Psi;\ \Phi \vdash \Pi(t_1).t_2 : s''}\ \Pi\textsc{Type}$$

$$\frac{\Psi;\ \Phi \vdash t_1 : s \qquad \Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Psi;\ \Phi \vdash \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|\dot{+}1} : s'}{\Psi;\ \Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|\dot{+}1}}\ \Pi\textsc{Intro}$$

$$\frac{\Psi;\ \Phi \vdash t_1 : \Pi(t).t' \qquad \Psi;\ \Phi \vdash t_2 : t}{\Psi;\ \Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (id_\Phi,\ t_2)}\ \Pi\textsc{Elim}$$

$$\frac{\Psi.i = T \qquad T = [\Phi']\, t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\ \textsc{MetaVar}$$

$\boxed{\vdash \Psi\ \text{wf}}$

$$\frac{}{\vdash \bullet\ \text{wf}}\ \textsc{ExtCEmpty} \qquad \frac{\vdash \Psi\ \text{wf} \qquad \Psi \vdash \Phi\ \text{wf}}{\vdash (\Psi,\ [\Phi]\, ctx)\ \text{wf}}\ \textsc{ExtCMeta}$$

$$\frac{\vdash \Psi\ \text{wf} \qquad \Psi \vdash [\Phi]\, t : [\Phi]\, s}{\vdash (\Psi,\ [\Phi]\, t)\ \text{wf}}\ \textsc{ExtCCtx}$$

$\boxed{\Psi \vdash T : K}$

$$\frac{\Psi;\ \Phi \vdash t : t'}{\Psi \vdash [\Phi]\, t : [\Phi]\, t'}\ \textsc{ExtCtxTerm} \qquad \frac{\Psi \vdash \Phi,\ \Phi'\ \text{wf}}{\Psi \vdash [\Phi]\, \Phi' : [\Phi]\, ctx}\ \textsc{ExtCtxInst}$$

$\boxed{\Psi \vdash \sigma_\Psi : \Psi'}$

$$\frac{}{\Psi \vdash \bullet : \bullet}\ \textsc{ExtSEmpty} \qquad \frac{\Psi \vdash \sigma_\Psi : \Psi' \qquad \Psi \vdash T : K \cdot \sigma_\Psi}{\Psi \vdash (\sigma_\Psi,\ T) : (\Psi',\ K)}\ \textsc{ExtSVar}$$

Figure 4.10: Extension of λHOL with parametric contexts: Typing (continued)

$\boxed{L \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
0 \cdot \sigma_\Psi & = & 0 \\
(L \dot{+} 1) \cdot \sigma_\Psi & = & (L \cdot \sigma_\Psi) \dot{+} 1 \\
* \quad (L \dot{+} |\phi_i|) \cdot \sigma_\Psi & = & (L \cdot \sigma_\Psi) \dot{+} |\Phi'| \text{ when } \sigma_\Psi.i = [\_]\,\Phi'
\end{array}
$$

$\boxed{t \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
* \quad v_L \cdot \sigma_\Psi & = & v_{L \cdot \sigma_\Psi} \\
(X_i/\sigma) \cdot \sigma_\Psi & = & t \cdot (\sigma \cdot \sigma_\Psi) \text{ when } \sigma_\Psi.i = [\_]\,t
\end{array}
$$

$\boxed{\sigma \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
\bullet \cdot \sigma_\Psi & = & \bullet \\
(\sigma,\ t) \cdot \sigma_\Psi & = & \sigma \cdot \sigma_\Psi,\ t \cdot \sigma_\Psi \\
* \quad (\sigma,\ \mathbf{id}(\phi_i)) \cdot \sigma_\Psi & = & \sigma \cdot \sigma_\Psi,\ id_{[\Phi]\,\Phi'} \text{ when } \sigma_\Psi.i = [\Phi]\,\Phi'
\end{array}
$$

$\boxed{\Phi \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
\bullet \cdot \sigma_\Psi & = & \bullet \\
(\Phi,\ t) \cdot \sigma_\Psi & = & \Phi \cdot \sigma_\Psi,\ t \cdot \sigma_\Psi \\
* \quad (\Phi,\ \phi_i) \cdot \sigma_\Psi & = & \Phi \cdot \sigma_\Psi,\ \Phi' \text{ when } \sigma_\Psi.i = [\_]\,\Phi'
\end{array}
$$

$\boxed{T \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
([\Phi]\,t) \cdot \sigma_\Psi & = & [\Phi \cdot \sigma_\Psi]\,(t \cdot \sigma_\Psi) \\
([\Phi]\,\Phi') \cdot \sigma_\Psi & = & [\Phi \cdot \sigma_\Psi]\,(\Phi' \cdot \sigma_\Psi)
\end{array}
$$

$\boxed{K \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
([\Phi]\,t) \cdot \sigma_\Psi & = & [\Phi \cdot \sigma_\Psi]\,(t \cdot \sigma_\Psi) \\
([\Phi]\,ctx) \cdot \sigma_\Psi & = & [\Phi \cdot \sigma_\Psi]\,ctx
\end{array}
$$

$\boxed{\sigma_\Psi \cdot \sigma'_\Psi}$

$$
\begin{array}{lcl}
\bullet \cdot \sigma'_\Psi & = & \bullet \\
(\sigma_\Psi,\ T) \cdot \sigma'_\Psi & = & \sigma_\Psi \cdot \sigma'_\Psi,\ T \cdot \sigma'_\Psi
\end{array}
$$

Figure 4.11: Extension of λHOL with parametric contexts: Extension substitution application

so that they can mention such variables. Well-formedness for contexts is similarly extended in Figure 4.9 with the rule CtxCVar. Notice that we are only allowed to use a context variable when its prefix exactly matches the current context; this is unlike metavariables where we allow them to be used in different contexts by specifying a substitution. This was guided by practical reasons, as we have not found cases where the extra flexibility of being able to provide a substitution is needed; still, it might be a useful addition in the future.

As described earlier, free logical variables are now indexed by parametric deBruijn levels L instead of natural numbers. This is reflected in the new syntax for logical terms which includes the form v_L instead of v_i in Figure 4.4. The new syntactic class of parametric deBruijn levels represents the number of variables from the current context that we have to skip over in order to reach the variable we are referring to. When the current context mentions a context variable ϕ_i, we need to skip over an unspecified number of variables – or rather, as many variables as the length of the instantiation of ϕ_i. We denote this length as $|\phi_i|$. Thus, instead of numbers, levels can be seen as first-order polynomials over the $|\phi_i|$ variables. More precisely, they are polynomials that only use noncommutative addition that we denote as $\dotplus$, reflecting the fact that contexts are ordered. It is easy to see that rearranging the uses of $|\phi_i|$ variables inside level polynomials we would lose the connection to the context they refer to.

Substitutions are extended similarly in Figure 4.4 to account for context variables, using the placeholder $\mathbf{id}(\phi_i)$ to stand for the identity substitution once ϕ_i gets instantiated. Notice that since ϕ_i is instantiated with a partial context $[\Phi]\,\Phi'$, the identity substitution expanded in the place of $\mathbf{id}(\phi_i)$ will similarly be a *partial identity substitution*, starting with the variable coming after Φ. We define this as a new syntactic operation in Figure 4.5. The new typing rule for substitutions SubstCVar given in Figure 4.9 requires that we can only use the identity substitution placeholder for ϕ_i if the current context includes ϕ_i. We make this notion of inclusion precise through the relation of *context subsumption*, formalized in Figure 4.8, where it is also defined for substitutions so that it can be used in some lemma statements.

The change to parametric levels immediately changes many aspects of the definition of λHOL: for example, accessing the L-th element of a context or a substitution is not directly

intuitively clear anymore, as L is more than a number and contexts or substitutions are more than lists of terms. We have adapted all such operations, including relations such as level comparisons (e.g. $L < L'$) and operations such as level addition (e.g. $L \dot{+} L' = L''$). We only give some of these here: substitution and context length and accessing operations in Figure 4.5; level comparison and variable limits in Figure 4.6; and freshening and binding in Figure 4.7.

Last, the most important new addition to the language is the operation of applying an extension substitution σ_Ψ, given in Figure 4.11 and especially the case for instantiating a context variable. To see how this works, consider the following example: we have the context variable $\phi_0 : [x : Nat]\, ctx$ and the contextual term:
$T = [x : Nat,\ \phi_0,\ y : Nat]\, plus\ x\ y$
represented as $T = [Nat,\ \phi_0,\ Nat]\, plus\ v_0\ v_{1\dot{+}|\phi_0|}$.
We instantiate ϕ_0 through the extension substitution:
$\sigma_\Psi = (\ \phi_0 \mapsto [x : Nat]\, y : Nat,\ z : Nat\ )$
represented as $\sigma_\Psi = (\ [Nat]\, Nat,\ Nat\ )$. The steps are as follows:

$$
\begin{aligned}
([Nat,\ \phi_0,\ Nat]\, plus\ v_0\ v_{1\dot{+}|\phi_0|}) \cdot \sigma_\Psi &= [(Nat,\ \phi_0,\ Nat) \cdot \sigma_\Psi]\, (plus\ v_0\ v_{1\dot{+}|\phi_0|}) \cdot \sigma_\Psi \\
&= [Nat,\ \underline{Nat,\ Nat},\ Nat]\, (plus\ v_0\ v_{1\dot{+}|\phi_0|}) \cdot \sigma_\Psi \\
&= [Nat,\ Nat,\ Nat,\ Nat]\, plus\ v_0\ v_{1\dot{+}(|\phi_0| \cdot \sigma_\Psi)} \\
&= [Nat,\ Nat,\ Nat,\ Nat]\, plus\ v_0\ v_{1\dot{+}2} \\
&= [Nat,\ Nat,\ Nat,\ Nat]\, plus\ v_0\ v_3
\end{aligned}
$$

Though not shown in the example, identity substitution placeholders $\mathbf{id}(\phi_i)$ are expanded to identity substitutions, similarly to how contexts are expanded in place of ϕ_i variables.

Metatheory

We will now extend the metatheoretic results we have for λHOL to account for the additions and adjustments presented here. The auxiliary lemmas that we proved in the previous section are simple to extend using the proof techniques we have shown. We will rather focus

on auxiliary lemmas whose proofs are rendered interesting because of the new additions, and also on covering the new cases for the main substitution theorems.

Lemma 4.2.1 *(Identity substitution leaves terms unchanged)*

$$1.\ \frac{t <^f L \qquad |\Phi| = L}{t \cdot id_\Phi = t} \qquad\qquad 2.\ \frac{\sigma <^f L \qquad |\Phi| = L}{\sigma \cdot id_\Phi = \sigma}$$

Proof. Part 1 is proved by induction on $t <^f L$. The interesting case is $v_{L'}$, with $L' < L$. In this case we have to prove $id_\Phi.L' = v_{L'}$. This is done by induction on $L' < L$.

When $L = L' \dot{+} 1$ we have by inversion of $|\Phi| = L$ that $\Phi = \Phi',\ t$ and $|\Phi'| = L'$. Thus $id_\Phi = id_{\Phi'},\ v_{L'}$ and hence the desired result.

When $L = L' \dot{+} |\phi_i|$, similarly.

When $L = L^* \dot{+} 1$ and $L' < L^*$, we have that $\Phi = \Phi^*,\ t$ and $|\Phi^*| = L^*$. By (inner) induction hypothesis we get that $id_{\Phi^*}.L' = v_{L'}$. From this we get directly that $id_\Phi.L' = v_{L'}$.

When $L = L^* \dot{+} |\phi_i|$ and $L' < L^*$, entirely as the previous case.

Part 2 is trivial to prove by induction and use of part 1 in cases $\sigma = \bullet$ or $\sigma = \sigma',\ t$. In the case $\sigma = \sigma',\ \mathbf{id}(\phi_i)$ we have: $\sigma' <^f L$ thus by induction $\sigma' \cdot id_\Phi = \sigma'$, and furthermore $(\sigma',\ \mathbf{id}(\phi_i)) \cdot id_\Phi = \sigma$. □

Theorem 4.2.2 (Extension of Theorem 4.1.5) *(Substitution)*

$$1.\ \frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi;\ \Phi' \vdash \sigma : \Phi}{\Psi;\ \Phi' \vdash t \cdot \sigma : t' \cdot \sigma} \qquad\qquad 2.\ \frac{\Psi;\ \Phi' \vdash \sigma : \Phi \qquad \Psi;\ \Phi'' \vdash \sigma' : \Phi'}{\Psi;\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi}$$

Proof. Part 1 is identical as before; all the needed lemmas are trivial to adjust, so the new form of indexes does not change the proof at all. Similarly for part 2, save for extending the previous proof by the new case for substitutions.

Case SubstCVar**.**

$$\left(\frac{\Psi;\ \Phi' \vdash \sigma : \Phi_0 \qquad \Psi.i = [\Phi_0]\ ctx \qquad (\Phi_0,\ \phi_i) \subseteq \Phi'}{\Psi;\ \Phi' \vdash (\sigma,\ \mathbf{id}(\phi_i)) : (\Phi_0,\ \phi_i)}\right)$$

By induction hypothesis for σ, we get: $\Psi;\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi_0$.

We need to prove that $(\Phi_0,\ \phi_i) \subseteq \Phi''$.

By induction on $(\Phi_0,\ \phi_i) \subseteq \Phi'$ and repeated inversions of the typing of σ' we arrive at a $\sigma'' \subseteq \sigma'$ such that:

$\Psi;\ \Phi'' \vdash \sigma'' : \Phi_0,\ \phi_i$

By inversion of this we get that $(\Phi_0,\ \phi_i) \subseteq \Phi''$.

Thus, using the same typing rule, we get:

$\Psi;\ \Phi'' \vdash (\sigma \cdot \sigma', \mathbf{id}(\phi_i)) : (\Phi_0,\ \phi_i)$, which is the desired. □

Lemma 4.2.3 *(Interaction of extensions substitution and element access)*

1. $(\sigma.L) \cdot \sigma_\Psi = (\sigma \cdot \sigma_\Psi).L \cdot \sigma_\Psi$
2. $(\Phi.L) \cdot \sigma_\Psi = (\Phi \cdot \sigma_\Psi).L \cdot \sigma_\Psi$

Proof. By induction on L and taking into account the implicit assumption that $L < |\sigma|$ or $L < |\Phi|$. □

Lemma 4.2.4 (Extension of Lemma 4.1.11) *(Substitution and extension substitution application distribute)*

1. $(t \cdot \sigma) \cdot \sigma_\Psi = (t \cdot \sigma_\Psi) \cdot (\sigma \cdot \sigma_\Psi)$
2. $(\sigma \cdot \sigma') \cdot \sigma_\Psi = (\sigma \cdot \sigma_\Psi) \cdot (\sigma' \cdot \sigma_\Psi)$

Proof. Part 1 is identical as before. Part 2 is trivial to prove for the new case of σ. □

Lemma 4.2.5 *(Extension substitutions are associative)*

1. $(L \cdot \sigma_\Psi) \cdot \sigma'_\Psi = L \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$
2. $(t \cdot \sigma_\Psi) \cdot \sigma'_\Psi = t \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$
3. $(\Phi \cdot \sigma_\Psi) \cdot \sigma'_\Psi = \Phi \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$
4. $(\sigma \cdot \sigma_\Psi) \cdot \sigma'_\Psi = \sigma \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$
5. $(T \cdot \sigma_\Psi) \cdot \sigma'_\Psi = T \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$

6. $(K \cdot \sigma_\Psi) \cdot \sigma'_\Psi = K \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$

7. $(\Psi \cdot \sigma_\Psi) \cdot \sigma'_\Psi = \Psi \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$

Proof.

Part 1 By induction on L. The interesting case is $L = L' \dot{+} \phi_i$. In that case we have:

$$\begin{aligned}
(L \cdot \sigma_\Psi) \cdot \sigma'_\Psi &= (L' \cdot \sigma_\Psi) \cdot \sigma'_\Psi \dot{+} |\Phi'| \cdot \sigma'_\Psi && \text{(assuming } \sigma_\Psi.i = [\Phi]\,\Phi') \\
&= (L' \cdot \sigma_\Psi) \cdot \sigma'_\Psi \dot{+} |\Phi' \cdot \sigma'_\Psi| \\
&= L' \cdot (\sigma_\Psi \cdot \sigma'_\Psi) \dot{+} |\Phi' \cdot \sigma'_\Psi| && \text{(by induction hypothesis)}
\end{aligned}$$

This is the desired, since $(\sigma_\Psi \cdot \sigma'_\Psi).i = [\Phi \cdot \sigma_\Psi]\,\Phi' \cdot \sigma'_\Psi$.

Part 2 By induction on t. The interesting case is $t = X_i/\sigma$. The left-hand-side is then equal to:

$$\begin{aligned}
(\sigma_\Psi.i \cdot (\sigma \cdot \sigma_\Psi)) \cdot \sigma'_\Psi &= (t \cdot (\sigma \cdot \sigma_\Psi)) \cdot \sigma'_\Psi && \text{(assuming } \sigma_\Psi.i = [\Phi]\,t) \\
&= (t \cdot \sigma'_\Psi) \cdot ((\sigma \cdot \sigma_\Psi) \cdot \sigma'_\Psi) && \text{(through Lemma 4.2.4)} \\
&= (t \cdot \sigma'_\Psi) \cdot (\sigma \cdot (\sigma_\Psi \cdot \sigma'_\Psi)) && \text{(through part 4)}
\end{aligned}$$

The right-hand side is written as:

$$\begin{aligned}
(X_i/\sigma) \cdot (\sigma_\Psi \cdot \sigma'_\Psi) &= ((\sigma_\Psi \cdot \sigma'_\Psi).i) \cdot (\sigma \cdot (\sigma_\Psi \cdot \sigma'_\Psi)) \\
&= ((\sigma_\Psi.i) \cdot \sigma'_\Psi) \cdot (\sigma \cdot (\sigma_\Psi \cdot \sigma'_\Psi)) \\
&= ([\Phi \cdot \sigma'_\Psi]\,(t \cdot \sigma'_\Psi)) \cdot (\sigma \cdot (\sigma_\Psi \cdot \sigma'_\Psi)) \\
&= (t \cdot \sigma'_\Psi) \cdot (\sigma \cdot (\sigma_\Psi \cdot \sigma'_\Psi))
\end{aligned}$$

Part 3 By induction on Φ. When $\Phi = \Phi,\ \phi_i$ and assuming $\sigma_\Psi.i = [\Phi]\,\Phi'$ we have that the left-hand side is equal to:

$$(\Phi \cdot \sigma_\Psi) \cdot \sigma'_\Psi,\ \Phi' \cdot \sigma'_\Psi = \Phi \cdot (\sigma_\Psi \cdot \sigma'_\Psi),\ \Phi' \cdot \sigma'_\Psi \qquad \text{(by induction hypothesis)}$$

Also, we have that $(\sigma_\Psi \cdot \sigma'_\Psi).i = [\Phi \cdot \sigma'_\Psi]\, \Phi' \cdot \sigma'_\Psi$ [3].

The right-hand-side is also equal to $\Phi \cdot (\sigma_\Psi \cdot \sigma'_\Psi),\ \Phi' \cdot \sigma'_\Psi$.

Rest Similarly as above. □

Theorem 4.2.6 (Extension of Theorem 4.1.13) *(Extension substitution application)*

$$1.\ \frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot \sigma_\Psi : t' \cdot \sigma_\Psi} \qquad 2.\ \frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi}$$

$$3.\ \frac{\Psi \vdash \Phi\ wf \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash \Phi \cdot \sigma_\Psi\ wf} \qquad 4.\ \frac{\Psi \vdash T : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi}$$

$$5.\ \frac{\Psi' \vdash \sigma_\Psi : \Psi \qquad \Psi'' \vdash \sigma'_\Psi : \Psi'}{\Psi'' \vdash \sigma_\Psi \cdot \sigma'_\Psi : \Psi}$$

Proof. We extend our previous proof as follows.

Case VAR**.**

$$\left(\frac{\Phi.L = t}{\Psi;\ \Phi \vdash v_L : t}\right)$$

We have $(\Phi \cdot \sigma_\Psi).(L \cdot \sigma_\Psi) = (\Phi.L) \cdot \sigma_\Psi = t \cdot \sigma_\Psi$ from Lemma 4.2.3. Thus using the same rule for $v_{L \cdot \sigma_\Psi}$ we get the desired result.

Case SUBSTCVAR**.**

$$\left(\frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi.i = [\Phi']\, ctx \qquad \Phi',\ \phi_i \subseteq \Phi}{\Psi;\ \Phi \vdash (\sigma,\ \mathbf{id}(\phi_i)) : (\Phi',\ \phi_i)}\right)$$

In this case we need to prove that $\Psi';\ \Phi \cdot \sigma_\Psi \vdash (\sigma \cdot \sigma_\Psi,\ id_{\sigma_\Psi.i}) : (\Phi' \cdot \sigma_\Psi,\ \sigma_\Psi.i)$.

By induction hypothesis for σ we get that $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi$.

Also, we have that $\Psi' \vdash \sigma_\Psi.i : \Psi.i \cdot \sigma_\Psi$.

Since $\Psi.i = [\Phi']\, ctx$ this can be rewritten as: $\Psi' \vdash \sigma_\Psi.i : [\Phi' \cdot \sigma_\Psi]\, ctx$.

3. Typing will require that $\Phi \cdot \sigma'_\Psi = \Phi \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$

By typing inversion get $\sigma_\Psi.i = [\Phi' \cdot \sigma_\Psi]\,\Phi''$ for some Φ'' and:

$\Psi' \vdash [\Phi' \cdot \sigma_\Psi]\,\Phi'' : [\Phi' \cdot \sigma_\Psi]\,ctx$.

Now proceed by induction on Φ'' to prove that

$\Psi';\ \Phi \cdot \sigma_\Psi \vdash (\sigma \cdot \sigma_\Psi,\ id_{\sigma_\Psi.i}) : (\Phi' \cdot \sigma_\Psi,\ \sigma_\Psi.i)$.

When $\Phi'' = \bullet$, trivial.

When $\Phi'' = \Phi''',\ t$, have $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi,\ id_{[\Phi' \cdot \sigma_\Psi]\,\Phi'''} : (\Phi' \cdot \sigma_\Psi,\ \Phi''')$ by induction hypothesis. We can append $v_{|\Phi' \cdot \sigma_\Psi| \dot{+} |\Phi'''|}$ to this substitution and get the desired, because $(|\Phi' \cdot \sigma_\Psi|, |\Phi'''|) < |\Phi \cdot \sigma_\Psi|$. This is because $(\Phi',\ \phi_i) \subseteq \Phi$ thus $(\Phi' \cdot \sigma_\Psi,\ \Phi''',\ t) \subseteq \Phi$ and $(|\Phi' \cdot \sigma_\Psi| \dot{+} |\Phi'''| \dot{+} 1) \leq |\Phi|$.

When $\Phi'' = \Phi''',\ \phi_j$, have $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi,\ id_{[\Phi' \cdot \sigma_\Psi]\,\Phi'''} : (\Phi' \cdot \sigma_\Psi,\ \Phi''')$. Now we have that $\Phi',\ \phi_i \subseteq \Phi$, which also means that $(\Phi' \cdot \sigma_\Psi,\ \Phi''',\ \phi_j) \subseteq \Phi \cdot \sigma_\Psi$. Thus we can apply the typing rule for $\mathbf{id}(\phi_j)$ to get that $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi,\ id_{[\Phi' \cdot \sigma_\Psi]\,\Phi'''},\ \mathbf{id}(\phi_j) : (\Phi' \cdot \sigma_\Psi,\ \Phi''',\ \phi_j)$, which is the desired.

Case CtxCVar**.**

$$\left(\frac{\Psi \vdash \Phi \text{ wf} \qquad \Psi.i = [\Phi]\,ctx}{\Psi \vdash (\Phi,\ \phi_i) \text{ wf}}\right)$$

By induction hypothesis we get $\Psi' \vdash \Phi \cdot \sigma_\Psi$ wf.

Also, we have that $\Psi' \vdash \sigma_\Psi.i : \Psi.i \cdot \sigma_\Psi$.

Since $\Psi.i = [\Phi]\,ctx$ the above can be rewritten as $\Psi' \vdash \sigma_\Psi.i : [\Phi \cdot \sigma_\Psi]\,ctx$.

By inversion of typing get that $\sigma_\Psi.i = [\Phi \cdot \sigma_\Psi]\,\Phi'$ and that $\Psi' \vdash \Phi \cdot \sigma_\Psi,\ \Phi'$ wf. This is exactly the desired result.

Case ExtCtxInst**.**

$$\left(\frac{\Psi \vdash \Phi,\ \Phi' \text{ wf}}{\Psi \vdash [\Phi]\,\Phi' : [\Phi]\,ctx}\right)$$

By use of part 3 we get $\Psi' \vdash \Phi \cdot \sigma_\Psi,\ \Phi' \cdot \sigma_\Psi$ wf.

Thus by the same typing rule we get exactly the desired.

Case ExtSVar**.**

$$\left(\frac{\Psi' \vdash \sigma_\Psi : \Psi \qquad \Psi' \vdash T : K \cdot \sigma_\Psi}{\Psi' \vdash (\sigma_\Psi,\ T) : (\Psi,\ K)}\right)$$

By induction hypothesis we get $\Psi'' \vdash \sigma_\Psi \cdot \sigma'_\Psi : \Psi$.

By use of part 4 we get $\Psi'' \vdash T \cdot \sigma'_\Psi : (K \cdot \sigma_\Psi) \cdot \sigma'_\Psi$.

This is equal to $K \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$ by use of Lemma 4.2.5. Thus we get the desired result by applying the same typing rule. □

4.3 Constant schemata in signatures

The λHOL logic as presented so far does not allow for polymorphism over kinds. For example, it does not support lists $\mathit{List}\ \alpha : \mathit{Type}$ of an arbitrary $\alpha : \mathit{Type}$ kind, or types for its associated constructor objects. Similarly, in order to define the elimination principle for natural numbers, we need to provide one constant in the signature for each individual return kind that we use, with the following form:

$$\mathit{elimNat}_{\mathcal{K}} : \mathcal{K} \to (\mathit{Nat} \to \mathcal{K} \to \mathcal{K}) \to (\mathit{Nat} \to \mathcal{K})$$

Another similar case is the proof of symmetricity for equality given in Section 3.2, which needs to be parametric over the kind of the involved terms. One way to solve this issue would be to include the PTS rule $(\mathit{Type}', \mathit{Type}, \mathit{Type})$ in the logic; but it is well known that this leads to inconsistency because of Girard's paradox [Girard, 1972, Coquand, 1986, Hurkens, 1995]. This in turn can be solved by supporting a predicative hierarchy of *Type* universes. We will consider a different alternative instead: viewing the signature Σ as a list of constant schemata instead of a (potentially infinite) list of constants. Each schema expects a number of parameters, just as *elimNat* above needs the return kind $\mathcal{K}$ as a parameter; every instantiation of the schema can then be viewed as a different constant. Therefore this change does not affect the consistency of the system, as we can still view the signature of constant schemata as a signature of constants.

This description of constant schemas corresponds exactly to the way that we have defined and used meta-variables above. We can see the context that a meta-variable depends on

Syntax:

$$\begin{array}{rll} (\textit{Signature}) & \Sigma ::= \bullet \mid \Sigma,\ c : [\Phi]\, t \\ (\textit{Logical terms}) & t \ ::= c/\sigma \mid \cdots \end{array}$$

$\boxed{\vdash \Sigma \text{ wf}}$

$$\frac{}{\vdash \bullet \text{ wf}}\ \textsc{SigEmpty} \qquad \frac{\vdash \Sigma \text{ wf} \qquad \bullet;\ \bullet \vdash_\Sigma [\Phi]\, t : [\Phi]\, s \qquad (c : \_) \notin \Sigma}{\vdash \Sigma,\ c : [\Phi]\, t \text{ wf}}\ \textsc{SigConst}$$

$\boxed{\Phi \vdash_\Sigma t : t'}$

the rule Constant *is replaced by:*

$$\frac{c : [\Phi']\, t' \in \Sigma \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash_\Sigma c/\sigma : t' \cdot \sigma}\ \textsc{ConstSchema}$$

Figure 4.12: Extension to λHOL with constant-schema-based signatures: Syntax and typing

as a *context of parameters* and the substitution required when using the meta-variable as the *instantiation of those parameters*. Therefore, we can support constant schematas by changing the signature Σ to be a context of meta-variables instead of a context of variables. We give the new syntax and typing rules for Σ in Figure 4.12. We give an example of using constant schemata for the definition of polymorphic lists in Figure 4.13, where we also provide constants for primitive recursion and induction over lists. We use named variables for presentation purposes. We specify the substitutions used together with the constant schemata explicitly; in the future we will omit these substitutions as they are usually trivial to infer from context. Our version of λHOL supports inductive definitions of kinds and propositions that generate a similar list of constants.

It is trivial to adapt the metatheory lemmas we have proved so far in order account for this change. Intuitively, we can view the variable-based Σ context as a special variable context Φ_Σ that is always prepended to the actual Φ context at hand; the metavariable-based Σ is a similar special metavariable context Ψ_Σ. The adaptation needed is therefore exactly equivalent to moving from the Constant rule that corresponds to the Var rule for using a variable out of a Φ context, to the ConstSchema rule corresponding to the MetaVar rule for using a metavariable out of Ψ. Since we have proved that all our lemmas still hold

$$
\begin{array}{lcl}
\mathit{List} & : & [\alpha : \mathit{Type}]\ \mathit{Type} \\
\mathit{nil} & : & [\alpha : \mathit{Type}]\ \mathit{List}/(\alpha) \\
\mathit{cons} & : & [\alpha : \mathit{Type}]\ \alpha \to \mathit{List}/(\alpha) \to \mathit{List}/(\alpha) \\
\mathit{elimList} & : & [\alpha : \mathit{Type},\ T : \mathit{Type}] \\
 & & \quad T \to (\alpha \to \mathit{List}/(\alpha) \to T \to T) \to (\mathit{List}/(\alpha) \to T) \\
\mathit{elimListNil} & : & [\alpha : \mathit{Type},\ T : \mathit{Type}] \\
 & & \quad \forall f_n : T. \forall f_c : \alpha \to \mathit{List}/\alpha \to T \to T. \\
 & & \quad \mathit{elimList}/(\alpha,\ T)\ f_n\ f_c\ \mathit{nil}/\alpha = f_n \\
\mathit{elimListCons} & : & [\alpha : \mathit{Type},\ T : \mathit{Type}] \\
 & & \quad \forall f_n : T. \forall f_c : \alpha \to \mathit{List}/\alpha \to T \to T. \forall t : \alpha. \forall l : \mathit{List}/\alpha. \\
 & & \quad \mathit{elimList}/(\alpha,\ T)\ f_n\ f_c\ (\mathit{cons}/\alpha\ t\ l) = \\
 & & \quad\ f_c\ t\ l\ (\mathit{elimList}/(\alpha,\ T)\ f_n\ f_c\ l) \\
\mathit{indList} & : & [\alpha : \mathit{Type},\ P : \mathit{List}/\alpha \to \mathit{Type}] \\
 & & \quad P\ \mathit{nil}/\alpha \to (\forall t : \alpha. \forall l : \alpha. P\ l \to P\ (\mathit{cons}/\alpha\ t\ l)) \to \\
 & & \quad \forall l : \mathit{List}/\alpha. P\ l
\end{array}
$$

Figure 4.13: Definition of polymorphic lists in λHOL through constant schemata

in the presence of the MetaVar rule, they will also hold in the presence of ConstSchema. With the formal description of constant schemata in place, we can make the reason why the consistency of the logic is not influenced precise: extension terms and their types, where the extra dependence on parameters is recorded, are not internalized within the logic – they do not become part of the normal logical terms. Thus $[\mathit{Type}]\ \mathit{Type}$ is not a Type, therefore Type is not predicative and Girard's paradox cannot be encoded.

4.4 Named extension variables

We have only considered the case where extension variables X_i and ϕ_i are free. This is enough in order to make use of extension variables in λHOL, as the logic itself does not include constructs that bind these kinds of variables. Our computational language, on the other hand, will need to be able to bind extension variables. We thus need to extend λHOL so that we can use such bound extension variables in further logical terms.

In the interests of avoiding needlessly complicating our presentation, we will use named extension variables instead in our description of the computational language, which follows in Chapter 6. As mentioned earlier in Section 4.1, our original choice of using hybrid deBruijn variables for extension variables is not as integral as it is for normal variables; we

$$
\begin{array}{rll}
(\textit{Logical terms}) & t & ::= X/\sigma \\
(\textit{Parametric deBruijn levels}) & L & ::= 0 \mid L \dot{+} 1 \mid L \dot{+} |\phi| \\
(\textit{Contexts}) & \Phi & ::= \cdots \mid \Phi,\ \phi \\
(\textit{Substitutions}) & \sigma & ::= \cdots \mid \sigma,\ \mathbf{id}(\phi) \\
(\textit{Extension variables}) & V & ::= X \mid \phi \\
(\textit{Extension terms}) & T & ::= [\Phi]\, t \mid [\Phi]\, \Phi' \\
(\textit{Extension types}) & K & ::= [\Phi]\, t \mid [\Phi]\, \mathit{ctx} \\
(\textit{Extension contexts}) & \Psi & ::= \bullet \mid \Psi,\ V : K \\
(\textit{Extension substitutions}) & \sigma_\Psi & ::= \bullet \mid \sigma_\Psi,\ T/V
\end{array}
$$

Figure 4.14: λHOL with named extension variables: Syntax

made it having future extensions in mind (such as meta-N-variables) and in order to be as close as possible to the variable representation we use in our implementation. We will thus not discuss the extension with bound extension variables further, as using named extension variables masks the distinction between free and bound variables. The extension closely follows the definitions and proofs for normal variables and their associated freshening and binding operations, generalizing them to the case where we bind multiple extension variables at once. The interested reader can find the full details of this extension in our published Technical Report [Stampoulis and Shao, 2012b].

We show the new syntax that uses named extension variables in Figure 4.14 and the new typing rules in Figure 4.15. The main change is that we have included variable names in extension contexts and extension substitutions. We use V for extension variables when it is not known whether they are metavariables X or context variables ϕ. Furthermore, instead of using integers to index into contexts and substitutions we use names. The connection with the formulation where extension variables are deBruijn indices is direct. Last, we will sometimes use the standard substitution notation $T'[T/V]$ to denote the operation $T' \cdot (id_\Psi,\ T/V)$, when the exact Ψ context needed is easily inferrable from the context.

$\boxed{\Psi \vdash_\Sigma \Phi \text{ wf}}$

$$\frac{\Psi \vdash \Phi \text{ wf} \qquad \Psi.\phi = [\Phi]\, ctx}{\Psi \vdash (\Phi,\ \phi) \text{ wf}}\ \textsc{CtxCVar}$$

$\boxed{\Psi;\ \Phi \vdash \sigma : \Phi'}$

$$\frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi.\phi = [\Phi']\, ctx \qquad (\Phi',\ \phi) \subseteq \Phi}{\Psi;\ \Phi \vdash (\sigma,\ \mathbf{id}(\phi)) : (\Phi',\ \phi)}\ \textsc{SubstCVar}$$

$\boxed{\Psi;\ \Phi \vdash t : t'}$

$$\frac{\Psi.X = T \qquad T = [\Phi']\, t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X/\sigma : t' \cdot \sigma}\ \textsc{MetaVar}$$

$\boxed{\vdash \Psi \text{ wf}}$

$$\frac{}{\vdash \bullet \text{ wf}}\ \textsc{ExtCEmpty} \qquad \frac{\vdash \Psi \text{ wf} \qquad \Psi \vdash \Phi \text{ wf}}{\vdash (\Psi,\ \phi : [\Phi]\, ctx) \text{ wf}}\ \textsc{ExtCMeta}$$

$$\frac{\vdash \Psi \text{ wf} \qquad \Psi \vdash [\Phi]\, t : [\Phi]\, s}{\vdash (\Psi,\ X : [\Phi]\, t) \text{ wf}}\ \textsc{ExtCCtx}$$

$\boxed{\Psi \vdash \sigma_\Psi : \Psi'}$

$$\frac{}{\Psi \vdash \bullet : \bullet}\ \textsc{ExtSEmpty} \qquad \frac{\Psi \vdash \sigma_\Psi : \Psi' \qquad \Psi \vdash T : K \cdot \sigma_\Psi}{\Psi \vdash (\sigma_\Psi,\ T/V) : (\Psi',\ V : K)}\ \textsc{ExtSVar}$$

Figure 4.15: λHOL with named extension variables: Typing

Chapter 5

The logic λHOL: Pattern matching

Having presented λHOL with extension variables, we are ready to proceed to the main operation through which λHOL terms are manipulated in VeriML: pattern matching. We will first describe the procedure and metatheoretic proofs for matching a scrutinee against a pattern in Section 5.1. In Section 5.2, we will present an approach that enables us to use the same pattern matching procedure when the scrutinee is not closed with respect to extension variables.

5.1 Pattern matching

In Section 2.3 we noted the central importance that pattern matching constructs have in VeriML. We presented two constructs: one for matching over contextual terms and one for matching over contexts. Having seen the details of extension terms in the previous chapter, it is now clear that these two constructs are in fact a single construct for pattern matching over extension terms T. In this section we will give the details of what *patterns* are, how unification of a pattern against a term works and prove the main metatheoretic result about this procedure.

Informally we can say that a pattern is the 'skeleton' of a term, where some parts are specified and some parts are missing. We name each missing subterm using *unification variables*. An example is the following pattern for a proposition, where we denote unification variables by prefixing them with a question mark:

$$?P \wedge (\forall x : Nat.?Q)$$

A pattern *matches* a term when we can find a substitution for the missing parts so that the pattern is rendered equal to the given term. For example, the above pattern matches the term $(\forall x : Nat.x + x \geq x) \wedge (\forall x : Nat.x \geq 0)$ using the substitution:

$$?P \mapsto (\forall x : Nat.x + x \geq x),\ ?Q \mapsto x \geq 0$$

As the above example suggests, unification variables can be used in patterns under the binding constructs of the λHOL logic. Because of this, if we view unification variables as typed variables, their type includes the information about the variables context of the missing subterm, in addition to its type. It is therefore evident that unification variables are a special kind of meta-variables; and that we can view the substitution returned from pattern matching as being composed from contextual terms to be substituted for these (unification) meta-variables. The situation for matching contexts against patterns that include context unification variables is similar. Therefore we can view patterns as a special kind of extension terms where the extension variables used are viewed as unification variables. We will place further restrictions on what terms are allowed as patterns, resulting in a new typing judgement that we will denote as $\Psi \vdash_p T : K$. Based on these, we can formulate pattern matching for λHOL as follows. Assuming a pattern T_P and an extension term T (the *scrutinee*) with the following typing:

$$\Psi_u \vdash_p T_P : K \qquad \text{and} \qquad \bullet \vdash T : K$$

where the extension context Ψ_u describes the unification variables used in T_P, pattern matching is a procedure which decides whether a substitution σ_Ψ exists so that:

$$\bullet \vdash \sigma_\Psi : \Psi_u \qquad \text{and} \qquad T_P \cdot \sigma_\Psi = T$$

We will proceed as follows.

1. First, we will define the typing judgement for patterns $\Psi \vdash_p T : K$. This will be mostly identical to the normal typing for terms, save for certain restrictions so that pattern matching is decidable and deterministic.

2. Then, we will define an operation to extract the *relevant variables* out of typing derivations; it will work by isolating the *partial context* that gets used in a given derivation.

This is useful for two reasons: first, to ensure that all the defined unification variables get used in a pattern; otherwise, pattern matching would not be deterministic as unused variables could be instantiated with any arbitrary term. Second, isolating the partial context is crucial in stating the induction principle needed for the pattern matching theorem that we will prove.

3. Using this operation, we will define an even more restrictive typing judgement for patterns, which requires all unification variables defined to be relevant, ruling out in this way variables that are not used. A sketch of its formal definition is as follows:

$$\frac{\Psi,\ \Psi_u \vdash_p T_P : K \qquad \mathit{relevant}\,(\Psi,\ \Psi_u \vdash_p T_P : K) = \cdots, \Psi_u}{\Psi \Vdash_p^* \Psi_u > T_P : K}$$

Let us explain this rule briefly. First, Ψ represents the normal extension variables whereas Ψ_u represents the newly-defined unification variables. The judgement $\Psi \Vdash_p^* \Psi_u > T_P : K$ is understood as: "in extension context Ψ, the introduction of the unification variables Ψ_u renders T_P a valid pattern of extension type K". The second premise of this rule uses the operation that extracts relevant variables out of a derivation in order to make sure that all unification variables are relevant. We will prove a substitution theorem for this rule, with the following formal statement:

$$\frac{\Psi \Vdash_p^* \Psi_u > T_P : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \Vdash_p^* \Psi_u \cdot \sigma_\Psi > T_P \cdot \sigma_\Psi : K \cdot \sigma_\Psi}\ \textbf{Substitution lemma}$$

This theorem extends the extension substitution theorem 4.2.6 so as to apply to the case of patterns as well.

4. Based on these, we will prove the fact that pattern matching is decidable and deterministic. We will carry out the proof in a constructive manner; its computational counterpart will be the pattern matching algorithm that we will use. Therefore the pattern matching algorithm will be sound and complete directly by construction. The algorithm thus produces a substitution σ_Ψ of instantiations to the unification variables making the pattern and the scrutinee match, when one such substitution exists; or fails, when there is no such substitution. We write $T_P \sim T = \sigma_\Psi$ for an execution

of this algorithm between the pattern T_P and the scrutinee T, when it results in a substitution σ_Ψ. Formally, the statements of the related theorems we will prove are as follows.

$$\frac{\bullet \mid\!\!\overset{*}{\underset{p}{}}\; \Psi_u > T_P : K \qquad \bullet \vdash T : K}{\exists^{\leq 1}\sigma_\Psi.(\quad \bullet \vdash \sigma_\Psi : \Psi_u \qquad T_P \cdot \sigma_\Psi = T \quad)} \quad \textbf{Pattern matching determinism and decidability}$$

$$\frac{\bullet \mid\!\!\overset{*}{\underset{p}{}}\; \Psi_u > T_P : K \qquad \bullet \vdash T : K \qquad T_P \sim T = \sigma_\Psi}{\bullet \vdash \sigma_\Psi : \Psi_u \qquad T_P \cdot \sigma_\Psi = T} \quad \textbf{Pattern matching algorithm soundness}$$

$$\frac{\bullet \mid\!\!\overset{*}{\underset{p}{}}\; \Psi_u > T_P : K \qquad \bullet \vdash T : K \qquad \exists\sigma_\Psi.(\quad \bullet \vdash \sigma_\Psi : \Psi_u \qquad T_P \cdot \sigma_\Psi = T \quad)}{T_P \sim T = \sigma_\Psi} \quad \textbf{Pattern matching algorithm completeness}$$

Pattern typing

In Figures 5.1 and 5.2 we give the details of pattern typing. The typing judgements use the extension context Ψ, Ψ_u, composed from the normal extension variable context Ψ and the context of unification variables Ψ_u. We will define the "actual" pattern matching process only between a 'closed' pattern and a closed extension term, so the case will be that $\Psi = \bullet$, as presented above. Yet being able to mention existing variables from an Ψ context will be useful in order to describe patterns that exactly match a yet-unspecified term. It is thus understood that even if a pattern depends on extension variables in a non-empty Ψ context, those will have been substituted by concrete terms by the time that the "actual" pattern matching happens.

The two kinds of variables are handled differently in the pattern typing rules themselves. For example, the rule for using an exact context variable PCtxCVarExact is different than the rule for using a unification context variable PCtxCVarUnify. The difference is subtle: the latter rule does not allow existing unification variables to inform the well-formedness of the Φ context; in other words, a unification context variable cannot be used in a pattern after another unification context variable has already been used. Consider the pattern

$\boxed{\Psi \vdash_{\overline{p}} \Psi_u \text{ wf}}$

$$\frac{}{\Psi \vdash_{\overline{p}} \bullet \text{ wf}}\ \textsc{UVarEmpty} \qquad \frac{\Psi \vdash_{\overline{p}} \Psi_u \text{ wf} \qquad \Psi,\ \Psi_u \vdash_{\overline{p}} [\Phi]\, t : [\Phi]\, s}{\Psi \vdash_{\overline{p}} (\Psi_u,\ [\Phi]\, t) \text{ wf}}\ \textsc{UVarMeta}$$

$$\frac{\Psi \vdash_{\overline{p}} \Psi_u \text{ wf} \qquad \Psi,\ \Psi_u \vdash_{\overline{p}} \Phi \text{ wf}}{\Psi \vdash_{\overline{p}} (\Psi_u,\ [\Phi]\, ctx) \text{ wf}}\ \textsc{UVarCtx}$$

$\boxed{\Psi,\ \Psi_u \vdash_{\overline{p}} T : K}$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t : t' \qquad \Psi,\ \Psi_u;\ \Phi \vdash t' : s}{\Psi,\ \Psi_u \vdash_{\overline{p}} [\Phi]\, t : [\Phi]\, t'}\ \textsc{PExtCtxTerm}$$

$$\frac{\Psi,\ \Psi_u \vdash_{\overline{p}} \Phi,\ \Phi' \text{ wf}}{\Psi,\ \Psi_u \vdash_{\overline{p}} [\Phi]\, \Phi' : [\Phi]\, ctx}\ \textsc{PExtCtxInst}$$

$\boxed{\Psi,\ \Psi_u \vdash_{\overline{p}} \Phi \text{ wf}}$

$$\frac{}{\Psi,\ \Psi_u \vdash_{\overline{p}} \bullet \text{ wf}}\ \textsc{PCtxExpty} \qquad \frac{\Psi,\ \Psi_u \vdash_{\overline{p}} \Phi \text{ wf} \qquad \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t : s}{\Psi,\ \Psi_u \vdash_{\overline{p}} (\Phi,\ t) \text{ wf}}\ \textsc{PCtxVar}$$

$$\frac{\Psi,\ \Psi_u \vdash_{\overline{p}} \Phi \text{ wf} \qquad i < |\Psi| \qquad (\Psi,\ \Psi_u).i = [\Phi]\, ctx}{\Psi,\ \Psi_u \vdash_{\overline{p}} (\Phi,\ \phi_i) \text{ wf}}\ \textsc{PCtxCVarExact}$$

$$\frac{\Psi \vdash_{\overline{p}} \Phi \text{ wf} \qquad i \geq |\Psi| \qquad (\Psi,\ \Psi_u).i = [\Phi]\, ctx}{\Psi,\ \Psi_u \vdash_{\overline{p}} (\Phi,\ \phi_i) \text{ wf}}\ \textsc{PCtxCVarUnif}$$

$\boxed{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \sigma : \Phi'}$

$$\frac{}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \bullet : \bullet}\ \textsc{PSubstEmpty}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \sigma : \Phi' \qquad \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t : t' \cdot \sigma}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} (\sigma,\ t) : (\Phi',\ t')}\ \textsc{PSubstVar}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \sigma : \Phi' \qquad (\Psi,\ \Psi_u).i = [\Phi']\, ctx \qquad \Phi',\ \phi_i \subseteq \Phi}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} (\sigma,\ \mathbf{id}(\phi_i)) : (\Phi',\ \phi_i)}\ \textsc{PSubstCVar}$$

Figure 5.1: Pattern typing for λHOL

$\boxed{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t : t'}$

$$\frac{c : t \in \Sigma \qquad \bullet;\ \bullet \vdash t : s \qquad s \neq \mathit{Prop}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} c : t}\ \textsc{PConstant}$$

$$\frac{\Phi.L = t \qquad \Psi,\ \Psi_u;\ \Phi \vdash t : s \qquad s \neq \mathit{Prop}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} v_L : t}\ \textsc{PVar} \qquad \frac{(s, s') \in \mathcal{A}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} s : s'}\ \textsc{PSort}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s \qquad \Psi,\ \Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_1).t_2 : s''}\ \textsc{P}\Pi\textsc{Type}$$

$$\frac{\begin{array}{c}\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s \\ \Psi,\ \Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|\dot{+}1} : s' \qquad s' \neq \mathit{Prop}\end{array}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|\dot{+}1}}\ \textsc{P}\Pi\textsc{Intro}$$

$$\frac{\begin{array}{c}\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : \Pi(t).t' \\ \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t \qquad \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t).t' : s' \qquad s' \neq \mathit{Prop}\end{array}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (id_\Phi, t_2)}\ \textsc{P}\Pi\textsc{Elim}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : t \qquad \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t \qquad \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t : \mathit{Type}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 = t_2 : \mathit{Prop}}\ \textsc{PEqType}$$

$$\frac{\begin{array}{c}(\Psi,\ \Psi_u).i = T \qquad T = [\Phi']\,t' \\ i < |\Psi| \qquad \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \sigma : \Phi' \qquad \Psi,\ \Psi_u \vdash t' : s' \qquad s \neq \mathit{Prop}\end{array}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} X_i/\sigma : t' \cdot \sigma}\ \textsc{PMetaVarExact}$$

$$\frac{\begin{array}{c}(\Psi,\ \Psi_u).i = T \qquad T = [\Phi']\,t' \qquad i \geq |\Psi| \qquad \Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} \sigma : \Phi' \\ \Phi' \subseteq \Phi \qquad \sigma = id_{\Phi'} \qquad \Psi,\ \Psi_u \vdash t' : s' \qquad s \neq \mathit{Prop}\end{array}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} X_i/\sigma : t' \cdot \sigma}\ \textsc{PMetaVarUnif}$$

Figure 5.2: Pattern typing for λHOL (continued)

ϕ_0, ϕ_1, where $\phi_1 : [\phi_0]\ ctx$. Any non-empty context could be matched against this pattern, yet the exact point of the split between which variables belong to the ϕ_0 context and which to ϕ_1 is completely arbitrary. This would destroy determinacy of pattern matching thus this form is disallowed using this rule.

Metavariables typing in patterns is similarly governed by two rules: PMetaVarExact and PMetaVarUnif. The former is the standard typing rule for metavariables save for a sort restriction which we will comment on shortly. The latter is the rule for using a unification variable. The main restriction that it has is that the substitution σ used must be the identity substitution – or more accurately, the identity substitution for a prefix of the current context. If arbitrary substitutions were allowed, matching a pattern X_i/σ against a term t would require finding an inverse substitution σ^{-1} and using $t \cdot \sigma^{-1}$ as the instantiation for X_i, so that $t \cdot \sigma^{-1} \cdot \sigma = t$. This destroys determinacy of pattern matching, as multiple inverse substitutions might be possible; but it also destroys decidability if further unification variables are allowed inside σ, as the problem becomes equivalent to higher-order unification, which is undecidable [Dowek, 2001]. Prefixes of the identity substitution for the current context can be used to match against scrutinees that only mention specific variables. For example we can use a pattern like $X_i/\bullet$ in order to match against closed terms.

Last, many pattern typing rules impose a sort restriction. The reason is that we want to disallow pattern matching against proof objects, so as to render the exact structure of proof objects computationally irrelevant. We cannot thus look inside proof objects: we are only interested in the existence of a proof object and not in its specific details. This will enable us to *erase* proof objects prior to evaluation of VeriML programs. By inspection of the rules it is evident that *Prop*-sorted terms are not well-typed as patterns; these terms are exactly the proof objects. The only *Prop*-sorted pattern that is allowed is the unification variable case, which we can understand as a wildcard rule for matching any proof object of a specific proposition.

We will need some metatheoretic facts about pattern typing: first, the fact that pattern typing implies normal typing; and second, the equivalent of the extension substitution theorem 4.2.6 for patterns. We will first need two definitions that lift identity substitutions and extension substitution application to unification contexts Ψ_u typed in a context Ψ,

Assuming $\Psi \vdash \Psi_u$ wf:

$\boxed{T = V_i}$

$$
\begin{array}{lcl}
X_i & = & [\Phi]\, X_i/id_\Phi \text{ when } (\Psi,\ \Psi_u).i = [\Phi]\, t' \\
\phi_i & = & [\Phi]\, \phi_i \text{ when } (\Psi,\ \Psi_u).i = [\Phi]\, ctx
\end{array}
$$

$\boxed{id_{\Psi_u}}$

$$
\begin{array}{lcl}
id_\bullet & = & \bullet \\
id_{\Psi'_u,\, K} & = & id_{\Psi'_u},\ V_{|\Psi|+|\Psi'_u|}
\end{array}
$$

$\boxed{\Psi_u \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
\bullet \cdot \sigma_\Psi & = & \bullet \\
(\Psi'_u,\ K) \cdot \sigma_\Psi & = & \Psi'_u \cdot \sigma_\Psi,\ K \cdot (\sigma_\Psi,\ id_{\Psi'_u})
\end{array}
$$

Figure 5.3: Syntactic operations for unification contexts

given in Figure 5.3. In the same figure we give an operation to include extension variables V_i as extension terms, by expanding them to the existing forms such as $[\Phi]\, V_i/id_\Phi$ and $[\Phi]\, V_i$. We remind the reader that V_i stands for either kind of an extension variable, that is, either for a meta-variable X_i or for a context variable ϕ_i.

Lemma 5.1.1 *(Pattern typing implies normal typing)*

$$1.\ \frac{\Psi \models_p \Psi_u\ wf}{\vdash \Psi,\ \Psi_u\ wf} \qquad 2.\ \frac{\Psi,\ \Psi_u \models_p T : K}{\Psi,\ \Psi_u \vdash T : K} \qquad 3.\ \frac{\Psi,\ \Psi_u \models_p \Phi\ wf}{\Psi,\ \Psi_u \vdash \Phi\ wf}$$

$$4.\ \frac{\Psi,\ \Psi_u \models_p \sigma : \Phi}{\Psi,\ \Psi_u \vdash \sigma : \Phi} \qquad 5.\ \frac{\Psi,\ \Psi_u;\ \Phi \models_p t : t'}{\Psi,\ \Psi_u;\ \Phi \vdash t : t'}$$

Proof. Trivial by induction on the typing derivations, as every pattern typing rule is a restriction of an existing normal typing rule. □

Theorem 5.1.2 *(Extension substitution application preserves pattern typing)*

Assuming $\Psi' \vdash \sigma_\Psi : \Psi$ *and* $\sigma'_\Psi = \sigma_\Psi,\ id_{\Psi_u}$,

$$1.\ \frac{\Psi,\ \Psi_u;\ \Phi \mathrel{\overline{\vdash}_p} t : t'}{\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \mathrel{\overline{\vdash}_p} t \cdot \sigma'_\Psi : t' \cdot \sigma'_\Psi} \qquad 2.\ \frac{\Psi,\ \Psi_u;\ \Phi \mathrel{\overline{\vdash}_p} \sigma : \Phi'}{\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \mathrel{\overline{\vdash}_p} \sigma \cdot \sigma'_\Psi : \Phi' \cdot \sigma'_\Psi}$$

$$3.\ \frac{\Psi,\ \Psi_u \mathrel{\overline{\vdash}_p} \Phi\ wf}{\Psi',\ \Psi_u \cdot \sigma_\Psi \mathrel{\overline{\vdash}_p} \Phi \cdot \sigma'_\Psi\ wf} \qquad 4.\ \frac{\Psi,\ \Psi_u \mathrel{\overline{\vdash}_p} T : K}{\Psi',\ \Psi_u \cdot \sigma_\Psi \mathrel{\overline{\vdash}_p} T \cdot \sigma'_\Psi : K \cdot \sigma'_\Psi} \qquad 5.\ \frac{\Psi \mathrel{\overline{\vdash}_p} \Psi_u\ wf}{\Psi \mathrel{\overline{\vdash}_p} \Psi_u \cdot \sigma_\Psi\ wf}$$

Proof. In most cases proceed similarly as before. The cases that deserve special mention are the ones that place extra restrictions compared to normal typing.

Case PCtxCVarUnif**.**

$$\left(\frac{\Psi \mathrel{\overline{\vdash}_p} \Phi \text{ wf} \qquad i \geq |\Psi| \qquad (\Psi,\ \Psi_u).i = [\Phi]\ ctx}{\Psi,\ \Psi_u \mathrel{\overline{\vdash}_p} (\Phi,\ \phi_i) \text{ wf}}\right)$$

We need to prove that $\Psi',\ \Psi_u \cdot \sigma_\Psi \mathrel{\overline{\vdash}_p} \Phi \cdot \sigma'_\Psi,\ \phi_i \cdot \sigma'_\Psi$ wf.

We have that $\phi_i \cdot \sigma'_\Psi = \phi_{i-|\Psi|+|\Psi'|}$.

By induction hypothesis for Φ, with $\Psi_u = \bullet$, we get:

$\Psi \mathrel{\overline{\vdash}_p} \Phi \cdot \sigma_\Psi$ wf from which $\Psi \mathrel{\overline{\vdash}_p} \Phi \cdot \sigma'_\Psi$ wf directly follows.

We have $i - |\Psi| + |\Psi'| \geq |\Psi'|$ because of $i \geq |\Psi|$.

Last, we have that $(\Psi',\ \Psi_u \cdot \sigma_\Psi).(i - |\Psi| + |\Psi'|) = [\Phi \cdot \sigma'_\Psi]\ ctx$.

Thus using the same rule PCtxCVarUnif we arrive at the desired.

Case PMetaVarUnif**.**

$$\left(\frac{(\Psi,\ \Psi_u).i = T \qquad T = [\Phi']\, t' \qquad i \geq |\Psi| \qquad \Psi,\ \Psi_u;\ \Phi \mathrel{\overline{\vdash}_p} \sigma : \Phi' \qquad \Phi' \subseteq \Phi \qquad \sigma = id_{\Phi'}}{\Psi,\ \Psi_u;\ \Phi \mathrel{\overline{\vdash}_p} X_i/\sigma : t' \cdot \sigma}\right)$$

Similar as above. Furthermore, we need to show that $id_{\Phi'} \cdot \sigma'_\Psi = id_{\Phi' \cdot \sigma'_\Psi}$, which follows trivially by induction on Φ'.

Case PVar**.**

$$\left(\frac{\Phi.L = t \qquad \Psi,\ \Psi_u;\ \Phi \vdash t : s \qquad s \neq \mathit{Prop}}{\Psi,\ \Psi_u;\ \Phi \vdash_{\overline{p}} v_L : t}\right)$$

We give the proof for this rule as an example of accounting for the sort restriction; other rules are proved similarly. By induction hypothesis for t we get:

$\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \vdash t \cdot \sigma'_\Psi : s \cdot \sigma'_\Psi$

But $s \cdot \sigma'_\Psi = s$ so the restriction $s \neq \mathit{Prop}$ still holds. □

The statement of the above lemma might be surprising at first. One could expect it to hold for the general case where $\Psi' \vdash \sigma_\Psi : (\Psi,\ \Psi_u)$. This stronger statement is not true however, if we take into account the restrictions we have placed: for example, substituting a unification variable for a proof object will result in a term that is not allowed as a pattern. The statement we have proved is in fact all we need, as we will use it only in cases where we are substituting the base Ψ context; the unification variables will only be substituted by new unification variables that are well-typed in the resulting Ψ' context. This becomes clearer if we consider a first sketch of the pattern typing rule used in the computational language:

$$\frac{\Psi \vdash T : K \qquad \Psi \vdash_{\overline{p}} \Psi_u\ \mathrm{wf} \qquad \Psi,\ \Psi_u \vdash_{\overline{p}} T_P : K \qquad \cdots}{\Psi \cdots \vdash \mathsf{match}\ T\ \mathsf{with}\ T_P \mapsto \cdots}$$

Type-safety of the computational language critically depends on the fact that applying a substitution σ_Ψ for the current Ψ context yields expressions that are typable using the same typing rule. This fact is exactly what can be proved using the lemma as stated.

Relevant typing

We will now proceed to define the notion of *relevant variables* and partial contexts. We say that a variable is *relevant* for a term if it is used either directly in the term or indirectly in its type. A partial context $\widehat{\Psi}$ is a context where only the types for the relevant variables are specified; the others are left unspecified, denoted as ?. We will define an operation $\mathit{relevant}\,(\Psi;\ \cdots)$ for derivations that isolates the partial context required for the relevant variables of the judgement. For example, considering the derivation resulting in:

$$\phi_0 : []\ \mathit{ctx},\ X_1 : [\phi_0]\ \mathit{Type},\ X_2 : [\phi_0]\,X_1,\ X_3 : [\phi_0]\,X_1 \vdash X_3 : [\phi_0]\,X_1$$

the resulting partial context will be:

$$\widehat{\Psi} = \phi_0 : [] \; ctx,\ X_1 : [\phi_0] \; Type,\ X_2 : ?,\ X_3 : [\phi_0] \, X_1$$

where the variable X_2 was unused in the derivation and is therefore left unspecified in the resulting partial context.

The typing rule for patterns in the computational language will require that all unification variables are relevant in order to ensure determinacy as noted above. Non-relevant variables can be substituted by arbitrary terms, thus pattern-matching would have an infinite number of valid solutions if we allowed them. Thus a refined sketch of the pattern match typing rule is:

$$\frac{\Psi \vdash T : K \qquad \Psi \models_{\overline{p}} \Psi_u \text{ wf} \qquad \Psi,\ \Psi_u \models_{\overline{p}} T_P : K \qquad ?, ?, \cdots, ?, \Psi_u \sqsubseteq \mathit{relevant}\,(\Psi,\ \Psi_u \models_{\overline{p}} T_P : K) \qquad \cdots}{\Psi \cdots \vdash \mathsf{match}\ T\ \mathsf{with}\ T_P \mapsto \cdots}$$

In this, the symbol $\sqsubseteq$ is used to mean that one partial context is less-specified than another one. Therefore, in the new restriction placed by the typing rule, we require that all unification variables are used in a pattern but normal extension variables can either be used or unused. Because of this extra restriction, we will need to prove a lemma regarding the interaction of relevancy with extension substitution application, so that the extension substitution lemma needed for the computational language will still hold.

We present the syntax and typing for partial contexts in Figure 5.4; these are entirely identical to normal Ψ contexts save for the addition of the case of an unspecified element. We present several operations involving partial contexts in Figure 5.5; we use these to define the operation of isolating the relevant variables of a judgement in Figures 5.6 and 5.7. The rules are mostly straightforward. For example, in the case of the rule MetaVar for using metavariables, the relevant variables are: the metavariable itself (by isolating just that variable from the context Ψ through the $\Psi \widehat{@} i$ operation); the relevant variables used for the substitution σ; and the variables that are relevant for well-typedness of the type of the metavariable $[\Phi'] \, t'$. The partial contexts so constructed are joined together through the $\Psi \circ \Psi'$ operation in order to get the overall result. This is equivalent to taking the union of the set of relevant variables in informal practice.

$$\widehat{\Psi} ::= \bullet \mid \widehat{\Psi},\ K \mid \widehat{\Psi},\ ?$$

$$\frac{}{\vdash \bullet \text{ wf}} \qquad \frac{\vdash \widehat{\Psi} \text{ wf} \qquad \widehat{\Psi} \vdash [\Phi]\, t : [\Phi]\, s}{\vdash (\widehat{\Psi},\ [\Phi]\, t) \text{ wf}} \qquad \frac{\vdash \widehat{\Psi} \text{ wf} \qquad \widehat{\Psi} \vdash \Phi \text{ wf}}{\vdash (\widehat{\Psi},\ [\Phi]\, ctx) \text{ wf}} \qquad \frac{\vdash \widehat{\Psi} \text{ wf}}{\vdash (\widehat{\Psi},\ ?) \text{ wf}}$$

Figure 5.4: Partial contexts (syntax and typing)

Fully unspecified context:
$unspec_{\Psi} = \widehat{\Psi}$

$$\begin{array}{lcl} unspec_{\bullet} & = & \bullet \\ unspec_{\Psi,\, K} & = & unspec_{\Psi},\ ? \end{array}$$

Partial context solely specified at i:
$\Psi \widehat{@} i = \widehat{\Psi}$

$$\begin{array}{lcl} (\Psi,\ K)\widehat{@}i & = & unspec_{\Psi},\ K \text{ when } |\Psi| = i \\ (\Psi,\ K)\widehat{@}i & = & (\Psi \widehat{@} i),\ ? \text{ when } |\Psi| > i \end{array}$$

Joining two partial contexts:
$\widehat{\Psi} \circ \widehat{\Psi}' = \widehat{\Psi}''$

$$\begin{array}{lcl} \bullet \circ \bullet & = & \bullet \\ (\widehat{\Psi},\ K) \circ (\widehat{\Psi}',\ K) & = & (\widehat{\Psi} \circ \widehat{\Psi}'),\ K \\ (\widehat{\Psi},\ ?) \circ (\widehat{\Psi}',\ K) & = & (\widehat{\Psi} \circ \widehat{\Psi}'),\ K \\ (\widehat{\Psi},\ K) \circ (\widehat{\Psi}',\ ?) & = & (\widehat{\Psi} \circ \widehat{\Psi}'),\ K \\ (\widehat{\Psi},\ ?) \circ (\widehat{\Psi}',\ ?) & = & (\widehat{\Psi} \circ \widehat{\Psi}'),\ ? \end{array}$$

Context $\widehat{\Psi}$ less specified than $\widehat{\Psi}'$:
$\widehat{\Psi} \sqsubseteq \widehat{\Psi}'$

$$\begin{array}{l} \bullet \sqsubseteq \bullet \\ (\widehat{\Psi},\ K) \sqsubseteq (\widehat{\Psi}',\ K) \quad \Leftarrow \quad \widehat{\Psi} \sqsubseteq \widehat{\Psi}' \\ (\widehat{\Psi},\ ?) \sqsubseteq (\widehat{\Psi}',\ K) \quad \Leftarrow \quad \widehat{\Psi} \sqsubseteq \widehat{\Psi}' \\ (\widehat{\Psi},\ ?) \sqsubseteq (\widehat{\Psi}',\ ?) \quad \Leftarrow \quad \widehat{\Psi} \sqsubseteq \widehat{\Psi}' \end{array}$$

Extension substitution application:
(assuming $\Psi \vdash \widehat{\Psi}_u$ wf)
$\widehat{\Psi} \cdot \sigma_{\Psi}$

$$\begin{array}{lcl} \bullet \cdot \sigma_{\Psi} & = & \bullet \\ (\widehat{\Psi}_u,\ K) \cdot \sigma_{\Psi} & = & \widehat{\Psi}_u \cdot \sigma_{\Psi},\ K \cdot (\sigma_{\Psi},\ id_{\widehat{\Psi}_u}) \\ (\widehat{\Psi}_u,\ ?) \cdot \sigma_{\Psi} & = & \widehat{\Psi}_u \cdot \sigma_{\Psi},\ ? \end{array}$$

Figure 5.5: Partial contexts (syntactic operations)

$$\boxed{\mathit{relevant}\,(\Psi \vdash T : K) = \widehat{\Psi}}$$

$$\mathit{relevant}\left(\frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi;\ \Phi \vdash t' : s}{\Psi \vdash [\Phi]\,t : [\Phi]\,t'}\right) = \mathit{relevant}\,(\Psi;\ \Phi \vdash t : t')$$

$$\mathit{relevant}\left(\frac{\Psi \vdash \Phi,\ \Phi' \text{ wf}}{\Psi \vdash [\Phi]\,\Phi' : [\Phi]\,\mathit{ctx}}\right) = \mathit{relevant}\,(\Psi \vdash \Phi,\ \Phi' \text{ wf})$$

$$\boxed{\mathit{relevant}\,(\Psi \vdash \Phi \text{ wf}) = \widehat{\Psi}}$$

$$\mathit{relevant}\left(\frac{}{\Psi \vdash \bullet \text{ wf}}\right) = \mathit{unspec}_{\Psi}$$

$$\mathit{relevant}\left(\frac{\Psi \vdash \Phi \text{ wf} \qquad \Psi;\ \Phi \vdash t : s}{\Psi \vdash (\Phi,\ t) \text{ wf}}\right) = \mathit{relevant}\,(\Psi;\ \Phi \vdash t : s)$$

$$\mathit{relevant}\left(\frac{\Psi \vdash \Phi \text{ wf} \qquad (\Psi).i = [\Phi]\,\mathit{ctx}}{\Psi \vdash (\Phi,\ \phi_i) \text{ wf}}\right) = \mathit{relevant}\,(\Psi \vdash \Phi \text{ wf}) \circ (\Psi \widehat{@} i)$$

$$\boxed{\mathit{relevant}\,(\Psi;\ \Phi \vdash \sigma : \Phi') = \widehat{\Psi}}$$

$$\mathit{relevant}\left(\frac{}{\Psi;\ \Phi \vdash \bullet : \bullet}\right) = \mathit{relevant}\,(\Psi \vdash \Phi \text{ wf})$$

$$\mathit{relevant}\left(\frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi;\ \Phi \vdash t : t' \cdot \sigma}{\Psi;\ \Phi \vdash (\sigma,\ t) : (\Phi',\ t')}\right) =$$

$$= \mathit{relevant}\,(\Psi;\ \Phi \vdash \sigma : \Phi') \circ \mathit{relevant}\,(\Psi;\ \Phi \vdash t : t' \cdot \sigma)$$

$$\mathit{relevant}\left(\frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi.i = [\Phi']\,\mathit{ctx} \qquad \Phi',\ \phi_i \subseteq \Phi}{\Psi;\ \Phi \vdash (\sigma,\ \mathbf{id}(\phi_i)) : (\Phi',\ \phi_i)}\right) = \mathit{relevant}\,(\Psi;\ \Phi \vdash \sigma : \Phi')$$

Figure 5.6: Relevant variables of λHOL derivations

$$\boxed{relevant\,(\Psi;\ \Phi \vdash t : t') = \widehat{\Psi}}$$

$$relevant\left(\frac{c : t \in \Sigma}{\Psi;\ \Phi \vdash c : t}\right) = relevant\,(\Psi \vdash \Phi \text{ wf})$$

$$relevant\left(\frac{\Phi.L = t}{\Psi;\ \Phi \vdash v_L : t}\right) = relevant\,(\Psi \vdash \Phi \text{ wf})$$

$$relevant\left(\frac{(s, s') \in \mathcal{A}}{\Psi;\ \Phi \vdash s : s'}\right) = relevant\,(\Psi \vdash \Phi \text{ wf})$$

$$relevant\left(\frac{\Psi;\ \Phi \vdash t_1 : s \qquad \Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Psi;\ \Phi \vdash \Pi(t_1).t_2 : s''}\right) =$$

$$= relevant\left(\Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s'\right)$$

$$relevant\left(\frac{\Psi;\ \Phi \vdash t_1 : s \qquad \Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Psi;\ \Phi \vdash \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1} : s'}{\Psi;\ \Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}}\right) =$$

$$= relevant\left(\Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t'\right)$$

$$relevant\left(\frac{\Psi;\ \Phi \vdash t_1 : \Pi(t).t' \qquad \Psi;\ \Phi \vdash t_2 : t}{\Psi;\ \Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (id_\Phi, t_2)}\right) =$$

$$= relevant\,(\Psi;\ \Phi \vdash t_1 : \Pi(t).t') \circ relevant\,(\Psi;\ \Phi \vdash t_2 : t)$$

$$relevant\left(\frac{\Psi;\ \Phi \vdash t_1 : t \qquad \Psi;\ \Phi \vdash t_2 : t \qquad \Psi;\ \Phi \vdash t : Type}{\Psi;\ \Phi \vdash t_1 = t_2 : Prop}\right) =$$

$$= relevant\,(\Psi;\ \Phi \vdash t_1 : t) \circ relevant\,(\Psi;\ \Phi \vdash t_2 : t)$$

$$relevant\left(\frac{(\Psi).i = T \qquad T = [\Phi']\,t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\right) =$$

$$= relevant\,(\Psi \vdash [\Phi']\,t' : [\Phi']\,s) \circ relevant\,(\Psi;\ \Phi \vdash \sigma : \Phi') \circ (\Psi \widehat{@} i)$$

Figure 5.7: Relevant variables of λHOL derivations (continued)

We are now ready to proceed to the metatheoretic proofs about isolating relevant variables.

Lemma 5.1.3 *(More specified contexts preserve judgements)*
Assuming $\widehat{\Psi} \sqsubseteq \widehat{\Psi}'$*:*

$$1.\ \frac{\widehat{\Psi} \vdash T : K}{\widehat{\Psi}' \vdash T : K} \qquad 2.\ \frac{\widehat{\Psi} \vdash \Phi\ wf}{\widehat{\Psi}' \vdash \Phi\ wf} \qquad 3.\ \frac{\widehat{\Psi};\ \Phi \vdash t : t'}{\widehat{\Psi}';\ \Phi \vdash t : t'} \qquad 4.\ \frac{\widehat{\Psi};\ \Phi \vdash \sigma : \Phi'}{\widehat{\Psi}';\ \Phi \vdash \sigma : \Phi'}$$

Proof. Simple by structural induction on the judgements. The interesting cases are the ones mentioning extension variables, as for example when $\Phi = \Phi',\ \phi_i$, or $t = X_i/\sigma$. In both such cases, the typing rule has a side condition requiring that $\widehat{\Psi}.i = T$. Since $\widehat{\Psi} \sqsubseteq \widehat{\Psi}'$, we have that $\widehat{\Psi}'.i = \widehat{\Psi}.i = T$. □

Lemma 5.1.4 *(Relevancy is decidable)*

$$1.\ \frac{\Psi \vdash T : K}{\exists!\widehat{\Psi}.\mathit{relevant}\,(\Psi \vdash T : K) = \widehat{\Psi}} \qquad 2.\ \frac{\Psi \vdash \Phi\ wf}{\exists!\widehat{\Psi}.\mathit{relevant}\,(\Psi \vdash \Phi\ wf) = \widehat{\Psi}}$$

$$3.\ \frac{\Psi;\ \Phi \vdash t : t'}{\exists!\widehat{\Psi}.\mathit{relevant}\,\big(\Psi;\ \Phi \vdash t : t'\big) = \widehat{\Psi}} \qquad 4.\ \frac{\Psi;\ \Phi \vdash \sigma : \Phi'}{\exists!\widehat{\Psi}.\mathit{relevant}\,\big(\Psi;\ \Phi \vdash \sigma : \Phi'\big) = \widehat{\Psi}}$$

Proof. The relevancy judgements are defined by structural induction on the corresponding typing derivations. It is crucial to take into account the fact that $\vdash \Psi$ wf and $\Psi \vdash \Phi$ wf are implicitly present in any typing derivation that mentions such contexts; thus these derivations themselves, as well as their sub-derivations, are structurally included in derivations like $\Psi;\ \Phi \vdash t : t'$. Furthermore, it is easy to see that all the partial context joins used are defined, as in all cases the joined contexts are less-specified versions of Ψ. This fact follows by induction on the derivation of the result of the relevancy operation and by inspecting the base cases for the partial contexts returned. □

Lemma 5.1.5 *(Relevancy soundness)*

$$1.\ \frac{\Psi \vdash T : K \qquad \mathit{relevant}\,(\Psi \vdash T : K) = \widehat{\Psi}}{\widehat{\Psi} \vdash T : K} \qquad 2.\ \frac{\Psi \vdash \Phi\ \mathit{wf} \qquad \mathit{relevant}\,(\Psi \vdash \Phi\ \mathit{wf}) = \widehat{\Psi}}{\widehat{\Psi} \vdash \Phi\ \mathit{wf}}$$

$$3.\ \frac{\Psi;\ \Phi \vdash t : t' \qquad \mathit{relevant}\,\big(\Psi;\ \Phi \vdash t : t'\big) = \widehat{\Psi}}{\widehat{\Psi};\ \Phi \vdash t : t'} \qquad 4.\ \frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \mathit{relevant}\,\big(\Psi;\ \Phi \vdash \sigma : \Phi'\big) = \widehat{\Psi}}{\widehat{\Psi};\ \Phi \vdash \sigma : \Phi'}$$

Proof. By induction on the typing derivations. □

Theorem 5.1.6 *(Interaction of relevancy and extension substitution)*

Assuming $\Psi' \vdash \sigma_\Psi : \Psi$, $\Psi \vdash \Psi_u\ \mathit{wf}$ *and* $\sigma'_\Psi = \sigma_\Psi,\ \mathit{id}_{\Psi_u}$,

$$1.\ \frac{\mathit{unspec}_\Psi, \widehat{\Psi}_u \sqsubseteq \mathit{relevant}\,(\Psi,\ \Psi_u \vdash T : K)}{\mathit{unspec}_{\Psi'}, \widehat{\Psi}_u \cdot \sigma_\Psi \sqsubseteq \mathit{relevant}\,\big(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash T \cdot \sigma'_\Psi : K \cdot \sigma'_\Psi\big)}$$

$$2.\ \frac{\mathit{unspec}_\Psi, \widehat{\Psi}_u \sqsubseteq \mathit{relevant}\,(\Psi,\ \Psi_u \vdash \Phi\ \mathit{wf})}{\mathit{unspec}_{\Psi'}, \widehat{\Psi}_u \cdot \sigma_\Psi \sqsubseteq \mathit{relevant}\,\big(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash \Phi \cdot \sigma'_\Psi\ \mathit{wf}\big)}$$

$$3.\ \frac{\mathit{unspec}_\Psi, \widehat{\Psi}_u \sqsubseteq \mathit{relevant}\,\big(\Psi,\ \Psi_u;\ \Phi \vdash t : t'\big)}{\mathit{unspec}_{\Psi'}, \widehat{\Psi}_u \cdot \sigma_\Psi \sqsubseteq \mathit{relevant}\,\big(\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \vdash t \cdot \sigma'_\Psi : t' \cdot \sigma'_\Psi\big)}$$

$$4.\ \frac{\mathit{unspec}_\Psi, \widehat{\Psi}_u \sqsubseteq \mathit{relevant}\,\big(\Psi,\ \Psi_u;\ \Phi \vdash \sigma : \Phi'\big)}{\mathit{unspec}_{\Psi'}, \widehat{\Psi}_u \cdot \sigma_\Psi \sqsubseteq \mathit{relevant}\,\big(\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \vdash \sigma \cdot \sigma'_\Psi : \Phi' \cdot \sigma'_\Psi\big)}$$

Proof. Proceed by induction on the typing judgements. We prove two interesting cases.

Case CtxCVar**.**

$$\left(\frac{\Psi,\ \Psi_u \vdash \Phi \text{ wf} \qquad (\Psi,\ \Psi_u).i = [\Phi]\ ctx}{\Psi,\ \Psi_u \vdash (\Phi,\ \phi_i) \text{ wf}}\right)$$

We have that $unspec_\Psi,\ \widehat{\Psi}_u \sqsubseteq relevant\,(\Psi,\ \Psi_u \vdash \Phi' \text{ wf}) \circ ((\Psi,\ \Psi_u)\widehat{@}i)$.

We split cases based on whether $i < |\Psi|$ or not.

In the first case, the desired follows trivially.

In the second case, we assume without loss of generality $\widehat{\Psi}'_u$ such that

$unspec_\Psi,\ \widehat{\Psi}'_u \sqsubseteq relevant\,(\Psi,\ \Psi_u \vdash \Phi' \text{ wf})$ and

$unspec_\Psi,\ \widehat{\Psi}_u = (unspec_\Psi,\ \widehat{\Psi}'_u) \circ ((\Psi,\ \Psi_u)\widehat{@}i)$

Then by induction hypothesis we get:

$unspec_{\Psi'},\ \widehat{\Psi}'_u \cdot \sigma_\Psi \sqsubseteq relevant\,(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash \Phi' \cdot \sigma'_\Psi \text{ wf})$

Now we have that $(\Phi',\ X_i) \cdot \sigma'_\Psi = \Phi' \cdot \sigma'_\Psi,\ X_{i-|\Psi|+|\Psi'|}$. Thus:

$$\begin{aligned} &relevant\,(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash (\Phi',\ X_i) \cdot \sigma'_\Psi \text{ wf}) = \\ &\quad = relevant\,\big(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash (\Phi' \cdot \sigma'_\Psi,\ X_{i-|\Psi|+|\Psi'|}) \text{ wf}\big) \\ &\quad = relevant\,(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash \Phi' \cdot \sigma'_\Psi \text{ wf}) \circ ((\Psi',\ \Psi_u \cdot \sigma_\Psi)\widehat{@}(i - |\Psi| + |\Psi'|)) \end{aligned}$$

Thus we have that:

$$\begin{aligned} &(unspec_{\Psi'},\ \widehat{\Psi}'_u \cdot \sigma_\Psi) \circ ((\Psi',\ \Psi_u \cdot \sigma_\Psi)\widehat{@}(i - |\Psi| + |\Psi'|)) \sqsubseteq \\ &\quad \sqsubseteq relevant\,(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash (\Phi',\ X_i) \cdot \sigma'_\Psi \text{ wf}) \end{aligned}$$

But $(unspec_{\Psi'},\ \widehat{\Psi}'_u \cdot \sigma_\Psi) \circ ((\Psi',\ \Psi_u \cdot \sigma_\Psi)\widehat{@}(i - |\Psi| + |\Psi'|)) = unspec_{\Psi'},\ \widehat{\Psi}_u \cdot \sigma_\Psi$.

This is because $(unspec_\Psi,\ \widehat{\Psi}_u) = (unspec_\Psi,\ \widehat{\Psi}'_u) \circ ((\Psi,\ \Psi_u)\widehat{@}i)$, so the i-th element is the only one where $unspec_\Psi, \widehat{\Psi}'_u$ might differ from $unspec_\Psi, \widehat{\Psi}_u$; this will be the $i - |\Psi| + |\Psi'|$-th element after σ'_Ψ is applied; and that element is definitely equal after the join.

Case MetaVar**.**

$$\left(\frac{(\Psi,\ \Psi_u).i = T \qquad T = [\Phi']\,t' \qquad \Psi,\ \Psi_u;\ \Phi \vdash \sigma : \Phi'}{\Psi,\ \Psi_u;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\right)$$

We split cases based on whether $i < |\Psi|$ or not. In case it is, the proof is trivial using part 4. We thus focus on the case where $i \geq |\Psi|$. We have that:

$$\begin{aligned} &unspec_\Psi,\ \widehat{\Psi} \sqsubseteq relevant\,(\Psi,\ \Psi_u \vdash [\Phi']\,t : [\Phi']\,s) \circ relevant\,(\Psi,\ \Psi_u;\ \Phi \vdash \sigma : \Phi') \circ \\ &\quad \circ((\Psi,\ \Psi_u)\widehat{@}i) \end{aligned}$$

Assume without loss of generality $\widehat{\Psi}^1_u, \widehat{\Psi}'_u, \widehat{\Psi}^2_u$ such that

$\widehat{\Psi}_u^1 = \widehat{\Psi}'_u, \quad \overbrace{?, \cdots, ?}^{|\Psi|+|\Psi_u|-i \text{ times}} \quad ,$

$\mathit{unspec}_\Psi,\ \widehat{\Psi}'_u \sqsubseteq \mathit{relevant}\,((\Psi, \Psi_u)|_i \vdash [\Phi']\, t : [\Phi']\, s),$

$\mathit{unspec}_\Psi,\ \widehat{\Psi}_u^2 \sqsubseteq \mathit{relevant}\,(\Psi,\ \Psi_u;\ \Phi \vdash \sigma : \Phi')$

and last that $\widehat{\Psi} = \widehat{\Psi}_u^1 \circ \widehat{\Psi}_u^2 \circ ((\Psi,\ \Psi_u \widehat{@} i))$.

By induction hypothesis for $[\Phi']\, t'$ we get that:

$\mathit{unspec}_{\Psi'},\ \widehat{\Psi}'_u \cdot \sigma_\Psi \sqsubseteq \mathit{relevant}\,(\Psi', \Psi_u \cdot \sigma_\Psi \vdash [\Phi' \cdot \sigma'_\Psi]\, t' \cdot \sigma'_\Psi : [\Phi' \cdot \sigma'_\Psi]\, s \cdot \sigma'_\Psi)$

By induction hypothesis for σ we get that:

$\mathit{unspec}_{\Psi'},\ \widehat{\Psi}_u^2 \cdot \sigma_\Psi \sqsubseteq \mathit{relevant}\,(\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \vdash \sigma \cdot \sigma'_\Psi : \Phi' \cdot \sigma'_\Psi)$

We combine the above to get the desired, using the properties of join and $\widehat{@}$ as we did earlier. □

We will now combine the theorems 5.1.2 and 5.1.6 into a single result. We first define the combined pattern typing judgement:

$$\frac{\Psi \vdash_p \Psi_u \text{ wf} \qquad \Psi,\ \Psi_u \vdash_p T_P : K \qquad \mathit{unspec}_\Psi, \Psi_u \sqsubseteq \mathit{relevant}\,(\Psi,\ \Psi_u \vdash_p T_P : K)}{\Psi \vdash_p^* \Psi_u > T_P : K}$$

Then the combined statement of the two theorems is:

Theorem 5.1.7 *(Extension substitution for combined pattern typing)*

$$\frac{\Psi \vdash_p^* \Psi_u > T_P : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash_p^* \Psi_u \cdot \sigma_\Psi > T_P \cdot \sigma_\Psi : K \cdot \sigma_\Psi}$$

Proof. Directly by typing inversion of typing for T_P and application of theorems 5.1.2 and 5.1.6. □

Pattern matching procedure

We are now ready to define the pattern matching procedure. We will do this by proving that for closed patterns and terms, there either exists a unique substitution making the pattern equal to the term, or no such substitution exists – that is, that pattern matching

$$\widehat{\sigma_\Psi} ::= \bullet \mid \widehat{\sigma_\Psi},\ T \mid \widehat{\sigma_\Psi},\ ?$$

$$\frac{}{\bullet \vdash \bullet : \bullet} \qquad \frac{\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi} \qquad \bullet \vdash T : K \cdot \widehat{\sigma_\Psi}}{\bullet \vdash_{\overline{p}} (\widehat{\sigma_\Psi},\ T) : (\widehat{\Psi},\ K)} \qquad \frac{\bullet \vdash_{\overline{p}} \widehat{\sigma_\Psi} : \widehat{\Psi}}{\bullet \vdash_{\overline{p}} (\widehat{\sigma_\Psi},\ ?) : (\widehat{\Psi},\ ?)}$$

Figure 5.8: Partial substitutions (syntax and typing)

is decidable and deterministic. The statement of this theorem will roughly be:

If $\Psi_u \vdash_{\overline{p}} T_P : K$ and $\bullet \vdash T : K$ then either there exists a unique σ_Ψ such that

$\bullet \vdash \sigma_\Psi : \Psi_u$ and $T_P \cdot \sigma_\Psi = T$ or no such substitution exists.

Note that the first judgement should be understood as a judgement of the form $\Psi,\ \Psi_u \vdash_{\overline{p}} T_P : K$ as seen above, where the normal context Ψ is empty. The computational content of this proof is the pattern matching procedure.

In order to do the proof of the pattern matching algorithm by induction, we need an equivalent lemma to hold at the sub-derivations of patterns and terms representing the induction principle, such as:

If $\Psi_u;\ \Phi \vdash_{\overline{p}} t_P : t'$ and $\bullet;\ \Phi \vdash t : t'$ then either there exists a unique σ_Ψ such that

$\bullet \vdash \sigma_\Psi : \Psi_u$ and $t_P \cdot \sigma_\Psi = t$ or no such substitution exists.

Such a lemma is not valid, for the simple reason that the Ψ_u variables that do not get used in t_P can be substituted by arbitrary terms. Our induction principle therefore needs to be refined so that only the relevant partial context $\widehat{\Psi}_u$ is taken into account. Also, the substitution established needs to be similarly a *partial substitution*, only instantiating the relevant variables.

We thus define this notion in Figure 5.8 (syntax and typing) and provide the operations involving partial substitutions in Figure 5.9. Note that applying a partial extension substitution $\widehat{\sigma_\Psi}$ is entirely identical to applying a normal extension substitution. It fails when an extension variable that is left unspecified in $\widehat{\sigma_\Psi}$ gets used, something that already happens from the existing definitions as they do not account for this case. With these definitions, we are able to state a stronger version of the earlier induction principle, as follows:

If $\widehat{\Psi}_u;\ \Phi \vdash_{\overline{p}} t_P : t'$ and $\bullet;\ \Phi \vdash t : t'$ then either there exists a unique $\widehat{\sigma_\Psi}$ such that

$\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u$ and $t_P \cdot \widehat{\sigma_\Psi} = t$ or no such substitution exists.

Joining two partial substitutions:
$\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}' = \widehat{\sigma_\Psi}''$

$$
\begin{array}{lcl}
\bullet \circ \bullet & = & \bullet \\
(\widehat{\sigma_\Psi},\ T) \circ (\widehat{\sigma_\Psi}',\ T) & = & (\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'),\ T \\
(\widehat{\sigma_\Psi},\ ?) \circ (\widehat{\sigma_\Psi}',\ T) & = & (\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'),\ T \\
(\widehat{\sigma_\Psi},\ T) \circ (\widehat{\sigma_\Psi}',\ ?) & = & (\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'),\ T \\
(\widehat{\sigma_\Psi},\ ?) \circ (\widehat{\sigma_\Psi}',\ ?) & = & (\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'),\ ?
\end{array}
$$

Less specified substitution:
$\widehat{\sigma_\Psi} \sqsubseteq \widehat{\sigma_\Psi}'$

$$
\begin{array}{lcl}
\bullet \sqsubseteq \bullet & & \\
(\widehat{\sigma_\Psi},\ T) \sqsubseteq (\widehat{\sigma_\Psi}',\ T) & \Leftarrow & \widehat{\sigma_\Psi} \sqsubseteq \widehat{\sigma_\Psi}' \\
(\widehat{\sigma_\Psi},\ ?) \sqsubseteq (\widehat{\sigma_\Psi}',\ T) & \Leftarrow & \widehat{\sigma_\Psi} \sqsubseteq \widehat{\sigma_\Psi}' \\
(\widehat{\sigma_\Psi},\ ?) \sqsubseteq (\widehat{\sigma_\Psi}',\ ?) & \Leftarrow & \widehat{\sigma_\Psi} \sqsubseteq \widehat{\sigma_\Psi}'
\end{array}
$$

Fully unspecified substitution:
$unspec_{\widehat{\Psi}} = \widehat{\sigma_\Psi}$

$$
\begin{array}{lcl}
unspec_\bullet & = & ? \\
unspec_{\widehat{\Psi},\ ?} & = & unspec_{\widehat{\Psi}},\ ? \\
unspec_{\widehat{\Psi},\ K} & = & unspec_{\widehat{\Psi}},\ ?
\end{array}
$$

Limiting to a partial context:
$\widehat{\sigma_\Psi}|_{\widehat{\Psi}} = \widehat{\sigma_\Psi}'$

$$
\begin{array}{lcl}
(\bullet)|_\bullet & = & \bullet \\
(\widehat{\sigma_\Psi},\ T)|_{\widehat{\Psi},\ ?} & = & \widehat{\sigma_\Psi}|_{\widehat{\Psi}},\ ? \\
(\widehat{\sigma_\Psi},\ T)|_{\widehat{\Psi},\ K} & = & \widehat{\sigma_\Psi}|_{\widehat{\Psi}},\ T \\
(\widehat{\sigma_\Psi},\ ?)|_{\widehat{\Psi},\ ?} & = & \widehat{\sigma_\Psi}|_{\widehat{\Psi}},\ ?
\end{array}
$$

Replacement of unspecified element at index:
$\widehat{\sigma_\Psi}[i \mapsto T] = \widehat{\sigma_\Psi}'$

$$
\begin{array}{lcl}
(\widehat{\sigma_\Psi},\ ?)[i \mapsto T] & = & \widehat{\sigma_\Psi},\ T \text{ when } i = |\widehat{\sigma_\Psi}| \\
(\widehat{\sigma_\Psi},\ ?)[i \mapsto T] & = & \widehat{\sigma_\Psi}[i \mapsto T],\ ? \text{ when } i < |\widehat{\sigma_\Psi}| \\
(\widehat{\sigma_\Psi},\ T')[i \mapsto T] & = & \widehat{\sigma_\Psi}[i \mapsto T],\ T' \text{ when } i < |\widehat{\sigma_\Psi}|
\end{array}
$$

Figure 5.9: Partial substitutions (syntactic operations)

$\boxed{(\Psi_u \Vdash_{\overline{p}} T : K) \sim (\bullet \vdash T' : K) \triangleright \widehat{\sigma_\Psi}}$

$$\frac{(\Psi_u;\ \Phi \Vdash_{\overline{p}} t : t_T) \sim \left(\bullet;\ \Phi \vdash t' : t_T\right) \triangleleft \mathit{unspec}_{\Psi_u} \triangleright \widehat{\sigma_\Psi}}{(\Psi_u \Vdash_{\overline{p}} [\Phi]\, t : [\Phi]\, t_T) \sim \left(\bullet \vdash [\Phi]\, t' : [\Phi]\, t_T\right) \triangleright \widehat{\sigma_\Psi}}$$

$$\frac{\left(\Psi_u \Vdash_{\overline{p}} \Phi,\ \Phi' \text{ wf}\right) \sim \left(\bullet \vdash \Phi,\ \Phi'' \text{ wf}\right) \triangleright \widehat{\sigma_\Psi}}{\left(\Psi_u \Vdash_{\overline{p}} [\Phi]\, \Phi' : [\Phi]\, \mathit{ctx}\right) \sim \left(\bullet \vdash [\Phi]\, \Phi'' : [\Phi]\, \mathit{ctx}\right) \triangleright \widehat{\sigma_\Psi}}$$

$\boxed{(\Psi_u \Vdash_{\overline{p}} \Phi \text{ wf}) \sim (\bullet \vdash \Phi' \text{ wf}) \triangleright \widehat{\sigma_\Psi}}$

$$\frac{}{(\Psi_u \Vdash_{\overline{p}} \bullet \text{ wf}) \sim (\bullet \vdash \bullet \text{ wf}) \triangleright \mathit{unspec}_{\Psi_u}}$$

$$\frac{(\Psi_u \Vdash_{\overline{p}} \Phi \text{ wf}) \sim \left(\bullet \vdash \Phi' \text{ wf}\right) \triangleright \widehat{\sigma_\Psi} \qquad (\Psi_u;\ \Phi \Vdash_{\overline{p}} t : s) \sim \left(\bullet;\ \Phi' \vdash t' : s\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}{(\Psi_u \Vdash_{\overline{p}} (\Phi,\ t) \text{ wf}) \sim \left(\bullet \vdash (\Phi',\ t') \text{ wf}\right) \triangleright \widehat{\sigma_\Psi}'}$$

$$\frac{}{(\Psi_u \Vdash_{\overline{p}} \Phi,\ \phi_i \text{ wf}) \sim \left(\bullet \vdash \Phi,\ \Phi' \text{ wf}\right) \triangleright \mathit{unspec}_{\Psi_u}[i \mapsto [\Phi]\, \Phi']}$$

Figure 5.10: Pattern matching algorithm for λHOL, operating on derivations (1/3)

This induction principle will indeed be valid.

We are now ready to proceed with the proof of the above theorem. As we have mentioned earlier, we will carry out the proof in a constructive manner; its computational content will be a pattern matching algorithm! Yet for presentation purposes we will make a slight diversion from the expected order: we will present the algorithm first and *then* proceed with the details of the proof out of which the algorithm is extracted.

Pattern matching algorithm. We illustrate the algorithm in Figures 5.10 through 5.12 by presenting it as a set of rules. If a derivation according to the given rules is not possible, the algorithm returns failure. The rules work on typing derivations for patterns and terms. The matching algorithm for terms accepts an input substitution $\widehat{\sigma_\Psi}$ and produces an output substitution $\widehat{\sigma_\Psi}'$. We denote this as $\triangleleft\widehat{\sigma_\Psi}\triangleright\widehat{\sigma_\Psi}'$. Most of the rules are straightforward, though the notation is heavy. The premises of each rule explicitly note which terms need to be syntactically equal and the recursive calls to pattern matching for sub-derivations. In the consequence, we sometimes unfold the last step of the derivation (easily inferrable through typing inversion), so as to assign names to the various subterms involved.

$$\boxed{(\Psi;\ \Phi \vdash_{\overline{p}} t : t_T) \sim (\bullet;\ \Phi' \vdash t' : t'_T) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}'}$$

$$\frac{}{(\Psi_u;\ \Phi \vdash_{\overline{p}} c : t) \sim \left(\bullet;\ \Phi' \vdash c : t'\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}} \qquad \frac{}{\left(\Psi_u;\ \Phi \vdash_{\overline{p}} s : s'\right) \sim \left(\bullet;\ \Phi' \vdash s : s''\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}}$$

$$\frac{L \cdot \widehat{\sigma_\Psi} = L'}{(\Psi_u;\ \Phi \vdash_{\overline{p}} v_L : t) \sim \left(\bullet;\ \Phi' \vdash v_{L'} : t'\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}}$$

$$\frac{\begin{array}{c} s = s_* \qquad (\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s) \sim \left(\bullet;\ \Phi' \vdash t'_1 : s_*\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}' \\ \left(\Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : s'\right) \sim \left(\bullet;\ \Phi',\ t'_1 \vdash_{\overline{p}} \left\lceil t'_2 \right\rceil_{|\Phi'|} : s'_*\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_\Psi}'' \end{array}}{\begin{array}{c} \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s \\ \Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : s' \\ (s, s', s'') \in \mathcal{R} \\ \hline \Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_1).t_2 : s'' \end{array} \sim \begin{array}{c} \bullet;\ \Phi \vdash t'_1 : s_* \\ \bullet;\ \Phi',\ t'_1 \vdash \left\lceil t'_2 \right\rceil_{|\Phi'|} : s'_* \\ (s_*, s'_*, s''_*) \in \mathcal{R} \\ \hline \bullet;\ \Phi' \vdash \Pi(t'_1).t'_2 : s''_* \end{array} \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}''}$$

$$\frac{\left(\Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : t'\right) \sim \left(\bullet;\ \Phi',\ t'_1 \vdash \left\lceil t'_2 \right\rceil_{|\Phi'|} : t''\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}'}{\begin{array}{c} \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s \\ \Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : t' \\ \Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_1).\left\lfloor t' \right\rfloor_{|\Phi|\dot{+}1} : s' \\ \hline \Psi_u;\ \Phi \vdash_{\overline{p}} \lambda(t_1).t_2 : \Pi(t_1).\left\lfloor t' \right\rfloor_{|\Phi|\dot{+}1} \end{array} \sim \begin{array}{c} \bullet;\ \Phi' \vdash t'_1 : s_* \\ \bullet;\ \Phi',\ t'_1 \vdash \left\lceil t'_2 \right\rceil_{|\Phi'|} : t'' \\ \bullet;\ \Phi' \vdash \Pi(t'_1).\left\lfloor t'' \right\rfloor_{|\Phi'|\dot{+}1} : s'_* \\ \hline \bullet;\ \Phi' \vdash \lambda(t'_1).t'_2 : \Pi(t'_1).\left\lfloor t'' \right\rfloor_{|\Phi'|\dot{+}1} \end{array} \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}'}$$

$$\frac{\begin{array}{c} s = s' \\ \mathit{relevant}\,(\Psi_u \vdash_{\overline{p}} \Phi \text{ wf}) = \widehat{\Psi} \qquad (\Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_a).t_b : s) \sim \left(\bullet;\ \Phi \vdash \Pi(t'_a).t'_b : s'\right) \lhd \widehat{\sigma_\Psi}|_{\widehat{\Psi}} \rhd \widehat{\sigma_\Psi}' \\ (\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : \Pi(t_a).t_b) \sim \left(\bullet;\ \Phi' \vdash t'_1 : \Pi(t'_a).t'_b\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_\Psi 1} \\ \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t_a : s) = \widehat{\Psi}' \qquad (\Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t_a) \sim \left(\bullet;\ \Phi' \vdash t'_2 : t'_a\right) \lhd \widehat{\sigma_\Psi}'|_{\widehat{\Psi}'} \rhd \widehat{\sigma_\Psi 2} \end{array}}{\begin{array}{c} \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : \Pi(t_a).t_b \\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t_a \\ \Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_a).t_b : s \\ \hline \Psi_u;\ \Phi \vdash_{\overline{p}} t_1\ t_2 : \lceil t_b \rceil_{|\Phi|} \cdot (id_\Phi, t_2) \end{array} \sim \begin{array}{c} \bullet;\ \Phi' \vdash t'_1 : \Pi(t'_a).t'_b \\ \bullet;\ \Phi' \vdash t'_2 : t'_a \\ \bullet;\ \Phi' \vdash \Pi(t'_a).t'_b : s' \\ \hline \bullet;\ \Phi' \vdash_{\overline{p}} t'_1\ t'_2 : \left\lceil t'_b \right\rceil_{|\Phi'|} \cdot (id_{\Phi'}, t'_2) \end{array} \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi 1} \circ \widehat{\sigma_\Psi 2}}$$

Figure 5.11: Pattern matching algorithm for λHOL, operating on derivations (2/3)

$$\boxed{(\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T) \sim (\bullet;\ \Phi' \vdash t' : t'_T) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}'}$$

$$\frac{\begin{array}{c}(\Psi_u;\ \Phi \vdash_{\overline{p}} t : \mathit{Type}) \sim \left(\bullet;\ \Phi' \vdash t' : \mathit{Type}\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}' \\ (\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : t) \sim \left(\bullet;\ \Phi' \vdash t'_1 : t'\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_\Psi}_1 \\ (\Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t) \sim \left(\bullet;\ \Phi' \vdash t'_2 : t'\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_\Psi}_2\end{array}}{\begin{array}{cc}\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : t & \bullet;\ \Phi \vdash t'_1 : t' \\ \Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t & \bullet;\ \Phi \vdash t'_2 : t' \\ \Psi_u;\ \Phi \vdash_{\overline{p}} t : \mathit{Type} & \bullet;\ \Phi \vdash t' : \mathit{Type} \\ \hline \Psi_u;\ \Phi \vdash_{\overline{p}} t_1 = t_2 : \mathit{Prop} & \sim \quad \bullet;\ \Phi \vdash t'_1 = t'_2 : \mathit{Prop}\end{array} \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}_1 \circ \widehat{\sigma_\Psi}_2}$$

$$\frac{\widehat{\sigma_\Psi}.i =? \qquad \Psi_u.i = [\Phi^*]\, t_T \qquad t' <^f |\Phi^* \cdot \widehat{\sigma_\Psi}|}{(\Psi_u;\ \Phi \vdash_{\overline{p}} X_i/\sigma : t_T \cdot \sigma) \sim \left(\bullet;\ \Phi' \vdash t' : t'_T\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}[i \mapsto [\Phi^* \cdot \widehat{\sigma_\Psi}]\, t']}$$

$$\frac{\widehat{\sigma_\Psi}.i = [\Phi^*]\, t \qquad t = t'}{(\Psi_u;\ \Phi \vdash_{\overline{p}} X_i/\sigma : t_T) \sim \left(\bullet;\ \Phi' \vdash t' : t'_T\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}}$$

Figure 5.12: Pattern matching algorithm for λHOL, operating on derivations (3/3)

Of special note are the two rules for unification variables, which represent an important case splitting in the proof that will follow. The essential reason why two rules are needed is the fact that patterns are allowed to be *non-linear*: that is, the same unification variable can be used multiple times. This is guided by practical considerations and is an important feature of our pattern matching support. Our formulation of pattern matching through partial contexts and substitutions allows such support without significantly complicating our proofs compared to the case where only linear patterns would be allowed. The two rules thus correspond to whether a unification variable is still unspecified or has already been instantiated at a previous occurence. In the former case, we just need to check that the scrutinee only uses variables from the allowed prefix of the current context (e.g. when our pattern corresponds to closed terms, we check to see that the scrutinee t' does not use any variables); if that is the case, we instantiate the unification variable as equal to the scrutinee. In the latter case we only need to check that the already-established substitution for the unification variable X_i is syntactically equal to the scrutinee.

Proof of pattern matching determinism and decidability. Let us now return to the pattern matching theorem we set out to prove in the beginning of this section. First, we will prove a couple of lemmas about partial substitution operations.

Lemma 5.1.8 *(Typing of joint partial substitutions)*

$$\frac{\bullet \vdash \widehat{\sigma_{\Psi 1}} : \widehat{\Psi}_1 \qquad \bullet \vdash \widehat{\sigma_{\Psi 2}} : \widehat{\Psi}_2 \qquad \widehat{\Psi}_1 \circ \widehat{\Psi}_2 = \widehat{\Psi}' \qquad \widehat{\sigma_{\Psi 1}} \circ \widehat{\sigma_{\Psi 2}} = \widehat{\sigma_\Psi}'}{\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'}$$

Proof. By induction on the derivation of $\widehat{\sigma_{\Psi 1}} \circ \widehat{\sigma_{\Psi 2}} = \widehat{\sigma_\Psi}'$ and use of typing inversion for $\widehat{\sigma_{\Psi 1}}$ and $\widehat{\sigma_{\Psi 2}}$. □

Lemma 5.1.9 *(Typing of partial substitution limitation)*

$$\frac{\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi} \qquad \bullet \vdash \widehat{\Psi}' \ wf \qquad \widehat{\Psi}' \sqsubseteq \widehat{\Psi}}{\widehat{\sigma_\Psi}|_{\widehat{\Psi}'} \sqsubseteq \widehat{\sigma_\Psi} \qquad \bullet \vdash \widehat{\sigma_\Psi}|_{\widehat{\Psi}'} : \widehat{\Psi}'}$$

Proof. Trivial by induction on the derivation of $\widehat{\sigma_\Psi}|_{\widehat{\Psi}'} = \widehat{\sigma_\Psi}'$. □

Let us now proceed to the main pattern matching theorem. To prove it we will assume well-sortedness derivations are subderivations of normal typing derivations. That is, if $\Psi;\ \Phi \models_{\overline{p}} t : t'$, with $t' \neq \mathit{Type}'$, the derivation $\Psi;\ \Phi \models_{\overline{p}} t' : s$ for a suitable s is a subderivation of the original derivation $\Psi;\ \Phi \models_{\overline{p}} t : t'$. The way we have stated the typing rules for λHOL this is actually not true, but an adaptation where the $t' : s$ derivation becomes part of the $t : t'$ derivation is possible, thanks to Lemma 4.1.7. Furthermore, we will use the quantifier $\exists^{\leq 1}$ as a shorthand for either unique existence or non-existence, that is, $\exists^{\leq 1} x.P \equiv (\exists! x.P) \vee (\neg \exists x.P)$

Theorem 5.1.10 *(Decidability and determinism of pattern matching)*

$$1.\ \frac{\Psi_u \models_{\overline{p}} \Phi \ wf \qquad \bullet \vdash \Phi' \ wf \qquad \mathit{relevant}\,(\Psi_u \models_{\overline{p}} \Phi \ wf) = \widehat{\Psi}_u}{\exists^{\leq 1} \widehat{\sigma_\Psi}. \Big(\quad \bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u \ \wedge \ \Phi \cdot \widehat{\sigma_\Psi} = \Phi' \quad \Big)}$$

$$\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T \qquad \bullet;\ \Phi' \vdash t' : t'_T \qquad \mathit{relevant}\left(\Psi_u;\ \Phi \vdash_{\overline{p}} t : t'_T\right) = \widehat{\Psi}'_u$$
$$\left(\quad \Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s \ \wedge\ \bullet;\ \Phi \vdash t'_T : s \ \wedge\ \mathit{relevant}\left(\Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s\right) = \widehat{\Psi}_u \quad\right)$$
$$\vee \left(\quad t_T = \mathit{Type} \ \wedge\ \Psi_u \vdash_{\overline{p}} \Phi\ \mathit{wf} \ \wedge\ \bullet \vdash \Phi\ \mathit{wf} \ \wedge\ \mathit{relevant}\left(\Psi_u \vdash_{\overline{p}} \Phi\ \mathit{wf}\right) = \widehat{\Psi}_u \quad\right)$$

2.
$$\frac{\exists! \widehat{\sigma_\Psi}. \left(\quad \bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u \ \wedge\ \Phi \cdot \widehat{\sigma_\Psi} = \Phi' \ \wedge\ t_T \cdot \widehat{\sigma_\Psi} = t'_T \quad\right)}{\exists^{\leq 1} \widehat{\sigma_\Psi}'. \left(\quad \bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'_u \ \wedge\ \Phi \cdot \widehat{\sigma_\Psi}' = \Phi' \ \wedge\ t_T \cdot \widehat{\sigma_\Psi}' = t'_T \ \wedge\ t \cdot \widehat{\sigma_\Psi}' = t' \quad\right)}$$

3.
$$\frac{\Psi_u \vdash_{\overline{p}} T : K \qquad \bullet \vdash T' : K \qquad \mathit{relevant}\left(\Psi_u \vdash_{\overline{p}} T : K\right) = \Psi_u}{\exists^{\leq 1} \sigma_\Psi. \left(\quad \bullet \vdash \sigma_\Psi : \Psi_u \ \wedge\ T \cdot \sigma_\Psi = T' \quad\right)}$$

Proof.

Part 1 By induction on the well-formedness derivation for Φ.

Case PCtxExpty**.**

$$\left(\frac{}{\Psi_u \vdash_{\overline{p}} \bullet \text{ wf}}\right)$$

Trivially, we either have $\Phi' = \bullet$, in which case unspec_Ψ is the unique substitution with the desired properties, or no substitution possibly exists.

Case PCtxVar**.**

$$\left(\frac{\Psi_u \vdash_{\overline{p}} \Phi \text{ wf} \qquad \Psi_u;\ \Phi \vdash_{\overline{p}} t : s}{\Psi_u \vdash_{\overline{p}} (\Phi,\ t) \text{ wf}}\right)$$

We either have that $\Phi' = \Phi',\ t'$ or no substitution possibly exists. By induction hypothesis get $\widehat{\sigma_\Psi}$ such that $\Phi \cdot \widehat{\sigma_\Psi} = \Phi'$ and $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u$ with $\widehat{\Psi}_u = \mathit{relevant}\,(\Psi_u \vdash_{\overline{p}} \Phi \text{ wf})$. Furthermore by typing inversion we get that $\bullet;\ \Phi' \vdash t' : s'$. We need $s' = s$ otherwise no suitable substitution exists. Now we use part 2 to either get a $\widehat{\sigma_\Psi}'$ which is obviously the substitution that we want, since $(\Phi,\ t) \cdot \widehat{\sigma_\Psi}' = \Phi',\ t'$ and $\mathit{relevant}\,(\Psi_u \vdash_{\overline{p}} (\Phi,\ t) \text{ wf}) = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t : s)$; or we get the fact that no such substitution possibly exists. In that case, we again conclude that no substitution for the current case exists either, otherwise it would violate the induction hypothesis.

Case PCtxCVarUnif**.**

$$\left(\frac{\Psi_u \vdash_{\overline{p}} \Phi \text{ wf} \qquad i \geq |\Psi| \qquad \Psi_u.i = [\Phi]\, ctx}{\Psi_u \vdash_{\overline{p}} (\Phi,\ \phi_i) \text{ wf}}\right)$$

We either have $\Phi' = \Phi,\ \Phi''$, or no substitution possibly exists (since Φ does not depend on unification variables, so we always have $\Phi \cdot \widehat{\sigma_\Psi} = \Phi$). We now consider the substitution $\widehat{\sigma_\Psi} = \mathit{unspec}_{\Psi_u}[i \mapsto [\Phi]\,\Phi'']$. We obviously have that $(\Phi,\ \phi_i) \cdot \widehat{\sigma_\Psi} = \Phi,\ \Phi''$, and also that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u$ with $\widehat{\Psi}_u = \Psi_u \widehat{@} i = \mathit{relevant}\,(\Psi \vdash_{\overline{p}} \Phi,\ \phi_i \text{ wf})$. Thus this substitution has the desired properties.

Part 2 By induction on the typing derivation for t.

Case PConstant**.**

$$\left(\frac{c : t \in \Sigma \qquad \bullet;\ \bullet \vdash t_T : s \qquad s \neq \mathit{Prop}}{\Psi_u;\ \Phi \vdash_{\overline{p}} c : t_T}\right)$$

We have $t \cdot \widehat{\sigma_\Psi}' = c \cdot \widehat{\sigma_\Psi}' = c$. So for any substitution to satisfy the desired properties we need to have that $t' = c$ also; if this isn't so, no $\widehat{\sigma_\Psi}'$ possibly exists. If we have that $t = t' = c$, then we choose $\widehat{\sigma_\Psi}' = \widehat{\sigma_\Psi}$ as provided by assumption, which has the desired properties considering that:

$\mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} c : t) = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s) = \mathit{relevant}\,(\Psi_u \vdash_{\overline{p}} \Phi \text{ wf})$

(since t_T comes from the definitions context and can therefore not contain extension variables).

Case PVar**.**

$$\left(\frac{\Phi.L = t_T \qquad \Psi_u;\ \Phi \vdash t_T : s \qquad s \neq \mathit{Prop}}{\Psi_u;\ \Phi \vdash_{\overline{p}} v_L : t_T}\right)$$

Similarly as above. First, we need $t' = v_{L'}$, otherwise no suitable $\widehat{\sigma_\Psi}'$ exists. From assumption we have a unique $\widehat{\sigma_\Psi}$ for $\mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s)$. If $L \cdot \widehat{\sigma_\Psi} = L'$, then $\widehat{\sigma_\Psi}$ has all the desired properties for $\widehat{\sigma_\Psi}'$, considering the fact that $\mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} v_L : t_T) = \mathit{relevant}\,(\Psi_u \vdash_{\overline{p}} \Phi \text{ wf})$ and $\mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} t_T : s) = \mathit{relevant}\,(\Psi \vdash_{\overline{p}} \Phi \text{ wf})$ (since $t_T = \Phi.i$). It is also unique, because an alternate $\widehat{\sigma_\Psi}'$ would violate the assumed uniqueness of $\widehat{\sigma_\Psi}$. If

$L \cdot \widehat{\sigma_\Psi} \neq L'$, no suitable substitution exists, because of the same reason.

Case PSort**.**

$$\left(\frac{(s, s') \in \mathcal{A}}{\Psi_u;\ \Phi \vdash_{\overline{p}} s : s'} \right)$$

Entirely similar to the case for PConstant.

Case PΠType**.**

$$\left(\frac{\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s \qquad \Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_1).t_2 : s''} \right)$$

First, we have either that $t' = \Pi(t'_1).t'_2$, or no suitable $\widehat{\sigma_\Psi}'$ exists. Thus by inversion for t' we get:

$\bullet;\ \Phi' \vdash_{\overline{p}} t'_1 : s_*, \quad \bullet;\ \Phi',\ t'_1 \vdash_{\overline{p}} \lceil t'_2 \rceil_{|\Phi'|} : s'_*, \quad (s_*, s'_*, s'') \in \mathcal{R}.$

Now, we need $s = s_*$, otherwise no suitable $\widehat{\sigma_\Psi}'$ possibly exists. To see why this is so, assume that a $\widehat{\sigma_\Psi}'$ satisfying the necessary conditions exists, and $s \neq s_*$; then we have that $t_1 \cdot \widehat{\sigma_\Psi}' = t'_1$, which means that their types should also match, a contradiction.

We use the induction hypothesis for t_1 and t'_1. We are allowed to do so because

$\mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} s'' : s''') = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} s : s'''')$

and the other properties for $\widehat{\sigma_\Psi}$ also hold trivially.

From that we either get a $\widehat{\sigma_\Psi}'$ such that: $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'_u, \quad t_1 \cdot \widehat{\sigma_\Psi}' = t'_1, \quad \Phi \cdot \widehat{\sigma_\Psi}' = \Phi'$ for $\widehat{\Psi}'_u = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s)$; or that no such substitution exists. Since a partial substitution unifying t with t' will also include a substitution that only has to do with $\widehat{\Psi}'_u$, we see that if no $\widehat{\sigma_\Psi}'$ is returned by the induction hypothesis, no suitable substitution for t and t' actually exists.

We can now use the induction hypothesis for t_2 and $\widehat{\sigma_\Psi}'$ with $\widehat{\Psi}''_u$ such that:

$\widehat{\Psi}''_u = \mathit{relevant}\left(\Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : s'\right)$. The induction hypothesis is applicable since

$\mathit{relevant}\,(\Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} s' : s''''') = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s)$

and the other requirements trivially hold. Especially for s' and s'_* being equal, this is trivial since both need to be equal to s'' (because of the form of our rule set $\mathcal{R}$).

From that we either get a $\widehat{\sigma_\Psi}''$ such that $\bullet \vdash \widehat{\sigma_\Psi}'' : \widehat{\Psi}''_u$, $\lceil t_2 \rceil_{|\Phi|} \cdot \widehat{\sigma_\Psi}'' = \lceil t'_2 \rceil_{|\Phi'|}$, $\Phi \cdot \widehat{\sigma_\Psi}'' = \Phi$

and $t_1 \cdot \widehat{\sigma_\Psi}'' = t_1'$, or that such $\widehat{\sigma_\Psi}''$ does not exist. In the second case we proceed as above, so we focus in the first case.

By use of properties of freshening we can deduce that $(\Pi(t_1).t_2) \cdot \widehat{\sigma_\Psi}'' = \Pi(t_1').(t_2')$, so the returned $\widehat{\sigma_\Psi}''$ has the desired properties, if we consider the fact that
$\mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_1).t_2 : s'') = \mathit{relevant}\left(\Psi_u;\ \Phi, t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : s'\right) = \widehat{\Psi}_u''$.

Case ΠΠIntro**.**

$$\left(\frac{\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : s \qquad \Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : t_3 \qquad \Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_1).\ \lfloor t_3 \rfloor_{|\Phi|\dot{+}1} : s' \qquad s' \neq \mathit{Prop}}{\Psi_u;\ \Phi \vdash_{\overline{p}} \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t_3 \rfloor_{|\Phi|\dot{+}1}}\right)$$

We have that either $t' = \lambda(t_1').t_2'$, or no suitable $\widehat{\sigma_\Psi}'$ exists. Thus by typing inversion for t' we get:

$\bullet;\ \Phi' \vdash_{\overline{p}} t_1' : s_*,\quad \bullet;\ \Phi',\ t_1' \vdash_{\overline{p}} \lceil t_2' \rceil_{|\Phi'|} : t_3',\quad \bullet;\ \Phi' \vdash_{\overline{p}} \Pi(t_1').\ \lfloor t_3 \rfloor_{|\Phi'|\dot{+}1} : s_*'$.

By assumption we have that there exists a unique $\widehat{\sigma_\Psi}$ such that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u,\quad \Phi \cdot \widehat{\sigma_\Psi} = \Phi'$, $(\Pi(t_1).\ \lfloor t_3 \rfloor) \cdot \widehat{\sigma_\Psi} = \Pi(t_1').\ \lfloor t_3' \rfloor$, if $\mathit{relevant}\left(\Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_1).\ \lfloor t_3 \rfloor_{|\Phi|\dot{+}1}\right) = \widehat{\Psi}_u$. From these we also get that $s' = s_*'$.

From the fact that $(\Pi(t_1).\ \lfloor t_3 \rfloor) \cdot \widehat{\sigma_\Psi} = \Pi(t_1').\ \lfloor t_3' \rfloor$, we get first of all that $t_1 \cdot \widehat{\sigma_\Psi} = t_1'$, and also that $t_3 \cdot \widehat{\sigma_\Psi} = t_3'$. Furthermore, we have that $\mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} \Pi(t_1).\ \lfloor t_3 \rfloor : s') = \mathit{relevant}\,(\Psi;\ \Phi,\ t_1 \vdash_{\overline{p}} t_3 : s')$.

From that we understand that $\widehat{\sigma_\Psi}$ is a suitable substitution to use for the induction hypothesis for $\lceil t_2 \rceil$.

Thus from induction hypothesis we either get a unique $\widehat{\sigma_\Psi}'$ with the properties:
$\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}_u',\quad \lceil t_2 \rceil \cdot \widehat{\sigma_\Psi}' = \lceil t_2' \rceil,\quad (\Phi,\ t_1) \cdot \widehat{\sigma_\Psi} = \Phi',\ t_1',\ t_3 \cdot \widehat{\sigma_\Psi} = t_3'$, if
$\mathit{relevant}\left(\Psi_u;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : t_3\right) = \widehat{\Psi}_u$, or that no such substitution exists. We focus on the first case; in the second case no unifying substitution for t and t' exists, otherwise the lack of existence of a suitable $\widehat{\sigma_\Psi}'$ would lead to a contradiction.

This substitution $\widehat{\sigma_\Psi}'$ has the desired properties with respect to matching of t against t' (again using the properties of freshening), and it is unique, because the existence of an alternate substitution with the same properties would violate the uniqueness assumption of the substitution returned by induction hypothesis.

Case PΠELIM**.**

$$\left(\frac{\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : \Pi(t_a).t_b \qquad \Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t_a \qquad \Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_a).t_b : s \qquad s' \neq \mathit{Prop}}{\Psi_u;\ \Phi \vdash_{\overline{p}} t_1\ t_2 : \lceil t_b \rceil_{|\Phi|} \cdot (id_\Phi, t_2)}\right)$$

Again we have that either $t' = t'_1\ t'_2$, or no suitable substitution possibly exists. Thus by inversion of typing for t' we get:

$\bullet;\ \Phi \vdash_{\overline{p}} t'_1 : \Pi(t'_a).t'_b,\quad \bullet;\ \Phi \vdash_{\overline{p}} t'_2 : t'_a,\quad t'_T = \lceil t'_b \rceil_{|\Phi'|} \cdot (id_{\Phi'},\ t'_2)$

Furthermore we have that $\Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_a).t_b : s$ and $\bullet;\ \Phi \vdash_{\overline{p}} \Pi(t'_a).t'_b : s'$ for suitable s, s'. We need $s = s'$, otherwise no suitable $\widehat{\sigma_\Psi}'$ exists (because if t_1 and t'_1 were matchable by substitution, their Π-types would match, and also their sorts, which is a contradiction).

We can use the induction hypothesis for $\Pi(t_a).t_b$ and $\Pi(t'_a).t'_b$, with the partial substitution $\widehat{\sigma_\Psi}$ limited only to those variables relevant in $\Psi_u \vdash_{\overline{p}} \Phi$ wf. That is, if $\widehat{\Psi}'_u = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} \Pi(t_a).t_b : s)$ then we use the substitution $\widehat{\sigma_\Psi}|_{\widehat{\Psi}_u}$. In that case all of the requirements for $\widehat{\sigma_\Psi}$ hold (the uniqueness condition also holds for this substitution, using part 1 for the fact that Φ and Φ' only have a unique matching substitution), so we get from the induction hypothesis either a $\widehat{\sigma_\Psi}'$ such that $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}_u,\quad \Phi \cdot \widehat{\sigma_\Psi} = \Phi'$ and $(\Pi(t_a).t_b) \cdot \widehat{\sigma_\Psi} = \Pi(t'_a).t'_b$, or that no such $\widehat{\sigma_\Psi}'$ exists. In the second case, again we can show that no suitable substitution for t and t' exists; so we focus in the first case.

We can now use the induction hypothesis for t_1, using this $\widehat{\sigma_\Psi}'$. From that, we get that either a $\widehat{\sigma_\Psi}_1$ exists for $\widehat{\Psi}_1 = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : \Pi(t_a).t_b)$ such that $t_1 \cdot \widehat{\sigma_\Psi}_1 = t'_1$, or that no such $\widehat{\sigma_\Psi}_1$ exists, in which case we argue that no global $\widehat{\sigma_\Psi}'$ exists for matching t with t' (because we could limit it to the $\widehat{\Psi}_1$ variables and yield a contradiction).

We now form $\widehat{\sigma_\Psi}'' = \widehat{\sigma_\Psi}'|_{\widehat{\Psi}''}$ for $\widehat{\Psi}'' = \mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} t_a : s_*)$. For $\widehat{\sigma_\Psi}''$, we have that $\bullet \vdash_{\overline{p}} \widehat{\sigma_\Psi}'' : \widehat{\Psi}''$, $\Phi \cdot \widehat{\sigma_\Psi}'' = \Phi'$ and $t_a \cdot \widehat{\sigma_\Psi}'' = t_a$. Also it is the unique substitution with those properties, otherwise the induction hypothesis for t_a would be violated.

Using $\widehat{\sigma_\Psi}''$ we can allude to the induction hypothesis for t_2, which either yields a substitution $\widehat{\sigma_\Psi}_2$ for $\widehat{\Psi}_2 = \mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} t_2 : t_a)$, such that $t_2 \cdot \widehat{\sigma_\Psi}_2 = t'_2$, or that no such substitution exists, which we prove implies no global matching substitution exists.

Having now the $\widehat{\sigma_\Psi}_1$ and $\widehat{\sigma_\Psi}_2$ specified above, we consider the substitution $\widehat{\sigma_\Psi}_r = \widehat{\sigma_\Psi}_1 \circ \widehat{\sigma_\Psi}_2$. This substitution, if it exists, has the desired properties: we have that

$\widehat{\Psi}_r = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t_1\ t_2 : \lceil t_b \rceil \cdot (id_\Phi,\ t_2)) =$

$= \mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} t_1 : \Pi(t_a).t_b) \circ \mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} t_2 : t_a)$

and thus $\bullet \vdash \widehat{\sigma_\Psi}_r : \widehat{\Psi}_r$. Also, $(t_1\ t_2) \cdot \widehat{\sigma_\Psi}_r = t'_1\ t'_2$, $t_T \cdot \widehat{\sigma_\Psi}_r = t'_T$, and $\Phi \cdot \widehat{\sigma_\Psi}_r = \Phi'$. It is also unique: if another substitution had the same properties, we could limit it to either the relevant variables for t_1 or t_2 and get a contradiction. Thus this is the desired substitution. If $\widehat{\sigma_\Psi}_r$ does not exist, then no suitable substitution for unifying t and t' exists. This is again because we could limit any potential such substitution to two parts, $\widehat{\sigma_\Psi}'_1$ and $\widehat{\sigma_\Psi}'_2$ (for $\widehat{\Psi}_1$ and $\widehat{\Psi}_2$ respectively), violating the uniqueness of the substitutions yielded by the induction hypotheses.

Case PEqType**.**

$$\left(\frac{\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : t_a \qquad \Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t_a \qquad \Psi_u;\ \Phi \vdash_{\overline{p}} t_a : \mathit{Type}}{\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 = t_2 : \mathit{Prop}} \right)$$

Similarly as above. First assume that $t' = (t'_1 = t'_2)$, with $t'_1 : t'_a$, $t'_2 : t'_a$ and $t'_a : \mathit{Type}$. Then, by induction hypothesis get a matching substitution $\widehat{\sigma_\Psi}'$ for t_a and t'_a. Use that $\widehat{\sigma_\Psi}'$ in order to allude to the induction hypothesis for t_1 and t_2 independently, yielding substitutions $\widehat{\sigma_\Psi}_1$ and $\widehat{\sigma_\Psi}_2$. Last, claim that the globally required substitution must actually be equal to $\widehat{\sigma_\Psi}_r = \widehat{\sigma_\Psi}_1 \circ \widehat{\sigma_\Psi}_2$.

Case PMetaVarUnif**.**

$$\left(\frac{\Psi_u.i = T \qquad T = [\Phi_*]\, t_T \qquad (i \geq |\Psi| = 0) \qquad \Psi_u;\ \Phi \vdash_{\overline{p}} \sigma : \Phi_* \qquad \Phi_* \subseteq \Phi \qquad \sigma = id_{\Phi_*}}{\Psi_u;\ \Phi \vdash_{\overline{p}} X_i/\sigma : t_T \cdot \sigma} \right)$$

We trivially have $t_T \cdot \sigma = t_T$. We split cases depending on whether $\widehat{\sigma_\Psi}.i$ is unspecified or not.

If $\widehat{\sigma_\Psi}.\mathbf{i} =?$, then we split cases further depending on whether t' uses any variables higher than $|\Phi_* \cdot \widehat{\sigma_\Psi}| - 1$ or not. That is, if $t' <^f |\Phi_* \cdot \widehat{\sigma_\Psi}|$ or not. In the case where this does not hold, it is obvious that there is no possible $\widehat{\sigma_\Psi}'$ such that $(X_i/\sigma) \cdot \widehat{\sigma_\Psi}' = t'$, since $\widehat{\sigma_\Psi}'$ must include $\widehat{\sigma_\Psi}$, and the term $(X_i/\sigma) \cdot \widehat{\sigma_\Psi}'$ can therefore not include variables outside the prefix $\Phi_* \cdot \widehat{\sigma_\Psi}$ of $\Phi \cdot \widehat{\sigma_\Psi}$. In the case where $t' <^f |\Phi_* \cdot \widehat{\sigma_\Psi}|$, we consider the substitution

$\widehat{\sigma_\Psi}' = \widehat{\sigma_\Psi}[i \mapsto [\Phi_* \cdot \widehat{\sigma_\Psi}]\, t']$. In that case we obviously have $\Phi \cdot \widehat{\sigma_\Psi}' = \Phi'$, $t_T \cdot \widehat{\sigma_\Psi}' = t_T$, and also $t \cdot \widehat{\sigma_\Psi} = t'$. Also, $\widehat{\Psi}' = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} X_i/\sigma : t_T \cdot \sigma) = (\mathit{relevant}\,(\Psi_u;\ \Phi_* \vdash_{\overline{p}} t_T : s)) \circ \mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} \sigma : \Phi_*) \circ (\Psi \widehat{@} i)$.

We need to show that $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'$. First, we have that $\mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} \sigma : \Phi^*) = \mathit{relevant}\,(\Psi \vdash_{\overline{p}} \Phi\ \mathrm{wf})$ since $\Phi^* \subseteq \Phi$. Second, we have that $\mathit{relevant}\,(\Psi;\ \Phi \vdash_{\overline{p}} t_T : s) = (\mathit{relevant}\,(\Psi_u;\ \Phi_* \vdash_{\overline{p}} t_T : s)) \circ \mathit{relevant}\,(\Psi \vdash_{\overline{p}} \Phi\ \mathrm{wf})$. Thus we have that $\widehat{\Psi}' = \widehat{\Psi} \circ (\Psi \widehat{@} i)$. It is now trivial to see that indeed $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'$.

If $\widehat{\sigma_{\boldsymbol{\Psi}}}.\mathbf{i} = \mathbf{t}_*$, then we split cases on whether $t_* = t'$ or not. If it is, then obviously $\widehat{\sigma_\Psi}$ is the desired unifying substitution for which all the desired properties hold. If it is not, then no substitution with the desired properties possibly exists, because it would violate the uniqueness assumption for $\widehat{\sigma_\Psi}$.

Part 3 By inversion on the typing for T.

Case PExtCtxTerm**.**

$$\left(\frac{\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T \qquad \Psi_u;\ \Phi \vdash t_T : s}{\Psi_u \vdash_{\overline{p}} [\Phi]\, t : [\Phi]\, t_T}\right)$$

By inversion of typing for T' we have: $T' = [\Phi]\, t'$, $\bullet;\ \Phi \vdash_{\overline{p}} t' : t_T$, $\bullet;\ \Phi \vdash_{\overline{p}} t_T : s$.

We thus have $\widehat{\Psi}_u = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s) = \mathit{unspec}_{\Psi_u}$ (since Φ is also well-formed in the empty extension context), and the substitution $\widehat{\sigma_\Psi} = \mathit{unspec}_\Psi$ is the unique substitution such that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u$, $\Phi \cdot \widehat{\sigma_\Psi} = \Phi$ and $t_T \cdot \widehat{\sigma_\Psi} = t_T$. We can thus use part 2 for attempting a match between t and t', yielding a $\widehat{\sigma_\Psi}'$ such that $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'_u$ with $\widehat{\Psi}'_u = \mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T)$ and $t \cdot \widehat{\sigma_\Psi}' = t'$. We have that $\mathit{relevant}\,(\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T) = \mathit{relevant}\,(\Psi_u \vdash_{\overline{p}} T : K)$, thus $\widehat{\Psi}'_u = \Psi$ by assumption. From that we realize that $\widehat{\sigma_\Psi}'$ is a fully-specified substitution since $\bullet \vdash \widehat{\sigma_\Psi}' : \Psi$, and thus this is the substitution with the desired properties.

If unification between t and t' fails, it is trivial to see that no substitution with the desired substitution exists, otherwise it would lead directly to a contradiction.

Case PExtCtxInst**.**

$$\left(\frac{\Psi_u \vdash_{\overline{p}} \Phi,\ \Phi' \text{ wf}}{\Psi_u \vdash_{\overline{p}} [\Phi]\,\Phi' : [\Phi]\ ctx} \right)$$

By inversion of typing for T' we have: $T' = [\Phi]\,\Phi''$, $\bullet \vdash_{\overline{p}} \Phi,\ \Phi''$ wf, $\bullet \vdash_{\overline{p}} \Phi$ wf. From part 1 we get a $\widehat{\sigma_\Psi}$ that matches $\Phi,\ \Phi'$ with $\Phi,\ \Phi''$, or the fact that no such $\widehat{\sigma_\Psi}$ exists. In the first case, as above, it is easy to see that this is the fully-specified substitution that we desire. In the second case, no suitable substitution exists.

□

The above proof is constructive. Its computational content is a pattern matching algorithm for patterns and closed terms – the algorithm we presented earlier. Notice that the rules presented in Figures 5.10 through 5.12 follow the inductive structure of the proof and make the same assumption about types-of-types being subderivations. Based on the statement of the above theorem, it is evident that this algorithm is sound (that is, if it finds a substitution, that substitution will be a matching substitution), and that it is also complete: if a matching substitution exists, then the algorithm will find it. The formal statement of soundness and completeness is as follows.

Lemma 5.1.11 *(Pattern matching algorithm soundness and completeness)*

$$1.\ \frac{\Psi_u \vdash_{\overline{p}} \Phi\ wf \qquad \bullet \vdash \Phi'\ wf \qquad relevant\,(\Psi_u \vdash_{\overline{p}} \Phi\ wf) = \widehat{\Psi}_u \qquad (\Psi_u \vdash_{\overline{p}} \Phi\ wf) \sim \left(\bullet \vdash \Phi'\ wf\right) \triangleright \widehat{\sigma_\Psi}}{\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u \qquad \Phi \cdot \widehat{\sigma_\Psi} = \Phi'}$$

$$2.\ \frac{\Psi_u \vdash_{\overline{p}} \Phi\ wf \qquad \bullet \vdash \Phi'\ wf \qquad relevant\,(\Psi_u \vdash_{\overline{p}} \Phi\ wf) = \widehat{\Psi}_u \qquad \exists \widehat{\sigma_\Psi}.(\quad \bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u \qquad \Phi \cdot \widehat{\sigma_\Psi} = \Phi' \quad)}{(\Psi_u \vdash_{\overline{p}} \Phi\ wf) \sim \left(\bullet \vdash \Phi'\ wf\right) \triangleright \widehat{\sigma_\Psi}}$$

$$3.\ \frac{\begin{array}{c}\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T \qquad \bullet;\ \Phi' \vdash t' : t'_T \qquad \mathit{relevant}\left(\Psi_u;\ \Phi \vdash_{\overline{p}} t : t'_T\right) = \widehat{\Psi}'_u \\ \left(\begin{array}{c}\Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s \qquad \bullet;\ \Phi \vdash t'_T : s \qquad \mathit{relevant}\left(\Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s\right) = \widehat{\Psi}_u\end{array}\right) \\ \vee \left(\begin{array}{c} t_T = \mathit{Type} \qquad \Psi_u \vdash_{\overline{p}} \Phi\ \mathit{wf} \qquad \bullet \vdash \Phi\ \mathit{wf} \qquad \mathit{relevant}\left(\Psi_u \vdash_{\overline{p}} \Phi\ \mathit{wf}\right) = \widehat{\Psi}_u\end{array}\right) \\ \bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u \qquad \Phi \cdot \widehat{\sigma_\Psi} = \Phi' \qquad t_T \cdot \widehat{\sigma_\Psi} = t'_T \\ \left(\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T\right) \sim \left(\bullet;\ \Phi' \vdash t' : t'_T\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'\end{array}}{\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'_u \qquad \Phi \cdot \widehat{\sigma_\Psi}' = \Phi' \qquad t_T \cdot \widehat{\sigma_\Psi}' = t'_T \qquad t \cdot \widehat{\sigma_\Psi}' = t'}$$

$$4.\ \frac{\begin{array}{c}\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T \qquad \bullet;\ \Phi' \vdash t' : t'_T \qquad \mathit{relevant}\left(\Psi_u;\ \Phi \vdash_{\overline{p}} t : t'_T\right) = \widehat{\Psi}'_u \\ \left(\begin{array}{c}\Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s \qquad \bullet;\ \Phi \vdash t'_T : s \qquad \mathit{relevant}\left(\Psi_u;\ \Phi \vdash_{\overline{p}} t_T : s\right) = \widehat{\Psi}_u\end{array}\right) \\ \vee \left(\begin{array}{c} t_T = \mathit{Type} \qquad \Psi_u \vdash_{\overline{p}} \Phi\ \mathit{wf} \qquad \bullet \vdash \Phi\ \mathit{wf} \qquad \mathit{relevant}\left(\Psi_u \vdash_{\overline{p}} \Phi\ \mathit{wf}\right) = \widehat{\Psi}_u\end{array}\right) \\ \bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}_u \qquad \Phi \cdot \widehat{\sigma_\Psi} = \Phi' \qquad t_T \cdot \widehat{\sigma_\Psi} = t'_T \\ \exists \widehat{\sigma_\Psi}'.\left(\begin{array}{c}\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'_u \qquad \Phi \cdot \widehat{\sigma_\Psi}' = \Phi' \qquad t_T \cdot \widehat{\sigma_\Psi}' = t'_T \qquad t \cdot \widehat{\sigma_\Psi}' = t'\end{array}\right)\end{array}}{\left(\Psi_u;\ \Phi \vdash_{\overline{p}} t : t_T\right) \sim \left(\bullet;\ \Phi' \vdash t' : t'_T\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}$$

$$5.\ \frac{\begin{array}{c}\Psi_u \vdash_{\overline{p}} T : K \qquad \bullet \vdash T' : K \qquad \mathit{relevant}\left(\Psi_u \vdash_{\overline{p}} T : K\right) = \Psi_u \\ \left(\Psi_u \vdash_{\overline{p}} T : K\right) \sim \left(\bullet \vdash T' : K\right) \triangleright \widehat{\sigma_\Psi}\end{array}}{\bullet \vdash \sigma_\Psi : \Psi_u \qquad T \cdot \sigma_\Psi = T'}$$

$$6.\ \frac{\begin{array}{c}\Psi_u \vdash_{\overline{p}} T : K \qquad \bullet \vdash T' : K \qquad \mathit{relevant}\left(\Psi_u \vdash_{\overline{p}} T : K\right) = \Psi_u \\ \exists \widehat{\sigma_\Psi}.\left(\begin{array}{c}\bullet \vdash \sigma_\Psi : \Psi_u \qquad T \cdot \sigma_\Psi = T'\end{array}\right)\end{array}}{\left(\Psi_u \vdash_{\overline{p}} T : K\right) \sim \left(\bullet \vdash T' : K\right) \triangleright \widehat{\sigma_\Psi}}$$

Proof. Valid by construction. □

Practical pattern matching. We can significantly simplify the presentation of the algorithm if instead of working directly on typing derivations we work on *annotated terms* which we will define shortly. We will not do this only for presentation purposes but for efficiency considerations too: the algorithm described so far would require us to keep full typing derivations at runtime. Instead, we will extract the information required by the algorithm from the typing derivations and make them available syntactically at the level of terms. The result will be the new notion of annotated terms and an alternative formulation

$$
\begin{array}{rl}
(\textit{Annotated terms}) & t^A ::= c \mid s \mid v_L \mid b_i \mid \lambda(t_1^A).t_2^A \mid \Pi_s(t_1^A).t_2^A \\
& \quad \mid (t_1^A : t^A : s)\ t_2^A \mid t_1^A =_{t^A} t_2^A \mid X_i/\sigma \\
(\textit{Annotated contexts}) & \Phi^A ::= \bullet \mid \Phi^A,\ (t^A : s) \mid \Phi^A,\ \phi_i \\
(\textit{Annotated extension terms}) & T^A ::= [-]\, t^A \mid [-]\, \Phi^A
\end{array}
$$

Figure 5.13: Annotated λHOL terms (syntax)

of the algorithm that works directly on such terms.

We present the syntax of annotated terms and annotated contexts in Figure 5.13. The main changes are:

1. the inclusion of sort information in Π-types,

2. type and sort information for functions in function application and

3. type information for the equality type.

Furthermore, each member of the context is annotated with its sort. Last, annotated extension terms do not need the context information anymore – that information is only relevant for typing purposes. We adapt the pattern matching rules so that they work on annotated terms instead in Figure 5.14. The algorithm accepts Ψ_u as an implicit argument in order to know what the length of the resulting substitution needs to be.

A well-typed annotated term t^A is straightforward to produce from a typing derivation of a normal term t; the inverse is also true. In the rest, we will thus use the two kinds of terms interchangeably and drop the A superscript when it is clear from context which kind we mean. For example, we will use the algorithm of Figure 5.14 algorithm directly to decide pattern matching between a well-typed pattern t and a scrutinee t'. It is understood that these have been converted to annotated terms after typing as needed. We give the exact rules for the conversion in Figure 5.15 as a type derivation-directed translation function.

We prove the following lemma about the correspondence between the two algorithms working on derivations and annotated terms; then soundness and completeness for the new algorithm follows directly.

$\boxed{T \sim T' \triangleright \widehat{\sigma_\Psi}}$

$$\frac{t \sim t' \triangleleft \mathit{unspec}_{\Psi_u} \triangleright \widehat{\sigma_\Psi}}{[-]\, t \sim [-]\, t' \triangleright \widehat{\sigma_\Psi}} \qquad \frac{\Phi' \sim \Phi'' \triangleright \widehat{\sigma_\Psi}}{[-]\, \Phi' \sim [-]\, \Phi'' \triangleright \widehat{\sigma_\Psi}}$$

$\boxed{\Phi' \sim \Phi'' \triangleright \widehat{\sigma_\Psi}}$

$$\frac{}{\bullet \sim \bullet \triangleright \mathit{unspec}_{\Psi_u}} \qquad \frac{s = s' \qquad \Phi \sim \Phi' \triangleright \widehat{\sigma_\Psi} \qquad t \sim t' \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}{(\Phi,\ (t : s)) \sim (\Phi',\ (t' : s')) \triangleright \widehat{\sigma_\Psi}'}$$

$$\frac{}{(\Phi,\ \phi_i) \sim \big(\Phi, \Phi'\big) \triangleright \mathit{unspec}_{\Psi_u}[i \mapsto [-]\, \Phi']}$$

$\boxed{t \sim t' \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}$

$$\frac{}{c \sim c \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}} \qquad \frac{}{s \sim s \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}} \qquad \frac{L \cdot \widehat{\sigma_\Psi} = L'}{v_L \sim v'_L \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}}$$

$$\frac{s = s' \qquad t_1 \sim t'_1 \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}' \qquad \lceil t_2 \rceil \sim \lceil t'_2 \rceil \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_\Psi}''}{\Pi_s(t_1).t_2 \sim \Pi_{s'}(t'_1).t'_2 \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}''} \qquad \frac{\lceil t_2 \rceil \sim \lceil t'_2 \rceil \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}{\lambda(t_1).t_2 \sim \lambda(t'_1).t'_2 \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}$$

$$\frac{\begin{array}{c} s = s' \qquad t \sim t' \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}' \\ t_1 \sim t'_1 \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_\Psi}_1 \qquad t_2 \sim t'_2 \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_\Psi}_2 \qquad \widehat{\sigma_\Psi}_1 \circ \widehat{\sigma_\Psi}_2 = \widehat{\sigma_\Psi}'' \end{array}}{((t_1 : t : s)\ t_2) \sim \big((t'_1 : t' : s')\ t'_2\big) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}''}$$

$$\frac{t \sim t' \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}' \qquad t_1 \sim t'_1 \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_\Psi}_1 \qquad t_2 \sim t'_2 \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_\Psi}_2 \qquad \widehat{\sigma_\Psi}_1 \circ \widehat{\sigma_\Psi}_2 = \widehat{\sigma_\Psi}''}{(t_1 =_t t_2) \sim \big(t'_1 =_{t'} t'_2\big) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}''}$$

$$\frac{\widehat{\sigma_\Psi}.i = ? \qquad t' <^f |\sigma \cdot \widehat{\sigma_\Psi}|}{X_i/\sigma \sim t' \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}[i \mapsto [-]\, t']} \qquad \frac{\widehat{\sigma_\Psi}.i = t'}{X_i/\sigma \sim t' \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}}$$

Figure 5.14: Pattern matching algorithm for λHOL, operating on annotated terms

$\boxed{[\![\bullet \vdash T : K]\!]^A = T^A}$

$$\frac{[\![\Psi;\ \Phi \vdash t : t_T]\!]^A = t^A}{[\![\Psi \vdash [\Phi]\, t : [\Phi]\, t_T]\!]^A = [-]\, t^A} \qquad \frac{\left[\!\left[\Psi \vdash \Phi,\ \Phi'\ \mathrm{wf}\right]\!\right]^A = \Phi^A,\ \Phi'^A}{\left[\!\left[\Psi \vdash [\Phi]\, \Phi' : [\Phi]\, ctx\right]\!\right]^A = [-]\, \Phi'^A}$$

$\boxed{[\![\bullet \vdash \Phi\ \mathrm{wf}]\!]^A = \Phi^A}$

$$\frac{}{[\![\Psi \vdash \bullet\ \mathrm{wf}]\!]^A = \bullet} \qquad \frac{[\![\Psi \vdash \Phi\ \mathrm{wf}]\!]^A = \Phi^A \qquad [\![\Psi;\ \Phi \vdash t : s]\!]^A = t^A}{[\![\Psi \vdash (\Phi,\ t)\ \mathrm{wf}]\!]^A = (\Phi^A,\ (t^A : s))}$$

$$\frac{[\![\Psi \vdash \Phi\ \mathrm{wf}]\!]^A = \Phi^A}{[\![\Psi \vdash \Phi,\ \phi_i\ \mathrm{wf}]\!]^A = (\Phi^A,\ \phi_i)}$$

$\boxed{[\![\Psi;\ \Phi \vdash t : t']\!]^A = t^A}$

$$[\![\Psi;\ \Phi \vdash c : t]\!]^A = c \qquad \left[\!\left[\Psi;\ \Phi \vdash_{\overline{p}} s : s'\right]\!\right]^A = s \qquad [\![\Psi;\ \Phi \vdash_{\overline{p}} v_L : t]\!]^A = v_L$$

$$\frac{[\![\Psi;\ \Phi \vdash_{\overline{p}} t_1 : s]\!]^A = t_1^A \qquad \left[\!\left[\Psi;\ \Phi,\ t_1 \vdash_{\overline{p}} \lceil t_2 \rceil_{|\Phi|} : s'\right]\!\right]^A = \lceil t_2^A \rceil}{\left[\!\left[\Psi;\ \Phi \vdash \Pi(t_1).t_2 : s''\right]\!\right]^A = \Pi_s(t_1^A).t_2^A}$$

$$\frac{[\![\Psi;\ \Phi \vdash t_1 : s]\!]^A = t_1^A \qquad \left[\!\left[\Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t'\right]\!\right]^A = \lceil t_2 \rceil^A}{\left[\!\left[\Psi;\ \Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}\right]\!\right]^A = \lambda(t_1^A).t_2^A}$$

$$\frac{[\![\Psi;\ \Phi \vdash t_1 : \Pi(t_a).t_b]\!]^A = t_1^A \qquad [\![\Psi;\ \Phi \vdash t_2 : t_a]\!]^A = t_2^A \qquad [\![\Psi;\ \Phi \vdash \Pi(t_a).t_b : s]\!]^A = t^A}{\left[\!\left[\Psi;\ \Phi \vdash t_1\ t_2 : \lceil t_b \rceil_{|\Phi|} \cdot (id_\Phi, t_2)\right]\!\right]^A = (t_1^A : t^A : s)\ t_2^A}$$

$$\frac{[\![\Psi_u;\ \Phi \vdash_{\overline{p}} t : \mathit{Type}]\!]^A = t^A \qquad [\![\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 : t]\!]^A = t_1^A \qquad [\![\Psi_u;\ \Phi \vdash_{\overline{p}} t_2 : t]\!]^A = t_2^A}{[\![\Psi_u;\ \Phi \vdash_{\overline{p}} t_1 = t_2 : \mathit{Prop}]\!]^A = (t_1^A =_{t^A} t_2^A)}$$

$$[\![\Psi_u;\ \Phi \vdash_{\overline{p}} X_i/\sigma : t_T \cdot \sigma]\!]^A = X_i/\sigma$$

Figure 5.15: Conversion of typing derivations to annotated terms

Lemma 5.1.12 *(Correspondence between pattern matching algorithms)*

$$1.\ \frac{(\Psi_u \vdash_{\overline{p}} T : K) \sim \big(\bullet \vdash T' : K\big) \triangleright \widehat{\sigma_\Psi} \qquad [\![\Psi_u \vdash T : K]\!]^A = T^A \qquad \big[\!\big[\bullet \vdash T' : K\big]\!\big]^A = T'^A}{T^A \sim T'^A \triangleright [\![\bullet \vdash \widehat{\sigma_\Psi} : \Psi_u]\!]^A}$$

$$2.\ \frac{[\![\Psi_u \vdash T : K]\!]^A = T^A \qquad \big[\!\big[\bullet \vdash T' : K\big]\!\big]^A = T'^A \qquad T^A \sim T'^A \triangleright \widehat{\sigma_\Psi}^A}{\exists \widehat{\sigma_\Psi}.(\quad [\![\bullet \vdash \widehat{\sigma_\Psi} : \Psi_u]\!]^A = \widehat{\sigma_\Psi}^A \qquad (\Psi_u \vdash_{\overline{p}} T : K) \sim \big(\bullet \vdash T' : K\big) \triangleright \widehat{\sigma_\Psi} \quad)}$$

$$3.\ \frac{(\Psi_u \vdash_{\overline{p}} \Phi\ wf) \sim \big(\bullet \vdash \Phi'\ wf\big) \triangleright \widehat{\sigma_\Psi} \qquad [\![\Psi_u \vdash \Phi\ wf]\!]^A = \Phi^A \qquad \big[\!\big[\bullet \vdash \Phi'\ wf\big]\!\big]^A = \Phi'^A}{\Phi^A \sim \Phi'^A \triangleright [\![\bullet \vdash \widehat{\sigma_\Psi} : \Psi_u]\!]^A}$$

$$4.\ \frac{[\![\Psi_u \vdash \Phi\ wf]\!]^A = \Phi^A \qquad \big[\!\big[\bullet \vdash \Phi'\ wf\big]\!\big]^A = \Phi'^A \qquad \Phi^A \sim \Phi'^A = \widehat{\sigma_\Psi}^A}{\exists \widehat{\sigma_\Psi}.(\quad [\![\bullet \vdash \widehat{\sigma_\Psi} : \Psi_u]\!]^A \triangleright \widehat{\sigma_\Psi}^A \qquad (\Psi_u \vdash_{\overline{p}} \Phi\ wf) \sim \big(\bullet \vdash \Phi'\ wf\big) \triangleright \widehat{\sigma_\Psi} \quad)}$$

$$5.\ \frac{\begin{array}{c}(\Psi_u;\ \Phi \vdash t : t_T) \sim \big(\bullet;\ \Phi' \vdash t' : t'_T\big) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}' \\ [\![\Psi_u;\ \Phi \vdash t : t_T]\!]^A = t^A \qquad [\![\bullet;\ \Phi' \vdash t' : t'_T]\!]^A = t'^A \qquad \left[\!\!\left[\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}\right]\!\!\right]^A = \widehat{\sigma_\Psi}^A\end{array}}{t^A \sim t'^A \triangleleft \widehat{\sigma_\Psi}^A \triangleright \left[\!\!\left[\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'\right]\!\!\right]^A}$$

$$6.\ \frac{\begin{array}{c}[\![\Psi_u;\ \Phi \vdash t : t_T]\!]^A = t^A \\ [\![\bullet;\ \Phi' \vdash t' : t'_T]\!]^A = t'^A \qquad \left[\!\!\left[\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}\right]\!\!\right]^A = \widehat{\sigma_\Psi}^A \qquad t^A \sim t'^A \triangleleft \widehat{\sigma_\Psi}^A \triangleright \widehat{\sigma_\Psi}'^A\end{array}}{\exists \widehat{\sigma_\Psi}.(\quad \left[\!\!\left[\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'\right]\!\!\right]^A \triangleright \widehat{\sigma_\Psi}'^A \qquad (\Psi_u;\ \Phi \vdash t : t_T) \sim \big(\bullet;\ \Phi' \vdash t' : t'_T\big) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}' \quad)}$$

Proof. By structural induction on the derivation of the pattern matching result. We also need to use Lemma 5.1.1. □

Lemma 5.1.13 *(Soundness and completeness for practical pattern matching algorithm)*

$$1.\ \frac{\begin{array}{c}\Psi_u \vdash_{\overline{p}} T : K \qquad [\![\Psi_u \vdash T : K]\!]^A = T^A \\ \bullet \vdash T' : K \qquad \big[\!\big[\bullet \vdash T' : K\big]\!\big]^A = T'^A \qquad \mathit{relevant}\,(\Psi_u \vdash_{\overline{p}} T : K) = \Psi_u \qquad T \sim T' \triangleright \sigma_\Psi^A\end{array}}{\exists \sigma_\Psi.(\quad \bullet \vdash \sigma_\Psi : \Psi_u \qquad [\![\bullet \vdash \sigma_\Psi : \Psi_u]\!]^A = \sigma_\Psi^A \qquad T \cdot \sigma_\Psi = T' \quad)}$$

$$2.\ \frac{\begin{array}{c}\Psi_u \Vdash_{\overline{p}} T : K \qquad [\![\Psi_u \vdash T : K]\!]^A = T^A \\ \bullet \vdash T' : K \qquad [\![\bullet \vdash T' : K]\!]^A = T'^A \qquad \mathit{relevant}(\Psi_u \Vdash_{\overline{p}} T : K) = \Psi_u \\ \exists \sigma_\Psi.(\quad \bullet \vdash \sigma_\Psi : \Psi_u \qquad T \cdot \sigma_\Psi = T' \quad)\end{array}}{T \sim T' \triangleright [\![\bullet \vdash \sigma_\Psi : \Psi_u]\!]^A}$$

Proof. By combining Lemma 5.1.11, Lemma 5.1.12 and Lemma 5.1.1. □

Summary

Let us briefly summarize the results of this section. For the typing judgement we defined in the end of Section 5.1, as follows:

$$\frac{\Psi \Vdash_{\overline{p}} \Psi_u \text{ wf} \qquad \Psi,\ \Psi_u \Vdash_{\overline{p}} T_P : K \qquad \mathit{unspec}_\Psi, \Psi_u \sqsubseteq \mathit{relevant}\,(\Psi,\ \Psi_u \Vdash_{\overline{p}} T_P : K)}{\Psi \Vdash^*_{\overline{p}} \Psi_u > T_P : K}$$

we have the following results:

$$\frac{\Psi \Vdash^*_{\overline{p}} \Psi_u > T_P : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \Vdash^*_{\overline{p}} \Psi_u \cdot \sigma_\Psi > T_P \cdot \sigma_\Psi : K \cdot \sigma_\Psi}\ \textbf{Substitution lemma}$$

$$\frac{\bullet \Vdash^*_{\overline{p}} \Psi_u > T_P : K \qquad \bullet \vdash T : K}{\exists^{\leq 1} \sigma_\Psi.(\quad \bullet \vdash \sigma_\Psi : \Psi_u \qquad T_P \cdot \sigma_\Psi = T \quad)}\ \textbf{Pattern matching determinism and decidability}$$

$$\frac{\bullet \Vdash^*_{\overline{p}} \Psi_u > T_P : K \qquad \bullet \vdash T : K \qquad T_P \sim T = \sigma_\Psi}{\bullet \vdash \sigma_\Psi : \Psi_u \qquad T_P \cdot \sigma_\Psi = T}\ \textbf{Pattern matching algorithm soundness}$$

$$\frac{\begin{array}{c}\bullet \Vdash^*_{\overline{p}} \Psi_u > T_P : K \qquad \bullet \vdash T : K \\ \exists \sigma_\Psi.(\quad \bullet \vdash \sigma_\Psi : \Psi_u \qquad T_P \cdot \sigma_\Psi = T \quad)\end{array}}{T_P \sim T = \sigma_\Psi}\ \textbf{Pattern matching algorithm completeness}$$

5.2 Collapsing extension variables

Let us consider the following situation. We are writing a tactic such as the simplify tactic given in Section 2.3 under the heading "Staging", a sketch of which is (using the new

notations for extension types):

$$\begin{array}{lcl} \mathsf{simplify} & : & (\phi : ctx) \to (P : [\phi]\, Prop) \to (Q : [\phi]\, Prop,\ [\phi]\, P \supset Q \wedge Q \supset P) \\ \mathsf{simplify}\ P & = & \mathsf{match}\ P\ \mathsf{with} \\ & & Q \wedge R \supset O \ \mapsto\ \langle Q \supset R \supset O, \cdots \rangle \end{array}$$

In order to fill the missing proof object (denoted as $\cdots$), we need to perform pattern matching on its inferred type: the open proposition

$$[\phi]\,((Q \wedge R \supset O) \supset (Q \supset R \supset O)) \wedge ((Q \supset R \supset O) \supset (Q \wedge R \supset O))$$

in the extension context Ψ that includes ϕ, P, Q, R, O. Yet the pattern matching procedure presented in the previous section is only applicable when the extension context is empty. Instead of changing our pattern matching procedure, we notice that we can prove an equivalent first-order lemma instead where only normal logical variables are used (and is therefore closed with respect to Ψ):

$$[P : Prop, Q : Prop, R : Prop, O : Prop]\,((Q \wedge R \supset O) \supset (Q \supset R \supset O)) \wedge ((Q \supset R \supset O) \supset (Q \wedge R \supset O))$$

Then, by instantiating this lemma with the variables of the open Ψ at hand, we can get a proof of the desired type. But is there always such an equivalent first-order lemma in the general case of extension contexts Ψ and can we find it if one exists? This will be the subject of this section.

More formally, we want to produce a proof object for a proposition P at a point where the extension variables context is Ψ. If we want to discover this proof object programmatically, we need to proceed by pattern matching on P (as the scrutinee). A problem arises: P is not closed, so the pattern matching methodology presented in the previous section does not apply – since only patterns are allowed to use extension variables (as unification variables). That is, we need to solve the following problem:

$$\text{If } \Psi,\ \Psi_u \vdash_{\overline{p}} P_P : [\Phi]\, Prop \text{ and } \Psi \vdash P : [\Phi]\, Prop,$$

$$\text{then there exists } \sigma_\Psi \text{ such that } \Psi \vdash \sigma_\Psi : (\Psi,\ \Psi_u) \text{ and } P_P \cdot \sigma_\Psi = P.$$

This is a significantly more complicated pattern matching problem, because P also contains meta-variables. To solve it properly, we need to be able to have unification variables that match against meta-variables; these unification variables will then be meta-2-variables.

In this section we will partially address this issue in a different way. We will show that under certain restrictions about the current non-empty extension context Ψ and its usage, we can *collapse* terms from Ψ to an empty extension context. A substitution exists that then transforms the collapsed terms into terms of the original Ψ context. That is, we can get an equivalent term that is closed with respect to the extension context and use our normal pattern matching procedure on that.

We will thus prove the following theorem for contextual terms:

$$\frac{\Psi;\ \Phi \vdash t : t' \qquad \mathit{collapsable}\left(\Psi \vdash [\Phi]\, t : [\Phi]\, t'\right)}{\exists t_{\downarrow}, t'_{\downarrow}, \Phi_{\downarrow}, \sigma.(\quad \bullet;\ \Phi_{\downarrow} \vdash t_{\downarrow} : t'_{\downarrow} \qquad \Psi;\ \Phi \vdash \sigma : \Phi_{\downarrow} \qquad t_{\downarrow} \cdot \sigma = t \qquad t'_{\downarrow} \cdot \sigma = t' \quad)}$$

Combined with the pattern matching algorithm given in the previous section, which applies when $\Psi = \bullet$, this will give us a way to perform pattern matching even when Ψ is not empty under the restrictions that we will express as the "collapsable" operation.

Let us now see the details of the limitations in the applicability of our solution. Intuitively, we will be able to collapse the extension variable context when all extension variables depend on contexts Φ which are prefixes of some Φ' context. The collapsable operation is actually a function that discovers such a context, if it exists. Furthermore, all uses of extension variables should be used only with identity substitutions. This restriction exists because there is no generic way of encoding the equalities that are produced by non-identity substitutions. We present the full details of the collapsable function in Figures 5.16 and 5.17.

The actual collapsing transformation amounts to replacing all instances of meta-variables with normal variables, instantiating all context variables with the empty context, and renaming normal variables appropriately, so that both meta-variables and normal variables can refer to one identical Φ context. We can define this transformation through a series of substitutions and meta-substitutions. Before we proceed to see the details of the proof through which the transformation is defined, we need two auxiliary lemmas.

$\boxed{\mathit{collapsable}\ (\Psi) \lhd \Phi \rhd \Phi'}$

$$\frac{}{\mathit{collapsable}\ (\bullet) \lhd \Phi \rhd \Phi} \qquad \frac{\mathit{collapsable}\ (\Psi) \lhd \Phi \rhd \Phi' \qquad \mathit{collapsable}\ (K) \lhd \Phi' \rhd \Phi''}{\mathit{collapsable}\ (\Psi,\ K) \lhd \Phi \rhd \Phi''}$$

$\boxed{\mathit{collapsable}\ (K) \lhd \Phi' \rhd \Phi''}$

$$\frac{T = [\Phi]\, t \qquad \mathit{collapsable}\ (T) \lhd \Phi' \rhd \Phi''}{\mathit{collapsable}\ ([\Phi]\, t) \lhd \Phi' \rhd \Phi''} \qquad \frac{\mathit{collapsable}\ (\Phi) \lhd \Phi' \rhd \Phi''}{\mathit{collapsable}\ ([\Phi]\ \mathit{ctx}) \lhd \Phi' \rhd \Phi''}$$

$\boxed{\mathit{collapsable}\ (T) \lhd \Phi' \rhd \Phi''}$

$$\frac{\mathit{collapsable}\ (\Phi) \lhd \Phi' \rhd \Phi'' \qquad \mathit{collapsable}\ (t) \lhd \Phi''}{\mathit{collapsable}\ ([\Phi]\, t) \lhd \Phi' \rhd \Phi''} \qquad \frac{\mathit{collapsable}\ (\Phi_1,\ \Phi_2) \lhd \Phi' \rhd \Phi''}{\mathit{collapsable}\ ([\Phi_1]\, \Phi_2) \lhd \Phi' \rhd \Phi''}$$

$\boxed{\mathit{collapsable}\ (\Psi \vdash T : K) \rhd \Phi'}$

$$\frac{\mathit{collapsable}\ (\Psi) \lhd \bullet \rhd \Phi' \qquad \mathit{collapsable}\ (K) \lhd \Phi' \rhd \Phi'' \qquad \mathit{collapsable}\ (T) \lhd \Phi'' \rhd \Phi''}{\mathit{collapsable}\ (\Psi \vdash T : K) \rhd \Phi''}$$

$\boxed{\mathit{collapsable}\ (\Psi \vdash \Phi\ \mathrm{wf}) \rhd \Phi'}$

$$\frac{\mathit{collapsable}\ (\Psi) \lhd \bullet \rhd \Phi' \qquad \mathit{collapsable}\ (\Phi) \lhd \Phi' \rhd \Phi''}{\mathit{collapsable}\ (\Psi \vdash \Phi\ \mathrm{wf}) \rhd \Phi''}$$

Figure 5.16: Operation to decide whether λHOL terms are collapsable with respect to the extension context

$\boxed{\mathit{collapsable}\ (\Phi) \lhd \Phi' \rhd \Phi''}$

$$\frac{}{\mathit{collapsable}\ (\bullet) \lhd \Phi' \rhd \Phi'} \qquad \frac{\mathit{collapsable}\ (\Phi) \lhd \Phi' \rhd \Phi'' \qquad \Phi = \Phi'' \qquad \mathit{collapsable}\ (t) \lhd \Phi}{\mathit{collapsable}\ (\Phi,\ t) \lhd \Phi' \rhd (\Phi,\ t)}$$

$$\frac{\mathit{collapsable}\ (\Phi) \lhd \Phi' \rhd \Phi'' \qquad \Phi \subset \Phi'' \qquad \mathit{collapsable}\ (t) \lhd \Phi''}{\mathit{collapsable}\ (\Phi,\ t) \lhd \Phi' \rhd \Phi''}$$

$$\frac{\mathit{collapsable}\ (\Phi) \lhd \Phi' \rhd \Phi'' \qquad \Phi = \Phi''}{\mathit{collapsable}\ (\Phi,\ \phi_i) \lhd \Phi' \rhd (\Phi,\ \phi_i)} \qquad \frac{\mathit{collapsable}\ (\Phi) \lhd \Phi' \rhd \Phi'' \qquad \Phi \subset \Phi''}{\mathit{collapsable}\ (\Phi,\ \phi_i) \lhd \Phi' \rhd \Phi''}$$

$\boxed{\mathit{collapsable}\ (t) \lhd \Phi'}$

$$\frac{}{\mathit{collapsable}\ (s) \lhd \Phi'} \qquad \frac{}{\mathit{collapsable}\ (c) \lhd \Phi'} \qquad \frac{}{\mathit{collapsable}\ (v_L) \lhd \Phi'}$$

$$\frac{\mathit{collapsable}\ (t_1) \lhd \Phi' \qquad \mathit{collapsable}\ (t_2) \lhd \Phi'}{\mathit{collapsable}\ (\lambda(t_1).t_2) \lhd \Phi'} \qquad \frac{\mathit{collapsable}\ (t_1) \lhd \Phi' \qquad \mathit{collapsable}\ (t_2) \lhd \Phi'}{\mathit{collapsable}\ (\Pi(t_1).t_2) \lhd \Phi'}$$

$$\frac{\mathit{collapsable}\ (t_1) \lhd \Phi' \qquad \mathit{collapsable}\ (t_2) \lhd \Phi'}{\mathit{collapsable}\ (t_1\ t_2) \lhd \Phi'}$$

$$\frac{\mathit{collapsable}\ (t_1) \lhd \Phi' \qquad \mathit{collapsable}\ (t_2) \lhd \Phi'}{\mathit{collapsable}\ (t_1 = t_2) \lhd \Phi'} \qquad \frac{\sigma \subseteq id_\Phi}{\mathit{collapsable}\ (X_i/\sigma) \lhd \Phi'}$$

Figure 5.17: Operation to decide whether λHOL terms are collapsable with respect to the extension context (continued)

Lemma 5.2.1

$$1.\ \frac{\mathit{collapsable}\,(\Phi) \lhd \Phi' \rhd \Phi''}{(\quad \Phi' \subseteq \Phi \qquad \Phi'' = \Phi \quad) \ \vee \ (\quad \Phi \subseteq \Phi' \qquad \Phi'' = \Phi' \quad)}$$

$$2.\ \frac{\mathit{collapsable}\,(\Psi \vdash [\Phi]\, t : [\Phi]\, t_T) \rhd \Phi'}{\Phi \subseteq \Phi'} \qquad 3.\ \frac{\mathit{collapsable}\,(\Psi \vdash [\Phi_0]\, \Phi_1 : [\Phi_0]\, \Phi_1) \rhd \Phi'}{\Phi_0, \Phi_1 \subseteq \Phi'}$$

Proof. By structural induction on the derivation of $\mathit{collapsable}\,(\cdot)$. □

Lemma 5.2.2

$$1.\ \frac{\mathit{collapsable}\,(\Psi) \lhd \bullet \rhd \Phi \qquad \Phi \subseteq \Phi'}{\mathit{collapsable}\,(\Psi) \lhd \Phi' \rhd \Phi'} \qquad 2.\ \frac{\mathit{collapsable}\,(K) \lhd \bullet \rhd \Phi \qquad \Phi \subseteq \Phi'}{\mathit{collapsable}\,(K) \lhd \Phi' \rhd \Phi'}$$

$$3.\ \frac{\mathit{collapsable}\,(T) \lhd \bullet \rhd \Phi \qquad \Phi \subseteq \Phi'}{\mathit{collapsable}\,(T) \lhd \Phi' \rhd \Phi'} \qquad 4.\ \frac{\mathit{collapsable}\,(\Phi_0) \lhd \bullet \rhd \Phi \qquad \Phi \subseteq \Phi'}{\mathit{collapsable}\,(\Phi_0) \lhd \Phi' \rhd \Phi'}$$

Proof. By structural induction on the derivation of $\mathit{collapsable}\,(\cdot)$. □

We are now ready to state and prove the main result of this section. Both the statement and proof are technically involved, so we will give a high-level picture prior to the actual proof. The main theorem that we will use later is a simple corollary of this result.

Theorem 5.2.3

$$\frac{\vdash \Psi \ wf \qquad collapsable\,(\Psi) \lhd \bullet \rhd \Phi^0}{\begin{array}{c}\exists \Psi^1, \sigma^1_\Psi, \sigma^2_\Psi, \Phi^1, \sigma^1, \sigma^{-1}, \sigma^{inv}. \\ \Psi^1 \vdash \sigma^1_\Psi : \Psi \qquad \bullet \vdash \sigma^2_\Psi : \Psi^1 \qquad \Psi^1 \vdash \Phi^1 \ wf \qquad \Psi^1;\ \Phi^1 \vdash \sigma^1 : \Phi^0 \cdot \sigma^1_\Psi \\ \Psi^1;\ \Phi^0 \cdot \sigma^1_\Psi \vdash \sigma^{-1} : \Phi^1 \qquad \sigma^1 \cdot \sigma^{-1} = id_{\Phi^0 \cdot \sigma^1_\Psi} \qquad \Psi;\ \Phi^0 \vdash \sigma^{inv} : \Phi^1 \cdot \sigma^2_\Psi \\ \forall t. \Psi;\ \Phi^0 \vdash t : t' \Rightarrow t \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = t \qquad \forall T \in \Psi^1. (T = [\Phi^*]\, t \wedge \Phi^* \subseteq \Phi^1)\end{array}}$$

Explanation of involved terms. The main idea of the proof is to maintain a number of substitutions to transform a term from the extension context Ψ to the empty context, by a series of context changes. We assume that the collapsable operation has established that all extension variables depend on prefixes of the context Φ^0. For example, this operation will determine Φ^0 for the extension context $\Psi = \phi : ctx, A : [\phi]\, Nat, B : [\phi, x : Nat]\, Nat, T : [\phi]\, Type, t : [\phi]\, T$ to be $\Phi^0 = \phi,\ x^0 : Nat$. Then, the substitutions and contexts defined by the theorem are:

Φ^1 is a context that is composed of 'collapsed' versions of all extension variables, as well as any normal variables required to type them. For the above example, this would be determined as $\Phi^1 = A^1 : Nat,\ x^1 : Nat,\ B^1 : Nat,\ T^1 : Type,\ t^1 : T^1$.

Ψ^1 is an extension context where all extension variables depend on the Φ^1 context. It has the same metavariables as Ψ, with the difference that all of them use prefixes of the Φ^1 context and dependencies on previous meta-variables are changed to dependencies on the Φ^1 context. Context variables are also elided. For the above example context we will therefore have $\Psi^1 = A' : [\Phi^1]\, Nat, B' : [\Phi^1]\, Nat, T' : [\Phi^1]\, Type, t' : [\Phi^1]\, T^1$, where we use primed names for meta-variables to distinguish them from the normal ones of Ψ.

σ^1_Ψ can be understood as a renaming of metavariables from the original Ψ context to the Ψ^1 context. Since the two contexts have essentially the same metavariables, the renaming is simple to carry out. Context variables are substituted with the empty context. This is fine since parametric contexts carry no information. For the example context above, we will have that $\sigma^1_\Psi = \phi \mapsto \bullet, A \mapsto A', B \mapsto B', T \mapsto T', t \mapsto t'$

σ^1 can be understood as a permutation from variables in Φ^0 to variables in Φ^1. The need for it arises from the fact that collapsed versions of metavariables get interspersed with normal variables in Φ^1. σ^{-1} is the permutation for the inverse direction, that is from variables in Φ^1 to variables in Φ^0. For the example, we have:

$\sigma^1 = x^0 \mapsto x^1$

$\sigma^{-1} = A^1 \mapsto A',\ x^1 \mapsto x^0,\ B^1 \mapsto B',\ T^1 \mapsto T',\ t^1 \mapsto t'$

σ^2_Ψ carries out the last step, which is to replace the extension variables from Ψ^1 with their collapsed versions in the Φ^1 context. In the example, we would have that $\sigma^2_\Psi = A' \mapsto [\Phi^1]\, A^1,\ B' \mapsto [\Phi^1]\, B^1,\ T' \mapsto [\Phi^1]\, T^1,\ t' \mapsto [\Phi^1]\, t^1$.

σ^{inv} is the substitution that inverses all the others, yielding a term in the original Ψ extension context from a closed collapsed term. In the example this would be $\sigma^{inv} = A^1 \mapsto A,\ x^1 \mapsto x^0,\ B^1 \mapsto B,\ T^1 \mapsto T,\ t^1 \mapsto t$.

Proof. By induction on the derivation of the relation $\mathit{collapsable}\,(\Psi) \triangleleft \bullet \triangleright \Phi^0$.

Case $\Psi = \bullet$. We choose $\Psi^1 = \bullet$; $\sigma^1_\Psi = \sigma^2_\Psi = \bullet$; $\Phi^1 = \bullet$; $\sigma^1 = \sigma^{-1} = \bullet$; $\sigma^{inv} = \bullet$ $\Phi^1 = \bullet$ and the desired trivially hold.

Case $\Psi = \Psi',\ [\Phi]\ \mathit{ctx}$. From the collapsable relation, we get: $\mathit{collapsable}\,(\Psi') \triangleleft \bullet \triangleright \Phi'^0$, $\mathit{collapsable}\,([\Phi]\ \mathit{ctx}) \triangleleft \Phi'^0 \triangleright \Phi^0$. By induction hypothesis for Ψ', get:

$\Psi'^1 \vdash \sigma'^1_\Psi : \Psi' \qquad \bullet \vdash \sigma'^2_\Psi : \Psi'^1 \qquad \Psi'^1 \vdash \Phi'^1$ wf $\qquad \Psi'^1;\ \Phi'^1 \vdash \sigma'^1 : \Phi'^0 \cdot \sigma'^1_\Psi$

$\Psi'^1;\ \Phi'^0 \cdot \sigma'^1_\Psi \vdash \sigma'^{-1} : \Phi'^1 \qquad \sigma'^1 \cdot \sigma'^{-1} = id_{\Phi'^0 \cdot \sigma'^1_\Psi} \qquad \Psi';\ \Phi'^0 \vdash \sigma'^{inv} : \Phi'^1 \cdot \sigma'^2_\Psi$

$\forall t.\Psi';\ \Phi'^0 \vdash t : t' \Rightarrow t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma'^2_\Psi \cdot \sigma'^{inv} = t \qquad \forall T \in \Psi'^1.T = [\Phi^*]\, t \wedge \Phi^* \subseteq \Phi'^1$

By inversion of typing for $[\Phi]\ \mathit{ctx}$ we get that $\Psi' \vdash \Phi$ wf.

We fix $\sigma^1_\Psi = \sigma'^1_\Psi,\ [\Phi'^0 \cdot \sigma'^1_\Psi] \bullet$ which is a valid choice as long as we select Ψ^1 so that $\Psi'^1 \subseteq \Psi^1$. This substitution has correct type by taking into account the substitution lemma for Φ'^0 and σ'^1_Ψ.

For choosing the rest, we proceed by induction on the derivation of $\Phi'^0 \subseteq \Phi^0$.

If $\Phi^0 = \Phi'^0$ then:

We have $\Phi \subseteq \Phi'^0$ because of Lemma 5.2.1.

Choose $\Psi^1 = \Psi'^1$; $\sigma^2_\Psi = \sigma'^2_\Psi$; $\Phi^1 = \Phi'^1$; $\sigma^1 = \sigma'^1$; $\sigma^{-1} = \sigma'^{-1}$; $\sigma^{inv} = \sigma'^{inv}$.

Everything holds trivially, other than σ^1_Ψ typing. This too is easy to prove by taking into account the substitution lemma for Φ and σ'^1_Ψ. Also, σ'^{inv} typing uses extension variable weakening. Last, for the cancellation part, terms that are typed under Ψ are also typed under Ψ' so this part is trivial too.

If $\Phi^0 = \Phi'^0,\ t$ then: (here we abuse notation slightly by identifying the context and substitutions from induction hypothesis with the ones we already have: their properties are the same for the new Φ'^0)

We have $\Phi = \Phi^0 = \Phi'^0,\ t$ because of Lemma 5.2.1 (Φ^0 is not Φ'^0 thus $\Phi^0 = \Phi$).

First, choose $\Phi^1 = \Phi'^1,\ t \cdot \sigma'^1_\Psi \cdot \sigma'^1$. This is a valid choice, because $\Psi';\ \Phi'^0 \vdash t : s$; by applying σ'^1_Ψ we get $\Psi'^1;\ \Phi'^0 \cdot \sigma'^1_\Psi \vdash t \cdot \sigma'^1_\Psi : s$; by applying σ'^1 we get $\Psi'^1;\ \Phi'^1 \vdash t \cdot \sigma'^1_\Psi \cdot \sigma'^1 : s$.

Thus $\Psi'^1 \vdash \Phi'^1,\ t \cdot \sigma'^1_\Psi \cdot \sigma'^1$ wf.

Now, choose $\Psi^1 = \Psi'^1,\ [\Phi_1]\, t \cdot \sigma'^1_\Psi \cdot \sigma'^1$. This is well-formed because of what we proved above about the substituted t, taking weakening into account. Also, the condition for the contexts in Ψ^1 being subcontexts of Φ^1 obviously holds.

Choose $\sigma^2_\Psi = \sigma'^2_\Psi,\ [\Phi_1]\, v_{|\Phi'^1|}$. We have $\bullet \vdash \sigma^2_\Psi : \Psi^1$ directly by our construction.

Choose $\sigma^1 = \sigma'^1,\ v_{|\Phi'^1|}$. We have that this latter term can be typed as:

$\Psi^1;\ \Phi^1 \vdash v_{|\Phi'^1|} : t \cdot \sigma'^1_\Psi \cdot \sigma'^1$, and thus we have $\Psi^1;\ \Phi^1 \vdash \sigma^1 : \Phi'^0 \cdot \sigma'^1_\Psi,\ t \cdot \sigma'^1_\Psi$.

Choose $\sigma^{-1} = \sigma'^{-1},\ v_{|\Phi'^0 \cdot \sigma'^1_\Psi|}$. The desired properties obviously hold.

Choose $\sigma^{inv} = \sigma'^{inv},\ v_{|\Phi'^0|}$, which is typed correctly since $t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma'^2_\Psi \cdot \sigma^{inv} = t$.

Last, assume $\Psi;\ \Phi'^0,\ t \vdash t_* : t'_*$. We prove $t_* \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = t_*$.

First, t_* is also typed under Ψ' because t_* cannot use the newly-introduced variable directly (even in the case where it would be part of Φ_0, there's still no extension variable that has $X_{|\Psi'|}$ in its context).

Thus it suffices to prove $t_* \cdot \sigma'^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = t_*$.

Then proceed by structural induction on t_*. The only interesting case occurs when $t_* = v_{|\Phi'^0|}$, in which case we have:

$v_{|\Phi'^0|} \cdot \sigma'^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = v_{|\Phi'^0 \cdot \sigma'^1_\Psi|} \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = v_{|\Phi'^1|} \cdot \sigma^2_\Psi \cdot \sigma^{inv} = v_{|\Phi'^1 \cdot \sigma^2_\Psi|} \cdot \sigma^{inv} = v_{|\Phi'^0|}$

If $\Phi^0 = \Phi'^0,\ \phi_i$ then:

By well-formedness inversion we get that $\Psi.i = [\Phi_*]\, ctx$, and by repeated inversions of the collapsable relation we get $\Phi_* \subseteq \Phi'^0$.

Choose $\Phi^1 = \Phi'^1$; $\Psi^1 = \Psi'^1$; $\sigma^2_\Psi = \sigma'^2_\Psi$; $\sigma^1 = \sigma'^1$; $\sigma^{-1} = \sigma'^{-1}$; $\sigma^{inv} = \sigma'^{inv}$.

Most desiderata are trivial. For σ^1, note that $(\Phi'^1,\ \phi_i) \cdot \sigma'^1_\Psi = \Phi'^1 \cdot \sigma'^1_\Psi$ since by construction we have that σ'^1_Ψ always substitutes parametric contexts by the empty context.

For substitutions cancellation, we need to prove that for all t such that $\Psi;\ \Phi'^0,\ \phi_i \vdash t_* : t'_*$, we have $t_* \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = t_*$. This is proved directly by noticing that t_* is typed also under Ψ' (ϕ_i as the just-introduced variable cannot refer to itself).

Case $\Psi = \Psi',\ [\Phi]\, t$**.** From the collapsable relation, we get:

$collapsable\ (\Psi') \lhd \bullet \rhd \Phi'^0,\ collapsable\ (\Phi) \lhd \Phi'^0 \rhd \Phi^0,\ collapsable\ (t) \lhd \Phi^0$.

By induction hypothesis for Ψ', we get:

$$\Psi'^1 \vdash \sigma'^1_\Psi : \Psi' \qquad \bullet \vdash \sigma'^2_\Psi : \Psi'^1 \qquad \Psi'^1 \vdash \Phi'^1 \text{ wf} \qquad \Psi'^1;\ \Phi'^1 \vdash \sigma'^1 : \Phi'^0 \cdot \sigma'^1_\Psi$$

$$\Psi'^1;\ \Phi'^0 \cdot \sigma'^1_\Psi \vdash \sigma'^{-1} : \Phi'^1 \qquad \sigma'^1 \cdot \sigma'^{-1} = id_{\Phi'^0 \cdot \sigma'^1_\Psi} \qquad \Psi';\ \Phi'^0 \vdash \sigma'^{inv} : \Phi'^1 \cdot \sigma'^2_\Psi$$

$$\forall t.\Psi';\ \Phi'^0 \vdash t : t' \Rightarrow t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma'^2_\Psi \cdot \sigma'^{inv} = t \qquad \forall T \in \Psi'^1.T = [\Phi^*]\, t \wedge \Phi^* \subseteq \Phi'^1$$

Also from typing inversion we get: $\Psi' \vdash \Phi$ wf and $\Psi';\ \Phi \vdash t : s$.

We proceed similarly as in the previous case, by induction on $\Phi'^0 \subseteq \Phi^0$, in order to redefine $\Psi'^1, \sigma'^1_\Psi, \sigma'^2_\Psi, \Phi'^1, \sigma'^1, \sigma'^{-1}, \sigma^{inv}$ with the same properties but for Φ^0 instead of Φ'^0:

$$\Psi'^1 \vdash \sigma'^1_\Psi : \Psi' \qquad \bullet \vdash \sigma'^2_\Psi : \Psi'^1 \qquad \Psi'^1 \vdash \Phi'^1 \text{ wf} \qquad \Psi'^1;\ \Phi'^1 \vdash \sigma'^1 : \Phi^0 \cdot \sigma'^1_\Psi$$

$$\Psi'^1;\ \Phi^0 \cdot \sigma'^1_\Psi \vdash \sigma'^{-1} : \Phi'^1 \qquad \sigma'^1 \cdot \sigma'^{-1} = id_{\Phi^0 \cdot \sigma'^1_\Psi} \qquad \Psi';\ \Phi^0 \vdash \sigma'^{inv} : \Phi'^1 \cdot \sigma'^2_\Psi$$

$$\forall t.\Psi';\ \Phi^0 \vdash t : t' \Rightarrow t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma'^2_\Psi \cdot \sigma'^{inv} = t \qquad \forall T \in \Psi'^1.T = [\Phi^*]\, t \wedge \Phi^* \subseteq \Phi'^1$$

Now we have $\Phi \subseteq \Phi^0$ thus $\Psi';\ \Phi^0 \vdash t : s$.

By applying σ'^1_Ψ and then σ'^1 to t we get $\Psi'^1;\ \Phi'^1 \vdash t \cdot \sigma'^1_\Psi \cdot \sigma'^1 : s$. We can now choose $\Phi^1 = \Phi'^1,\ t \cdot \sigma'^1_\Psi \cdot \sigma'^1$.

Choose $\Psi^1 = \Psi'^1,\ [\Phi^1]\, t \cdot \sigma'^1_\Psi \cdot \sigma'^1$. It is obviously well-formed.

Choose $\sigma^1_\Psi = \sigma'^1_\Psi,\ [\Phi]\, t^1$. We need $\Psi^1;\ \Phi \cdot \sigma'^1_\Psi \vdash t^1 : t \cdot \sigma'^1_\Psi$.

Assuming $t^1 = X_{|\Phi'^1|}/\sigma$, we need $t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma = t \cdot \sigma'^1_\Psi$ and $\Psi^1;\ \Phi \cdot \sigma'^1_\Psi \vdash \sigma : \Phi^1$.

Therefore $\sigma = \sigma'^{-1}$ and $\sigma^1_\Psi = \sigma'^1_\Psi,\ [\Phi]\, X_{|\Phi'^1|}/\sigma'^{-1}$ with the desirable properties.

Choose $\sigma^2_\Psi = \sigma'^2_\Psi,\ [\Phi^1]\, v_{|\Phi'^1|}$. We trivially have $\bullet \vdash \sigma^2_\Psi : \Psi^1$.

Choose $\sigma^1 = \sigma'^1$, with typing holding obviously.

Choose $\sigma^{-1} = \sigma^{-1},\ X_{|\Psi'^1|}/id_{\Phi^1}$. Since σ^1 does not use the newly introduced variable in Φ^1, the fact that σ^{-1} is its inverse follows trivially.

Choose $\sigma^{inv} = \sigma'^{inv},\ X_{|\Psi'|}/id(\Phi)$. This is well-typed if we consider the substitutions cancellation fact.

It remains to prove that for all t_* such that $\Psi', [\Phi]\, t;\ \Phi^0 \vdash t_* : t'_*$, we have $t_* \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = t_*$.

This is done by structural induction on t_*, with the interesting case being $t_* = X_{|\Psi'|}/\sigma_*$. By inversion of collapsable relation, we get that $\sigma_* = id_\Phi$.

Thus $(X_{|\Psi'|}/id_\Phi) \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = (X_{|\Phi'^1|}/(id_\Phi \cdot \sigma^1_\Psi)) \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = (X_{|\Phi'^1|}/(id_{\Phi \cdot \sigma^1_\Psi})) \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{inv} = (X_{|\Phi'^1|}/(id_{\Phi \cdot \sigma^1_\Psi} \cdot \sigma^1)) \cdot \sigma^2_\Psi \cdot \sigma^{inv} = (X_{|\Phi'^1|}/(id_{\Phi^1})) \cdot \sigma^2_\Psi \cdot \sigma^{inv} = (v_{|\Phi'^1|} \cdot (id_{\Phi^1} \cdot \sigma^2_\Psi)) \cdot \sigma^{inv} = (v_{|\Phi'^1|} \cdot id_{\Phi^1 \cdot \sigma^2_\Psi}) \cdot \sigma^{inv} = v_{|\Phi'^1 \cdot \sigma^2_\Psi|} \cdot \sigma^{inv} = X_{|\Psi'|}/id_\Phi$. □

Theorem 5.2.4

$$\frac{\Psi \vdash [\Phi]\, t : [\Phi]\, t_T \qquad \mathit{collapsable}\,(\Psi \vdash [\Phi]\, t : [\Phi]\, t_T) \triangleright \Phi_*}{\exists \Phi', t', t'_T, \sigma.}$$

$$\bullet \vdash \Phi'\ \mathit{wf} \qquad \bullet \vdash [\Phi']\, t' : [\Phi']\, t'_T \qquad \Psi;\ \Phi \vdash \sigma : \Phi' \qquad t' \cdot \sigma = t \qquad t'_T \cdot \sigma = t_T$$

Proof. Trivial application of Theorem 5.2.3. Set $\Phi' = \Phi^1 \cdot \sigma^2_\Psi$, $t' = t \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi$, $t'_T = t_T \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi$, and also set $\sigma = \sigma^{-1}$. □

Chapter 6

The VeriML computational language

Having presented the details of the λHOL logic and its associated operations in the previous chapters, we are now ready to define the core VeriML computational language. I will first present the basic constructs of the language for introducing and manipulating λHOL logical terms. I will state the type safety theorem for this language and prove it using the standard syntactic approach of Wright and Felleisen [1994]. I will then present some extensions to the language that will be useful for the applications of VeriML we will consider in the rest.

6.1 Base computation language

6.1.1 Presentation and formal definition

The core of VeriML is a functional language with call-by-value semantics and support for side-effects such as mutable references in the style of ML. This core calculus is extended with constructs that are tailored to working with terms of the λHOL logic. Specifically, we include constructs in order to be able to introduce *extension terms* T – that is, contextual terms and contexts – and also to look into their structure through pattern matching. The type K of an extension term T is retained at the level of VeriML computational types. Types K can mention variables from the extension context Ψ and can therefore depend on

$$
\begin{array}{rrl}
(\textit{Kinds}) & k & ::= \star \mid k \to k \\
(\textit{Types}) & \tau & ::= \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha : k.\tau \mid \mathsf{ref}\ \tau \\
 & & \quad \mid \forall\alpha : k.\tau \mid \lambda\alpha : k.\tau \mid \tau_1\ \tau_2 \mid \alpha \\
(\textit{Expressions}) & e & ::= () \mid \lambda x : \tau.e \mid e\ e' \mid x \mid (e,\ e') \mid \mathsf{proj}_i\ e \mid \mathsf{inj}_i\ e \\
 & & \quad \mid \mathsf{case}(e,\ x.e',\ x.e'') \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \mathsf{ref}\ e \mid e := e' \\
 & & \quad \mid\ !e \mid l \mid \Lambda\alpha : k.e \mid e\ \tau \mid \mathsf{fix}\ x : \tau.e \\
(\textit{Contexts}) & \Gamma & ::= \bullet \mid \Gamma,\ x : \tau \mid \Gamma,\ \alpha : k \\
(\textit{Values}) & v & ::= () \mid \lambda x : \tau.e \mid (v,\ v') \mid \mathsf{inj}_i\ v \mid \mathsf{fold}\ v \mid l \mid \Lambda\alpha : k.e \\
(\textit{Evaluation contexts}) & \mathcal{E} & ::= \bullet \mid \mathcal{E}\ e' \mid v\ \mathcal{E} \mid (\mathcal{E},\ e) \mid (v,\ \mathcal{E}) \mid \mathsf{proj}_i\ \mathcal{E} \mid \mathsf{inj}_i\ \mathcal{E} \\
 & & \quad \mid \mathsf{case}(\mathcal{E},\ x.e_1,\ x.e_2) \mid \mathsf{fold}\ \mathcal{E} \mid \mathsf{unfold}\ \mathcal{E} \mid \mathsf{ref}\ \mathcal{E} \\
 & & \quad \mid \mathcal{E} := e' \mid v := \mathcal{E} \mid\ !\mathcal{E} \mid \mathcal{E}\ \tau \\
(\textit{Stores}) & \mu & ::= \bullet \mid \mu,\ l \mapsto v \\
(\textit{Store typing}) & \mathcal{M} & ::= \bullet \mid \mathcal{M},\ l : \tau
\end{array}
$$

Figure 6.1: VeriML computational language: ML core (syntax)

$$
\begin{array}{rrl}
(\textit{Kinds}) & k & ::= \cdots \mid \Pi V : K.k \\
(\textit{Types}) & \tau & ::= \cdots \mid (V : K) \to \tau \mid (V : K) \times \tau \mid \lambda V : K.\tau \mid \tau\ T \\
(\textit{Expressions}) & e & ::= \cdots \mid \lambda V : K.e \mid e\ T \mid \langle\ T,\ e\ \rangle_{(V:K)\times\tau} \\
 & & \quad \mid \mathsf{let}\ \langle\ V,\ x\ \rangle = e\ \mathsf{in}\ e' \\
 & & \quad \mid \mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau\ \mathsf{with}\ \Psi_u.T' \mapsto e' \\
(\textit{Values}) & v & ::= \cdots \mid \lambda V : K.e \mid \langle\ T,\ e\ \rangle_{(V:K)\times\tau} \\
(\textit{Evaluation contexts}) & \mathcal{E} & ::= \cdots \mid \mathcal{E}\ T \mid \langle\ T,\ \mathcal{E}\ \rangle_{(V:K)\times\tau} \mid \mathsf{let}\ \langle\ V,\ x\ \rangle = \mathcal{E}\ \mathsf{in}\ e'
\end{array}
$$

Figure 6.2: VeriML computational language: λHOL-related constructs (syntax)

terms that were introduced earlier. Thus λHOL constructs available at the VeriML level are dependently typed. We will see more details of this in the rest of this section.

Let us now present the formal details of VeriML, split into the ML core and the λHOL-related extensions. Figure 6.1 defines the syntax of the ML core of VeriML; Figures 6.3 through 6.5 give the relevant typing rules and Figure 6.8 gives the small-step operational semantics. This ML core supports the following features: higher-order functional programming (through function types $\tau_1 \to \tau_2$); general recursion (through the fixpoint construct $\mathsf{fix}\ x : \tau.e$); algebraic data types (combining product types $\tau_1 \times \tau_2$, sum types $\tau_1 + \tau_2$, recursive types $\mu\alpha : k.\tau$ and the unit type unit); mutable references $\mathsf{ref}\ \tau$; polymorphism over types supporting full System F – also referred to as rank-N-polymorphism (through the $\forall\alpha : k.\tau$ type); and type constructors in the style of System F_ω (by including arrows at the kind level $k_1 \to k_2$ and type constructors $\lambda\alpha : k.\tau$ at the type level). Data type definitions

of the form:

$$\textbf{data}\ \mathsf{list}\ \alpha = \mathsf{Nil} \mid \mathsf{Cons\ of}\ \alpha \times \mathsf{list}\ \alpha$$

can be desugared to their equivalent using the base algebraic data type formers:

$$\mathsf{let\ list} = \lambda\alpha : \star.\mu\mathsf{list} : \star.\mathsf{unit} + \alpha \times \mathsf{list}$$

$$\mathsf{let\ Nil} = \lambda\alpha : \star.\mathsf{fold}\ (\mathsf{inj}_1\ ())$$

$$\mathsf{let\ Cons} = \lambda\alpha : \star.\lambda\mathsf{hd} : \alpha.\lambda\mathsf{tl} : \mathsf{list}\ \alpha.\mathsf{fold}\ (\mathsf{inj}_2\ (\mathsf{hd},\ \mathsf{tl}))$$

We do not present features such as exception handling and the module system available in Standard ML [Milner, 1997], but these are straightforward to add using the standard mechanisms. Typing derivations are of the form $\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{J}$, where Ψ is a yet-unused extension variables context that will be used in the λHOL-related extensions; $\mathcal{M}$ is the store typing, i.e. a context assigning types to locations in the current store μ; and Γ is the context of computational variables, including type-level α and expression-level x variables. The operational semantics work on machine states of the form $(\ \mu\ ,\ e\ )$ which bundle the current store μ with the current expression e. We formulate the semantics through evaluation contexts $\mathcal{E}$ [Felleisen and Hieb, 1992]. All of these definitions are entirely standard [e.g. Pierce, 2002] so we will not comment further on them.

The new λHOL-related constructs are dependently-typed tuples over λHOL extension terms, dependently-typed functions, dependent pattern matching and type constructors dependent on λHOL terms. We present their details in Figures 6.2 (syntax), 6.6 and 6.7 (typing) and 6.9 (operational semantics). We will now briefly discuss these constructs.

Dependent λHOL tuples. We use tuples over λHOL extension terms in order to incorporate such terms inside our computational language expressions. These tuples are composed from a λHOL part T and a computational part e. We write them as $\langle\ T,\ e\ \rangle$. The type that VeriML assigns to such tuples includes the typing information coming from the logic. We will denote their type as $K \times \tau$ for the time being, with K being the λHOL type of T and τ being the type of e. The simplest case occurs when the computational part is empty – more accurately when it is equal to the unit expression that carries no information ($e = ()$). In this case, a λHOL tuple simply gives us a way to create a VeriML computational value v out of a λHOL extension term T, for example:

$\boxed{\Psi \vdash k \text{ wf}}$

$$\frac{\vdash \Psi \text{ wf}}{\Psi \vdash \star \text{ wf}}\ \textsc{CKnd-CType} \qquad \frac{\Psi \vdash k \text{ wf} \qquad \Psi \vdash k' \text{ wf}}{\Psi \vdash k \to k' \text{ wf}}\ \textsc{CKnd-Arrow}$$

$\boxed{\Psi;\ \Gamma \vdash \tau : k}$

$$\frac{}{\Psi;\ \Gamma \vdash \mathsf{unit} : \star}\ \textsc{CTyp-Unit} \qquad \frac{\Psi;\ \Gamma \vdash \tau_1 : \star \qquad \Psi;\ \Gamma \vdash \tau_2 : \star}{\Psi;\ \Gamma \vdash \tau_1 \to \tau_2 : \star}\ \textsc{CTyp-Arrow}$$

$$\frac{\Psi;\ \Gamma \vdash \tau_1 : \star \qquad \Psi;\ \Gamma \vdash \tau_2 : \star}{\Psi;\ \Gamma \vdash \tau_1 \times \tau_2 : \star}\ \textsc{CTyp-Prod} \qquad \frac{\Psi;\ \Gamma \vdash \tau_1 : \star \qquad \Psi;\ \Gamma \vdash \tau_2 : \star}{\Psi;\ \Gamma \vdash \tau_1 + \tau_2 : \star}\ \textsc{CTyp-Sum}$$

$$\frac{\Psi \vdash k \text{ wf} \qquad \Psi;\ \Gamma,\ \alpha : k \vdash \tau : k}{\Psi;\ \Gamma \vdash (\mu\alpha : k.\tau) : k}\ \textsc{CTyp-Rec} \qquad \frac{\Psi;\ \Gamma \vdash \tau : \star}{\Psi;\ \Gamma \vdash \mathsf{ref}\ \tau : \star}\ \textsc{CTyp-Ref}$$

$$\frac{\Psi \vdash k \text{ wf} \qquad \Psi;\ \Gamma,\ \alpha : k \vdash \tau : \star}{\Psi;\ \Gamma \vdash (\forall\alpha : k.\tau) : \star}\ \textsc{CTyp-Poly}$$

$$\frac{\Psi \vdash k \text{ wf} \qquad \Psi;\ \Gamma,\ \alpha : k \vdash \tau : k'}{\Psi;\ \Gamma \vdash (\lambda\alpha : k.\tau) : k \to k'}\ \textsc{CTyp-Lam}$$

$$\frac{\Psi;\ \Gamma \vdash \tau_1 : k \to k' \qquad \Psi;\ \Gamma \vdash \tau_2 : k}{\Psi;\ \Gamma \vdash \tau_1\ \tau_2 : k'}\ \textsc{CTyp-App} \qquad \frac{(\alpha : k) \in \Gamma}{\Psi;\ \Gamma \vdash \alpha : k}\ \textsc{CTyp-Var}$$

$\boxed{\Psi \vdash \Gamma \text{ wf}}$

$$\frac{}{\Psi \vdash \bullet \text{ wf}}\ \textsc{CCtx-Empty} \qquad \frac{\Psi \vdash \Gamma \text{ wf} \qquad \Psi;\ \Gamma \vdash k \text{ wf}}{\Psi \vdash (\Gamma,\ \alpha : k) \text{ wf}}\ \textsc{CCtx-TVar}$$

$$\frac{\Psi \vdash \Gamma \text{ wf} \qquad \Psi;\ \Gamma \vdash \tau : \star}{\Psi \vdash (\Gamma,\ x : \tau) \text{ wf}}\ \textsc{CCtx-CVar}$$

Figure 6.3: VeriML computational language: ML core (typing, 1/3)

$\boxed{\vdash \mathcal{M}\ \text{wf}}$

$$\frac{}{\vdash \bullet\ \text{wf}}\ \textsc{Store-Empty} \qquad \frac{\vdash \mathcal{M}\ \text{wf} \qquad \bullet;\ \bullet \vdash \tau : \star}{\vdash (\mathcal{M},\ l : \tau)}\ \textsc{Store-Loc}$$

$\boxed{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau}$

$$\frac{}{\Psi;\ \mathcal{M};\ \Gamma \vdash () : \mathsf{unit}}\ \textsc{CExp-Unit} \qquad \frac{\Psi;\ \mathcal{M};\ \Gamma,\ x : \tau \vdash e : \tau'}{\Psi;\ \mathcal{M};\ \Gamma \vdash \lambda x : \tau.e : \tau \to \tau'}\ \textsc{CExp-FunI}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau \to \tau' \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e' : \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash e\ e' : \tau'}\ \textsc{CExp-FunE} \qquad \frac{(x : \tau) \in \Gamma}{\Psi;\ \mathcal{M};\ \Gamma \vdash x : \tau}\ \textsc{CExp-Var}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e_1 : \tau_1 \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e_2 : \tau_2}{\Psi;\ \mathcal{M};\ \Gamma \vdash (e_1,\ e_2) : \tau_1 \times \tau_2}\ \textsc{CExp-ProdI}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau_1 \times \tau_2 \qquad i = 1 \text{ or } 2}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{proj}_i\ e : \tau_i}\ \textsc{CExp-ProdE}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau_i \qquad i = 1 \text{ or } 2}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{inj}_i\ e : \tau_1 + \tau_2}\ \textsc{CExp-SumI}$$

$$\frac{\begin{array}{c}\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau_1 + \tau_2 \\ \Psi;\ \mathcal{M};\ \Gamma,\ x : \tau_1 \vdash e_1 : \tau \qquad \Psi;\ \mathcal{M};\ \Gamma,\ x : \tau_2 \vdash e_2 : \tau\end{array}}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{case}(e,\ x.e_1,\ x.e_2) : \tau}\ \textsc{CExp-SumE}$$

$$\frac{\Psi;\ \Gamma \vdash (\mu\alpha : k.\tau) : \star \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau[\mu\alpha : k.\tau/\alpha]}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{fold}\ e : \mu\alpha : k.\tau}\ \textsc{CExp-RecI}$$

$$\frac{\Psi;\ \Gamma \vdash (\mu\alpha : k.\tau) : \star \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e : \mu\alpha : k.\tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{unfold}\ e : \tau[\mu\alpha : k.\tau/\alpha]}\ \textsc{CExp-RecE}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma,\ \alpha : k \vdash e : \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash \Lambda\alpha : k.e : \Pi\alpha : k.\tau}\ \textsc{CExp-PolyI}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \Pi\alpha : k.\tau' \qquad \Psi;\ \Gamma \vdash \tau : k}{\Psi;\ \mathcal{M};\ \Gamma \vdash e\ \tau : \tau'[\tau/\alpha]}\ \textsc{CExp-PolyE}$$

Figure 6.4: VeriML computational language: ML core (typing, 2/3)

$\boxed{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau}$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ \tau}\ \textsc{CExp-NewRef}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \mathsf{ref}\ \tau \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e' : \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash e := e' : \mathsf{unit}}\ \textsc{CExp-Assign}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \mathsf{ref}\ \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash !e : \tau}\ \textsc{CExp-Deref} \qquad \frac{(l : \tau) \in \mathcal{M}}{\Psi;\ \mathcal{M};\ \Gamma \vdash l : \mathsf{ref}\ \tau}\ \textsc{CExp-Loc}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau \qquad \tau =_\beta \tau'}{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau'}\ \textsc{CExp-Conv} \qquad \frac{\Psi;\ \mathcal{M};\ \Gamma,\ x : \tau \vdash e : \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{fix}\ x : \tau.e : \tau}\ \textsc{CExp-Fix}$$

Figure 6.5: VeriML computational language: ML core (typing, 3/3)

$\boxed{\Psi \vdash k\ \mathrm{wf}}$

$$\frac{\vdash \Psi,\ V : K\ \mathrm{wf} \qquad \Psi,\ V : K \vdash k\ \mathrm{wf}}{\Psi \vdash \Pi V : K.k\ \mathrm{wf}}\ \textsc{CKnd-}\Pi\textsc{Hol}$$

$\boxed{\Psi;\ \Gamma \vdash \tau : k}$

$$\frac{\Psi,\ V : K;\ \Gamma \vdash \tau : \star}{\Psi;\ \Gamma \vdash (V : K) \to \tau : \star}\ \textsc{CTyp-}\Pi\textsc{Hol} \qquad \frac{\Psi,\ V : K;\ \Gamma \vdash \tau : \star}{\Psi;\ \Gamma \vdash (V : K) \times \tau : \star}\ \textsc{CTyp-}\Sigma\textsc{Hol}$$

$$\frac{\Psi,\ V : K;\ \Gamma \vdash \tau : k}{\Psi;\ \Gamma \vdash (\lambda V : K.\tau) : (\Pi V : K.k)}\ \textsc{CTyp-LamHol}$$

$$\frac{\Psi;\ \Gamma \vdash \tau : \Pi V : K.k \qquad \Psi \vdash T : K}{\Psi;\ \Gamma \vdash \tau\ T : k[T/V]}\ \textsc{CTyp-AppHol}$$

Figure 6.6: VeriML computational language: λHOL-related constructs (typing)

$\boxed{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau}$

$$\frac{\Psi,\ V : K;\ \mathcal{M};\ \Gamma \vdash e : \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash (\lambda V : K.e) : ((V : K) \to \tau)}\ \textsc{CExp-}\Pi\textsc{HolI}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : (V : K) \to \tau \qquad \Psi \vdash T : K}{\Psi;\ \mathcal{M};\ \Gamma \vdash e\ T : \tau[T/V]}\ \textsc{CExp-}\Pi\textsc{HolE}$$

$$\frac{\Psi;\ \Gamma \vdash (V : K) \times \tau : \star \qquad \Psi \vdash T : K \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau[T/V]}{\Psi;\ \mathcal{M};\ \Gamma \vdash \langle\ T,\ e\ \rangle_{(V:K)\times\tau} : ((V : K) \times \tau)}\ \textsc{CExp-}\Sigma\textsc{HolI}$$

$$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : (V : K) \times \tau \qquad \Psi,\ V : K;\ \mathcal{M};\ \Gamma,\ x : \tau \vdash e' : \tau' \qquad \Psi;\ \Gamma \vdash \tau' : \star}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{let}\ \langle\ V,\ x\ \rangle = e\ \mathsf{in}\ e' : \tau'}\ \textsc{CExp-}\Sigma\textsc{HolE}$$

$$\frac{\Psi \vdash T : K \qquad \Psi,\ V : K;\ \Gamma \vdash \tau : \star \qquad \Psi \models^{*}_{p} \Psi_u > T_P : K \qquad \Psi,\ \Psi_u;\ \mathcal{M};\ \Gamma \vdash e' : \tau[T_P/V]}{\Psi;\ \mathcal{M};\ \Gamma \vdash \begin{array}{l}\mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau \\ \mathsf{with}\ \ \Psi_u.\ T_P \mapsto e'\end{array} : \tau[T/V] + \mathsf{unit}}\ \textsc{CExp-HolMatch}$$

$$\frac{\Psi;\ \Gamma \vdash (\mu\alpha : k.\tau) : k \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau[\mu\alpha : k.\tau/\alpha]\ a_1\ a_2\ \cdots\ a_n \qquad a_i = \tau_i \text{ or } T_i}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{fold}\ e : (\mu\alpha : k.\tau)\ a_1\ a_2\ \cdots\ a_n}\ \textsc{CExp-RecI}$$

$$\frac{\Psi;\ \Gamma \vdash (\mu\alpha : k.\tau) : k \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e : (\mu\alpha : k.\tau)\ a_1\ a_2\ \cdots\ a_n \qquad a_i = \tau_i \text{ or } T_i}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{unfold}\ e : \tau[\mu\alpha : k.\tau/\alpha]\ a_1\ a_2\ \cdots\ a_n}\ \textsc{CExp-RecE}$$

Figure 6.7: VeriML computational language: λHOL-related constructs (typing, continued)

$\boxed{(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu\ ,\ e'\ )}$

$$\frac{(\ \mu\ ,\ e\ ) \longrightarrow \left(\ \mu'\ ,\ e'\ \right) \qquad (\mathcal{E} \neq \bullet)}{(\ \mu\ ,\ \mathcal{E}[e]\ ) \longrightarrow \left(\ \mu'\ ,\ \mathcal{E}[e']\ \right)}\ \textsc{Op-Env}$$

$$\frac{}{(\ \mu\ ,\ (\lambda x : \tau.e)\ v\ ) \longrightarrow (\ \mu\ ,\ e[v/x]\ )}\ \textsc{Op-Beta}$$

$$\frac{}{(\ \mu\ ,\ \mathsf{proj}_i(v_1,\ v_2)\ ) \longrightarrow (\ \mu\ ,\ v_i\ )}\ \textsc{Op-Proj}$$

$$\frac{}{(\ \mu\ ,\ \mathsf{case}(\mathsf{inj}_i\ v,\ x.e_1,\ x.e_2)\ ) \longrightarrow (\ \mu\ ,\ e_i[v/x]\ )}\ \textsc{Op-Case}$$

$$\frac{}{(\ \mu\ ,\ \mathsf{unfold}\ (\mathsf{fold}\ v)\ ) \longrightarrow (\ \mu\ ,\ v\ )}\ \textsc{Op-Unfold}$$

$$\frac{\neg(l \mapsto \_ \in \mu)}{(\ \mu\ ,\ \mathsf{ref}\ v\ ) \longrightarrow (\ (\mu,\ l \mapsto v)\ ,\ l\ )}\ \textsc{Op-NewRef}$$

$$\frac{l \mapsto \_ \in \mu}{(\ \mu\ ,\ l := v\ ) \longrightarrow (\ \mu[l \mapsto v]\ ,\ ()\ )}\ \textsc{Op-Assign} \qquad \frac{l \mapsto v \in \mu}{(\ \mu\ ,\ !l\ ) \longrightarrow (\ \mu\ ,\ v\ )}\ \textsc{Op-Deref}$$

$$\frac{}{(\ \mu\ ,\ (\Lambda\alpha : k.e)\ \tau\ ) \longrightarrow (\ \mu\ ,\ e[\tau/\alpha]\ )}\ \textsc{Op-PolyInst}$$

$$\frac{}{(\ \mu\ ,\ \mathsf{fix}\ x : \tau.e\ ) \longrightarrow (\ \mu\ ,\ e[\mathsf{fix}\ x : \tau.e/x]\ )}\ \textsc{Op-Fix}$$

$\boxed{\mu[l := v]}$

$$\begin{array}{lcl}(\mu,\ l' \mapsto v')[l := v] & = & \mu[l := v],\ l' \mapsto v' \\ (\mu,\ l \mapsto v')[l := v] & = & \mu, l \mapsto v\end{array}$$

$\boxed{(l \mapsto v) \in \mu}$

$$\begin{array}{lcl}(l \mapsto v) & \in & (\mu,\ l \mapsto v) \\ (l \mapsto v) & \in & (\mu,\ l' \mapsto v') \Leftarrow (l \mapsto v) \in \mu\end{array}$$

Figure 6.8: VeriML computational language: ML core (semantics)

$\boxed{(\mu\ ,\ e\ ) \longrightarrow (\mu\ ,\ e'\ )}$

$$\frac{}{(\mu\ ,\ (\lambda V : K.e)\ T\ ) \longrightarrow (\mu\ ,\ e[T/V]\ )}\ \textsc{Op-}\Pi\textsc{Hol-Beta}$$

$$\frac{}{\big(\mu\ ,\ \mathsf{let}\ \langle\ V,\ x\ \rangle = \langle\ T,\ v\ \rangle\ \mathsf{in}\ e'\ \big) \longrightarrow \big(\mu\ ,\ (e'[T/V])[v/x]\ \big)}\ \textsc{Op-}\Sigma\textsc{Hol-Unpack}$$

$$\frac{T_P \sim T = \sigma_\Psi}{\big(\mu\ ,\ \mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_u.T_P \mapsto e'\ \big) \longrightarrow \big(\mu\ ,\ \mathsf{inj}_1\ (e' \cdot \sigma_\Psi)\ \big)}\ \textsc{Op-HolMatch}$$

$$\frac{T_P \sim T = \mathsf{fail}}{\big(\mu\ ,\ \mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_u.T_P \mapsto e'\ \big) \longrightarrow (\mu\ ,\ \mathsf{inj}_2\ ()\ )}\ \textsc{Op-HolNoMatch}$$

Figure 6.9: VeriML computational language: λHOL-related constructs (semantics)

$$v = \langle\ [P : \mathit{Prop}]\ \lambda x : P.x,\ ()\ \rangle : ([P : \mathit{Prop}]\ P \to P) \times \mathsf{unit}$$

Note that this construct works over extension terms T, not normal logical terms t, thus the use of the contextual term inside v. Consider now the case where the computational part e of a λHOL tuple $\langle\ T,\ e\ \rangle$ is non-empty and more specifically is another λHOL tuple, for example:

$$v = \langle\ [P : \mathit{Prop}]\ P \to P,\ \langle\ [P : \mathit{Prop}]\ \lambda x : P.x,\ ()\ \rangle\ \rangle$$

What type should this value have? Again using informal syntax, one possible type could be:

$$\begin{array}{rcl} v & = & \langle\ [P : \mathit{Prop}]\ P \to P,\ \langle\ [P : \mathit{Prop}]\ \lambda x : P.x,\ ()\ \rangle\ \rangle \\ & : & ([P : \mathit{Prop}]\ \mathit{Prop}) \times ([P : \mathit{Prop}]\ P \to P) \times \mathsf{unit} \end{array}$$

While this is a valid type, it is perhaps not capturing our intention: if we view the value v as a package of a proposition P together with its proof π, the fact that π actually proves the proposition P and not some other proposition is lost using this type. We say that the tuple is *dependent*: the *type* of its second component depends on the *value* of the first component. Thus the actual syntax for the type of λHOL tuples given in Figure 6.2 is $(V : K) \times \tau$, where the variable V can be used to refer to the value of the first component inside the type τ. The example would thus be typed as:

$$\begin{array}{rcl} v & = & \langle\ [P : \mathit{Prop}]\ P \to P,\ \langle\ [P : \mathit{Prop}]\ \lambda x : P.x,\ ()\ \rangle\ \rangle \\ & : & (X : [P : \mathit{Prop}]\ \mathit{Prop}) \times ([P : \mathit{Prop}]\ X) \times \mathsf{unit} \end{array}$$

We could also use dependent tuples to create a value that packages together the variable context that the proposition depends on as well, by using the other kind of extension terms – contexts:

$$\begin{array}{rcl} v & = & \langle\, [P : \mathit{Prop}],\ \langle\, [P : \mathit{Prop}]\, P \to P,\ \langle\, [P : \mathit{Prop}]\, \lambda x : P.x,\ ()\, \rangle\, \rangle\, \rangle \\ & : & (\phi : \mathit{ctx}) \times (X : [\phi]\, \mathit{Prop}) \times ([\phi]\, X) \times \mathsf{unit} \end{array}$$

We have only seen the introduction form of dependent tuples. The elimination form is the binding construct $\mathsf{let}\ \langle\, V,\ x\, \rangle = e\ \mathsf{in}\ e'$ that simply makes the two components available in e'. Combined with the normal ML features, we can thus write functions that create λHOL terms at runtime – whose exact value is only decided dynamically. An illustratory example is a function that takes a proposition P and a number n as arguments and produces a conjunction $P \wedge P \wedge \cdots \wedge P$ of length 2^n. In the rest, the λHOL terms that we will be most interested in producing dynamically will be proof objects.

$$\begin{array}{rcl} \mathsf{conjN} & : & ((]\, \mathit{Prop}) \times \mathsf{unit}) \to \mathsf{int} \to ((]\, \mathit{Prop}) \times \mathsf{unit}) \\ & = & \lambda x : ((]\, \mathit{Prop}) \times \mathsf{unit}).\lambda n : \mathsf{int}. \\ & & \mathsf{if}\ n = 0\ \mathsf{then}\ p \\ & & \mathsf{else\ let}\ \langle\, P,\ \_\, \rangle = x\ \mathsf{in\ conjN}\ \langle\, []\, P \wedge P,\ ()\, \rangle\ (n-1) \end{array}$$

Let us now see the formal details of the dependent tuples. The actual syntax that we use is $\langle\, T,\ e\, \rangle_{(V:K)\times\tau}$ where the return type of the tuple is explicitly noted; this is because there are multiple valid types for tuples as our example above shows. We usually elide the annotation when it can be easily inferred from context. The type $(V : K) \times \tau$ introduces a bound variable V that can be used inside τ as the kinding rule CTyp-ΣHol in Figure 6.6 shows. The typing rule for introducing a dependent tuple CExp-ΣHolI in Figure 6.7 uses the λHOL typing judgement $\Psi \vdash T : K$ in order to type-check the first component. Thus the λHOL type checker is *embedded within the type checker of VeriML*. Furthermore it captures the fact that typing for the second component e of the tuple depends on the value T of the first through the substitution $\sigma_\Psi = id_\Psi,\ T/V$. Note that applying an extension substitution to a computational kind, type or expression applies it structurally to all the λHOL extension terms and kinds it contains. The elimination rule for tuples CExp-ΣHolE makes its two components available in an expression e' yet not at the type level – as the exact value of T is in the general case not statically available as our example above demonstrates.

These typing rules are standard for dependent tuples (also for existential packages, which is just another name for the same construct).

As a notational convenience, we will use the following syntactic sugar:

$$\begin{array}{lcl} \langle\, T \,\rangle & = & \langle\, T,\, () \,\rangle \\ (K) & = & (V : K) \times \mathsf{unit} \\ \mathsf{let}\ \langle\, V \,\rangle\ = e\ \mathsf{in}\ e' & = & \mathsf{let}\ \langle\, V,\, \_ \,\rangle = e\ \mathsf{in}\ e' \\ \langle\, T_1,\, T_2,\, \cdots, T_n \,\rangle & = & \langle\, T_1,\, \langle\, T_2,\, \cdots,\, \langle\, T_n \,\rangle \,\rangle \,\rangle \end{array}$$

Dependent λHOL functions. Dependent functions are a necessary complement to dependent tuples. They allow us to write functions where the type of the *results* depends on the values of the *input*. VeriML supports dependent functions over λHOL extension terms. We denote their type as $(V : K) \to \tau$ and introduce them through the notation $\lambda V : K.e$. For example, the type of an automated prover that searches for a proof object for a given proposition is:

$$\mathsf{auto} : (\phi : \mathit{ctx}) \to (P : [\phi]\ \mathit{Prop}) \to \mathsf{option}\ \big([\phi]\ P\big)$$

We have used the normal ML option type as the prover might fail; the standard definition is:

$$\mathbf{data}\ \mathsf{option}\ \alpha = \mathsf{None} \mid \mathsf{Some\ of}\ \alpha$$

Another example is the computational function that lifts the modus-ponens logical rule to the computational level:

$$\begin{array}{lcl} \mathsf{mp} & : & (\phi : \mathit{ctx}) \to (P : [\phi]\ \mathit{Prop}) \to (Q : [\phi]\ \mathit{Prop}) \to \\ & & \quad ([\phi]\ P \to Q) \to ([\phi]\ P) \to \big([\phi]\ Q\big) \\ \mathsf{mp} & = & \lambda\phi.\lambda P.\lambda Q.\lambda H_1 : [\phi]\ P \to Q.\lambda H_2 : [\phi]\ P.\, \langle\, [\phi]\ H_1\ H_2 \,\rangle \end{array}$$

The typing rules for dependent functions CExp-ΠHolI and CExp-ΠHolE are entirely standard as is the small-step semantics rule Op-ΠHolBeta. Note that the substitution T/V used in this rule is the one-element σ_Ψ extension substitution, applied structurally to the body of the function e.

Dependent λHOL pattern matching. The most important λHOL-related construct in VeriML is a pattern matching construct for looking into the structure of λHOL extension

terms. We can use this construct to implement functions such as the auto automated prover suggested above, as well as the tautology prover presented in Section 2.3. An example is as follows: (note that this is a formal version of the same example as in Section 2.3)

$$
\begin{array}{l}
\mathsf{tautology} \quad : \quad (\phi : ctx) \to (P : [\phi]\, Prop) \to \mathsf{option}\ \big([\phi]\, Prop\big) \\
\mathsf{tautology}\ \phi\ P \quad = \\
\quad \mathsf{holmatch}\ P\ \mathsf{with} \\
\quad\quad (Q : [\phi]\, Prop,\ R : [\phi]\, Prop).[\phi]\, Q \wedge R \quad \mapsto \\
\quad\quad\quad \mathsf{do}\ \ X \leftarrow \mathsf{tautology}\ \phi\ ([\phi]\, Q)\ ; \\
\quad\quad\quad\quad\ \ Y \leftarrow \mathsf{tautology}\ \phi\ ([\phi]\, R)\ ; \\
\quad\quad\quad\quad\ \ \mathsf{return}\ \langle\ [\phi]\, andI\ X\ Y\ \rangle \\
\quad\quad | \ (Q : [\phi]\, Prop,\ R : [\phi]\, Prop).[\phi]\, Q \to R \quad \mapsto \\
\quad\quad\quad \mathsf{do}\ \ X \leftarrow \mathsf{tautology}\ (\phi,\ Q)\ ([\phi,\ Q]\, R) \\
\quad\quad\quad\quad\ \ \mathsf{return}\ \langle\ [\phi]\, \lambda(Q).X\ \rangle \\
\quad\quad | \quad (P : [\phi]\, Prop).[\phi]\, P \ \mapsto \ \ \mathsf{findHyp}\ \phi\ ([\phi]\, P)
\end{array}
$$

The variables in the parentheses represent the unification variables context Ψ_u, whereas the following extension term is the pattern. Though we do not give the details of findHyp, this is mostly similar as in Section 2.3; it works by using the same pattern matching construct over contexts. It is worth noting that pattern matching is dependently typed too, as the result type of the overall construct depends on the value of the scrutinee. In this example, the return type is a proof object of type P – the scrutinee itself. Therefore each branch needs to return a proof object that matches the pattern itself.

In the formal definition of VeriML we give a very simple pattern matching construct that just matches a term against one pattern; more complicated pattern matching constructs that match several patterns at once are derived forms of this basic construct. We write this construct as:

$$\mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau\ \mathsf{with}\ \Psi_u.T_P \mapsto e$$

The term T represents the scrutinee, Ψ_u the unification context, T_P the pattern and e the body of the match. The return clause specifies what the return type of the construct will be, using V to refer to T. We omit this clause when it is directly inferrable from context. As the typing rule CExp-HolMatch demonstrates, the return type is an option

type wrapping $\tau[T/V]$ in order to capture the possibility of pattern match failure. The operational semantics of the construct Op-HolMatch and Op-HolNoMatch use the pattern matching algorithm $T_P \sim T$ defined in Section 5.1; the latter rule is used when no matching substitution exists.

Type-level λHOL functions. The last λHOL-related construct available in VeriML is functions over λHOL terms at the type level, giving us type constructors over λHOL terms. This allows us to specify type constructors such as the following:

$$\textbf{type}\ \mathsf{eqlist} = \lambda\phi : ctx.\lambda T : [\phi]\ \mathit{Type}.\mathsf{list}\ ((t_1 : [\phi]\, T) \times (t_2 : [\phi]\, T) \times \big([\phi]\, t_1 = t_2\big))$$

This type constructor expects a context and a type and represents lists of equality proofs for terms of that specific context and type. Furthermore, in order to support recursive types that are indexed by λHOL we adjust the typing rules for recursive type introduction and elimination CExp-RecI and CExp-RecE. This allows us to define *generalized algebraic data types* (GADTs) in VeriML as supported in various languages (e.g. Dependent ML [Xi and Pfenning, 1999] and Haskell [Peyton Jones et al., 2006]), through the standard encoding based on explicit equality predicates [Xi et al., 2003]. An example is the vector data type which is indexed by a λHOL natural number representing its length:

$$\begin{array}{lll} \textbf{data}\ \mathsf{vector}\ \alpha\ n = & \mathsf{Vnil} :: \mathsf{vector}\ \alpha\ 0 \\ & \mid\ \mathsf{Vcons} :: \alpha \to \mathsf{vector}\ \alpha\ n' \to \mathsf{vector}\ \alpha\ (\mathit{succ}\ n') \end{array}$$

The formal counterpart of this definition is:

$$\begin{array}{rcl} \mathsf{vector} & : & \star \to \Pi n : []\ \mathit{Nat}.\star \\ & = & \lambda\alpha : \star.\mu\mathsf{vector} : (\Pi n : []\ \mathit{Nat}.\star).\lambda n : []\ \mathit{Nat}. \\ & & \quad \big([]\, n = 0\big) + \\ & & \quad ((n' : []\ \mathit{Nat}) \times \alpha \times (\mathsf{vector}\ []\ n') \times \big([]\, n = \mathit{succ}\ n'\big)) \end{array}$$

Notational conventions. Before we conclude our presentation of the base VeriML computational language, let us present some notational conventions used in the rest of this dissertation. We use $(V_1 : K_1,\ V_2 : K_2,\ \cdots,\ V_n : K_n) \to \tau$ to denote successive dependent function types $(V_1 : K_1) \to (V_2 : K_2) \to \cdots (V_n : K_n) \to \tau$. Similarly we use $\big(V_1 : K_1,\ V_2 : K_2,\ \cdots,\ V_n : K_n\big)$ for the iterated dependent tuple type $(V_1 : K_1) \times (V_2 : K_2) \times \cdots (V_n : K_n) \times \mathsf{unit}$. We elide the context Φ in contextual terms

$[\Phi]\,t$ when it is easy to infer; similarly we omit the substitution σ in the form X/σ for using the metavariable X. Thus we write $(\phi : \mathit{ctx},\ T : \mathit{Type},\ t : T) \to \tau$ instead of $(\phi : \mathit{ctx},\ T : [\phi]\ \mathit{Type},\ t : [\phi]\,T/\mathit{id}_\phi) \to \tau$. Last, we sometimes omit abstraction over contexts or types and arguments to functions that are determined by further arguments. Thus the above type could be written as $(t : T) \to \tau$ instead; a function of this type can be called with a single argument which determines both ϕ and $T : [\phi]\ \mathit{Type}$.

6.1.2 Metatheory

We will now present the main metatheoretic proofs about VeriML, culminating in a *type-safety* theorem. The usual informal description of type-safety is that well-typed programs do not go wrong. A special case of this theorem is particularly revealing about what this theorem entails for VeriML: the case of programs e typed as (P), where P is a proposition. As discussed in the previous section, values $v = \langle\ \pi\ \rangle$ having a type of this form include a proof object π proving P. Expressions of such a type are arbitrary programs that may use all VeriML features (functions, λHOL pattern matching, general recursion, mutable references etc.) in order to produce a proof object; we refer to these expressions as proof expressions. Type safety guarantees that *if evaluation of a proof expression of type* (P) *terminates, then it will result in a value* $\langle\ \pi\ \rangle$ *where* π *is a valid proof object for the proposition* P. Thus we do not need to check the proof object produced by a proof expression again using a proof checker. Another way to view the same result is that we cannot evade the λHOL proof checker embedded in the VeriML type system – we cannot cast a value of a different type into a proof object type (P) only to discover at runtime that this value does not include a proper proof object. Of course the type safety theorem is much more general than just the case of proof expressions, establishing that an expression of *any* type that terminates evalutes to a value of the same type. A sketch of the formal statement of this theorem is as follows:

$$\frac{\vdash e : \tau \qquad e \text{ terminates}}{e \longrightarrow^* v \qquad \vdash v : \tau}\ \textbf{Type safety}$$

As is standard, we split this proof into two main lemmas: preservation and progress.

Compatibility of store with store typing:

$$\frac{\forall l, v.(l \mapsto v) \in \mu \Rightarrow \exists \tau.(l : \tau) \in \mathcal{M} \wedge \bullet;\ \mathcal{M};\ \bullet \vdash v : \tau \quad \forall l, \tau.(l : \tau) \in \mathcal{M} \Rightarrow \exists v.(l \mapsto v) \in \mu \wedge \bullet;\ \mathcal{M};\ \bullet \vdash v : \tau}{\mu \sim \mathcal{M}}$$

Store typing subsumption:

$$\frac{\forall l, \tau.(l : \tau) \in \mathcal{M} \Rightarrow (l : \tau) \in \mathcal{M}'}{\mathcal{M} \subseteq \mathcal{M}'}$$

β-equivalence for types τ (used in CExp-Conv) is the compatible closure of the relation:

$$\begin{array}{lll} (\lambda \alpha : \mathcal{K}.\tau)\ \tau' & =_\beta & \tau[\tau'/\alpha] \\ (\lambda V : K.\tau)\ T & =_\beta & \tau[T/V] \end{array}$$

Canonical and neutral types:

$$\begin{array}{rll} (\textit{Canonical types}) & \tau^c ::= & \mathsf{unit} \mid \tau_1^c \to \tau_2^c \mid \tau_1^c \times \tau_2^c \mid \tau_1^c + \tau_2^c \mid \mathsf{ref}\ \tau^c \mid \forall \alpha : k.\tau^c \\ & & \mid \lambda \alpha : k.\tau^c \mid \tau^n \mid (V : K) \to \tau^c \mid (V : K) \times \tau^c \\ & & \mid \lambda V : K.\tau^c \\ (\textit{Neutral types}) & \tau^n ::= & \alpha \mid \mu \alpha : k.\tau^c \mid \tau_1^n\ \tau_2^c \mid \tau^n\ T \end{array}$$

Figure 6.10: VeriML computational language: definitions used in metatheory

$$\frac{\vdash e : \tau \qquad e \longrightarrow e'}{\vdash e' : \tau}\ \textbf{Preservation} \qquad \frac{\vdash e : \tau \qquad e \neq v}{\exists e'.e \longrightarrow e'}\ \textbf{Progress}$$

In a way, we have presented the bulk of the proof already: establishing the metatheoretic proofs for λHOL and its extensions in Chapters 4 and 5 is the main effort required. We directly use these proofs in order to prove type-safety for the new constructs of VeriML; and type-safety for the constructs of the ML core is entirely standard [e.g. Pierce, 2002, Harper, 2011].

We give some needed definitions in Figure 6.10 and proceed to prove a number of auxiliary lemmas.

Lemma 6.1.1 *(Distributivity of computational substitutions with extension substitution application)*

1. $(\tau[\tau'/\alpha]) \cdot \sigma_\Psi = \tau \cdot \sigma_\Psi[\tau' \cdot \sigma_\Psi/\alpha]$
2. $(e[\tau/\alpha]) \cdot \sigma_\Psi = e \cdot \sigma_\Psi[\tau \cdot \sigma_\Psi/\alpha]$
3. $(e[e'/x]) \cdot \sigma_\Psi = e \cdot \sigma_\Psi[e' \cdot \sigma_\Psi/x]$

Proof. By induction on the structure of τ or e. □

Lemma 6.1.2 *(Normalization for types) For every type τ, there exists a canonical type τ^c such that $\tau =_\beta \tau^c$.*

Proof. The type and kind level of the computational language can be viewed as a simply-typed lambda calculus, with all type formers other than $\lambda\alpha : k.\tau$ and $\lambda V : K.\tau$ being viewed as constant applications. Reductions of the form $(\lambda V : K.\tau)\ T \to_\beta \tau[T/V]$ do not generate any new redeces of either form; therefore normalization for STLC directly gives us the desired. □

Based on this lemma, we can view types modulo β-equivalence and only reason about their canonical forms.

Lemma 6.1.3 *(Extension substitution application for computational language)*

$$1.\ \frac{\Psi \vdash \Gamma\ wf \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash \Gamma \cdot \sigma_\Psi\ wf} \qquad 2.\ \frac{\Psi \vdash k\ wf \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash k \cdot \sigma_\Psi\ wf}$$

$$3.\ \frac{\Psi;\ \Gamma \vdash \tau : k \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi';\ \Gamma \cdot \sigma_\Psi \vdash \tau \cdot \sigma_\Psi : k \cdot \sigma_\Psi} \qquad 4.\ \frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi';\ \mathcal{M};\ \Gamma \cdot \sigma_\Psi \vdash e \cdot \sigma_\Psi : \tau \cdot \sigma_\Psi}$$

Proof. By structural induction on the typing derivation for k, τ or e. We prove two representative cases.

Case CTyp-LamHol**.**

$$\left(\frac{\Psi,\ V : K;\ \Gamma \vdash \tau : k}{\Psi;\ \Gamma \vdash (\lambda V : K.\tau) : (\Pi V : K.k)} \right)$$

By induction hypothesis for τ with the substitution $\sigma'_\Psi = \sigma_\Psi,\ V/V$ typed as
$\Psi',\ V : K \cdot \sigma_\Psi \vdash (\sigma_\Psi,\ V/V) : (\Psi,\ V : K)$, we get:
$\Psi',\ V : K \cdot \sigma_\Psi;\ \Gamma \cdot \sigma'_\Psi \vdash \tau \cdot \sigma'_\Psi : k \cdot \sigma'_\Psi$
Note that V/V is stands either for $([\Phi \cdot \sigma_\Psi]\, X/id_{\Phi \cdot \sigma_\Psi})/X$ when $V = X$ and $K = [\Phi]\, t'$; or for $([\Phi \cdot \sigma_\Psi]\, \phi)/\phi$ when $V = \phi$ and $K = [\Phi]\, ctx$, as is understood from the inclusion of extension variables into extension terms defined in Figure 5.3.
From the fact that $\Psi \vdash \Gamma$ wf we get that $\Gamma \cdot \sigma'_\Psi = \Gamma \cdot \sigma_\Psi$. We also have that $\tau \cdot \sigma'_\Psi = \tau \cdot \sigma_\Psi$

and $k \cdot \sigma'_\Psi = k \cdot \sigma_\Psi$. *(Note that such steps are made more precise in our Technical report [Stampoulis and Shao, 2012b] through the use of hybrid deBruijn variables for Ψ)*

Thus:

$\Psi',\ V : K \cdot \sigma_\Psi;\ \Gamma \cdot \sigma_\Psi \vdash \tau \cdot \sigma_\Psi : k \cdot \sigma_\Psi$

Using the same typing rule CTyp-LamHol we get:

$\Psi';\ \Gamma \cdot \sigma_\Psi \vdash (\lambda V : K \cdot \sigma_\Psi . \tau \cdot \sigma_\Psi) : (\Pi V : K \cdot \sigma_\Psi . k \cdot \sigma_\Psi)$, which is the desired.

Case CExp-ΠHolE**.**

$$\left(\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash e : (V : K) \to \tau \qquad \Psi \vdash T : K}{\Psi;\ \mathcal{M};\ \Gamma \vdash e\ T : \tau \cdot (id_\Psi,\ T/V)} \right)$$

By induction hypothesis for e we have:

$\Psi';\ \mathcal{M};\ \Gamma \cdot \sigma_\Psi \vdash e \cdot \sigma_\Psi : (V : K \cdot \sigma_\Psi) \to \tau \cdot \sigma_\Psi$

Using the extension substitution theorem of λHOL (Theorem 4.2.6) for T we get:

$\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi$

Using the same typing rule we get:

$\Psi';\ \mathcal{M};\ \Gamma \cdot \sigma_\Psi \vdash (e \cdot \sigma_\Psi)\ (T \cdot \sigma_\Psi) : (\tau \cdot \sigma_\Psi \cdot (id_{\Psi'},\ T \cdot \sigma_\Psi / V))$

Through properties of extension substitution application we have that:

$\sigma_\Psi \cdot (id_{\Psi'},\ T \cdot \sigma_\Psi / V) = (id_\Psi,\ T/V) \cdot \sigma_\Psi$ thus this is the desired.

Case CExp-HolMatch**.**

$$\left(\frac{\Psi \vdash T : K \qquad \Psi,\ V : K;\ \Gamma \vdash \tau : \star \qquad \Psi \mid^*_p \Psi_u > T_P : K \qquad \Psi,\ \Psi_u;\ \mathcal{M};\ \Gamma \vdash e' : \tau \cdot (id_\Psi,\ T_P/V)}{\Psi;\ \mathcal{M};\ \Gamma \vdash \begin{array}{l} \mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau \\ \mathsf{with}\ \Psi_u . T_P \mapsto e' \end{array} : \tau \cdot (id_\Psi,\ T/V) + \mathsf{unit}} \right)$$

We have:

$\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi$, through Theorem 4.2.6 for T

$\Psi',\ V : K \cdot \sigma_\Psi;\ \Gamma \cdot \sigma_\Psi \vdash \tau \cdot \sigma_\Psi : \star$, using part 3 for τ with $\sigma'_\Psi = \sigma_\Psi,\ V/V$

$\Psi' \mid^*_p \Psi_u \cdot \sigma_\Psi > T_P \cdot \sigma_\Psi : K \cdot \sigma_\Psi$, directly using the extension substitution application theorem for patterns (Theorem 5.1.7)

$\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash e' \cdot (\sigma_\Psi,\ id_{\Psi_u}) : \tau \cdot (id_\Psi,\ T_P/V) \cdot (\sigma_\Psi,\ id_{\Psi_u})$, through induction hypothesis

for e' with $\sigma_\Psi = (\sigma_\Psi,\ id_{\Psi_u})$ typed as $\Psi';\ \Psi_u \cdot \sigma_\Psi \vdash (\sigma_\Psi,\ id_{\Psi_u}) : (\Psi,\ \Psi_u)$

$\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash e' \cdot \sigma_\Psi : \tau \cdot (id_{\Psi'},\ T_P \cdot \sigma_\Psi / V)$, through properties of extension substitutions

Thus using the same typing rule we get:

$$\Psi';\ \mathcal{M};\ \Gamma \cdot \sigma_\Psi \vdash \begin{array}{l}\mathsf{holmatch}\ T \cdot \sigma_\Psi \\ \mathsf{return}\ V : K \cdot \sigma_\Psi . \tau \cdot \sigma_\Psi \\ \mathsf{with}\ \Psi_u \cdot \sigma_\Psi . T_P \cdot \sigma_\Psi \mapsto e' \cdot \sigma_\Psi\end{array} : (\tau \cdot \sigma_\Psi) \cdot (id_{\Psi'},\ (T \cdot \sigma_\Psi)/V) + \mathsf{unit}$$

This is the desired since $(\tau \cdot \sigma_\Psi) \cdot (id_{\Psi'},\ (T \cdot \sigma_\Psi)/V) = (\tau \cdot (id_\Psi,\ T/V)) \cdot \sigma_\Psi$. □

Lemma 6.1.4 *(Computational substitution)*

$$1.\ \frac{\Psi,\ \Psi' \vdash \Gamma,\ \alpha' : k',\ \Gamma'\ wf \qquad \Psi;\ \Gamma \vdash \tau' : k'}{\Psi,\ \Psi' \vdash \Gamma,\ \Gamma'[\tau'/\alpha']\ wf}$$

$$2.\ \frac{\Psi,\ \Psi';\ \Gamma,\ \alpha' : k',\ \Gamma' \vdash \tau : k \qquad \Psi;\ \Gamma \vdash \tau' : k'}{\Psi,\ \Psi';\ \Gamma,\ \Gamma'[\tau'/\alpha'] \vdash \tau[\tau'/\alpha'] : k}$$

$$3.\ \frac{\Psi,\ \Psi';\ \mathcal{M};\ \Gamma,\ \alpha' : k',\ \Gamma' \vdash e : \tau \qquad \Psi;\ \Gamma \vdash \tau' : k'}{\Psi,\ \Psi';\ \mathcal{M};\ \Gamma,\ \Gamma'[\tau'/\alpha'] \vdash e[\tau'/\alpha'] : \tau[\tau'/\alpha']}$$

$$4.\ \frac{\Psi,\ \Psi';\ \mathcal{M};\ \Gamma,\ x' : \tau',\ \Gamma' \vdash e : \tau \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e' : \tau'}{\Psi,\ \Psi';\ \mathcal{M};\ \Gamma,\ \Gamma' \vdash e[e'/x'] : \tau}$$

Proof. By structural induction on the typing derivations for Γ, τ or e. □

Theorem 6.1.5 *(Preservation)*

$$\frac{\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau \qquad (\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ ) \qquad \mu \sim \mathcal{M}}{\exists \mathcal{M}'.(\quad \bullet;\ \mathcal{M}';\ \bullet \vdash e' : \tau \qquad \mathcal{M} \subseteq \mathcal{M}' \qquad \mu' \sim \mathcal{M}' \quad)}$$

Proof. By induction on the derivation of $(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )$. In the following, when we don't specify a different μ', we have that $\mu' = \mu$. We present the λHOL-related constructs first; the cases for the ML core follow.

Case Op-ΠHol-Beta**.**

$$\left[(\ \mu\ ,\ (\lambda V : K.e)\ T\ ) \longrightarrow (\ \mu\ ,\ e \cdot (T/V)\ ) \right]$$

By typing inversion we have:

$\bullet;\ \mathcal{M};\ \bullet \vdash (\lambda V : K.e) : (V : K) \to \tau',\quad \bullet \vdash T : K,\quad \tau = \tau' \cdot (T/V)$

By further typing inversion for $\lambda V : K.e$ we get:

$V : K;\ \mathcal{M};\ \bullet \vdash e : \tau'$

For $\sigma_\Psi = (\bullet,\ T/V)$ (also written as $\sigma_\Psi = (T/V)$) we have directly from typing for T:

$\bullet \vdash (T/V) : (V : K)$

Using Lemma 6.1.3 for σ_Ψ we get that:

$\bullet;\ \mathcal{M};\ \bullet \vdash e \cdot (T/V) : \tau' \cdot (T/V)$

Case Op-ΣHol-Unpack**.**

$$\left[\Big(\ \mu\ ,\ \mathsf{let}\ \langle\ V,\ x\ \rangle = \langle\ T,\ v\ \rangle_{(V:K)\times\tau''}\ \mathsf{in}\ e'\ \Big) \longrightarrow (\ \mu\ ,\ (e' \cdot (T/V))[v/x]\ ) \right]$$

By inversion of typing we get:

$\bullet;\ \mathcal{M};\ \bullet \vdash \langle\ T,\ v\ \rangle_{(V:K)\times\tau''} : (V : K) \times \tau';\qquad V : K;\ \mathcal{M};\ x : \tau' \vdash e' : \tau;\qquad \bullet;\ \bullet \vdash \tau : \star$

By further typing inversion for $\langle\ T,\ V : K\ \rangle\, \tau'' v$ we get:

$\tau'' = \tau';\qquad \bullet \vdash T : K;\qquad V : K;\ \bullet \vdash \tau' : \star;\qquad \bullet;\ \mathcal{M};\ \bullet \vdash v : \tau' \cdot (T/V)$

First by Lemma 6.1.3 for e' with $\sigma_\Psi = T/V$ we get:

$\bullet;\ \mathcal{M};\ x : \tau' \cdot (T/V) \vdash e' \cdot (T/V) : \tau \cdot (T/V)$

We trivially have that $\tau \cdot (T/V) = \tau$, thus this equivalent to:

$\bullet;\ \mathcal{M};\ x : \tau' \cdot (T/V) \vdash e' \cdot (T/V) : \tau$

Last by Lemma 6.1.4 for $[v/x]$ we get the desired:

$\bullet;\ \mathcal{M};\ \bullet \vdash (e' \cdot (T/V))[v/x] : \tau$

Case Op-HolMatch**.**

$$\left[\frac{T_P \sim T = \sigma_\Psi}{(\ \mu\ ,\ \mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_u.T_P \mapsto e'\ ) \longrightarrow (\ \mu\ ,\ \mathsf{inj}_1\ (e' \cdot \sigma_\Psi)\ )} \right]$$

By typing inversion we get:

$\bullet \vdash T : K,\ \ V : K;\ \bullet \vdash \tau' : \star,\ \ \bullet \vdash^*_p \Psi_u > T_P : K,\ \ \Psi_u \vdash e' : \tau' \cdot (T_P/V),\ \ \tau = \tau' \cdot (T/V) + \mathsf{unit}$

Directly by soundness of pattern matching (Lemma 5.1.13) we get:

$\bullet \vdash \sigma_\Psi : \Psi_u, \quad T_P \cdot \sigma_\Psi = T$

Through Lemma 6.1.3 for e' and σ_Ψ we have:

$\bullet \vdash e' \cdot \sigma_\Psi : \tau' \cdot (T_P/V) \cdot \sigma_\Psi$

We can now use the typing rule CExp-SumI to get the desired, since:

$(T_P/V) \cdot \sigma_\Psi = (T_P \cdot \sigma_\Psi/V) = (T/V)$

Case Op-HolNoMatch**.**

$$\left[\frac{T_P \sim T = \mathsf{fail}}{(\ \mu\ ,\ \mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_u.T_P \mapsto e'\ ) \longrightarrow (\ \mu\ ,\ \mathsf{inj}_2\ ()\ )}\right]$$

Trivial using the typing rules CExp-SumI and CExp-Unit. *Note:* Though completeness for pattern matching is not used for type-safety, it guarantees that the rule Op-HolNoMatch will not be used unless no matching substitution exists.

Case Op-Env**.**

$$\left[\frac{(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )}{(\ \mu\ ,\ \mathcal{E}[e]\ ) \longrightarrow (\ \mu'\ ,\ \mathcal{E}[e']\ )}\right]$$

By induction hypothesis for $(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )$ we get a $\mathcal{M}'$ such that $\mathcal{M} \subseteq \mathcal{M}'$, $\mu' \sim \mathcal{M}'$ and $\bullet;\ \mathcal{M};\ \bullet \vdash e' : \tau$. By inversion of typing for $\mathcal{E}[e]$ and re-application of the same typing rule for $\mathcal{E}[e']$ we get that $\bullet;\ \mathcal{M}';\ \bullet \vdash \mathcal{E}[e'] : \tau$.

Case Op-Beta**.**

$$\left[(\ \mu\ ,\ (\lambda x : \tau.e)\ v\ ) \longrightarrow (\ \mu\ ,\ e[v/x]\ )\right]$$

By inversion of typing we get: $\bullet;\ \mathcal{M};\ \bullet \vdash \lambda x : \tau.e : \tau' \to \tau, \quad \bullet;\ \mathcal{M};\ \bullet \vdash v : \tau'$

By further typing inversion for $\lambda x : \tau.e$ we get: $\bullet;\ \mathcal{M};\ x : \tau \vdash e : \tau$

By lemma 6.1.4 for $[v/x]$ we get: $\bullet;\ \mathcal{M};\ \bullet \vdash e[v/x] : \tau$

Case Op-Proj**.**

$$\left[(\ \mu\ ,\ \mathsf{proj}_i(v_1,\ v_2)\ ) \longrightarrow (\ \mu\ ,\ v_i\ )\right]$$

By typing inversion we get: $\bullet;\ \mathcal{M};\ \bullet \vdash (v_1,\ v_2) : \tau_1 \times \tau_2, \quad \tau = \tau_i$

By further inversion for $(v_1,\ v_2)$ we have: $\bullet;\ \mathcal{M};\ \bullet \vdash v_i : \tau_i$

Case Op-Case**.**

$$\left[(\ \mu\ ,\ \mathsf{case}(\mathsf{inj}_i\ v,\ x.e_1,\ x.e_2)\ ) \longrightarrow (\ \mu\ ,\ e_i[v/x]\ ) \right]$$

By typing inversion we get: $\bullet;\ \mathcal{M};\ \bullet \vdash v : \tau_i;\quad \bullet;\ \mathcal{M};\ x : \tau_i \vdash e_i : \tau$

Using the lemma 6.1.4 for [v/x] we get: $\bullet;\ \mathcal{M};\ \bullet \vdash e_i[v/x] : \tau$

Case Op-Unfold**.**

$$\left[(\ \mu\ ,\ \mathsf{unfold}\ (\mathsf{fold}\ v)\ ) \longrightarrow (\ \mu\ ,\ v\ ) \right]$$

By typing inversion we get: $\bullet;\ \bullet \vdash \mu\alpha : k.\tau' : k;\quad \bullet;\ \mathcal{M};\ \bullet \vdash \mathsf{fold}\ v : (\mu\alpha : k.\tau')\ a_1\ a_2\ \cdots\ a_n;$

every $a_i = \tau_i$ or T_i; $\quad \tau = \tau'[\mu\alpha : k.\tau']\ a_1\ a_2\ \cdots\ a_n$

By further typing inversion for $\mathsf{fold}\ v$ we have $\bullet;\ \mathcal{M};\ \bullet \vdash v : \tau'[\mu\alpha : k.\tau/\alpha]\ a_1\ a_2\ \cdots\ a_n$

Case Op-NewRef**.**

$$\left[\frac{\neg(l \mapsto \_ \in \mu)}{(\ \mu\ ,\ \mathsf{ref}\ v\ ) \longrightarrow (\ (\mu,\ l \mapsto v)\ ,\ l\ )} \right]$$

By typing inversion we get: $\bullet;\ \mathcal{M};\ \bullet \vdash v : \tau'$

For $\mathcal{M}' = \mathcal{M},\ l : \tau$ and $\mu' = \mu,\ l \mapsto v$ we have that $\mu' \sim \mathcal{M}'$ and $\bullet;\ \mathcal{M}';\bullet \vdash l : \mathsf{ref}\ \tau'$.

Case Op-Assign**.**

$$\left[\frac{l \mapsto \_ \in \mu}{(\ \mu\ ,\ l := v\ ) \longrightarrow (\ \mu[l \mapsto v]\ ,\ ()\ )} \right]$$

By typing inversion get: $\bullet;\ \mathcal{M};\ \bullet \vdash l : \mathsf{ref}\ \tau';\quad \bullet;\ \mathcal{M};\ \bullet \vdash v : \tau'$

Thus for $\mu' = \mu[l \mapsto v]$ we have that $\mu' \sim \mathcal{M}$. Obviously $\bullet;\ \mathcal{M};\bullet \vdash () : \mathsf{unit}$.

Case Op-Deref**.**

$$\left[\frac{l \mapsto v \in \mu}{(\ \mu\ ,\ !l\ ) \longrightarrow (\ \mu\ ,\ v\ )} \right]$$

By typing inversion get: $\bullet;\ \mathcal{M};\ \bullet \vdash l : \mathsf{ref}\ \tau$

By inversion of $\mu \sim \mathcal{M}$ get:

$\bullet;\ \mathcal{M};\ \bullet \vdash v : \tau$, which is the desired.

Case Op-PolyInst**.**

$$\left[(\ \mu\ ,\ (\Lambda\alpha : k.e)\ \tau''\ ) \longrightarrow (\ \mu\ ,\ e[\tau''/\alpha]\ ) \right]$$

By typing inversion we get: $\bullet;\ \mathcal{M};\ \bullet \vdash \Lambda\alpha : k.e : \Pi\alpha : k.\tau';\quad \bullet;\ \bullet \vdash \tau'' : k;\quad \tau = \tau'[\tau''/\alpha]$

By further typing inversion for $\Lambda\alpha : k.e$ we get: $\bullet;\ \mathcal{M};\ \alpha : k \vdash e : \tau'$

Using Lemma 6.1.4 for e and $[\tau''/\alpha]$ we get: $\bullet;\ \mathcal{M};\ \bullet \vdash e[\tau''/\alpha] : \tau'[\tau''/\alpha]$

Case Op-Fix**.**

$$\left[(\ \mu\ ,\ \mathsf{fix}\ x : \tau.e\ ) \longrightarrow (\ \mu\ ,\ e[\mathsf{fix}\ x : \tau.e/x]\ ) \right]$$

By typing inversion get: $\bullet;\ \mathcal{M};\ x : \tau \vdash e : \tau$

By application of Lemma 6.1.4 for e and $[\mathsf{fix}\ x : \tau.e/x]$ we get:

$\bullet;\ \mathcal{M};\ \bullet \vdash e[\mathsf{fix}\ x : \tau.e/x] : \tau$ □

Lemma 6.1.6 *(Canonical forms) If* $\bullet;\ \mathcal{M};\ \bullet \vdash v : \tau$ *then:*

If $\tau = \cdots$	*then exists* $\cdots$	*such that* $v = \cdots$
$(V : K) \to \tau'$	e	$\Lambda V : K.e$
$(V : K) \times \tau'$	T, v'	$\langle\ T,\ v'\ \rangle_{(V:K)\times\tau''}$ *with* $\tau' =_\beta \tau''$
unit		$()$
$\tau_1 \to \tau_2$	e	$\lambda x : \tau_1.e$
$\tau_1 \times \tau_2$	v_1, v_2	$(v_1,\ v_2)$
$\tau_1 + \tau_2$	v'	$\mathsf{inj}_1\ v'$ *or* $\mathsf{inj}_2\ v'$
$(\mu\alpha : k.\tau')\ a_1\ a_2\ \cdots\ a_n$	v'	$\mathsf{fold}\ v'$
$\mathsf{ref}\ \tau'$	l	l
$\forall\alpha : k.\tau'$	e	$\Lambda\alpha : k.e$

Proof. Directly by typing inversion on the derivation for v. □

Theorem 6.1.7 *(Progress)*

$$\frac{\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau \qquad \mu \sim \mathcal{M} \qquad e \neq v}{\exists \mu', e'.\ (\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )}$$

Proof. By induction on the typing derivation for e. We do not consider cases where $e = \mathcal{E}[e'']$, without loss of generality: by typing inversion we can get that e'' is well-typed under the empty context, so by induction hypothesis we get μ'', e''' such that $(\ \mu\ ,\ e''\ ) \longrightarrow (\ \mu''\ ,\ e'''\ )$. Thus set $\mu' = \mu''$ and $e' = \mathcal{E}[e''']$; by Op-Env get $(\ \mu\ ,\ \mathcal{E}[e'']\ ) \longrightarrow (\ \mu\ ,\ \mathcal{E}[e''']\ )$. Most cases follow from use of canonical forms lemma (Lemma 6.1.6), typing inversion and application of the appropriate operational semantics rule. We give the λHOL-related cases as a sample.

Case CExp-ΠHolE**.**

$$\left(\frac{\bullet;\ \mathcal{M};\ \bullet \vdash v : (V : K) \to \tau' \qquad \bullet \vdash T : K}{\bullet;\ \mathcal{M};\ \bullet \vdash v\ T : \tau' \cdot (T/V)}\right)$$

By use of canonical forms lemma we get that $v = \lambda V : K.e$. Through typing inversion for v we get $V : K;\ \mathcal{M};\ \bullet \vdash e : \tau'$. Thus $e \cdot (T/V)$ is well-defined (it does not depend on variables other than V). Using Op-ΠHol-Beta we get $e' = e \cdot (T/V)$ with the desired properties for $\mu' = \mu$.

Case CExp-ΣHolE**.**

$$\left(\frac{\bullet;\ \mathcal{M};\ \bullet \vdash v : (V : K) \times \tau' \qquad V : K;\ \mathcal{M};\ x : \tau' \vdash e'' : \tau \qquad \bullet;\ \bullet \vdash \tau : \star}{\bullet;\ \mathcal{M};\ \bullet \vdash \mathsf{let}\ \langle\ V,\ x\ \rangle = v\ \mathsf{in}\ e'' : \tau}\right)$$

Through canonical forms lemma we get that $v = \langle\ T,\ v'\ \rangle_{(V:K)\times\tau'}$. From typing of e' we have that $e'' \cdot (T/V)$ is well-defined. Therefore using Op-ΣHol-Unpack we get $e' = (e'' \cdot (T/V))[v/x]$ with the desired properties.

Case CExp-HolMatch**.**

$$\left(\frac{\bullet \vdash T : K \qquad V : K;\ \bullet \vdash \tau' : \star \qquad \bullet \mid\!\!\mid^{*}_{p} \bullet_u > T_P : K \qquad \bullet_u;\ \mathcal{M};\ \bullet \vdash e'' : \tau' \cdot (T_P/V)}{\bullet;\ \mathcal{M};\ \bullet \vdash \begin{array}{l}\mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau' \\ \mathsf{with}\ \bullet_u\ .T_P \mapsto e''\end{array} : \tau' \cdot (T/V) + \mathsf{unit}}\right)$$

We split cases on whether $T_P \sim T = \sigma_\Psi$ or $T_P \sim T = \mathsf{fail}$. In the first case rule Op-HolMatch applies, taking into account the fact that $e'' \cdot \sigma_\Psi$ is well-defined based on typing for e''. In the second case Op-NoHolMatch directly applies. □

Theorem 6.1.8 *(Type safety for VeriML)*

$$\frac{\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau \qquad \mu \sim \mathcal{M} \qquad e \neq v}{\exists \mu', \mathcal{M}', e'.(\quad (\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ ) \qquad \bullet;\ \mathcal{M}';\ \bullet \vdash e' : \tau \qquad \mu' \sim \mathcal{M}' \quad)}$$

Proof. Directly by combining Theorem 6.1.5 and Theorem 6.1.7. □

6.2 Derived pattern matching forms

The pattern matching construct available in the above definition of VeriML is the simplest possible: it allows matching a scrutinee against a single pattern. In this section, we will cover a set of increasingly more complicated pattern matching forms and show how they are derived forms of the simple case. For example, we will add support for multiple branches and for multiple simultaneous scrutinees. We will present each extension as *typed syntactic sugar*: every new construct will have its own typing rules and operational semantics, yet we will give a syntax- or type-directed translation into the original VeriML constructs. Furthermore, we will show that the translation respects the semantics of the new construct. In this way, the derived forms are indistinguishable to programmers from the existing forms, as they can be understood through their typing rules and semantics; yet our metatheoretic proofs such as type safety do not need to be adapted.

The first extension is *multiple branches support* presented in Figure 6.11. This is a simple extension that allows us to match the same scrutinee against multiple patterns, for example:

$$\mathsf{holMultMatch}\ P\ \mathsf{with}\ Q \wedge R \mapsto e_1 \mid Q \vee R \mapsto e_2$$

Syntax

$$
\begin{array}{l}
e ::= \cdots \\
\quad \mid \mathsf{holMultMatch}\ T\ \mathsf{return}\ V : K.\tau\ \mathsf{with}\ \overrightarrow{\Psi_u.T_P \mapsto e'}
\end{array}
$$

Typing

$$
\frac{\begin{array}{c}\Psi \vdash T : K \qquad \Psi,\ V : K;\ \Gamma \vdash \tau : \star \\ \forall i.(\quad \Psi \mid\!\!\stackrel{*}{\vdash}_p \Psi_{u,i} > T_{P,i} : K \qquad \Psi,\ \Psi_{u,i};\ \mathcal{M};\ \Gamma \vdash e_i' : \tau \cdot (id_\Psi,\ T_{P,i}/V) \quad)\end{array}}{\Psi;\ \mathcal{M};\ \Gamma \vdash \left(\begin{array}{l}\mathsf{holMultMatch}\ T \\ \mathsf{return}\ V : K.\tau \\ \mathsf{with}\ \overrightarrow{\Psi_u.T_P \mapsto e'}\end{array}\right) : \tau \cdot (id_\Psi,\ T/V) + \mathsf{unit}}
$$

Semantics

$$
\frac{T_{P,1} \sim T = \sigma_\Psi}{\mathsf{holMultMatch}\ T\ \mathsf{with}\ (\Psi_{u,1}.T_{P,1} \mapsto e_1 \mid \cdots \mid \Psi_{u,n}.T_{P,n} \mapsto e_n) \longrightarrow \mathsf{inj}_1\ (e_1 \cdot \sigma_\Psi)}
$$

$$
\frac{T_{P,1} \sim T = \mathsf{error}}{\begin{array}{r}\mathsf{holMultMatch}\ T\ \mathsf{with}\ (\Psi_{u,1}.T_{P,1} \mapsto e_1 \mid \cdots \mid \Psi_{u,n}.T_{P,n} \mapsto e_n) \longrightarrow \\ \mathsf{holMultMatch}\ T\ \mathsf{with}\ (\Psi_{u,2}.T_{P,2} \mapsto e_2 \mid \cdots \mid \Psi_{u,n}.T_{P,n} \mapsto e_n)\end{array}}
$$

$$
\frac{}{\mathsf{holMultMatch}\ T\ \mathsf{with}\ () \longrightarrow \mathsf{inj}_2\ ()}
$$

Translation

$$
\begin{array}{l}
[\![\mathsf{holMultMatch}\ T\ \mathsf{with}\ \overrightarrow{\Psi_u.T_P \mapsto e}]\!] = \\
\quad \mathsf{case}(\mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_{u,1}.T_{P,1} \mapsto e_1,\ x.\mathsf{inj}_1\ x, \\
\qquad y.\mathsf{case}(\mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_{u,2}.T_{P,2} \mapsto e_2,\ x.\mathsf{inj}_1\ x, \\
\qquad\quad y.\cdots(\mathsf{case}(\mathsf{holmatch}\ T\ \mathsf{with}\ T_{u,n}.T_{P,n} \mapsto e_n,\ x.\mathsf{inj}_1\ x,\ y.\mathsf{inj}_2\ y))))
\end{array}
$$

Figure 6.11: Derived VeriML pattern matching constructs: Multiple branches

The pattern matches are attempted sequentially, with no backtracking; the branch with the first pattern that matches successfully is followed and the rest of the branches are forgotten. The pattern list is not checked for coverage or for non-redundancy. The construct returns a type of the form $\tau + \mathsf{unit}$; the unit value is returned in the case where no pattern successfully matches, similarly to the base pattern matching construct.

We show that our definitions of the typing rule and semantics for this construct are sensible by proving that they correspond to the typing and semantics of its translated form.

Lemma 6.2.1 *(Pattern matching with multiple branches is well-defined)*

$$1.\ \frac{\Psi;\ \mathcal{M};\ \Gamma \vdash \left(\mathit{holMultMatch}\ T\ \mathit{with}\ \overrightarrow{\Psi_u.T_P \mapsto e'}\right) : \tau'}{\Psi;\ \mathcal{M};\ \Gamma \vdash \left[\!\left[\mathit{holMultMatch}\ T\ \mathit{with}\ \overrightarrow{\Psi_u.T_P \mapsto e'}\right]\!\right] : \tau'}$$

$$2.\ \frac{\mathit{holMultMatch}\ T\ \mathit{with}\ \overrightarrow{\Psi_u.T_P \mapsto e} \longrightarrow e'}{\left[\!\left[\mathit{holMultMatch}\ T\ \mathit{with}\ \overrightarrow{\Psi_u.T_P \mapsto e}\right]\!\right] \longrightarrow^* e'}$$

Proof. Part 1 follows directly from typing; similarly part 2 is a direct consequence of the operational semantics for the translated form. □

The second extension we will consider is more subtle; we will introduce it through an example. Consider the case of a tactic that accepts a proposition and its proof as arguments; it then returns an equivalent proposition as well as a proof of it.

$$\begin{aligned}\mathsf{tactic}\ &:\ \ (P : \mathit{Prop},\ H : P) \to \left(P' : \mathit{Prop},\ H' : P'\right)\\ &=\ \ \lambda P : \mathit{Prop}.\lambda H : P.\mathsf{holMultMatch}\ P\ \mathsf{with}\\ &\qquad Q \wedge R \mapsto \langle\ R \wedge Q,\ \cdots\ \rangle\\ &\qquad Q \vee R \mapsto \langle\ R \vee Q,\ \cdots\ \rangle\end{aligned}$$

In each branch of the pattern match, we know that P is of the form $Q \wedge R$ or $Q \vee R$; yet the input proof is still typed as $H : P$ instead of $H : Q \wedge R$ or $H : Q \vee R$ respectively. The reason is that pattern matching only refines its return type but not the environment as well. We cannot thus directly deconstruct the proof H in order to fill in the missing output proofs. Yet an easy workaround exists by abstracting again over the input proof, owing to

Syntax

$$e ::= \cdots \mid \mathsf{holEnvMatch}\ V_s\ \mathsf{return}\ \tau\ \mathsf{with}\ \Psi_u.T_P \mapsto e'$$

Typing

$$\frac{\begin{array}{c}\Psi = \Psi_0,\ V_s : K,\ \Psi_1 \qquad \Psi;\ \Gamma \vdash \tau : \star \qquad \Psi \models^*_p \Psi_u > T_P : K \\ \sigma_\Psi = id_{\Psi_0},\ T_P/V_s,\ id_{\Psi_1} \qquad \Psi' = \Psi_0,\ V_s : K,\ \Psi_1 \cdot \sigma_\Psi \\ \Psi',\ \Psi_u;\ \mathcal{M};\ \Gamma \cdot \sigma_\Psi \vdash e' : \tau \cdot \sigma_\Psi\end{array}}{\Psi;\ \mathcal{M};\ \Gamma \vdash \left(\begin{array}{l}\mathsf{holEnvMatch}\ V_s \\ \mathsf{return}\ \tau \\ \mathsf{with}\ \Psi_u.T_P \mapsto e'\end{array}\right) : \tau \cdot \sigma_\Psi + \mathsf{unit}}$$

Semantics $(e \cdot \sigma_\Psi = e' \text{ when } \sigma_\Psi.V_s = T)$

$$\begin{array}{rl}\text{If } T = V_s' : & (\mathsf{holEnvMatch}\ V_s\ \mathsf{return}\ \tau\ \mathsf{with}\ \Psi_u.T_P \mapsto e') \cdot \sigma_\Psi = \\ & \mathsf{holEnvMatch}\ V_s'\ \mathsf{return}\ \tau \cdot \sigma_\Psi\ \mathsf{with}\ \Psi_u \cdot \sigma_\Psi.T_P \cdot \sigma_\Psi \mapsto e' \cdot \sigma_\Psi \\ \text{If } T \neq V_s' : & (\mathsf{holEnvMatch}\ V_s\ \mathsf{return}\ \tau\ \mathsf{with}\ \Psi_u.T_P \mapsto e') \cdot \sigma_\Psi = \\ & \mathsf{holmatch}\ T\ \mathsf{return}\ V_s.\tau \cdot \sigma_\Psi\ \mathsf{with}\ \Psi_u \cdot \sigma_\Psi.T_P \cdot \sigma_\Psi \mapsto e' \cdot \sigma_\Psi\end{array}$$

Translation

$$\frac{\Psi = \Psi_0,\ V_s : K,\ \Psi_1}{\begin{array}{l}[\![\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{holEnvMatch}\ V_s\ \mathsf{return}\ \tau\ \mathsf{with}\ \Psi_u.T_P \mapsto e : \tau']\!] = \\ (\mathsf{holmatch}\ V_s\ \mathsf{return}\ V_s : K.(\Psi) \to \Gamma \to \tau\ \mathsf{with}\ \Psi_u.T_P \mapsto \lambda\Psi.\lambda\Gamma.e)\ \Psi\ \Gamma\end{array}}$$

Figure 6.12: Derived VeriML pattern matching constructs: Environment-refining matching

the fact that the matching construct *does* refine the return type based on the pattern:

$$\begin{array}{rcl}\mathsf{tactic} & : & (P : Prop,\ H : P) \to \big(P' : Prop,\ H' : P'\big) \\ & = & \lambda P : Prop.\lambda H : P. \\ & & (\mathsf{holMultMatch}\ P \\ & & \mathsf{return}\ P' : Prop.(H : P') \to \big(P' : Prop,\ H' : P'\big)\ \mathsf{with} \\ & & \quad Q \wedge R \mapsto \lambda H : Q \wedge R.\langle\ R \wedge Q,\ \cdots\ \rangle \\ & & \quad Q \vee R \mapsto \lambda H : Q \vee R.\langle\ R \vee Q,\ \cdots\ \rangle)\ H\end{array}$$

More formally, we introduce *environment-refining pattern matching*: when the scrutinee of the pattern matching is a variable V, all the types in the current context that depend on V get refined by the current pattern. We introduce this construct in Figure 6.12. We only show the case of matching a scrutinee against a single pattern; multiple patterns can be supported as above. We are mostly interested in the typing for this construct. The

construct itself does not have operational semantics, as it requires that the scrutinee is a variable. Since the operational semantics only handle closed computational terms, an occurrence of this construct is impossible. The only semantics that we define have to do with extension substitution application: when the scrutinee variable V_s is substituted by a concrete term, the environment-refining construct is replaced with the normal, non-refining pattern matching construct. Our definition of this construct follows the approach of Crary and Weirich [1999], yet both the typing and semantics are unsatisfyingly complex. A cleaner approach would be the introduction of a distinguished computational type $T \approx T'$ reflecting the knowledge that the two extension terms unify, as well as casting constructs to make explicit use of such knowledge [similar to Goguen et al., 2006]; still this approach is beyond the scope of this section.

Lemma 6.2.2 *(Pattern matching with environment refining is well-defined)*

$$1.\ \frac{\Psi;\ \mathcal{M};\ \Gamma \vdash \left(\mathit{holEnvMatch}\ T\ \mathit{with}\ \Psi_u.T_P \mapsto e'\right) : \tau' \qquad \left[\!\left[\Psi;\ \mathcal{M};\ \Gamma \vdash \mathit{holEnvMatch}\ T\ \mathit{with}\ \Psi_u.T_P \mapsto e' : \tau'\right]\!\right] = e''}{\Psi;\ \mathcal{M};\ \Gamma \vdash e'' : \tau'}$$

$$2.\ \frac{\left(\mathit{holEnvMatch}\ V_s\ \mathit{with}\ \Psi_u.T_P \mapsto e\right) \cdot \sigma_\Psi = e' \qquad \left[\!\left[\Psi;\ \mathcal{M};\ \Gamma \vdash \mathit{holEnvMatch}\ T\ \mathit{with}\ \Psi_u.T_P \mapsto e' : \tau'\right]\!\right] = e''}{e'' \cdot \sigma_\Psi = e'}$$

Proof. Similarly as above; direct consequence of typing and extension substitution application. □

The last derived pattern matching form that we will consider is simultaneous matching of multiple scrutinees against multiple patterns. The pattern match only succeeds if all scrutinees can be matched against the corresponding pattern. This is an extension that is especially useful for comparing two terms structurally:

$$\mathsf{holSimMatch}\ P,\ Q\ \mathsf{with}\ A \wedge B,\ A' \wedge B' \mapsto e$$

Syntax

$$e ::= \cdots \mid \mathsf{holSimMatch}\ \overrightarrow{T}\ \mathsf{return}\ \overrightarrow{V : K}.\tau\ \mathsf{with}\ \overrightarrow{(\Psi_u.T_P)} \mapsto e'$$

Typing

$$\frac{\begin{array}{c}\forall i.\Psi \vdash T_i : K_i \qquad \Psi,\ V_1 : K_1,\ \cdots,\ V_n : K_n;\ \Gamma \vdash \tau : \star \\ \Psi \mid\!\!\stackrel{*}{p}\ \Psi_{u,1} > T_{P,1} : K_1 \qquad \Psi,\ \Psi_{u,1} \mid\!\!\stackrel{*}{p}\ \Psi_{u,2} > T_{P,2} : K_2 \qquad \cdots \\ \Psi,\ \Psi_{u,1},\ \cdots,\ \Psi_{u,n};\ \mathcal{M};\ \Gamma \vdash e' : \tau \cdot (id_\Psi,\ T_{P,1}/V_1,\ \cdots,\ T_{P,n}/V_n)\end{array}}{\Psi;\ \mathcal{M};\ \Gamma \vdash \left(\begin{array}{l}\mathsf{holSimMatch}\ \overrightarrow{T} \\ \mathsf{return}\ \overrightarrow{V : K}.\tau \\ \mathsf{with}\ \overrightarrow{\Psi_u.T_P} \mapsto e'\end{array}\right) : \tau \cdot (id_\Psi,\ T_1/V_1,\ \cdots,\ T_n/V_n) + \mathsf{unit}}$$

Semantics

$$\frac{T_{P,1} \sim T_1 = \sigma^1_\Psi \qquad \cdots \qquad T_{P,n} \sim T_n = \sigma^n_\Psi}{\mathsf{holSimMatch}\ \overrightarrow{T}\ \mathsf{with}\ \overrightarrow{(\Psi_{u,1}.T_{P,1})} \mapsto e \longrightarrow \mathsf{inj}_1\ (e \cdot (\sigma^1_\Psi,\ \cdots,\ \sigma^n_\Psi))}$$

$$\frac{T_{P,i} \sim T_i = \mathsf{error}}{\mathsf{holSimMatch}\ \overrightarrow{T}\ \mathsf{with}\ \overrightarrow{(\Psi_{u,1}.T_{P,1})} \mapsto e \longrightarrow \mathsf{inj}_2\ ()}$$

Translation

$$\begin{array}{l}\left[\!\left[\mathsf{holSimMatch}\ \overrightarrow{T}\ \mathsf{with}\ \overrightarrow{\Psi_u.T_P} \mapsto e\right]\!\right] = \\ \quad \mathsf{holmatch}\ T_1\ \mathsf{with}\ \Psi_{u,1}.T_{P,1} \mapsto \mathsf{do}\ \mathsf{holmatch}\ T_2\ \mathsf{with}\ \Psi_{u,2}.T_{P,2} \mapsto \cdots \mapsto \\ \qquad\qquad (\mathsf{holmatch}\ T_n\ \mathsf{with}\ \Psi_{u,n}.T_{P,n} \mapsto \mathsf{do}\ e)\end{array}$$

Figure 6.13: Derived VeriML pattern matching constructs: Simultaneous matching

We can also use it in combination with environment-refining matching in order to match a term whose type is not yet specified:

$$\lambda T : \mathit{Type}.\lambda t : T.\mathsf{holSimMatch}\ T,\ t\ \mathsf{with}\ \mathit{Nat},\ \mathsf{zero} \mapsto e$$

We present the details of this construct in Figure 6.13. Typing and semantics are straightforward; the only subtle point is that the unification variables of one pattern become exact variables in the following pattern, as matching one scrutinee against a pattern provides them with concrete instantiations.

Lemma 6.2.3 *(Simultaneous pattern matching is well-defined)*

$$1.\ \frac{\Psi;\ \mathcal{M};\ \Gamma \vdash \left(\mathit{holSimMatch}\ \overrightarrow{T}\ \mathit{with}\ \overrightarrow{(\Psi_u.T_P)} \mapsto e'\right) : \tau'}{\Psi;\ \mathcal{M};\ \Gamma \vdash \left[\!\left[\mathit{holSimMatch}\ \overrightarrow{T}\ \mathit{with}\ \overrightarrow{(\Psi_u.T_P)} \mapsto e'\right]\!\right] : \tau'}$$

$$2.\ \frac{\mathit{holSimMatch}\ \overrightarrow{T}\ \mathit{with}\ \overrightarrow{(\Psi_u.T_P)} \mapsto e \longrightarrow e'}{\left[\!\left[\mathit{holSimMatch}\ \overrightarrow{T}\ \mathit{with}\ \overrightarrow{(\Psi_u.T_P)} \mapsto e\right]\!\right] \longrightarrow^* e'}$$

Proof. Similar to Lemma 6.2.1. □

In the rest we will freely use the holmatch construct to refer to combinations of the above derived constructs.

6.3 Staging

In the simplify example presented in Section 2.3 and also in Section 5.2, we mentioned that we want to evaluate certain tactic calls at the time of definition of a new tactic. We use this in order to 'fill in' proof objects in the new tactic through existing automation code; furthermore, the proof objects are created once and for all when the tactic is defined and not every time the tactic is invoked. We will see more details of this feature of VeriML in Section 7.2, but for the time being it will suffice to say that it is the combination of two features: a *logic-level feature*, which is the transformation of λHOL open terms with respect to the extension context Ψ to equivalent closed terms, presented in Section 5.2; and a *computational-level feature*, namely the ability to evaluate subexpressions of an expression during a distinct evaluation stage prior to normal execution. This latter feature is supported by extending the VeriML computational language with a *staging construct* and is the subject of this section.

We will start with an informal description of staging. Note that the application we are interested in is not the normal use case of staging (namely, the ability to write programs that generate executable code). Our description is thus more geared to the use case we have in mind. Similarly, the staging construct we will add is quite limited compared to language

extensions such as MetaML [Taha and Sheard, 2000] but will suffice for our purposes.

Constant folding is an extremely common compiler optimization, where subexpressions composed only from constants are replaced by their result at compilation time. A simple example is the following:

$$\lambda n : \mathsf{int}.n * 10 * 10 \rightsquigarrow \lambda n : \mathsf{int}.n * 100$$

The subexpression $10 * 10$ can trivially be replaced by 100 yielding an equivalent program, avoiding the need to perform this multiplication at runtime. A more sophisticated version of the same optimization is constant propagation, which takes into account definitions of variables to constant values:

$$\lambda n : \mathsf{int}.\mathsf{let}\ x = 10\ \mathsf{in}\ n * x * x \rightsquigarrow \lambda n : \mathsf{int}.n * 100$$

Now consider the case where instead of a primitive operation like multiplication, we call a user-defined function with constant arguments, for example:

$$\lambda n : \mathsf{int}.n * \mathsf{factorial}(5)$$

We could imagine a similar optimization of evaluating the call to factorial at compilation time and replacing it with the result. Of course this optimization is not safe in the presence of side-effects and non-termination, as it changes the evaluation order of the program. Still we can support such an optimization if we make it a language feature: the user can annotate a particular constant subexpression to be evaluated at compile-time and replaced by its result:

$$\lambda n : \mathsf{int}.n * \{\mathsf{factorial}(5)\}_{static} \rightsquigarrow \lambda n : \mathsf{int}.n * 120$$

Compile-time evaluation is thus a distinct *static evaluation phase* where annotated subexpressions are evaluated to completion; another way to view this is that the original expression e is evaluated to a *residual expression* or *dynamic expression* e_d with all the static subexpressions replaced; the residual expression e_d can then be evaluated normally to a value. We can also view this as a form of staged computation, where the expression $\{\mathsf{factorial}(5)\}_{static}$ produces code that is to be evaluated at the next stage – namely the constant expression 120.

This feature is very useful in the case of VeriML if instead of integer functions like factorial we consider *proof expressions* with constant arguments – for example a call to a

decision procedure with a closed proposition as input. Note that we mean 'closed' with respect to runtime λHOL extension variables; it *can* be open with respect to the normal logical variable environment.

$$\begin{array}{l}\lambda\phi : ctx.\lambda P : [\phi]\, Prop. \\ \quad \mathsf{let}\ \langle\ V\ \rangle\ = \{\mathsf{tautology}\ [P' : Prop]\ ([P' : Prop]\, P \to P)\}_{static}\ \mathsf{in} \\ \quad \langle\ [\phi]\, V/[P/id_\phi]\ \rangle\end{array}$$

Here the call to tautology will be evaluated in the static evaluation phase, at the definition time of this function. The residual program e_d will be a function with a proof object instead:

$$\begin{array}{l}\lambda\phi : ctx.\lambda P : [\phi]\, Prop. \\ \quad \mathsf{let}\ \langle\ V\ \rangle\ = \langle\ [P' : Prop]\, \lambda H : P'.H\ \rangle\ \mathsf{in} \\ \quad \langle\ [\phi]\, V/[P/id_\phi]\ \rangle\end{array}$$

The actual construct we will support is slightly more advanced: we will allow static expressions to be bound to *static variables* and also for static expressions to depend on them. The actual staging construct we support is therefore written as $\mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e'$, where x is a static variable and e can only mention static variables. Returning to our analogy of the $\{\cdot\}_{static}$ construct to constant folding, the letstatic construct is the generalization of constant propagation. An example of its use would be to programmatically construct a proposition and then its proof:

$$\begin{array}{l}\lambda\phi : ctx.\lambda P : [\phi,\ x : Nat]\, Prop. \\ \quad \mathsf{letstatic}\ indProp\ =\ \mathsf{inductionPrincipleStatement}\ Nat\ \mathsf{in} \\ \quad \mathsf{letstatic}\ ind\ =\ \mathsf{inductionPrincipleProof}\ Nat\ \mathsf{indProp}\ \mathsf{in} \\ \quad \ldots\end{array}$$

We now present the formal details of our staging mechanism. Figure 6.14 gives the syntax extensions to expressions and contexts, adding the letstatic construct and the notion of static variables in the context Γ denoted as $x :_s \tau$. Figure 6.15 gives the typing rules and figure 6.16 adapts the operational semantics; all the syntactic classes used in the semantics are presented in Figure 6.14.

The typing rule CExp-LetStatic for the letstatic construct reflects the fact that only static variables are allowed in staged expressions e by removing non-static variables from the context through the context limiting operation $\Gamma|_{\text{static}}$. Through the two added typing rules we see that static evaluation has a *comonadic structure* with the rule CExp-StaticVar

$$
\begin{array}{rl}
\textit{(Expressions)} & e ::= \cdots \mid \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' \\
\textit{(Contexts)} & \Gamma ::= \cdots \mid \Gamma,\ x :_s \tau \\
\\
\textit{(Values)} & v ::= () \mid \lambda x : \tau.e_d \mid (v,\ v') \mid \mathsf{inj}_i\ v \mid \mathsf{fold}\ v \mid l \mid \Lambda\alpha : k.e_d \\
 & \quad \mid \lambda V : K.e_d \mid \langle\ T,\ e_d\ \rangle_{(V:K)\times\tau} \\
\textit{(Residual expressions)} & e_d ::= () \mid \lambda x : \tau.e_d \mid e_d\ e'_d \mid x \mid (e_d,\ e'_d) \mid \mathsf{proj}_i\ e_d \mid \mathsf{inj}_i\ e_d \\
 & \quad \mid \mathsf{case}(e_d,\ x.e'_d,\ x.e''_d) \mid \mathsf{fold}\ e_d \mid \mathsf{unfold}\ e_d \mid \mathsf{ref}\ e_d \\
 & \quad \mid e_d := e'_d \mid !e_d \mid l \mid \Lambda\alpha : k.e_d \mid e_d\ \tau \mid \mathsf{fix}\ x : \tau.e_d \\
 & \quad \mid \lambda V : K.e_d \mid e_d\ T \mid \langle\ T,\ e_d\ \rangle_{(V:K)\times\tau} \\
 & \quad \mid \mathsf{let}\ \langle\ V,\ x\ \rangle = e_d\ \mathsf{in}\ e'_d \\
 & \quad \mid \mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau\ \mathsf{with}\ \Psi_u.T' \mapsto e'_d \\
\textit{(Static binding evaluation contexts)} & \mathcal{S} ::= \mathcal{E}_s[\mathcal{S}] \mid \mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e' \mid \mathsf{letstatic}\ x\ =\ \mathcal{S}\ \mathsf{in}\ e' \\
 & \quad \mid \lambda x : \tau.\mathcal{S} \mid \mathsf{case}(e_d,\ x.\mathcal{S},\ x.e_2) \mid \mathsf{case}(e_d,\ x.e_d,\ x.\mathcal{S}) \\
 & \quad \mid \Lambda\alpha : k.\mathcal{S} \mid \mathsf{fix}\ x : \tau.\mathcal{S} \mid \lambda V : K.\mathcal{S} \\
 & \quad \mid \mathsf{let}\ \langle\ V,\ x\ \rangle = e_d\ \mathsf{in}\ \mathcal{S} \\
 & \quad \mid \mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau\ \mathsf{with}\ \Psi_u.T' \mapsto \mathcal{S} \\
\textit{(Static evaluation contexts)} & \mathcal{E}_s ::= \mathcal{E}_s\ e' \mid e_d\ \mathcal{E}_s \mid (\mathcal{E}_s,\ e) \mid (e_d,\ \mathcal{E}_s) \mid \mathsf{proj}_i\ \mathcal{E}_s \mid \mathsf{inj}_i\ \mathcal{E}_s \\
 & \quad \mid \mathsf{case}(\mathcal{E}_s,\ x.e_1,\ x.e_2) \mid \mathsf{fold}\ \mathcal{E}_s \mid \mathsf{unfold}\ \mathcal{E}_s \mid \mathsf{ref}\ \mathcal{E}_s \\
 & \quad \mid \mathcal{E}_s := e' \mid e_d := \mathcal{E}_s \mid !\mathcal{E}_s \mid \mathcal{E}_s\ \tau \mid \mathcal{E}_s\ T \\
 & \quad \mid \langle\ T,\ \mathcal{E}_s\ \rangle_{(V:K)\times\tau} \mid \mathsf{let}\ \langle\ V,\ x\ \rangle = \mathcal{E}_s\ \mathsf{in}\ e' \\
\textit{(Dynamic evaluation contexts)} & \mathcal{E} ::= \bullet \mid \mathcal{E}\ e_d \mid v\ \mathcal{E} \mid (\mathcal{E},\ e_d) \mid (v,\ \mathcal{E}) \mid \mathsf{proj}_i\ \mathcal{E} \mid \mathsf{inj}_i\ \mathcal{E} \\
 & \quad \mid \mathsf{case}(\mathcal{E},\ x.e'_d,\ x.e''_d) \mid \mathsf{fold}\ \mathcal{E} \mid \mathsf{unfold}\ \mathcal{E} \mid \mathsf{ref}\ \mathcal{E} \\
 & \quad \mid \mathcal{E} := e_d \mid v := \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E}\ \tau \mid \mathcal{E}\ T \mid \langle\ T,\ \mathcal{E}\ \rangle_{(V:K)\times\tau} \\
 & \quad \mid \mathsf{let}\ \langle\ V,\ x\ \rangle = \mathcal{E}\ \mathsf{in}\ e_d
\end{array}
$$

Figure 6.14: VeriML computational language: Staging extension (syntax)

$\boxed{\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau}$

$$
\frac{\bullet;\ \mathcal{M};\ \Gamma|_{\text{static}} \vdash e : \tau \qquad \Psi;\ \mathcal{M};\ \Gamma, x :_s \tau \vdash e' : \tau}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' : \tau}\ \textsc{CExp-LetStatic}
$$

$$
\frac{x :_s \tau \in \Gamma}{\Psi;\ \mathcal{M};\ \Gamma \vdash x : \tau}\ \textsc{CExp-StaticVar}
$$

$\boxed{\Gamma|_{\text{static}} = \Gamma'}$ *(Limiting a context to static variables)*

$$
\begin{array}{lcl}
\bullet|_{\text{static}} & = & \bullet \\
(\Gamma,\ x :_s t)|_{\text{static}} & = & \Gamma|_{\text{static}},\ x : t \\
(\Gamma,\ x : t)|_{\text{static}} & = & \Gamma|_{\text{static}} \\
(\Gamma,\ \alpha : k)|_{\text{static}} & = & \Gamma|_{\text{static}}
\end{array}
$$

Figure 6.15: VeriML computational language: Staging extension (typing)

$\boxed{(\ \mu\ ,\ e\ ) \longrightarrow_s (\ \mu\ ,\ e'\ )}$

$$\frac{(\ \mu\ ,\ e_d\ ) \longrightarrow (\ \mu'\ ,\ e'_d\ )}{(\ \mu\ ,\ \mathcal{S}[e_d]\ ) \longrightarrow_s (\ \mu'\ ,\ \mathcal{S}[e'_d]\ )}\ \textsc{Op-StaticEnv}$$

$$(\ \mu\ ,\ \mathcal{S}[\mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e]\ ) \longrightarrow_s (\ \mu\ ,\ \mathcal{S}[e[v/x]]\ )\ \textsc{Op-LetStaticEnv}$$

$$(\ \mu\ ,\ \mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e\ ) \longrightarrow_s (\ \mu\ ,\ e[v/x]\ )\ \textsc{Op-LetStaticTop}$$

Figure 6.16: VeriML computational language: Staging extension (operational semantics)

corresponding to extraction and CExp-LetStatic corresponding to iterated extension over all static variables in the context.

The operational semantics is adapted as follows: we define a new small-step reduction relation $\longrightarrow_s$ between machine states, capturing the static evaluation stage where expressions inside letstatic are evaluated. If this stage terminates, it yields a residual expression e_d which is then evaluated using the normal reduction relation $\longrightarrow$. With respect to the first stage, residual expressions can thus be viewed as *values*. Furthermore, we define static binding evaluation contexts $\mathcal{S}$: these are the equivalent of evaluation contexts $\mathcal{E}$ for static evaluation. The key difference is that static binding evaluation contexts can also go under non-static binders. The reason is that such binders do not influence static evaluation as we cannot refer to their variables due to typing.

A subtle point of the semantics is that the static evaluation phase and the normal evaluation phase share the same state. The store μ' yielded by static evaluation needs to be retained in order to evaluate the residual expression: $(\ \mu\ ,\ e\ ) \longrightarrow_s^* (\ \mu'\ ,\ e_d\ ) \longrightarrow^* (\ \mu''\ ,\ v\ )$. This can be supported in an interpreter-based evaluation strategy, but poses significant problems in a compilation-based strategy, as the compiler would need to yield an initial store together with the executable program. In practice, we get around this problem by allowing expressions of only a small set of types to be statically evaluated (e.g. allowing λHOL tuples but disallowing functions or mutable references); repeating top-level definitions during static and normal evaluation (so that an equivalent global state is established in both stages); and last we assume that the statically evaluated expressions preserve the global state. We will comment further on this issue when we cover the VeriML implementation in more detail,

specifically in Subsection 8.3.3. As these constraints are not necessary for type-safety, we do not capture them in our type system; we leave their metatheoretic study to future work.

Metatheory

We will now establish type-safety for the staging extension by proving progress and preservation for evaluation of well-typed expressions e through the static small-step semantics. Type-safety for the normal small step semantics $(\mu , e_d) \longrightarrow (\mu' , e'_d)$ is entirely as before, as residual expressions are identical to the expressions of the previous section and the relation $\longrightarrow$ has not been modified.

Lemma 6.3.1 (Extension of Lemma 6.1.4) *(Computational substitution)*

$$1.\ \frac{\Psi,\ \Psi';\ \Gamma,\ \alpha' : k',\ \Gamma' \vdash \tau : k \qquad \Psi;\ \Gamma \vdash \tau' : k'}{\Psi,\ \Psi';\ \Gamma,\ \Gamma'[\tau'/\alpha'] \vdash \tau[\tau'/\alpha'] : k}$$

$$2.\ \frac{\Psi,\ \Psi';\ \mathcal{M};\ \Gamma,\ \alpha' : k',\ \Gamma' \vdash e : \tau \qquad \Psi;\ \Gamma \vdash \tau' : k'}{\Psi,\ \Psi';\ \mathcal{M};\ \Gamma,\ \Gamma'[\tau'/\alpha'] \vdash e[\tau'/\alpha'] : \tau[\tau'/\alpha']}$$

$$3.\ \frac{\Psi,\ \Psi';\ \mathcal{M};\ \Gamma,\ x' : \tau',\ \Gamma' \vdash e_d : \tau \qquad \Psi;\ \mathcal{M};\ \Gamma \vdash e'_d : \tau'}{\Psi,\ \Psi';\ \mathcal{M};\ \Gamma,\ \Gamma' \vdash e_d[e'_d/x'] : \tau}$$

$$4.\ \frac{\Psi;\ \Gamma,\ x :_s \tau,\ \Gamma' \vdash e : \tau' \qquad \bullet;\ \mathcal{M};\ \bullet \vdash v : \tau}{\Psi;\ \mathcal{M};\ \Gamma,\ \Gamma' \vdash e[v/x] : \tau'}$$

$$5.\ \frac{\Psi;\ \Gamma,\ x : \tau,\ \Gamma' \vdash e : \tau' \qquad \bullet;\ \mathcal{M};\ \bullet \vdash v : \tau}{\Psi;\ \mathcal{M};\ \Gamma,\ \Gamma' \vdash e[v/x] : \tau'}$$

$$6.\ \frac{\Psi,\ \Psi' \vdash \Gamma,\ \alpha' : k',\ \Gamma'\ wf \qquad \Psi;\ \Gamma \vdash \tau' : k'}{\Psi,\ \Psi' \vdash \Gamma,\ \Gamma'[\tau'/\alpha']\ wf}$$

Proof. By induction on the typing derivation for τ, e or e_d. We prove the cases for the two new typing rules. cases follow.

Part 3. Case CExp-StaticVar**.**

$$\left(\frac{x :_s \tau \in \Gamma}{\Psi;\ \mathcal{M};\ \Gamma \vdash x : \tau}\right)$$

We have that $[e'_d/x] = e'_d$, and $\Psi;\ \mathcal{M};\ \Gamma \vdash e'_d : \tau$, which is the desired.

Case CExp-LetStatic**.**

$$\left(\frac{\bullet;\ \mathcal{M};\ \Gamma|_{\text{static}} \vdash e : \tau \qquad \Psi;\ \mathcal{M};\ \Gamma,\ x :_s \tau \vdash e' : \tau'}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' : \tau'}\right)$$

Impossible case by syntactic inversion on e_d.

Part 4. Case CExp-LetStatic**.**

$$\left(\frac{\bullet;\ \mathcal{M};\ \Gamma|_{\text{static}},\ x : \tau,\ \Gamma'|_{\text{static}} \vdash e : \tau \qquad \Psi;\ \mathcal{M};\ \Gamma,\ x :_s \tau,\ \Gamma',\ x' :_s \tau'' \vdash e' : \tau'}{\Psi;\ \mathcal{M};\ \Gamma,\ x :_s \tau,\ \Gamma' \vdash \mathsf{letstatic}\ x'\ =\ e\ \mathsf{in}\ e' : \tau'}\right)$$

We use part 5 for e to get that $\bullet;\ \mathcal{M};\ \Gamma|_{\text{static}},\ \Gamma'|_{\text{static}} \vdash e[v/x] : \tau$

By induction hypothesis for e' we get $\Psi;\ \mathcal{M};\ \Gamma,\ \Gamma', x' :_s \tau'' \vdash e'[v/x] : \tau'$

Thus using the same typing rule we get the desired result. □

Lemma 6.3.2 *(Types of decompositions)*

1. $$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{S}[e] : \tau \qquad \Gamma|_{static} = \bullet}{\exists \tau'.(\quad \bullet;\ \mathcal{M};\ \bullet \vdash e : \tau' \qquad \forall e'.\bullet;\ \mathcal{M};\ \bullet \vdash e' : \tau' \Rightarrow \Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{S}[e'] : \tau \quad)}$$

2. $$\frac{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{E}_s[e] : \tau}{\exists \tau'.(\quad \Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau' \qquad \forall e.\Psi;\ \mathcal{M};\ \Gamma \vdash e' : \tau' \Rightarrow \Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{E}_s[e'] : \tau \quad)}$$

Proof.

Part 1. By structural induction on $\mathcal{S}$.

Case $\mathcal{S} = \mathsf{letstatic}\ \mathbf{x}\ =\ \bullet\ \mathsf{in}\ \mathbf{e'}$**.**

By inversion of typing we get that $\bullet;\ \mathcal{M};\ \Gamma|_{\text{static}} \vdash e : \tau$. We have $\Gamma|_{\text{static}} = \bullet$, thus we get

$\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau'$. Using the same typing rule we get the desired result for $\mathcal{S}[e']$.

Case $\mathcal{S} = \mathsf{letstatic\ x\ =\ \mathcal{S}'\ in\ e''}$**.** Directly by inductive hypothesis for $\mathcal{S}'$.

Case $\mathcal{S} = \mathcal{E}_\mathbf{s}[\mathcal{S}]$**.** We have that $\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{E}_s[\mathcal{S}[e_d]] : \tau$. Using part 2 for $\mathcal{E}_s$ and $\mathcal{S}[e_d]$ we get that $\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{S}[e_d] : \tau'$ for some τ' and also that for all e' such that $\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau'$ we have $\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{E}_s[e'] : \tau$. Then using induction hypothesis we get a τ'' such that $\bullet;\ \mathcal{M};\ \bullet \vdash e_d : \tau''$. For this type, we also have that $\bullet;\ \mathcal{M};\ \bullet \vdash e'_d : \tau''$ implies $\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{S}[e'_d] : \tau'$, which further implies $\Psi;\ \mathcal{M};\ \Gamma \vdash \mathcal{E}_s[\mathcal{S}[e'_d]] : \tau$.

Part 2. By induction on the structure of $\mathcal{E}_s$. In each case, we use inversion of typing to get the type for e and then use the same typing rule to get the derivation for $\mathcal{E}_s[e']$. □

Theorem 6.3.3 (Extension of Theorem 6.1.5) *(Preservation)*

$$1.\ \frac{\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau \qquad (\ \mu\ ,\ e\ ) \longrightarrow_s (\ \mu'\ ,\ e'\ ) \qquad \mu \sim \mathcal{M}}{\exists \mathcal{M}'.(\quad \bullet;\ \mathcal{M}';\ \bullet \vdash e' : \tau \qquad \mathcal{M} \subseteq \mathcal{M}' \qquad \mu' \sim \mathcal{M}' \quad)}$$

$$2.\ \frac{\bullet;\ \mathcal{M};\ \bullet \vdash e_d : \tau \qquad (\ \mu\ ,\ e_d\ ) \longrightarrow (\ \mu'\ ,\ e'_d\ ) \qquad \mu \sim \mathcal{M}}{\exists \mathcal{M}'.(\quad \bullet;\ \mathcal{M}';\ \bullet \vdash e'_d : \tau \qquad \mathcal{M} \subseteq \mathcal{M}' \qquad \mu' \sim \mathcal{M}' \quad)}$$

Proof.

Part 1. We proceed by induction on the derivation of $(\ \mu\ ,\ e\ ) \longrightarrow_s (\ \mu'\ ,\ e'\ )$.

Case Op-StaticEnv**.**

$$\left[\frac{(\ \mu\ ,\ e_d\ ) \longrightarrow (\ \mu'\ ,\ e'_d\ )}{(\ \mu\ ,\ \mathcal{S}[e_d]\ ) \longrightarrow_s (\ \mu'\ ,\ \mathcal{S}[e'_d]\ )}\right]$$

Using Lemma 6.3.2 we get $\bullet;\ \mathcal{M};\ \bullet \vdash e_d : \tau'$. Using part 2, we get that $\bullet;\ \mathcal{M};\ \bullet \vdash e'_d : \tau'$. Using the same lemma again we get the desired.

Case Op-LetStaticEnv**.**

$$\left[(\ \mu\ ,\ \mathcal{S}[\mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e]\ ) \longrightarrow_s (\ \mu\ ,\ \mathcal{S}[e[v/x]]\ ) \right]$$

Using Lemma 6.3.2 we get $\bullet;\ \mathcal{M};\ \bullet \vdash \mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e : \tau'$. By typing inversion we get that $\bullet;\ \mathcal{M};\ \bullet \vdash v : \tau''$ and also that $\bullet;\ \mathcal{M};\ x :_s \tau'' \vdash e : \tau'$. Using the substitution lemma (Lemma 6.3.1) we get the desired result.

Case Op-LetStaticTop**.**

$$\left[(\ \mu\ ,\ \mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e\ ) \longrightarrow_s (\ \mu\ ,\ e[v/x]\ ) \right]$$

Similar to the above.

Part 2. Exactly as before by induction on the typing derivation for e_d, as e_d entirely matches the definition of expressions prior to the extension. □

Lemma 6.3.4 *(Unique decomposition)*

1. *Every expression e is either a residual expression e_d or there either exists a unique decomposition $e = \mathcal{S}[e_d]$.*

2. *Every residual expression e_d can be uniquely decomposed as $e_d = \mathcal{E}[v]$.*

Proof.

Part 1. By induction on the structure of the expression e. We give some representative cases.

Case $\mathbf{e} = \lambda\mathbf{V} : \mathbf{K}.\mathbf{e}'$. Splitting cases on whether $e' = e'_d$ or not. If $e' = e'_d$ then trivially $e = e_d$. If $e' \neq e'_d$ then by induction hypothesis we get a unique decomposition of e' into $\mathcal{S}'[e'']$. The original expression e can be uniquely decomposed as $\mathcal{S}[e'']$ with $\mathcal{S} = \lambda V : K.\mathcal{S}'$. This decomposition is unique because the outer frame is uniquely determined; the uniqueness of the inner frames and the expression filling the hole are already established by induction hypothesis.

Case $\mathbf{e} = \mathbf{e}'\ \mathbf{T}$. By induction hypothesis we get that either $e' = e'_d$, or there is a unique

decomposition of e' into $\mathcal{S}'[e'']$. The former case is trivial. In the latter case, e is uniquely decomposed using $\mathcal{S} = \mathcal{E}_s[\mathcal{S}']$ with $\mathcal{E}_s = \bullet\ T$, into $e = \mathcal{S}'[e'']\ T$.

Case $\mathbf{e} = (\mathsf{let}\ \langle\ \mathbf{e}',\ \mathbf{V}\ \rangle = \mathbf{x}\ \mathsf{in}\ \mathbf{e}'')$. Using induction hypothesis on e'; if it is a residual expression, then using induction hypothesis on e''; if that too is a residual expression, then the original expression e is too. Otherwise, use the unique decomposition of $e'' = \mathcal{S}'[e''']$ to get the unique decomposition $e = \mathsf{let}\ \langle\ e_d,\ x\ \rangle = \mathcal{S}'[e''']\ \mathsf{in}\ .$ If e' is not a residual expression, use the unique decomposition of $e' = \mathcal{S}''[e'''']$ to get the unique decomposition $e = \mathsf{let}\ \langle\ \mathcal{S}''[e'''']\ ,\ x\ \rangle = e''\ \mathsf{in}\ .$

Case $\mathbf{e} = \mathsf{letstatic}\ \mathbf{x}\ =\ \mathbf{e}'\ \mathsf{in}\ \mathbf{e}''$. Using the induction hypothesis on e'.
In the case that $e' = e_d$, then trivially we have the unique decomposition
$e = (\mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e'')[e_d]$.
In the case where $e' = \mathcal{S}[e_d]$, we have the unique decomposition
$e = (\mathsf{letstatic}\ x\ =\ \mathcal{S}\ \mathsf{in}\ e'')[e_d]$.

Part 2. Similarly, by induction on the structure of the dynamic expression e_d. □

Theorem 6.3.5 (Extension of Theorem 6.1.7) *(Progress)*

$$1.\ \frac{\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau \qquad \mu \sim \mathcal{M} \qquad e \neq e_d}{\exists \mu', e'.\ (\ \mu\ ,\ e\ ) \longrightarrow_s (\ \mu'\ ,\ e'\ )} \qquad 2.\ \frac{\bullet;\ \mathcal{M};\ \bullet \vdash e_d : \tau \qquad \mu \sim \mathcal{M} \qquad e_d \neq v}{\exists \mu', e'_d.\ (\ \mu\ ,\ e_d\ ) \longrightarrow (\ \mu'\ ,\ e'_d\ )}$$

Proof.

Part 1 First, we use the unique decomposition lemma (Lemma 6.3.4) for e. We get that either e is a dynamic expression e_d, in which case we are done; or a decomposition into $\mathcal{S}[e'_d]$. In that case, we use Lemma 6.3.2 and part 2 to get that either e'_d is a value or that some progress can be made. In the latter case we have $(\ \mu\ ,\ e'_d\ ) \longrightarrow (\ \mu'\ ,\ e''_d\ )$ for some μ', e''_d. Using Op-StaticEnv we get $(\ \mu\ ,\ \mathcal{S}[e'_d]\ ) \longrightarrow_s (\ \mu'\ ,\ \mathcal{S}[e''_d]\ )$, thus we have the desired for $e' = \mathcal{S}[e''_d]$. In the former case, where $e = \mathcal{S}[v]$, we split cases based on whether $\mathcal{S} = (\mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e)$ or not; in the former case we use Op-LetStaticTop and in the latter the rule Op-LetStaticEnv to make progress.

Part 2 Identical as before. □

Theorem 6.3.6 (Extension of Theorem 6.1.8) *(Type safety for VeriML)*

$$1.\ \frac{\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau \qquad \mu \sim \mathcal{M} \qquad e \neq e_d}{\exists \mu', \mathcal{M}', e'.(\quad (\ \mu\ ,\ e\ ) \longrightarrow_s (\ \mu'\ ,\ e'\ ) \qquad \bullet;\ \mathcal{M}';\ \bullet \vdash e' : \tau \qquad \mu' \sim \mathcal{M}' \quad)}$$

$$2.\ \frac{\bullet;\ \mathcal{M};\ \bullet \vdash e_d : \tau \qquad \mu \sim \mathcal{M} \qquad e_d \neq v}{\exists \mu', \mathcal{M}', e'_d.(\quad (\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'_d\ ) \qquad \bullet;\ \mathcal{M}';\ \bullet \vdash e'_d : \tau \qquad \mu' \sim \mathcal{M}' \quad)}$$

Proof. Directly by combining Theorem 6.3.3 and Theorem 6.3.5. □

6.4 Proof erasure

Consider the following program:

$$\mathsf{tautology}\ [P : Prop, Q : Prop]\ ((P \wedge Q) \rightarrow (Q \wedge P)) : \big([P, Q]\ (P \wedge Q) \rightarrow (Q \wedge P)\big)$$

assuming the following type for the tautology tactic:

$$\mathsf{tautology} : (\phi : ctx) \rightarrow (P : [\phi]\ Prop) \rightarrow \big([\phi]\ P\big)$$

where it is understood that the function throws an exception in cases where it fails to prove the given proposition. This program can be seen as a proof script that proves the proposition $(P \wedge Q) \rightarrow (Q \wedge P)$. If it evaluates successfully it will yield a proof object for that proposition. The proof checker for λHOL is included in the type system of VeriML and therefore we know that the yielded proof object is guaranteed to be valid – this is a direct consequence of type safety for VeriML. There are many situations where we are interested in the exact structure of the proof object. For example, we might want to ship the proof object to a third party for independent verification. In most situations though, we are interested merely in the existence of the proof object. In this section we will show that in this case, we can evalute the original VeriML expression without producing proof objects at any intermediate step but still maintain the same guarantees about the existence of proof objects. We will do this by showing that an alternative semantics for VeriML using *proof erasure* simulate the normal semantics – that is, if we erase all proof objects from an expression prior to execution, the expression evaluates through exactly the same steps.

Evaluating expressions under proof erasure results in significant space savings, as proof objects can become very large. Still, there is a trade-off between the space savings and the amount of code that we need to trust. If we evaluate expressions under the normal semantics and re-validate the yielded proof objects using the base proof checker for λHOL, the proof checker is the only part of the system that we need to trust, resulting in a very small *trusted base*. On the other hand, if we evaluate all expressions under proof erasure, we need to include the implementation of the whole VeriML language in our trusted base, including its type checker and compiler. In our implementation, we address this trade-off by allowing users to selectively evaluate expressions under proof erasure. In this case, the trusted core is extended only with the proof-erased version of the "trusted" expressions, which can be manually inspected.

More formally, we will define a type-directed translation from expressions to expressions with all proofs erased $[\![\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau]\!]_E = e'$. We will then show a strict bisimulation between the normal semantics and the semantics where the erasure function has first been applied – that is, that every evaluation step under the normal semantics $(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )$ is simulated by a step like $(\ [\![\mu]\!]_E\ ,\ [\![e]\!]_E\ ) \longrightarrow (\ [\![\mu']\!]_E\ ,\ [\![e']\!]_E\ )$, and also that the inverse holds. The essential reason why this bisimulation exists is that the pattern matching construct we have presented does not allow patterns for individual constructors of proof objects because of the sort restrictions in the pattern typing rules (Figures 5.1 and 5.2). Since pattern matching is the only computational construct available for looking into the structure of proof objects, this restriction is enough so that proof objects will not influence the runtime behavior of VeriML expressions.

We present the details of the proof erasure function in Figure 6.17. It works by replacing all proof objects in an expression e by the special proof object *admit*, directly using the erasure rule for extension terms EraseProof. We only show the main rules; the rest of the cases only use the erasure procedure recursively. In the same figure, we also show how the same function is lifted to extension substitutions σ_Ψ and stores μ when they are compatible with the store typing context $\mathcal{M}$. We call the term *admit* the prove-anything constant and it is understood as a logic constant that cannot be used by the user but proves any proposition. It is added to the syntax and typing of λHOL in Figure 6.18.

$\boxed{[\![\Psi;\ \Phi \vdash t : t']\!]_E = t^E}$

$$\frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi;\ \Phi \vdash t' : \mathit{Prop}}{[\![\Psi;\ \Phi \vdash t : t']\!]_E = \mathit{admit}}\ \textsc{EraseProof}$$

$\boxed{[\![\Psi \vdash T : K]\!]_E = T^E}$

$$\frac{[\![\Psi;\ \Phi \vdash t : t']\!]_E = t^E}{[\![\Psi \vdash [\Phi]\, t : [\Phi]\, t']\!]_E = [\Phi]\, t^E} \qquad \frac{}{[\![\Psi \vdash [\Phi]\, \Phi' : [\Phi]\, \mathit{ctx}]\!]_E = [\Phi]\, \Phi'}$$

$\boxed{[\![\Psi' \vdash \sigma_\Psi : \Psi]\!]_E = \sigma_\Psi^E}$

$$\frac{}{[\![\Psi' \vdash \bullet : \bullet]\!]_E = \bullet} \qquad \frac{[\![\Psi' \vdash \sigma_\Psi : \Psi]\!]_E = \sigma_\Psi^E \qquad [\![\Psi' \vdash T : K \cdot \sigma_\Psi]\!]_E = T^E}{[\![\Psi' \vdash (\sigma_\Psi,\ V : T) : (\Psi,\ V : K)]\!]_E = (\sigma_\Psi^E,\ T^E)}$$

$\boxed{[\![\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau]\!]_E = e^E}$

$$\frac{[\![\Psi,\ V : K;\ \mathcal{M};\ \Gamma \vdash e : \tau]\!]_E = e^E}{[\![\Psi;\ \mathcal{M};\ \Gamma \vdash (\lambda V : K.e) : ((V : K) \to \tau)]\!]_E = \lambda V : K.e^E}$$

$$\frac{[\![\Psi;\ \mathcal{M};\ \Gamma \vdash e : (V : K) \to \tau]\!]_E = e^E \qquad [\![\Psi \vdash T : K]\!]_E = T^E}{[\![\Psi;\ \mathcal{M};\ \Gamma \vdash e\ T : \tau \cdot (id_\Psi,\ T/V)]\!]_E = e^E\ T^E}$$

$$\frac{[\![\Psi \vdash T : K]\!]_E = T^E \qquad [\![\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau \cdot (id_\Psi,\ T/V)]\!]_E = e^E}{\left[\!\!\left[\Psi;\ \mathcal{M};\ \Gamma \vdash \langle\ T,\ e\ \rangle_{(V:K)\times\tau} : ((V : K) \times \tau)\right]\!\!\right]_E = \left\langle\ T^E,\ e^E\ \right\rangle_{(V:K)\times\tau}}$$

$$\frac{[\![\Psi;\ \mathcal{M};\ \Gamma \vdash e : (V : K) \times \tau]\!]_E = e^E \qquad [\![\Psi,\ V : K;\ \mathcal{M};\ \Gamma,\ x : \tau \vdash e' : \tau']\!]_E = e'^E}{[\![\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{let}\ \langle\ V,\ x\ \rangle = e\ \mathsf{in}\ e' : \tau']\!]_E = (\mathsf{let}\ \langle\ V,\ x\ \rangle = e^E\ \mathsf{in}\ e'^E)}$$

$$\frac{[\![\Psi \vdash T : K]\!]_E = T^E \qquad [\![\Psi,\ \Psi_u;\ \mathcal{M};\ \Gamma \vdash e' : \tau \cdot (id_\Psi,\ T_P/V)]\!]_E = e'^E}{\left[\!\!\left[\Psi;\ \mathcal{M};\ \Gamma \vdash \begin{array}{l}\mathsf{holmatch}\ T\ \mathsf{return}\ V : K.\tau \\ \mathsf{with}\ \ \Psi_u.\ T_P \mapsto e'\end{array} : \tau \cdot (id_\Psi,\ T/V) + \mathsf{unit}\right]\!\!\right]_E = \\ = \mathsf{holmatch}\ T^E\ \mathsf{return}\ V : K.\tau\ \mathsf{with}\ \Psi_u.T_P \mapsto e'^E}$$

$\boxed{[\![\mu]\!]_E = \mu'}$ when $\mu \sim \mathcal{M}$

$$\frac{}{[\![\bullet]\!]_E = \bullet} \qquad \frac{[\![\bullet;\ \mathcal{M};\ \bullet \vdash v : \tau]\!]_E = v^E}{[\![\mu,\ (l \mapsto v)]\!]_E = \mu,\ (l \mapsto v^E)}$$

Figure 6.17: Proof erasure for VeriML

Syntax

$$t ::= \cdots \mid \mathit{admit}$$

$\boxed{\Psi;\ \Phi \vdash t : t'}$

$$\frac{\Psi;\ \Phi \vdash t' : \mathit{Prop}}{\Psi;\ \Phi \vdash \mathit{admit} : t'}\ \textsc{ProveAnything}$$

Figure 6.18: Proof erasure: Adding the prove-anything constant to λHOL

We first prove the following auxiliary lemmas which capture the essential reason behind the bisimulation proof that follows.

Lemma 6.4.1 *(Proof object patterns do not exist)*

$$\frac{\Psi \stackrel{*}{\vdash_p} \Psi_u > T_P : [\Phi]\, t' \qquad \Psi,\ \Psi_u;\ \Phi \vdash t' : s}{s \neq \mathit{Prop}}$$

Proof. By structural induction on the pattern typing for T_P. □

Lemma 6.4.2 *(Pattern matching is preserved under proof erasure)*

$$\frac{\bullet \stackrel{*}{\vdash_p} \Psi_u > T_P : K \qquad \bullet \vdash T : K \qquad [\![\bullet \vdash T : K]\!]_E = T^E}{(T_P \sim T) = (T_P \sim T^E)}$$

Proof. We view the consequence as an equivalence: $T_P \sim T = \sigma_\Psi \Leftrightarrow T_P \sim T^E = \sigma_\Psi$. We prove each direction by induction on the derivation of the pattern matching result and using the previous lemma. □

Lemma 6.4.3 *(Erasure commutes with extension substitution application)*

1. $$\frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi' \vdash \sigma_\Psi : \Psi \qquad [\![\Psi;\ \Phi \vdash t : t']\!]_E = t^E \qquad [\![\Psi' \vdash \sigma_\Psi : \Psi]\!]_E = \sigma_\Psi^E}{t^E \cdot \sigma_\Psi^E = [\![\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot \sigma_\Psi : t' \cdot \sigma_\Psi]\!]_E}$$

$$2.\ \frac{\Psi \vdash T : K \qquad \Psi' \vdash \sigma_\Psi : \Psi \qquad [\![\Psi \vdash T : K]\!]_E = T^E \qquad [\![\Psi' \vdash \sigma_\Psi : \Psi]\!]_E = \sigma_\Psi^E}{T^E \cdot \sigma_\Psi^E = [\![\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi]\!]_E}$$

$$3.\ \frac{\begin{array}{c}\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau \\ \Psi' \vdash \sigma_\Psi : \Psi \qquad [\![\Psi;\ \mathcal{M};\ \Gamma \vdash e : \tau]\!]_E = e^E \qquad [\![\Psi' \vdash \sigma_\Psi : \Psi]\!]_E = \sigma_\Psi^E\end{array}}{e^E \cdot \sigma_\Psi^E = [\![\Psi';\ \mathcal{M};\ \Gamma \cdot \sigma_\Psi \vdash e \cdot \sigma_\Psi : \tau \cdot \sigma_\Psi]\!]_E}$$

Proof. By induction on the structure of T. □

We are now ready to prove the bisimulation theorem.

Theorem 6.4.4 *(Proof erasure semantics for VeriML are bisimilar to normal semantics)*

$$1.\ \frac{\begin{array}{c}\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau \qquad \mu \sim \mathcal{M} \qquad (\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ ) \qquad [\![\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau]\!]_E = e^E \\ [\![\bullet;\ \mathcal{M};\ \bullet \vdash e' : \tau]\!]_E = e'^E \qquad [\![\mu]\!]_E = \mu^E \qquad [\![\mu']\!]_E = \mu'^E\end{array}}{(\ \mu^E\ ,\ e^E\ ) \longrightarrow (\ \mu'^E\ ,\ e'^E\ )}$$

$$2.\ \frac{\begin{array}{c}\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau \\ \mu \sim \mathcal{M} \qquad [\![\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau]\!]_E = e^E \qquad [\![\mu]\!]_E = \mu^E \qquad (\ \mu^E\ ,\ e^E\ ) \longrightarrow (\ \mu'^E\ ,\ e'^E\ )\end{array}}{\exists e', \mu', \mathcal{M}'. \left(\begin{array}{c}\bullet;\ \mathcal{M}';\ \bullet \vdash e' : \tau \quad \mu' \sim \mathcal{M}' \quad \mathcal{M} \subseteq \mathcal{M}' \\ [\![\bullet;\ \mathcal{M}';\ \bullet \vdash e' : \tau]\!]_E = e'^E \quad [\![\mu']\!]_E = \mu'^E \quad (\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )\end{array}\right)}$$

Proof. **Part 1.** By induction on the step relation $(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )$. In all cases the result follows trivially using Lemmas 6.4.2 and 6.4.3.

Part 2. By induction on the step relation $(\ \mu^E\ ,\ e^E\ ) \longrightarrow (\ \mu'^E\ ,\ e'^E\ )$. We prove three representative cases.

Case Op-ΠHol-Beta**.**

$$\left[(\ \mu^E\ ,\ (\lambda V : K.e'^E)\ T^E\ ) \longrightarrow (\ \mu^E\ ,\ e'^E \cdot (T^E/V)\ )\right]$$

By inversion of $[\![\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau]\!]_E = (\lambda V : K.e'^E)\ T^E$ we get that $e = (\lambda V : K.e'')\ T$ with $[\![V : K;\ \mathcal{M};\ \bullet \vdash e'' : (V : K) \to \tau']\!]_E = e'^E$ and $[\![\bullet \vdash T : K]\!]_E = T^E$. Choosing $e' =$

$e'' \cdot (T/V)$, $\mu' = \mu^E$ and $\mathcal{M}' = \mathcal{M}$ we have the desired, using Lemma 6.4.3.

Case Op-HolMatch**.**

$$\left[\frac{T_P \sim T^E = \sigma^E_\Psi}{\left(\ \mu^E\ ,\ \mathsf{holmatch}\ T^E\ \mathsf{with}\ \Psi_u.T_P \mapsto e'^E\ \right) \longrightarrow \left(\ \mu^E\ ,\ \mathsf{inj}_1\ (e'^E \cdot \sigma^E_\Psi)\ \right)}\right]$$

By inversion of $[\![\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau]\!]_E = e^E$ for $e^E = \mathsf{holmatch}\ T^E\ \mathsf{with}\ \Psi_u.T_P \mapsto e'^E$ we get T and e'' such that $e = (\mathsf{holmatch}\ T\ \mathsf{with}\ \Psi_u.T_P \mapsto e'')$ with $[\![\bullet \vdash T : K]\!]_E = T^E$ and $[\![\Psi_u;\ \mathcal{M};\ \bullet \vdash e'' : \tau']\!]_E = e'^E$. By Lemma 6.4.2 we get that $T_P \sim T = \sigma^E_\Psi$ with $\sigma_\Psi = \sigma^E_\Psi$. Thus setting $e' = \mathsf{inj}_1(e'' \cdot \sigma^E_\Psi)$ and using Lemma 6.4.3 we get the desired.

Case Op-NewRef**.**

$$\left[\frac{\neg(l \mapsto \_ \in \mu^E)}{\left(\ \mu^E\ ,\ \mathsf{ref}\ v^E\ \right) \longrightarrow \left(\ (\mu^E,\ l \mapsto v^E)\ ,\ l\ \right)}\right]$$

By inversion of $[\![\bullet;\ \mathcal{M};\ \bullet \vdash e : \tau]\!]_E = \mathsf{ref}\ v^E$ we get a value v such that $[\![\bullet;\ \mathcal{M};\ \bullet \vdash v : \tau']\!]_E = v^E$. Therefore setting $\mu' = (\mu,\ l \mapsto v)$ we get the desired. □

Chapter 7

User-extensible static checking

The type system of the VeriML computational language includes the λHOL type system. This directly means that proof objects within VeriML expressions and tactics that are automatically checked for logical validity. As we argued in Chapter 2, this is one of the main departure points of VeriML compared to traditional proof assistants. Yet we often need to check things beyond logical validity which the VeriML type system does not directly allow.

A common example is checking whether two propositions or terms are equivalent. This check requires proof search: we first need to find a proof object for the equivalence and then check it for validity using the VeriML type system. Yet such equivalence checks are very common when writing λHOL proofs; we would like to check them statically, as if they were part of type checking. In this way, we will know at the definition time of proof scripts and tactics that the equivalence checks contained in proofs are indeed valid. Checks involving proof search are undecidable in the general case, thus fixing a search algorithm beforehand inside the type system will not be able to handle all equivalence checks that users need in an efficient way. Users should instead be able to provide their own search algorithms. We say that *static checking* – checks that happen at the definition time of a program, just as typing – should be *user-extensible*. In this chapter we will show that user-extensible static checking is possible for proofs and tactics written in VeriML; we will also briefly present how the same technique can be viewed as a general extensible type checking mechanism, in the context of dependently-typed languages.

In order to make our description of the issue we will address more precise, let us consider an example. Consider the following logical function:

$$twice \ : Nat \rightarrow Nat \ \equiv \ \lambda x : Nat.x + x$$

Given the following theorem for even numbers:

$$evenMult2 \ : \ \forall x : Nat.even \ (2x)$$

we want to prove the following proposition:

$$evenTwice \ : \ \forall x : Nat.even \ (twice \ x)$$

In normal mathematical practice, we would be able to say that *evenTwice* follows directly from *evenMult2*. Though this is true, a formal proof object would need to include more details, such as the fact that $x + x = 2x$, the β-reduction that reduces *twice* x to $x + x$ and the fact that if $even(2x)$ is true, so is $even(x + x)$ since the arguments to *even* are equal (called *congruence*). The formal proof is thus roughly as follows[1]:

$even \ (2x)$	(theorem)
$twice \ x = x + x$	(β-conversion)
$x + x = 2x$	(arithmetic conversion)
$even \ (twice \ x) = even \ (2x)$	(congruence)
$even \ (twice \ x)$	(follows from the above)

We can extend the base λHOL type system, so that some of these equivalences are decided automatically; thus the associated step of the proof can be omitted. This is done by including a *conversion rule* in λHOL, as suggested in Section 3.2, which can be viewed as a fixed decision procedure that directly handles a class of equivalences. For example, including a conversion rule that automatically checks β-equivalence would allow us to omit the second step of the above proof. As more checks are included in the λHOL logic directly, smaller proofs are possible, as shown in Figure 7.1. Yet there is a tradeoff between the sophistication

1. We are being a bit imprecise here for presentation purposes. All three equalities would actually hold trivially through the conversion rule if definitional equality includes β- and $\iota-$reduction. Yet other cases of arithmetic simplification such as $x + x = x \cdot 2$ would not hold directly. Supporting the more general case of simplification based on arithmetic properties is what we refer to in this chapter when referring to an arithmetic conversion rule. Similarly, conversion with congruence closure reasons about arbitrary equality rewriting steps taking the available hypotheses into account. Thus it is more general than the congruence closure included in definitional equality, where simply definitionally equal subterms lead to definitionally equal terms.

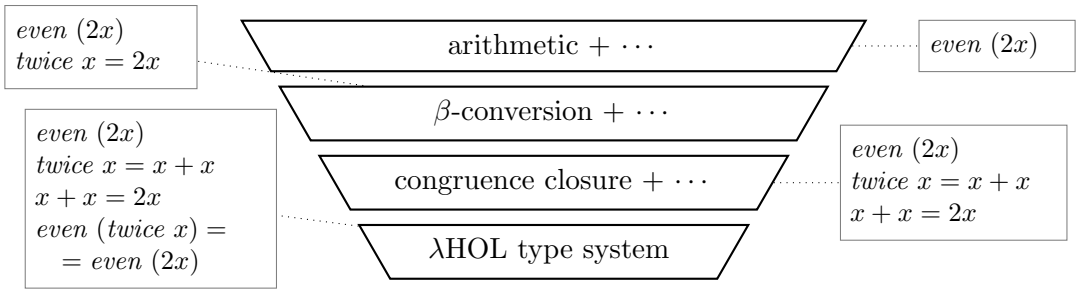

Figure 7.1: Layers of static checking and steps in the corresponding formal proof of *even* (*twice* x)

of the conversion rule included in the logic and the amount of trust we need to place in the logic metatheory and implementation. Put otherwise, there is a tradeoff between the size of the resulting proofs and the size of the trusted core of the logic. Furthermore, even if we include checking for all steps – β-conversion, arithmetic conversion and congruence – there will always be steps that are common and trivial yet the fixed conversion rule cannot handle. The reason is that by making any fixed conversion rule part of the logic, the metatheory of the logic and the trust we can place in the logic checker are dependent on the details of the conversion rule, thus disallowing user extensions. For example, if we later prove the following theorem:

$$\forall x, \mathit{even}\ x \to \mathit{odd}\ (x + 1)$$

we can use the above lemma *evenTwice* to show that *odd* (*twice* $x + 1$). Yet we will need to allude to *evenTwice* explicitly, as the fixed conversion rule described does not include support for proving whether a number is even or odd.

A different approach is thus needed, where user extensions to the conversion rule are allowed, leading to proofs that omit the trivial details that are needed even in user-specified domains. This has to be done in a safe manner: user extensions should not introduce equivalences that are not provable in the original logic as this would directly lead to unsoundness. Here is where the VeriML type system becomes necessary: we can use the rich typing offered by VeriML in order to impose this safety guarantee for user extensions. We require user extensions to provide proof for the claimed equivalences by expecting them to obey a specific type, such as the following:

$$\mathsf{userExtension}\ :\ (t_1 : T) \to (t_2 : T) \to (t_1 = t_2)$$

VeriML is an expressive language, allowing for a large number of such extensions to be implemented. This includes the equivalence checkers for β-conversion, congruence closure as well as arithmetic simplifications. Thus none of these need to be included inside the logic – they can all be implemented in VeriML and taken outside the trusted core of the system. Based on this idea, we will present the details of how to support a *safe* and *user-extensible* alternative to the conversion rule in Section 7.1, where conversion is implemented as a call to tactics of a specific type. We have implemented a version of the conversion rule that includes all the equivalence checkers we have mentioned. Supporting these has previously required a large metatheoretic extension to the CIC logic and involved significant reengineering effort as part of the CoqMT project [Strub, 2010].

So far we have only talked about using the conversion rule as part of proof objects. Let us now consider what user-extensible checking amounts to in the case of tactics. Imagine a tactic that proves whether a number is even or odd.

$$\mathsf{evenOrOdd} \ : \ (\phi : ctx) \rightarrow (n : Nat) \rightarrow (even\ n) + (odd\ n)$$

The cases for $2x$ and $twice\ x$ would be:

$$\begin{array}{l} \mathsf{evenOrOdd}\ \phi\ n \ = \ \mathsf{holmatch}\ n\ \mathsf{with} \\ \qquad\qquad 2x \ \mapsto \ \mathsf{inj}_1 \ \langle \ evenMult2\ x \ \rangle \\ \qquad\qquad twice\ x \ \mapsto \ \mathsf{inj}_1 \ \left\langle \ \cdots : even\ (twice\ x) \ \right\rangle \\ \qquad\qquad \ldots \end{array}$$

In the case of $twice\ x$, we can directly use the *evenTwice* lemma. Yet this lemma is proved through one step (the allusion to $evenMult2\ x$) when the conversion rule includes the equivalence checkers pictured in Figure 7.1. We would rather avoid stating and proving the lemma separately, alluding instead to the $evenMult2\ x$ theorem directly, even in this case, handling the trivial rewriting details through the conversion rule. That is, we would like proofs contained within tactics to be checked using the extensible conversion rule as well. Furthermore, these checks should be *static* so as to know as soon as possible whether the proofs are valid or not. As we said above, the conversion rule amounts to a call to a VeriML tactic. Therefore we need to evaluate the calls to the conversion rule statically, at the definition time of tactics such as evenOrOdd. We achieve this by making critical use of

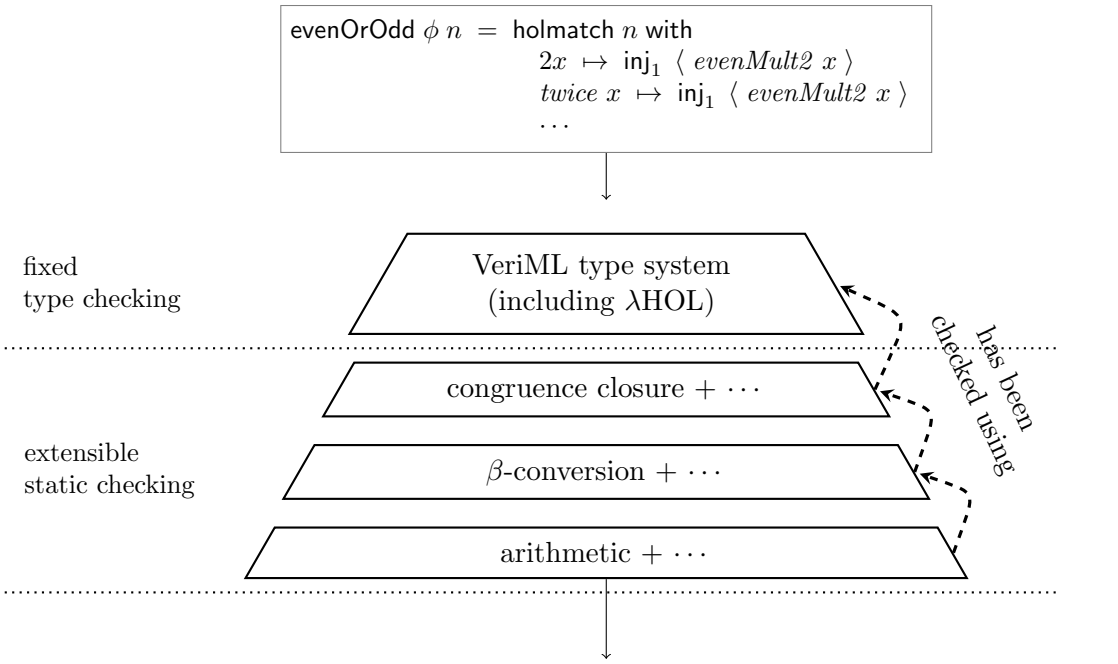

Figure 7.2: User-extensible static checking of tactics

two features of VeriML we have already described: the staging extension of Section 6.3 and the collapsed logical terms of Section 5.2.

The essential idea is to view static checking as a two-phase procedure: first the normal VeriML typechecker is used; then, during the static evaluation phase of VeriML semantics, additional checks are done. These checks might involve arbitrary proof search and are user-defined and user-extensible, through normal VeriML code. In this way, proofs contained within tactics can use calls to powerful procedures, such as the conversion rule, yet be statically checked for validity – just as if the checks became part of the original λHOL logic. Owing to the expressive VeriML type system, this is done without needing to extend the logic at all. This amounts to *user-extensible static checking* within tactics, which we cover in more detail in Section 7.2.

The support of user-extensible static checking enables us to *layer checking functions*, so as to simplify their implementation, as shown in Figure 7.2. This corresponds to mathematical practice: when working on proofs of a domain such as arithmetic, we omit trivial details such as equality rewriting steps. Once we have a fair amount of intuition about arithmetic, we can move to further domains such as linear algebra – where arithmetic steps

themselves become trivial and are omitted; and so on. Similarly, by layering static checking procedures for increasingly more complicated domains, we can progressively omit more trivial details from the proofs we use in VeriML. This is true whether we are interested in the proofs themselves, or when the proofs are parts of tactics, used to provide automation for further domains. For example, once we have programmed a conversion rule that supports congruence closure, programming a conversion rule that supports β-reduction is simplified. The reason is that the proofs contained within the β-conversion rule do not have to mention the congruence proof steps, just as we could skip details in evenOrOdd above. Similarly, implementing an arithmetic conversion rule benefits from the existence of congruence closure and β-conversion.

Our approach to user-extensible static checking for proofs and tactics can in fact be viewed as a more general feature of VeriML: the ability to support *extensible type checking*. Functional languages such as ML and Haskell allow users to define their own algebraic datatypes. Dependently typed languages, such as versions of Haskell with GADTs [Peyton Jones et al., 2006] and Agda [Norell, 2007], allow users to define and enforce their own invariants on those datatypes, through dependent types. Yet, since the type checking procedure of such languages is fixed, these extra invariants cannot always be checked statically. VeriML allows us to program the automated procedures to prove such invariants within the same language and evaluate them statically through the staging mechanism. In this way, type checking becomes extensible by user-defined properties and user-defined checking procedures, while type safety is maintained. The details of this idea are the subject of Section 7.3.

7.1 User-extensible static checking for proofs

7.1.1 The extensible conversion rule

Issues with the explicit equality approach. In Section 3.2 we presented various alternatives to handling equality within the λHOL logic. We chose the explicit equality approach, where every equality step is explicitly witnessed inside proofs. As we discussed, this leads to the simplest possible type-checker for the logic. Yet equality steps are so ubiquitous inside

proofs that proof objects soon become prohibitively large. One such example is proving that $(succ\ x) + y = succ\ (x + y)$. Assuming the following constants in the signature Σ concerning natural numbers, where *elimNat* represents primitive recursion over them:

$$\begin{array}{lcl} Nat & : & Type \\ zero & : & Nat \\ succ & : & Nat \to Nat \\ elimNat_{\mathcal{K}} & : & \mathcal{K} \to (Nat \to \mathcal{K} \to \mathcal{K}) \to Nat \to \mathcal{K} \\ elimNatZero & : & \forall f_z f_s, elimNat_{\mathcal{K}}\ f_z\ f_s\ zero = f_z \\ elimNatSucc & : & \forall f_z f_s p, elimNat_{\mathcal{K}}\ f_z\ f_s\ (succ\ p) = f_s\ p\ (elimNat_{\mathcal{K}}\ f_z\ f_s\ p) \end{array}$$

we can define natural number addition as:

$$plus\ =\ \lambda a : Nat.\lambda b : Nat.elimNat_{Nat}\ b\ (\lambda p.\lambda r.succ\ r)\ a$$

To prove the desired proposition, we need to go through the following steps:

$$\begin{array}{l} plus\ (succ\ x)\ y = \\ = (\lambda a : Nat.\lambda b : Nat.elimNat\ b\ (\lambda p.\lambda r.succ\ r)\ a)\ (succ\ x)\ y \\ \qquad\qquad \text{(by definition of } plus\text{)} \\ = (\lambda b : Nat.elimNat\ b\ (\lambda p.\lambda r.succ\ r)\ (succ\ x))\ y \\ \qquad\qquad \text{(by EqBeta)} \\ = elimNat\ y\ (\lambda p.\lambda r.succ\ r)\ (succ\ x) \\ \qquad\qquad \text{(by EqBeta)} \\ = (\lambda p.\lambda r.succ\ r)\ x\ (elimNat\ y\ (\lambda p.\lambda r.succ\ r)\ x) \\ \qquad\qquad \text{(by } elimNatSucc\text{)} \\ = succ\ (elimNat\ y\ (\lambda p.\lambda r.succ\ r)\ x) \\ \qquad\qquad \text{(by EqBeta, twice)} \\ = succ\ (plus\ x\ y) \\ \qquad\qquad \text{(by EqBeta and EqSymm, twice)} \end{array}$$

The resulting proof object explicitly mentions this steps, as well as uses of equality transitivity in order to combine the steps together. Generating and checking such objects with such painstaking level of detail about equality soon becomes a bottleneck. For example, imagine the size of a proof object proving that $100000 + 100000 = 200000$: we would need at least 100000 applications of logical rules such as EqBeta and *elimNatSucc* in order to prove a proposition which otherwise follows trivially by computation.

Extensions to typing (stratified version)

$$\frac{\Phi \vdash_C \pi : P \qquad P \equiv_{\beta\mathbb{N}} P'}{\Phi \vdash_C \pi : P'} \textsc{Conversion}$$

Extensions to typing (PTS version)

$$\frac{\Phi \vdash_C t : t' \qquad \Phi \vdash_C t' : \mathit{Prop} \qquad t' \equiv_{\beta\mathbb{N}} t''}{\Phi \vdash_C t : t''} \textsc{Conversion}$$

$t \to_{\beta\mathbb{N}} t'$

$$(\lambda x : \mathcal{K}.t)\ t' \to_{\beta\mathbb{N}} t[t'/x]$$
$$\mathit{elimNat}_{\mathcal{K}}\ t_z\ t_s\ \mathit{zero} \to_{\beta\mathbb{N}} t_z$$
$$\mathit{elimNat}_{\mathcal{K}}\ t_z\ t_s\ (\mathit{succ}\ t) \to_{\beta\mathbb{N}} t_s\ t\ (\mathit{elimNat}_{\mathcal{K}}\ t_z\ t_s\ t)$$

$t \equiv_{\beta\mathbb{N}} t'$

The compatible ($t \equiv t' \Rightarrow t''[t/x] \equiv t''[t'/x]$), reflexive, symmetric and transitive closure of $t \to_{\beta\mathbb{N}} t'$

Figure 7.3: The logic λHOL_C: Adding the conversion rule to the λHOL logic

Introducing the conversion rule. In order to solve this problem, various logics introduce a notion of terms that are *definitionally equal*: that is, they are indistinguishable with respect to the logic. For example, we can say that terms that are equivalent based on computation alone such as $100000 + 100000$ and 200000 or $(\mathit{succ}\ x) + y$ and $\mathit{succ}\ (x + y)$ correspond to the *same term*. In this way, the rewriting steps to go from one to the other do not need to be explicitly witnessed in a proof object and a simple application of reflexivity is enough to prove their equality:

$$\vdash (\mathit{refl}\ ((\mathit{succ}\ x) + y)) : ((\mathit{succ}\ x) + y = \mathit{succ}\ (x + y))$$

This typing derivation might be surprising at first: we might expect the type of this proof object to be $((\mathit{succ}\ x) + y) = ((\mathit{succ}\ x) + y)$. Indeed, this is a valid type for the proof object. But since $((\mathit{succ}\ x) + y)$ is *indistinguishable* from $\mathit{succ}\ (x + y)$, the type we gave above for this proof object is just as valid.

In order to include this notion of definitional equality, we introduce a *conversion rule* in the λHOL logic, with details shown in Figure 7.3. We denote definitional equality as $\equiv$ and include β-reduction and natural number primitive recursion as part of it. The conversion

rule is practically what makes definitionally equal terms indistinguishable, allowing to view a proof object for a proposition P as a proof object for any definitionally equal proposition P'. In order to differentiate between the notion of λHOL with explicit equality that we have considered so far, we refer to this version with the conversion rule as λHOL_C and its typing judgement as $\Psi;\ \Phi \vdash_c t : t'$; whereas the explicit equality version is denoted as λHOL_E.

Including the conversion rule in λHOL means that when we prove the metatheory of the logic, we need to prove that the conversion rule does not introduce any unsoundness to the logic. This proof complicates the metatheory of the logic significantly. Furthermore, the trusted base of a system relying on this logic becomes bigger. The reason is that the implementation of the logic type checker now needs to include the implementation of the conversion rule. This implementation can be rather large, especially if we want the conversion rule to be efficient. The implementation essentially consists of a partial evaluator for the functional language defined through β-reduction and natural number elimination. This can be rather complicated, especially as evaluation strategies such as compilation to bytecode have been suggested [Grégoire, 2003]. Furthermore, it is desirable to support even more sophisticated definitional equality as part of the conversion rule, as evidenced in projects such as Blanqui et al. [1999], Strub [2010] and Blanqui et al. [2010]. For example, by including congruence closure in the conversion rule, proof steps that perform rewriting based on hypotheses can be omitted: if we know that $x = y$ and $y = z$, then proving $f\ x = f\ z$ becomes trivial. The size of the resulting proof objects is further reduced. Of course, extending definitional equality also complicates the metatheory further and enlarges the trusted base. Last, user extensions to definitional equality are impossible: any new extension requires a new metatheoretic result in order to make sure that it does not violate logical soundness.

The conversion rule in VeriML. We will now see how VeriML makes it possible to reconcile the explicit equality based approach with the conversion rule: we will gain the conversion rule back, albeit it will remain completely outside the logic. Therefore we will be free to extend it, all the while without risking introducing unsoundness in the logic, since the logic remains fixed (λHOL_E as presented above).

The crucial step is to recognize that the conversion rule essentially consists of a *trusted tactic* that is hardcoded within the logic type checker. This tactic checks whether two terms are definitionally equal or not; if it claims that they are, we trust that they are indeed so, and no proof object is produced. This is where the space gains come from. We can thus view the definitional equality checker as a trusted function of type:

$$\mathsf{defEqual} \ : \ (P : \mathit{Prop}) \to (P' : \mathit{Prop}) \to \mathsf{bool}$$

and the conversion rule as follows:

$$\frac{\Phi \vdash_C \pi : P \qquad \mathsf{defEqual}\ P\ P' \longrightarrow^*_{\mathrm{ML}} \mathsf{true}}{\Phi \vdash_C \pi : P'}\ \textsc{Conversion}$$

where $\longrightarrow^*_{\mathrm{ML}}$ stands for the operational semantics of ML – the meta-language where the type checker for λHOL is implemented.

This key insight leads to our alternative treatment of the conversion rule in VeriML. First, instead of hardcoding the trusted definitional equality checking tactic inside the logic type checker, we program a *type-safe definitional equality tactic*, utilizing the features of VeriML. Based on typing alone, we require that this function returns a valid proof object of the claimed equivalences:

$$\mathsf{defEqual} \ : \ (\phi : \mathit{ctx},\ P : \mathit{Prop},\ P' : \mathit{Prop}) \to \mathsf{option}\ \left(P = P'\right)$$

Second, we evaluate this tactic under *proof erasure semantics*. This means that no proof object for $P = P'$ is produced, leading to the same space gains as the original conversion rule. In fact, under this semantics, the type $\mathsf{option}\ \left(P = P'\right)$ is isomorphic to bool – so the correspondence with the original conversion rule is direct. Third, we use the staging construct in order to *check conversion statically*. In this way, we will know whether the conversion checks are successful as soon as possible, after stage-one evaluation. The main reason why this is important will be mostly evident in the following section. Informally, the conversion rule now becomes:

$$\frac{\Phi \vdash_E \pi : P \qquad \left\{\left\{\mathsf{defEqual}\ \Phi\ P\ P'\right\}_{\mathrm{erase}}\right\}_{\mathrm{static}} \longrightarrow^*_{\mathrm{VeriML}} \mathsf{Some}\ \langle\ \mathit{admit}\ \rangle}{\Phi \vdash_E \pi : P'}$$

Through this treatment, the choice of what definitional equality corresponds to can be decided at will by the VeriML programmer, rather than being fixed at the time of the definition of λHOL.

As it stands the rule above mixes typing for λHOL terms with the operational semantics of a VeriML expression. Since typing for VeriML expressions depends on λHOL typing, adding a dependence on the VeriML operational semantics would severely complicate the definition of the VeriML language. We would not be able to use the standard techniques of proving type safety in the style of Chapter 6. We resolve this by leaving the conversion rule *outside* the logical core. Instead of redefining typing for λHOL proof objects we will define a notion of *proof object expressions*, denoted as e_π. These are normal VeriML expressions which statically evaluate to a proof object. The only computational part they contain are calls to the definitional equality tactic. Thus the combination of VeriML typing and VeriML static evaluation for proof object expressions exactly corresponds to λHOL_C typing for proof objects:

$$\Phi \vdash_C \pi : P \quad \Leftrightarrow \quad \bullet;\ \bullet;\ \Gamma \vdash e_\pi : (P) \wedge (\ \mu\ ,\ e_\pi\ ) \longrightarrow_s^* (\ \mu\ ,\ \langle\ \pi'\ \rangle\ )$$

We will show that we can convert any λHOL_C proof object π into an equivalent proof object expression through a type-directed translation: $[\![\Phi \vdash_C \pi : P]\!] = e_\pi$. Through proof-erasure the calls to the definitional equality tactic will not produce proof objects at runtime, yielding the space savings inherrent in the λHOL_C approach. Furthermore, since the use of proof erasure semantics is entirely optional in VeriML, we can also prove that a full proof object in the original λHOL_E logic can be produced:

$$\frac{\Phi \vdash_C \pi : P \qquad [\![\Phi \vdash_C \pi : P]\!] = e_\pi \qquad (\ \mu\ ,\ e_\pi\ ) \longrightarrow_s^* \left(\ \mu\ ,\ \langle\ \pi'\ \rangle\ \right)}{\Phi \vdash_E \pi' : P}$$

This proof object only requires the original type checker for λHOL_E without any conversion-related extensions, as all conversion steps are explicitly witnessed.

Formal details. We are now ready to present our approach formally. We give the implementation of the definitional equality checker in VeriML in Code Listings 7.1 and 7.2. We support the notion of definitional equality presented above: the compatible closure over β-reduction and natural number elimination reduction. As per standard practice, equality

```
rewriteBetaNatStep : (φ : ctx, T : Type, t : T) → (t' : T) × (t = t')
rewriteBetaNatStep φ T t =
  holmatch t with
    (t1 : T' → T) (t2 : T') ↦
      let ⟨ t1', H1 : t1 = t1' ⟩ = rewriteBetaNat φ (T' → T) t1 in
      let ⟨ t', H2 : t1' t2 = t' ⟩ =
        holmatch t1' with
            λx : T'.tf  ↦  ⟨ tf/[id_φ, t2], G1 ⟩
          | t1'         ↦  ⟨ t1' t2, G2 ⟩
      in
      ⟨ t', G3 ⟩
    | elimNat_T fz fs n ↦
        let ⟨ n', H3 ⟩ = rewriteBetaNat φ Nat n in
        let ⟨ t', H4 : elimNat_T fz fs n' = t' ⟩ =
          holmatch n' with
              zero     ↦  ⟨ fz, G4 ⟩
            | succ n'  ↦  ⟨ fs n' (elimNat_T fz fs n'), G5 ⟩
            | n'       ↦  ⟨ elimNat_T fz fs n', G6 ⟩
        in
        ⟨ t', G7 ⟩
    | t ↦ ⟨ t, G8 ⟩

rewriteBetaNat : (φ : ctx, T : Type, t : T) → (t' : T) × (t = t')
rewriteBetaNat φ T t =
  let ⟨ t', H5 : t = t' ⟩ = rewriteBetaNatStep φ T t in
  holmatch t' with
      t   ↦  ⟨ t, G9 ⟩
    | t'  ↦  let ⟨ t'', H6 ⟩ = rewriteBetaNat φ T t' in ⟨ t'', G10 ⟩
```

Code Listing 7.1: Implementation of the definitional equality checker in VeriML (1/2)

```
defEqual : (φ : ctx, T : Type, t1 : T, t2 : T) → option (t1 = t2)
defEqual φ T t1 t2 =
  let ⟨ t1', H1 : t1 = t1' ⟩ = rewriteBetaNat φ T t1 in
  let ⟨ t2', H2 : t2 = t2' ⟩ = rewriteBetaNat φ T t2 in
  do ⟨ H : t1' = t2' ⟩ ← holmatch t1', t2' with

      (ta : T' → T) tb, tc td ↦
        do ⟨ Ha : ta = tc ⟩ ← defEqual φ T' ta tc
           ⟨ Hb : tb = td ⟩ ← defEqual φ T tb td
           return ⟨ G1 : ta tb = tc td ⟩

    | ((ta : T') = tb), (tc = td) ↦
        do ⟨ Ha : ta = tc ⟩ ← defEqual φ T' ta tc
           ⟨ Hb : tb = td ⟩ ← defEqual φ T tb td
           return ⟨ G2 : (ta = tb) = (tc = td) ⟩

    | (ta → tb), (tc → td) ↦
        do ⟨ Ha : ta = tc ⟩ ← defEqual φ Prop ta tc
           ⟨ Hb : tb = td ⟩ ← defEqual φ Prop tb td
           return ⟨ G3 : (ta → tb) = (tc → td) ⟩

    | (∀x : T', ta), (∀x : T', tb) ↦
        do ⟨ Ha : ta = tb ⟩ ← defEqual (φ, x : T') Prop ta tb
           return ⟨ G4 : (∀x : T', ta) = (∀x : T', tb) ⟩

    | (λx : T'.(ta : T'')), (λx : T'.tb) ↦
        do ⟨ Ha : ta = tb ⟩ ← defEqual (φ, x : T') T'' ta tb
           return ⟨ G5 : (λx : T'.ta) = (λx : T'.tb) ⟩

    | ta, ta ↦ do return ⟨ G6 : ta = ta ⟩

    | ta, tb ↦ None

  in return ⟨ G : t1 = t2 ⟩

reqEqual : (φ : ctx, T : Type, t1 : T, t2 : T) → (t1 = t2)
reqEqual φ T t1 t2 = match defEqual φ T t1 t2 with
                       Some ⟨ H : t1 = t2 ⟩ ↦ ⟨ H ⟩
                     | None                 ↦ error
```

Code Listing 7.2: Implementation of the definitional equality checker in VeriML (2/2)

checking is split into two functions: one which reduces a term to weak-head normal form (rewriteBetaNat in Code Listing 7.1) and the actual checking function defEqual which compares two terms structurally after reduction. We also define a version of the tactic that raises an error instead of returning an option type if we fail to prove the terms equal, which we call reqEqual. For presentation reasons we omit the proof objects contained within the tactics. We instead name the respective proof obligations within the tactics as G_i and give their type. Most obligations are easy to fill in based on our presentation of λHOL_E in Section 3.2 and can be found in our prototype implementation of VeriML.

The code of the defEqual tactic is entirely similar to the code one would write for the definitional equality checking routine inside the λHOL_C logic type checker, save for the extra types and proof objects. It therefore follows trivially that everything that holds for the standard implementation of the conversion check also holds for this code: e.g. it corresponds exactly to the $\equiv_{\beta\mathbb{N}}$ relation as defined in the logic; it is bound to terminate because of the strong normalization theorem for this relation; and its proof-erased version is at least as trustworthy as the standard implementation. Formally, we write:

Lemma 7.1.1

$$1.\ \frac{\Phi \vdash t : T \qquad t \equiv_{\beta\mathbb{N}} t'}{\mathit{reqEqual}\ \Phi\ T\ t\ t' \longrightarrow^* \langle\ \pi\ \rangle \qquad \Phi \vdash \pi : t = t'}$$

$$2.\ \frac{\Phi \vdash t : T \qquad \mathit{reqEqual}\ \Phi\ T\ t\ t' \longrightarrow^* \mathit{error}}{t \not\equiv_{\beta\mathbb{N}} t'}$$

Proof. Through standard techniques show that $\equiv_{\beta\mathbb{N}}$ is equivalent to structural equality modulo rewriting to $\beta\mathbb{N}$-weak head-normal form. Then, by showing that the combination of defEqual and rewriteBetaNat accurately implement the latter relation. □

With this implementation of definitional equality in VeriML, we are now ready to translate λHOL_E proof objects into VeriML expressions. Essentially, we will replace the implicit definitional equality checks used in the conversion rule within the proof objects with explicit calls to the reqEqual tactic. We define a type-directed translation function in Figure 7.4

$\boxed{[\![\Phi \vdash_C \pi_C : P]\!]' = \langle\ \pi_E \mid l\ \rangle}$

$$\frac{[\![\Phi,\ x : P \vdash_C \pi_C : P']\!]' = \langle\ \pi_E \mid l\ \rangle}{[\![\Phi \vdash_C \lambda x : P.\pi_C : P \to P']\!]' = \langle\ \lambda x : P.\pi_E \mid l\ \rangle}\ {\to}\textsc{Intro}$$

$$\frac{[\![\Phi \vdash_C \pi_C^1 : P' \to P]\!]' = \langle\ \pi_E^1 \mid l^1\ \rangle \qquad [\![\Phi \vdash_C \pi_C^2 : P']\!]' = \langle\ \pi_E^2 \mid l^2\ \rangle}{[\![\Phi \vdash_C \pi_C^1\ \pi_C^2 : P]\!]' = \langle\ \pi_E^1\ \pi_E^2 \mid l^1 \uplus l^2\ \rangle}\ {\to}\textsc{Elim}$$

$$\frac{[\![\Phi,\ x : \mathcal{K} \vdash_C \pi_C : P]\!]' = \langle\ \pi_E \mid l\ \rangle}{[\![\Phi \vdash_C \lambda x : \mathcal{K}.\pi_C : \forall x : \mathcal{K}.P]\!]' = \langle\ \lambda x : \mathcal{K}.\pi_E \mid l\ \rangle}\ \forall\textsc{Intro}$$

$$\frac{[\![\Phi \vdash_C \pi_C : \forall x : \mathcal{K}.P]\!]' = \langle\ \pi_E \mid l\ \rangle}{[\![\Phi \vdash_C \pi_C\ d : P[d/x]]\!]' = \langle\ \pi_E\ d \mid l\ \rangle}\ \forall\textsc{Elim} \qquad \frac{}{[\![\Phi \vdash_C x : P]\!]' = \langle\ x \mid \emptyset\ \rangle}\ \textsc{Var}$$

$$\frac{[\![\Phi \vdash_C \pi_C : P]\!] = \langle\ \pi_E \mid l\ \rangle \qquad P \equiv_{\beta\mathbb{N}} P' \qquad G_n \notin l}{[\![\Phi \vdash_C \pi_C : P']\!] = \langle\ conv\ \pi_E\ G_n \mid l \uplus [G_n \mapsto (\Phi,\ P,\ P')]\ \rangle}\ \textsc{Conversion}$$

$\boxed{[\![\Phi \vdash_C \pi_C : P]\!] = e_\pi}$

$$\frac{[\![\Phi \vdash_C \pi_C : P]\!]' = \langle\ \pi_E \mid l\ \rangle \qquad |l| = n \qquad l = \overrightarrow{G_i \mapsto (\Phi_i,\ P_i,\ P_i')}}{\begin{array}{l}[\![\Phi \vdash_C \pi_C : P]\!] = (\mathsf{letstatic}\ \langle\ G_1\ \rangle\ =\ [\![\mathsf{reqEqual}\ \Phi_1\ \mathit{Prop}\ P_1\ P_1']\!]_E\ \mathsf{in} \\ \quad \cdots \\ \quad \mathsf{letstatic}\ \langle\ G_n\ \rangle\ =\ [\![\mathsf{reqEqual}\ \Phi_n\ \mathit{Prop}\ P_n\ P_n']\!]_E\ \mathsf{in} \\ \quad \langle\ [\Phi]\ \pi_E\ \rangle)\end{array}}$$

Figure 7.4: Conversion of $\lambda\mathrm{HOL}_C$ proof objects into VeriML proof object expressions

that does exactly that. The auxiliary function $[\![\Phi \vdash_C \pi_C : P]\!]'$ forms the main part of the translation. It yields an equivalent λHOL_E proof object π_E, which is mostly identical to the original λHOL_C proof object, save for uses of the conversion rule. A use of the conversion rule for terms t and t' is replaced with an explicit proof object that has a placeholder G_i for the proof of $t = t'$. The same auxiliary function also yields a map from the placeholders G_i into triplets $(\Phi,\ t,\ t')$ that record the relevant information of each use of the conversion rule [2]. The main translation function $[\![\Phi \vdash_C \pi_C : P]\!]$ then combines these two parts into a valid VeriML expression, calling the reqEqual tactic in order to fill in the G_i placeholders based on the recorded information.

While this translation explicitly records the information for every use of conversion – that is, the triplet $(\Phi,\ t,\ t')$, a very important point we need to make is that VeriML type information can be used to infer this information. Therefore, an alternate translation could produce VeriML proof object expressions of the form:

$$\begin{array}{l} \mathsf{letstatic}\ G_1\ =\ [\![\mathsf{reqEqual}\ \_\ \_\ \_\ \_]\!]_E\ \mathsf{in} \\ \ldots \\ \mathsf{letstatic}\ G_n\ =\ [\![\mathsf{reqEqual}\ \_\ \_\ \_\ \_]\!]_E\ \mathsf{in} \\ \langle\ \pi_E\ \rangle \end{array}$$

This evidences a clear benefit of recording logic-related information in types: based on typing alone, we can infer normal arguments to functions, even when these have an actual run-time role, such as t and t'. The function reqEqual pattern matches on these objects, yet their values can be inferred automatically by the type system based on their usage in a larger expression. This benefit can be pushed even further to eliminate the need to record the uses of the conversion rule, allowing the user to write proof object expressions that look *identical* to λHOL_C proof objects. Yet, as we will see shortly, proof object expressions are free to use an arbitrary conversion rule instead of the fixed conversion rule in λHOL_C; thus, by extending the conversion rule appropriately, we can get small proof object expressions even in cases where the corresponding λHOL_C proof object would be prohibitively large.

We will not give the full details of how to hide uses of the conversion rule in proof object expressions. Briefly we will say that this goes through an algorithmic typing formulation

2. We assume that the names and uses of the G_i are α-converted accordingly when joining lists together.

for λHOL_C proof objects: instead of having an explicit conversion rule, conversion checks are interspersed in the typing rules for the other proof object constructors. For example, implication elimination is now typed as follows:

$$\frac{\Phi \vdash_C \pi : P \to P'' \qquad \Phi \vdash_C \pi' : P' \qquad P \equiv_{\beta\mathbb{N}} P'}{\Phi \vdash_C \pi\ \pi' : P''}$$

Having such a formulation at hand, we implement each such typing rule as an always-terminating VeriML function with an appropriate type. Conversion checks are replaced with a requirement for proof evidence of the equivalence. For example:

$$\mathsf{implElim}\ :\ (\phi : ctx,\ P, P', P'' : Prop,\ H_1 : P \to P',\ H_2 : P',\ H_3 : P = P') \to \big(P''\big)$$

Then, the translation from λHOL_C proof objects into proof object expressions replaces every typing rule with the VeriML tactic that implements it. Type inference is used to omit all but the necessary arguments to these tactics; and static, proof-erased calls to reqEqual are used to solve the equivalence goals. Thus through appropriate syntactic sugar to hide the inferred arguments, the calls to the corresponding VeriML tactics look identical to the original proof object.

Correspondence with original proof object. We use the term *proof object expressions* to refer to the VeriML expressions resulting from this translation and denote them as e_π. We will now elucidate the correspondence between such proof object expressions and the original λHOL_C proof object. To do that, it is fruitful to view the proof object expressions as a proof certificate, sent to a third party. The steps required to check whether it constitutes a valid proof are the following. First, the whole expression is checked using the VeriML type checker; this includes checking the included λHOL_E proof object π_E through the λHOL_E type checker. Then, the calls to the reqEqual function are evaluated during stage one, using proof erasure semantics and thus do not produce additional proof objects. We expect such calls to reqEqual to be successful, just as we expect the conversion rule to be applicable in the original λHOL_C proof object when it is used. Last, stage-two evaluation is performed, but the proof object expression has already been reduced to a value after the first evaluation stage. Thus checking the validity of the proof expression is entirely equivalent to the behavior of type-checking the λHOL_C proof object, save for pushing all

conversion checks towards the end. Furthermore, the size of the proof object expression e_π is comparable to the original π_C proof object; the same is true for the resulting π_E proof object when proof erasure semantics are used for the calls to reqEqual. Of course, use of the proof erasure semantics is entirely optional. If we avoid using them (e.g. in cases where the absolute minimum trusted base is required), the resulting π_E proof object will be exponentially larger due to the full definitional equality proofs.

Formally, we have the following lemma, assuming no proof erasure semantics are used for VeriML evaluation.

Lemma 7.1.2 *(λHOL_C proof object – VeriML proof object expression equivalence)*

$$\frac{\Phi \vdash_C \pi_C : P \qquad [\![\Phi \vdash_C \pi_C : P]\!] = e_\pi}{\bullet;\ \bullet;\ \bullet \vdash e_\pi : \big([\Phi]\, P\big) \qquad e_\pi \longrightarrow_s^* \langle\ [\Phi]\, \pi_E\ \rangle \qquad \Phi \vdash_E \pi_E : P}$$

Extending conversion at will. In our treatment of the conversion rule we have so far focused on regaining the conversion rule with $\equiv_{\beta\mathbb{N}}$ as the definitional equality in our framework. Still, there is nothing confining us to supporting this notion of definitional equality only. As long as we can program an equivalence checker in VeriML that has the right type, it can safely be made part of the definitional equality used by our notion of the conversion rule. That is, any equivalence relation R can be viewed as the definitional equality $\equiv_R$, provided that a $\mathsf{defEqual}_R$ tactic that implements it can be written in VeriML, with the type:

$$\mathsf{defEqual}_R\ :\ (\phi : ctx,\ T : Type,\ t : T,\ t' : T) \to \mathsf{option}\ \big(t = t'\big)$$

For example, we have written an eufEqual function, which checks terms for equivalence based on the equality with uninterpreted functions decision procedure – also referred to as congruence. This equivalence checking tactic isolates hypotheses of the form $t_1 = t_2$ from the current context; then, it constructs a union-find data structure in order to form equivalence classes of terms. Based on this structure, and using code similar to defEqual (recursive calls on subterms), we can decide whether two terms are equal up to simple uses of the equality hypotheses at hand. We have combined this tactic with the original defEqual tactic, making the implicit equivalence supported similar to the one in the Calculus

of Congruent Constructions [Blanqui et al., 2005]. This demonstrates the flexibility of this approach: equivalence checking is extended with a sophisticated decision procedure, which is programmed using its original, imperative formulation. We have extended defEqual further in order to support rewriting based on arithmetic facts, so that simple arithmetic simplifications are taken into account as part of the conversion rule.

In our implementation, we have programmed the equality checking procedure defEqual in an extensible manner, so that we can globally register further extensions. We present details of this in Chapter 9. We can thus view defEqual as being composed from a number of equality checkers $\mathsf{defEqual}_{R_1}, \cdots, \mathsf{defEqual}_{R_n}$, corresponding to the definitional equality relation $\equiv_{R_1 \cup \cdots \cup R_n}$. We will refer to this relation as $\equiv_R$.

We would like to stress that by keeping the conversion rule outside the logic, the user extensions cannot jeopardize the soundness of the logic. This choice also allows for adding undecidable equivalences to the conversion rule (e.g. functional extensionality) or potentially non-terminating decision procedures. Soundness is guaranteed thanks to the VeriML type system, by requiring that extensions return a proof of the claimed equivalences. Through type safety we know that such a proof exists, even if it is not produced at runtime.

We actually make use of non-terminating extensions to the conversion rule in our developments. For example, before implementing the eufEqual tactic for deciding equivalence based on equality hypotheses, we implement a naïve version of the same tactic, that blindly rewrites terms based on equality hypotheses – that is, given a hypothesis $a = b$ and two terms t and t' to test for equivalence, it rewrites all occurrences of a to b inside t and t'. While this strategy would lead to non-termination in the general case, it simplifies implementing eufEqual significantly, as many proof obligations within that tactic can be handled automatically through the naïve version. After eufEqual is implemented, the naïve version does not get used. Still, this usage scenario demonstrates that associating decidability of equivalence checking with the soundness of the conversion rule is solely an artifact of the traditional way to integrate the conversion rule within the internals of the logic.

7.1.2 Proof object expressions as certificates

In terms of size and checking behavior, a proof object expression e_π that makes calls to an appropriately extended defEqual procedure corresponds closely to proof objects with a version of λHOL$_C$ with $\equiv_R$ as the definitional equality used in the conversion rule. These proof object expressions can be utilized as an alternative proof certificate format, to be sent to a third party. Their main benefit over proof objects is that they allow the receiving third party to decide on the tradeoff between the trusted base and the cost of validity checking. This is due to the receiver having a number of choices regarding the evaluation of each defEqual$_{R_i}$ implementing the sub-relations $\equiv_{R_i}$ of $\equiv_R$. First, the receiver can use proof erasure for evaluation as suggested above. In this case, the proof-erased version of defEqual$_{R_i}$ becomes part of the trusted base – though the fact that it has been verified through the VeriML type system gives a much better assurance that the function can indeed be trusted. Another choice would be to use non-erasure semantics and have the function return an actual proof object. This is then checked using the original λHOL$_E$ type checker. In that case, the VeriML type system does not need to become part of the trusted base of the system. Last, the 'safest possible' choice would be to avoid doing any evaluation of the function, and ask the proof certificate provider to do the evaluation of defEqual$_{R_i}$ themselves. In that case, no evaluation of its code would need to happen at the proof certificate receiver's side. This mitigates any concerns one might have for code execution as part of proof validity checking, and guarantees that the small λHOL$_E$ type checker is the trusted base in its entirety.

The receiver can decide on the above choices separately for each defEqual$_{R_i}$ – e.g. use proof erasure for rewriteBetaNat but not for eufEqual, leading to a trusted base identical to the λHOL$_C$ case. This means that *the choice of what the conversion rule is rests with the proof certificate receiver and not with the designer of the logic.* Thus the proof certificate receiver can choose the level of trust they require at will, while taking into account the space and time resources they want to spend on validation. Still, the producer of the proof certificate has the option of further extensions to the conversion rule. Thus proof object expressions constitute an *extensible proof certificate format* –that is, a certificate format

with an extensible checking procedure– that still allows for the actual size of the checking procedure, or the trusted base, to be decided by the checking party.

Disambiguating between notions of proof. So far we have introduced a number of different notions of proof: proof objects in λHOL_C and λHOL_E; proof object expressions and proof expressions; we have also mentioned proof scripts in traditional proof assistants. Let us briefly try to contrast and disambiguate between these notions.

As we have said, proof objects in λHOL_E are a kind of proof certificate where every single proof step is recorded in full detail; therefore, the size of the trusted base required for checking them is the absolute minimum of the frameworks we consider. Yet their size is often prohibitively large, rendering their transfer to a third party and their validation impractical. Proof object expressions in VeriML, as generated by the translation offered in this chapter, can be seen as proof objects with some holes that are filled in computationally. That is, some simple yet tedious proof steps are replaced with calls to *verified* VeriML proof-producing functions. The fact that these functions are verified is a consequence of requiring them to pertain to a certain type. The type system of VeriML guarantees that these functions are safe to trust: that is, if they return successfully, the missing part of the proof object can indeed be filled in. Seen as proof certificates, they address the impracticality of proof objects, without imposing an increased trusted base: their size is considerably smaller, validating them is fast as the missing parts do not need to be reconstructed, yet an absolutely skeptical third party is indeed able to perform this reconstruction.

We have seen the conversion rule as an example of these simple yet tedious steps that are replaced by calls to verified VeriML functions. More generally, we refer to the mechanism of avoiding such steps as *small-scale automation*. The defining characteristics of small-scale automation are: it is used to automate *ubiquitous, yet simple* steps; therefore the VeriML functions that we can use should be relatively *inexpensive*; as our earlier discussion shows, the calls to small-scale automation tactics can be made invisible to the programmer, and therefore happen *implicitly*; and we expect, in the standard case, that small-scale automation functions are evaluated through proof-erasure semantics and therefore *do not leave a trace in the produced proof object*. We have also made the choice of evaluating small-

scale automation *statically*. In this way, after stage-one evaluation, small-scale automation has already been checked. Based on this view, our insight is that proof objects in λHOL_C should actually be seen as proof object expressions, albeit where a specific small-scale automation procedure is used, which is fixed a priori.

What about proof expressions? We have defined them earlier as VeriML expressions of propositional type and which therefore evaluate to a proof object. In these expressions we are free to use any VeriML proof-producing function, evaluated either statically or dynamically, through proof-erasure semantics or not. Indeed, proof object expressions are meant to be seen as a special case of proof expressions, where only static calls to (user-specified) small-scale automation functions are allowed and evaluated under proof-erasure semantics. The main distinction is that proof expressions can also use other, potentially expensive, automated proof-search functions. Using arbitrary proof expressions as proof certificates would thus not be desirable despite their compactness, as the receiving third party would have to perform lengthy computation. We refer to these extra automation mechanisms that can be used in proof expressions as *large-scale automation*.

The distinction between small- and large-scale automation in VeriML might seem arbitrary, but indeed it is deliberately so: the user should be free to decide what constitutes trivial detail, taking into account the complexity of the domain they are working on and of the automation functions they have developed. There is no special provision that needs to be taken in order to render a VeriML proof automation function part of small-scale automation, save perhaps for syntactic sugar to hide its uses. Typing is therefore of paramount importance in moving freely between the two kinds of automation: in the case of small-scale automation, the fact that calls to tactics are made implicitly, means that all the relevant information must be inferred through the type system instead of being explicitly specified by the user. It is exactly this inferrence that the VeriML type system permits.

Contrasting proof expressions with proof scripts reinforces this point. Proof scripts in traditional proof assistants are similar to VeriML proof expressions, where all the logic-related typing information is absent. [This is why we refer to VeriML proof expressions as *typed proof scripts* as well.] Since the required typing information is missing, the view of small-scale vs. large-scale automation as presented above is impossible in traditional proof

assistants. Another way to put this, is that individual proof steps in proof scripts depend on information that is not available statically, and therefore choosing to validate certain proof steps ahead of time (i.e. calls to small-scale automation tactics) is impossible.

7.2 User-extensible static checking for tactics

In the previous section we presented a novel treatment of the conversion rule which allows safe user extensions. Uses of the conversion rule are replaced with special user-defined, type-safe tactics, evaluated statically and under proof erasure semantics. This leads to a new notion of proof certificate, which we referred to as proof object expressions. The main question we will concern ourselves with in this section is what happens when the proof object expression is contained within a VeriML function and is therefore open, instead of being a closed expression: can we still check its validity statically?

Upon closer scrutiny, another way to pose the same question is the following: can we evaluate VeriML proof-producing function calls during stage-one evaluation? The reason this question is equivalent is that the conversion rule is implemented as nothing other than normal VeriML functions that produce proofs of a certain type.

We will motivate this question through a specific usage case where it is especially useful. When developing a new extension to the conversion rule, the programmer has to solve a number of proof obligations witnessing the claimed equivalences – remember the obligations denoted as G_i in Code Listings 7.1 and 7.2. Ideally, we would like to solve such proof obligations with minimal manual effort, through a call to a suitable proof-producing function. Yet we would like to know whether these obligations were successfully proved *at the definition time of the tactic*, i.e. statically – rather than having the function fail seemingly at random upon a specific invocation. The way to achieve this is to use static evaluation for the function calls.

A rewriter for plus. Let us consider the case of developing a rewriter –similar to rewriteBetaNatStep– for simplifying expressions of the form $x + y$ depending on the second argument. We will then register this rewriter with defEqual, so that such simplifications are

```
type rewriterT = (φ : ctx, T : Type, E : T) → (E' : T) × (E = E')

plusRewriter1 : rewriterT → rewriterT
plusRewriter1 recurse φ T E =
  holmatch E with
    X + Y ↦
      let ⟨ Y', H1 : Y = Y' ⟩ = recurse φ Y in
      let ⟨ E', H2 : X + Y' = E' ⟩ =
        holmatch Y' with
          zero    ↦ ⟨ X, G1 : X + zero = X ⟩
        | succ Y' ↦ ⟨ succ (X + Y'), G2 : X + succ Y' = succ (X + Y') ⟩
        | Y'      ↦ ⟨ X + Y', G3 : X + Y' = X + Y' ⟩
        in
        ⟨ E', G4 : X + Y = E' ⟩
    | E ↦ ⟨ E, G5 : E = E ⟩
```

Code Listing 7.3: VeriML rewriter that simplifies uses of the natural number addition function in logical terms

automatically taken into account as part of definitional equality.

The addition function is defined by induction on the first argument, as follows:

$$(+) = \lambda x.\lambda y.elimNat\ y\ (\lambda p.\lambda r.succ\ r)\ x$$

Based on this definition, it is evident that defEqual, that is, the definitional equality tactic, presented already in the previous section performs simplification based on the first argument, allowing the conversion rule to decide implicitly that $(succ\ x) + y = succ\ (x + y)$ or even that $(succ\ zero) + (x + y) = succ\ (x + y)$. With the extension we will describe here, similar equivalences such as $x + (succ\ y) = succ(x + y)$ will also be handled by definitional equality.

We give the code of this rewriter as the plusRewriter1 function in Code Listing 7.3. In order for rewriters to be able to use existing as well as future rewriters to perform their recursive calls, we write them in the open recursion style – they receive a function of the same type that corresponds to the "current" rewriter. As always, we have noted the types of proof hypotheses and obligations in the program text; it is understood that this information is not part of the actual text but can be reported to the user through interactive use of the

VeriML type checker. Contrary to our practice so far, in this code we have used only capital letters for the names of logical metavariables. This we do in order to have a clear distinction between normal logical variables, denoted through lowercase letters, and metavariables.

How do we fill in the missing obligations? For the interesting cases of $X + zero = X$ and $X + succ\ Y' = succ\ (X + Y')$, we would certainly need to prove the corresponding lemmas. But for the rest of the cases, the corresponding lemmas would be uninteresting and tedious to state and prove, such as the following for the $G_4 : X + Y = E'$ case:

$$\begin{aligned} lemma1 &: \forall x, y, y', e'.y = y' \rightarrow (x + y' = e') \rightarrow x + y = e' \\ &= \lambda x, y, y', e', H_a, H_b.conv\ H_b\ (subst\ (z.x + z = e')\ (symm\ H_a)) \\ G_4 &= lemma1\ X\ Y\ Y'\ E'\ H_1\ H_2 \end{aligned}$$

An essentially identical alternative would be to directly fill in the proof object for G_4 instead of stating the lemma:

$$G_4 = conv\ H_2\ (subst\ (z.X + z = E')\ (symm\ H_1))$$

Still, stating and proving such lemmas, or writing proof objects explicitly, soon becomes a hindrance when writing tactics. A better alternative is to use the congruence closure conversion rule to solve this trivial obligation for us directly at the point where it is required. Assuming that the current reqEqual function includes congruence closure as part of definitional equality, our first attempt to do this would be:

$$\begin{aligned} G_4 &: x + y = t' \equiv \\ &\mathsf{let}\ \Phi' = [\phi,\ h_1 : Y = Y',\ h_2 : X + Y' = E']\ \mathsf{in} \\ &\mathsf{let}\ \left\langle H : [\Phi']\, X + Y = E' \right\rangle = \mathsf{reqEqual}\ \Phi'\ (X + Y)\ E'\ \mathsf{in} \\ &\left\langle [\phi]\, H/[id_\phi,\ H_1/h_1,\ H_2/h_2] : [\phi]\, X + Y = E' \right\rangle \end{aligned}$$

The benefit of this approach is more evident when utilizing implicit arguments, since many details can be inferred and therefore omitted. Here we had to alter the environment passed to requireEqual, which includes several extra hypotheses. Once the resulting proof has been computed, the hypotheses are substituted by the actual proofs that we have.

Moving to static proof expressions. This is where using the letstatic construct becomes essential. We can evaluate the call to reqEqual statically, during stage one interpretation. Thus we will know at the time that plusRewriter1 is defined whether the call succeeded;

also, it will be replaced by a concrete value, so it will not affect the runtime behavior of each invocation of plusRewriter1 anymore. To do that, we need to avoid mentioning any of the metavariables that are bound during runtime, like X, Y, and E'. This is done by specifying an appropriate environment in the call to reqEqual, similarly to the way we incorporated the extra knowledge above and substituted it later. Using this approach, we have:

$$
\begin{aligned}
G_4 \;=\; & \mathsf{letstatic}\ \langle\ H\ \rangle\ = \\
& \quad \mathsf{let}\ \Phi'' \;= [x, y, y', e' : Nat, h_1 : y = y', h_2 : x + y' = e']\ \mathsf{in} \\
& \quad \mathsf{requireEqual}\ \Phi''\ (x + y)\ t' \\
& \mathsf{in}\ \langle\ [\phi]\, H/[X/id_\phi, Y/id_\phi, Y'/id_\phi, E'/id_\phi, H_1/id_\phi, H_2/id_\phi]\ \rangle
\end{aligned}
$$

What we are essentially doing here is replacing the meta-variables by normal logical variables, which our tactics can deal with. The meta-variable context is "collapsed" into a normal context; proofs are constructed using tactics in this environment; last, the resulting proofs are transported back into the desired context by substituting meta-variables for variables. We have explicitly stated the substitutions in order to distinguish between normal logical variables and meta-variables. In the actual program code, most of these details can be omitted.

The reason why this transformation needs to be done is that functions in our computational language can only manipulate logical terms that are open with respect to a normal variables context; not logical terms that are open with respect to the meta-variables context too. A much more complicated, but also more flexible alternative to using this "collapsing" trick would be to support meta^n-variables within our computational language directly.

It is worth noting that the transformation we describe here is essentially the collapsing transformation we have detailed in Section 5.2. Therefore this transformation is not always applicable. It is possible only under the conditions that we describe there – roughly, that all involved contextual terms must depend on the same variables context and use the identity substitution.

Overall, this approach is entirely similar to proving the auxiliary lemma *lemma1* mentioned above, prior to the tactic definition. The benefit is that by leveraging the type information together with type inference, we can avoid stating such lemmas explicitly, while retaining the same runtime behavior. We thus end up with very concise proof expressions to

```
natInduction: (φ : ctx, P : [φ] Nat → Prop) →
                (H_z : P 0, H_s : ∀n.P n → P (succ n)) → (∀n.P n)
instantiate : (φ : ctx, T : Type, P : [φ, x : T] Prop) →
                (H : ∀x.P, a : T) → (P/[a/x])
plusRewriter1 : rewriterT → rewriterT
plusRewriter1 recurse φ T E =
  holmatch E with
    X + Y ↦
      let ⟨ Y', H1 : Y = Y' ⟩ = recurse φ Y in
      let ⟨ E', H2 : X + Y' = E' ⟩ =
        holmatch Y' with
          zero    ↦ ⟨ X, {{instantiate (natInduction reqEqual reqEqual) X}}
                                                  : X + zero = X ⟩
        | succ Y' ↦ ⟨ succ (X + Y'),
                      {{instantiate (natInduction reqEqual reqEqual) X}}
                                                  : X + succ Y' = succ (X + Y') ⟩
        | Y'      ↦ ⟨ X + Y', {{reqEqual}} : X + Y' = X + Y' ⟩
      in
      ⟨ E', {{reqEqual}} : X + Y = E' ⟩
  | E ↦ ⟨ E, {{reqEqual}} : E = E ⟩
```

Code Listing 7.4: VeriML rewriter that simplifies uses of the natural number addition function in logical terms, with proof obligations filled-in

solve proof obligations within tactics, which are statically validated. We introduce syntactic sugar for binding the result of a static proof expression to a variable, and then performing a substitution to bring it into the current context, since this is a common operation.

$$\{\{e\}\} \equiv \mathsf{letstatic} \langle H \rangle = e \ \mathsf{in} \ \langle [\phi] H/\sigma \rangle$$

More details of how this construct works are presented in Subsection 8.3.1. For the time being, it will suffice to say that it effectively implements the collapsing transformation for the logical terms involved in e and also fills in the σ substitution to bring the collapsed proof object back to the required context.

Based on these, the trivial proofs in the above tactic can be filled in using a simple {{reqEqual}} call. The other two we actually need to prove by induction. We give the full code with the proof obligations filled in Code Listing 7.4; we use implicit arguments to omit

information that is easy to infer through typing and end up with very concise code.

After we define plusRewriter1, we can register it with the global equivalence checking procedure. Thus, all later calls to reqEqual will benefit from this simplification. It is then simple to prove commutativity for addition:

$$\begin{array}{lcl} \mathsf{plusComm} & : & (\forall x, y.x + y = y + x) \\ \mathsf{plusComm} & = & \mathsf{NatInduction\ reqEqual\ reqEqual} \end{array}$$

Based on this proof, we can write a naive rewriter that takes commutativity into account and uses the hash values of logical terms to avoid infinite loops; similarly we can handle associativity and distributivity with multiplication. Such rewriters can then be used for a better arithmetic simplification rewriter, where all the proofs are statically solved through the naive rewriters. The simplifier works by converting expressions into a list of monomials, sorting the list based on the hash values of the variables, and then factoring monomials on the same variable. Also, the eufEqual procedure mentioned earlier has all of its associated proofs automated through static proof expressions, using a naive, potentially non-terminating, equality rewriter. In this way, we can separate the 'proving' part of writing an extension to the conversion rule – by incorporating into a rewriter with a naive mechanism – from its 'programming' part – where we are free to use sophisticated data structures and not worry about the generated proof obligations. More details about these can be found in Chapter 9.

Formal statement. Let us return now to the question we asked at the very beginning of this section: can we statically check proof object expressions that are contained within VeriML functions – that is, that are open with respect to the extension variable context Ψ? The answer is perhaps obvious: yes, under the limitations about collapsable terms that we have already seen. We will now state this with more formal detail.

Let us go back to the structure of proof object expressions, as presented in Section 7.1. We have defined them as VeriML expressions of the form:

$$\mathsf{letstatic}\ G_1\ =\ [\![\mathsf{reqEqual}\ \Phi_1\ \mathit{Prop}\ P_1\ P_1']\!]_E\ \mathsf{in}$$

$$\ldots$$

$$\mathsf{letstatic}\ G_n\ =\ [\![\mathsf{reqEqual}\ \Phi_n\ \mathit{Prop}\ P_n\ P_n']\!]_E\ \mathsf{in}$$

$$\langle\ \pi_E\ \rangle$$

The main limitation to checking such expressions statically comes from having to evaluate the calls to reqEqual statically: if the logical terms $P_i,\ \ P_i'$ involve metavariables, their value will only be known at runtime. But as we have seen in this chapter, if these logical terms are collapsable, we are still able to evaluate such calls statically, after the transformation that we described. Therefore we are still able to check open proof object expressions statically, if we use the special $\{\{\cdot\}\}$ construct to evaluate the calls to reqEqual:

$$\mathsf{let}\ G_1\ =\ \{\{[\![\mathsf{reqEqual}\ \Phi_1\ \mathit{Prop}\ P_1\ P_1']\!]_E\}\}\ \mathsf{in}$$

$$\ldots$$

$$\mathsf{let}\ G_n\ =\ \{\{[\![\mathsf{reqEqual}\ \Phi_n\ \mathit{Prop}\ P_n\ P_n']\!]_E\}\}\ \mathsf{in}$$

$$\langle\ \pi_E\ \rangle$$

Based on this, we can extend the lemma 7.1.2 to open $\lambda\mathrm{HOL}_C$ proof objects as well.

Lemma 7.2.1 (Extension of Lemma 7.1.2) *(λHOL_C proof object – VeriML proof object expression equivalence)*

$$1.\ \frac{\Psi;\ \Phi \vdash_C \pi_C : P \qquad \mathit{collapsable}\,(\Psi;\ \Phi \vdash_C \pi_C : P) \qquad [\![\Psi;\ \Phi \vdash_C \pi_C : P]\!] = e_\pi}{\Psi;\ \bullet;\ \bullet \vdash e_\pi : \big([\Phi]\,P\big) \qquad e_\pi \longrightarrow_s^* e_\pi'}$$

$$2.\ \frac{\Psi;\ \bullet;\ \bullet \vdash e_\pi : \big([\Phi]\,P\big) \qquad e_\pi \longrightarrow_s^* e_{\pi'} \qquad \bullet \vdash \sigma_\Psi : \Psi}{e_\pi' \cdot \sigma_\Psi \longrightarrow^* \langle\ \pi_E\ \rangle \qquad \bullet;\ \Phi \cdot \sigma_\Psi \vdash_E \pi_E : P \cdot \sigma_\Psi}$$

Proof. Straightforward by the adaptation of the translation of $\lambda\mathrm{HOL}_C$ proof object to proof object expressions suggested above; the feasibility of this adaptation is proved directly through Theorem 5.2.4. The second part is proved directly by the form of the proof object expressions e_π (a series of let $\cdots$ in constructs), which based on the semantics of VeriML and its progress theorem always evaluate successfully. □

Let us describe this lemma in a little bit more detail. The first part of the lemma states that we can always translate an open (with respect to the extension variables context Ψ)

λHOL_C proof object to an open VeriML proof object expression of the same type. Also, that the expression is guaranteed to successfully evaluate statically into another expression, instead of throwing an exception or going into an infinite loop. Informally, we can say that the combination of typing and static evaluation yields the same checking behavior as typing for the original λHOL_C proof object. Formally, static evaluation results in an expression e'_π and not a value; it would not be possible otherwise, since its type contains extension variables that are only instantiated at runtime. Thus it would still be possible for the expression to fail to evaluate successfully at runtime, which would destroy the informal property we are claiming. This is why the second part is needed: under any possible instantiation of the extension variables, the runtime e'_π expression will evaluate successfully to a valid λHOL_E proof object.

Overall this lemma allows us one more way to look at VeriML with the staging, proof erasure and collapsing constructs we have described. First, we can view VeriML as a generic language design VeriML(X), where X is a typed object language. For example, though we have described VeriML(λHOL_E), an alternate version VeriML(λHOL_C) where λHOL_C is used as the logic is easy to imagine. The main VeriML constructs –dependent abstraction, dependent tuples and dependent pattern matching– allow us to manipulate the typed terms of X in a type-safe way. The lemma above suggests that the combination of staging, proof erasure and collapsing are the minimal constructs that *allow us to retain this genericity at the user level, despite the a priori choice of* λHOL_E.

What we mean by this is the following: the lemma above shows that we can embed λHOL_C proof objects within arbitrary VeriML(λHOL_E) expressions and still check them statically, even though λHOL_E does not include an explicit conversion rule at its design time. Furthermore, we have seen how further extensions to the conversion rule can be checked in entirely the same manner. Therefore, these constructs allow VeriML(λHOL_E) to effectively appear as VeriML(λHOL_{E+X}) to the user, where X is *the user-defined conversion rule*. Last, user-extensible static checking does not have to be limited to equivalences as we do in the conversion rule; any other property encodable within λHOL_E can be checked in a similar manner too. Thus the combination of VeriML(λHOL_E) with these three constructs allow the user to treat it as the language VeriML(λHOL_E+X) where X is a user-defined

set of properties that is statically checked in a user-defined manner. Put simply, VeriML can be viewed as a language that allows user-extensible static checking.

Of course, the above description is somewhat marred by the existence of the side-condition of collapsable logical terms and therefore does not hold in its full generality. It is a fact that we cannot statically prove that terms containing meta-variables are $\beta\mathbb{N}$-equivalent; this would be possible already in the VeriML(λHOL_C) version of the language. Lifting this side-condition requires further additions to VeriML, such as substitution variables and meta-meta-variables. The reason is that VeriML would need to manipulate λHOL_C terms that are open not only with respect to the Φ normal variables context, but also with respect to the Ψ metavariables context, in order to support type-safe manipulation of terms that include metavariables. For example, our definitional equality tactic would have roughly the following form, where we have only written the metavariables-matching case and include the context details we usually omit:

$$
\begin{array}{l}
\mathsf{defEqual}\ :\ (\psi : ctx^2,\ \phi : [\psi]\, ctx^1) \rightarrow \\
\qquad (T : [\psi]\,[\phi]\ \mathit{Type},\ t_1, t_2 : [\psi]\,[\phi]\, T) \rightarrow \\
\qquad \mathsf{option}\ \big([\psi]\,[\phi]\, t_1 = t_2\big) \\
\\
\mathsf{defEqual}\ \psi\ \phi\ T\ t_1\ t_2\ =\ \mathsf{holmatch}\ t_1,\ t_2\ \mathsf{with} \\
\quad (\phi' : [\psi]\, ctx^1,\ X : [\psi]\,[\phi']\, T',\ s, s' : [\psi]\, \phi' \rhd \phi).X/s,\ X/s' \ \mapsto \\
\quad\ \mathsf{do}\ \Big\langle\ H : [\psi]\, s = s'\ \Big\rangle \leftarrow\ \mathsf{defEqualSubst}\ \psi\ \phi'\ \phi\ s\ s' \\
\quad\ \mathsf{return}\ \langle\ \mathit{eqMeta}\ X\ H\ \rangle
\end{array}
$$

Though the principles presented in our technical development for VeriML could be used to provide an account for such extensions, we are not yet aware of a way to hide the considerable extra complexity from the user. Thus we have left further investigation of how to properly lift the collapsing-related limitation to future work.

7.3 Programming with dependently-typed data structures

In this section we will see a different application of the static checking of proof expressions that we presented in the previous section. We will show how the same approach can be used to simplify programming with dependently-typed data structures in the computational

part of VeriML, similar to programming in Haskell with GADTs [Peyton Jones et al., 2006]. Though this is a tangent to the original application area that we have in mind for VeriML, namely proof assistant-like scenarios, we will see that in fact the two areas are closely related. Still, our current VeriML implementation has not been tuned to this usage scenario, so most of this section should be perceived as a description of an interesting future direction.

First, let us start with a brief review of what dependently-typed programming is: the manipulation of data structures (or terms) whose type depends somehow on the *value* of other data structures – that is, programming with terms whose type *depends* on other terms. The standard concrete example of such a data structure is the type of lists of a specific length, usually called *vectors* in the literature. The type of a vector depends on a *type*, representing the type of its elements (similar to polymorphic lists); and on the *value* of a natural number as well, representing the length of the vector. Vector values are constrained through this type so that only vectors of specific length are allowed to have the corresponding type. For example, the following vector:

```
[ "hello", "world", "!" ]
```

has type vector string 3. We write that the *kind* of vectors is:

$$\mathsf{vector} \ : \ \star \rightarrow \mathit{Nat} \rightarrow \star$$

We remind the reader that "$\star$" stands for the kind of computational types. The definition of the vector type looks as follows:

$$\begin{array}{lll} \textbf{type}\ \mathsf{vector}\ (\alpha : \star)\ (n : \mathit{Nat}) & = & \mathsf{nil}\ ::\ \mathsf{vector}\ \alpha\ 0 \\ & & \mid\ \mathsf{cons}\ ::\ \alpha \rightarrow \mathsf{vector}\ \alpha\ n \rightarrow \mathsf{vector}\ \alpha\ (\mathit{succ}\ n) \end{array}$$

We see that the constructor for an empty vector imposes the constraint that its length should be zero; the case for prefixing a vector with a new element constrains the length of the resulting vector accordingly.

The dependently-typed vector datatype offers increased expressiveness in the type system compared to the simply-typed list datatype familiar from ML and Haskell. This becomes apparent when considering functions that manipulate such a datatype. By having the extra length argument, we can specify richer properties of such functions. For example, we might say that the head and tail functions on vectors require that the vector be non-empty; that

the map function returns a vector of the same length; and that append returns a vector with the number of elements of both arguments. Formally we would write these as follows:

$$
\begin{array}{lcl}
\mathsf{head} & : & \mathsf{vector}\ \alpha\ (\mathit{succ}\ n) \to \mathsf{vector}\ \alpha\ n \\
\mathsf{tail} & : & \mathsf{vector}\ \alpha\ (\mathit{succ}\ n) \to \mathsf{vector}\ \alpha\ n \\
\mathsf{map} & : & (\alpha \to \beta) \to \mathsf{vector}\ \alpha\ n \to \mathsf{vector}\ \beta\ n \\
\mathsf{append} & : & \mathsf{vector}\ \alpha\ n \to \mathsf{vector}\ \alpha\ m \to \mathsf{vector}\ \alpha\ (n + m)
\end{array}
$$

In the case of head, this allows us to omit the redundant case of the nil vector:

$$\mathsf{head}\ l\ = \mathsf{match}\ l\ \mathsf{with}\ \mathsf{cons\ hd\ tl}\ \mapsto\ \mathsf{hd}$$

The compiler can determine that the nil-case is indeed redundant, as vector $\alpha\ (n+1)$ could never be inhabited by an empty vector per definition, and therefore will not issue a warning about incomplete pattern matching. Furthermore, the richer types allow us to catch more errors statically. For example, consider the following erroneous version of map:

$$
\begin{array}{lcll}
\mathsf{map}\ f\ l & = & \mathsf{match}\ l\ \mathsf{with} & \\
 & & \mathsf{nil} & \mapsto\ \mathsf{nil} \\
 & & \mid\ \mathsf{cons\ hd\ tl} & \mapsto\ \mathsf{let}\ \mathsf{hd}' = f\ \mathsf{hd}\ \mathsf{in}\ \mathsf{map}\ f\ \mathsf{tl}
\end{array}
$$

In the second case, the user made a typo, forgetting to prefix the resulting list with hd'. Since the resulting list does not have the expected length, the compiler will issue an error when type-checking this function.

There are a number of languages supporting this style of programming. Some of the languages that pioneered this style are Cayenne [Augustsson, 1999], Dependent ML [Xi and Pfenning, 1999] and Epigram [McBride, 2005]. Modern dependently-typed languages include Haskell with the generalized abstract datatypes extension (GADTs) [Peyton Jones et al., 2006], Agda [Norell, 2007] and Idris [Brady, 2011]. Various logics based on Martin-Löf type theory such as CIC as supported by Coq [Barras et al., 2012] and NuPRL [Constable et al., 1986] support dependently-typed programming as well. Coq includes a surface language specifically for this style of programming, referred to as Russell [Sozeau, 2006].

VeriML does fall under the definition of dependently-typed programming: for example, in a dependent tuple $(P,\ \pi)$, the type of the proof object π depends on the (runtime) value of P. In this section we will show that it can indeed encode types such as vector exactly as presented above and the advantages it offers over such languages.

Behind the scenes. Let us now dig deeper into what is happening behind the scenes in the type checker of a compiler, when manipulating dependently-typed data structures such as vector. We will start with the elimination form, namely pattern matching. Consider the following piece of code:

$$\begin{array}{lcll} \mathsf{append}\ l_1\ l_2 & = & \mathsf{match}\ l_1\ \mathsf{with} & \\ & & \mathsf{nil} & \mapsto\ \cdots \\ & & \mathsf{cons}\ hd\ tl & \mapsto\ \cdots \end{array}$$

Just as in the simply-typed version of lists, in each branch we learn the top-most constructor used. But in the dependently-typed version, we learn more information: namely, that the length of the list (the dependent argument n) is zero in the first case; and that it can be written as $n = succ\ n'$ for suitable n' in the second case.

This information might be crucial for further typing decisions. Typing imposes additional constraints on the length of resulting lists, as the type of the example append functio does. To show that these constraints are indeed satisfied, the extra information coming from each pattern matching branch needs to be taken into account. Similar information might come from function calls yielding dependent data structures. These are more evident when we consider the full code of the append function:

$$\begin{array}{lcll} \mathsf{append} & : & \mathsf{vector}\ \alpha\ n \to \mathsf{vector}\ \alpha\ m \to \mathsf{vector}\ \alpha\ (n+m) & \\ \mathsf{append}\ l_1\ l_2 & = & \mathsf{match}\ l_1\ \mathsf{with} & \\ & & \quad \mathsf{nil} & \mapsto\ l_2 \\ & & \mid\ \mathsf{cons}\ hd\ tl & \mapsto\ \mathsf{let}\ tl' = \mathsf{append}\ tl\ l_2\ \mathsf{in} \\ & & & \quad \mathsf{cons}\ hd\ tl' \end{array}$$

Let's annotate this code with all the information coming from typing. We drop the polymorphic type for presentation purposes.

$$\begin{array}{l} \mathsf{append}\ (l_1 : \mathsf{vector}\ n)\ (l_2 : \mathsf{vector}\ m)\ = \\ \quad \mathsf{match}\ l_1\ \mathsf{with} \\ \qquad \mathsf{nil}\ \mathrm{where}\ n = 0\ \mapsto \\ \qquad\quad (l_2 : \mathsf{vector}\ m) \\ \qquad \mid\ \mathsf{cons}\ \mathrm{where}\ n = succ\ n'\ hd\ (tl : \mathsf{vector}\ n')\ \mapsto \\ \qquad\quad \mathsf{let}\ (tl' : \mathsf{vector}\ (n' + m)) = \mathsf{append}\ tl\ l_2\ \mathsf{in} \\ \qquad\quad (\mathsf{cons}\ hd\ tl' : \mathsf{vector}\ (succ\ (n' + m))) \end{array}$$

This information can directly be figured out through the type inference algorithm based on the already-available information about the dependent terms, using adaptations of unification-based mechanisms such as Algorithm W [Damas and Milner, 1982]. Yet there is one piece missing from deeming the above code well-typed: we must ensure that the type of the returned lists in both branches indeed satisfies the desired type, as specified in the type signature of append. Namely, in the first branch we need to reconcile the type vector m with the expected type vector $(n + m)$; similarly for the second branch, vector $(succ\ (n' + m))$ needs to be reconciled with vector $(n + m)$. What the type checker essentially needs to do is *prove* that the two types are equal. It needs to make sure that $m = n + m$ and $succ\ (n' + m) = n + m$ taking into account the extra available information about n in both cases.

Though these two cases might seem straightforward conclusions of the available information, it is important to realize that the proof obligations that might be generated in such dependently-typed programs are arbitrarily complex. It is important for users to be able to specify their own functions such as + and predicates other than the implicit equality constraints shown above, in order to be able to capture precisely the properties that they want the dependent data structures to have. Taking these into account, it is evident that proving the generated proof obligations involves proving theorems about functions such as + and predicates such as less-than. Thus discharging the proof obligations automatically is undecidable in the general case, if we allow a rich enough set of properties to be captured at the type level.

In summary, we might say that *positive occurrences* of dependent terms, such as in the body of a function or of a let declaration, are associated with *proof obligations*; *negative occurrences*, such as in a pattern matching branch or in the binding part of a let declaration, yield *proof hypotheses*; and that *type-checking* of dependent terms is intermixed with *theorem proving* to discharge the generated proof obligations[3]. The astute reader familiar with ML-style languages might note that such a view is even true for polymorphic types in the

3. This summary assumes let-normal form so that hypotheses are not missed: for example, writing cons hd (append $tl\ l_2$) in the second branch of the append function would make us lose the information about the length of the returned list. Transforming this into let-normal form allows us to capture the extra information.

```
append : (n : Nat, l1 : vector n, m : Nat, l2 : vector m) →
           (r : Nat) × vector r × (r = n + m)
append n (l1 : vector n) m (l2 : vector m) =
  match l1 with
        nil(H1 : n = 0) ↦
          (m, l2, G1 : m = n + m)
    |   cons(n', hd, tl : vector n', H2 : n = succ n') ↦
          let (r', tl' : vector r', H3 : r' = n' + m) = append tl l2 in
          (n, cons hd tl', G2 : succ r' = n + m)
```

Code Listing 7.5: Version of vector append with explicit proofs

presence of existential types (e.g. through the ML module system). Yet in that case we are only dealing with syntactic equality, rendering a decidable decision procedure for the theorem proving part feasible.

Explicit version. Based on the above discussion, we can provide an intermediate representation of dependently-typed programs where the proof-related parts are explicitly evident. For example, the implicit equality constraints in the type of vector's constructors and in the type of append can be viewed instead as requiring explicit proof terms for the equalities. Such a representation was first presented by Xi et al. [2003] in order to reconcile the GADT-style definitions with ML-style definitions of data types: instead of explicitly assigning a type to each constructor, each constructor can be presented as a list of components, some of which are explicit equality constraints. Still, recent work [e.g. Goguen et al., 2006, Sulzmann et al., 2007, Schrijvers et al., 2009, Vytiniotis et al., 2012] suggests that this representation is more fundamental; the current version of the GHC compiler for Haskell actually uses the explicit version and even keeps the equality proofs in subsequent intermediate languages as it is useful for compilation purposes.

Based on the above, we give the explicit version of the definition of vector becomes:

$$\begin{aligned}\textbf{type}\ \mathsf{vector}\ \alpha\ n\ =\ & \quad \mathsf{nil} \quad \textbf{of}\ \big(n = 0\big) \\ & \mid\ \mathsf{cons} \quad \textbf{of}\ (n' : \mathit{Nat}) \times \alpha \times \mathsf{vector}\ \alpha\ n' \times \big(n = \mathit{succ}\ n'\big)\end{aligned}$$

Similarly, the explicit version of append assigns names to hypotheses and notes the types of obligations that need to be proved; we present it in Code Listing 7.5. It is quite similar

to our fully-annotated version from above. In this code fragment we have left the proof obligations G_1 and G_2 unspecified; we have only given their types. In the real version of the code they would need to be replaced by sufficient evidence to show that the proof obligations indeed hold.

We have on purpose used syntax close to the one we use in VeriML. In fact, the above two code fragments can be directly programmed within VeriML, showing that VeriML can be used for the style of dependently-typed programming supported by languages such as Haskell at its computational level. Domain objects of λHOL such as natural numbers can be used as dependent arguments (also called *dependent indexes*); λHOL propositions can be used to ascribe constraints to such arguments; and λHOL proof objects can be used as evidence for the discharging of proof obligations. All of the involved λHOL terms are *closed* and the contextual terms support is not required for this style of programming. Note that while the dependent indexes are at the level of the logical language, dependent types such as vector are part of the ML-style computational level of VeriML. The full expressiveness of λHOL is available both for defining the types of dependent indexes and for defining their required properties.

To show that the above encoding of the dependent vector datatype does not need any new features in VeriML, let us give its fully explicit version with no syntactic sugar, following the language definition in Chapter 6:

$$\begin{aligned}\mathsf{vector} \;=\; & \lambda\alpha : \star.\mu t : (\square\, Nat) \to \star.\lambda N : \square\, Nat. \\ & \big(\square\, N = zero\big) + ((N' : \square\, Nat) \times \alpha \times t\ N' \times \big(\square\, N = succ\ N'\big))\end{aligned}$$

One question remains: how do we go from the simply-typed presentation of dependent programs as shown above to the explicit version presented here? By comparison of the two versions of the code of append we identify two primary difficulties, both related to positive occurrences of dependent terms: first, we need to 'invent' instantiations for the indexes – e.g. for the length of the vectors returned; second, we need to discharge the proof obligations. As suggested above, the instantiations for the indexes can be handled through type inferrence using a unification-based algorithm, assuming that sufficient typing annotations are available from the context; though limiting the number of required annotations is an interesting research problem in itself [e.g. Peyton Jones et al., 2006], we will not be

further concerned with this. The latter problem, discharging proof obligations such as G_1 and G_2 in append, amounts to general theorem proving as discussed above and is therefore undecidable. We will now focus on approaches to handling this problem.

Options for discharging obligations. The first option we have with respect to discharging obligations is to restrict their form, so that the problem becomes decidable. For example, Dependent ML restricts indices to be natural numbers and the obligations to be linear equalities. The type-checker might then employ a decision procedure for deciding the validity of all the obligations. If the decision procedure is proof-producing, such calls are viewed as part of elaboration of the surface-level program into the explicit version, where obligations such as G_1 and G_2 are actually filled in with the proof returned through the decision procedure. The benefit of this choice is that it requires no special user input in order to discharge obligations; the obvious downside is that the expressibility of the dependent types is severely limited. If we take the view that dependently-typed programming is really about being able to define data structures with user-defined invariants and functions with user-defined pre- and post-conditions, it is absolutely critical that we have an expressive logic in order to describe them; this is something that this option entirely precludes.

A second option is to have the user give full, explicit evidence that the proof obligations hold by providing a suitable proof object. This is the other end of the spectrum: it offers no automation with respect to discharging obligations, yet it offers maximum extensibility – programs with arbitrarily complex proof obligations can be deemed well-typed, as long as proofs exist (and the user can somehow discover it). The proof objects can be rather large even for simple proof obligations, therefore this choice is clearly not suitable for a surface language. The maximum expressivity and the ease of type-checking makes this choice well-suited for compiler intermediate languages; indeed, GHC uses this form as mentioned above.

In practice, most dependently-typed languages offer a mix between these two choices, where some obligations are solved directly through some built-in decision procedures and others require user input, in the form of explicit proofs or auxiliary lemmas. For example, the type class resolution mechanism offered by Haskell can be viewed as a decision procedure

for Prolog-style obligations (first-order minimal logic with uninterpreted functions); the conversion rule in languages such as CIC or Agda, as noted in Section 7.1, can be viewed as a decision procedure for $\beta\iota$-equality; languages with an extended conversion rule such as CoqMT also include decision procedures for arithmetic and other first-order theories; and last, dependent pattern matching as supported by languages such as Agda or Idris can be viewed as providing a further decision procedure over heterogeneous equality. All these languages also provide ways to give an explicit proof object in order to discharge specific proof obligations arising in dependently-typed programs. Still, the automation offered in these languages falls short of user expectations in many cases: for example, while the above version of `append` would be automatically found to be well-typed in all these languages, the version where the arguments are flipped is only well-typed in CoqMT and requires explicit proof in others. This fact is especially true as the properties specified for dependent indices become more complex.

Yet another choice is to use the full power of a proof assistant in order to discharge obligations. This choice is primarily offered by the Russell language, which is a surface language in Coq specifically for writing dependently-typed programs. It allows the user to write the simply-typed version of such programs, as we did before for `append`; during elaboration, the generated proof obligations are presented to the user as goals; after the user has solved all the goals, the fully explicit dependently-typed version is emitted. Obligations are first attempted to be solved through a default automation (semi-)decision procedure. They are only presented to the user if the decision procedure fails to solve them, in which case the user can develop a suitable proof script in the interactive style offered by Coq. We consider this approach to be the current state-of-the-art: it offers a considerable amount of automation; it is maximally expressive, as the user is always free to provide a proof script; it separates the programming part clearly from the proof-related part; and, most importantly, the amount of automation offered is *user-extensible*. This last point is true because the user can specify their own more sophisticated automation decision procedure to be used as the 'default' one.

Our main point in this section is that VeriML offers this same choice, yet with a significant improvement: proof obligations can be discharged through statically-evaluated calls

to decision procedures; but *the decision procedures themselves can be programmed within the same language*. First, we can use the same mechanism used in the previous section for statically discharging obligations. Consider the following version of append within VeriML:

$$
\begin{array}{l}
\mathsf{append}\ :\ (n : Nat, l_1 : \mathsf{vector}\ n, m : Nat, l_2 : \mathsf{vector}\ m) \rightarrow \\
\qquad (r : Nat) \times \mathsf{vector}\ r \times (r = n + m) \\
\mathsf{append}\ n\ (l_1 : \mathsf{vector}\ n)\ m\ (l_2 : \mathsf{vector}\ m)\ = \\
\quad \mathsf{match}\ l_1\ \mathsf{with} \\
\qquad \mathsf{nil}(H_1 : n = 0)\ \mapsto \\
\qquad\quad (m,\ l_2,\ \{\{\mathsf{Auto}\}\} : (m = n + m))\ \\
\qquad |\ \ \mathsf{cons}(n',\ hd,\ tl : \mathsf{vector}\ n',\ H_2 : n = succ\ n')\ \mapsto \\
\qquad\quad \mathsf{let}\ (r',\ tl' : \mathsf{vector}\ r',\ H_3 : r' = n' + m) = \mathsf{append}\ tl\ l_2\ \mathsf{in} \\
\qquad\quad (n,\ \mathsf{cons}\ hd\ tl',\ \{\{\mathsf{Auto}\}\} : \big(succ\ r' = n + m\big))
\end{array}
$$

Obligations are discharged through static calls to the Auto function of the type

$$\mathsf{Auto}\ :\ (\phi : ctx,\ P : Prop) \rightarrow (P)$$

where both ϕ and P are implicit arguments. The collapsing transformation used in the previous section turns the extra proof hypotheses H_1 through H_3 into elements of the ϕ context that Auto is called with. Thus the static calls to the function can take this information into account, even though the actual proofs of these hypotheses will only be available at runtime. Since discharging happens during stage-one evaluation, we will know statically, at the definition time of the append function whether the obligations are successfully discharged or not.

The departure from the Russell approach is subtle: the Auto function itself is written within the same programming language; whereas the tactics used in Russell are written through other languages provided by Coq. In fact, Auto is a dependently-typed function in itself. Therefore, some of its proof obligations can be solved statically and automatically through another function and so on. Another way to view this is that *proof obligations generated during VeriML type-checking are discharged through proof functions that are written in VeriML themselves*. Using one conversion rule to help us solve the obligations of another, as shown in the previous section, is one example where this idea is put to practice.

VeriML does not yet offer a surface language for dependently-typed programming similar to Russell, so that surface programs can be written in a simply-typed style. Thus for the time being developing such programs is tedious. Still, such an extension is conceptually simple and could be provided through typed syntactic sugar. As suggested above, positive occurrences of dependent data types can be replaced by suitable dependent tuples: indices, such as the length of the list in the vector example, are left as implicit arguments to be inferred by type checking; and proof obligations are discharged through a static call to a default automation tactic. Negative occurrences are replaced by fully unpacking the dependent tuples and assigning unique names to their components, so that they can be used in the rest of the program. Furthermore, the user can be asked to explicitly provide a proof expression if one of the static tactic calls fail. Based on this, type-checking a dependently-typed program has a fixed part (using type inference to infer missing indices) and a user-extensible part (using the automation tactic statically). They respectively correspond to fixed type-checking and user-extensible static evaluation, resembling the case of the extensible conversion rule.

Overall the VeriML approach to dependently-typed programming described here leads to a form of *extensible type checking*: dependent types offer users a way to specify rich properties about data structures and functions; and the VeriML constructs offer a way to specify how to statically check whether these properties hold using a rich programming model. We believe that exploiting this possibility is a promising future research direction.

Further considerations.

We will now draw attention to further issues that have to do with dependently-typed programming in VeriML and other languages.

Phase distinction. Throughout the above discussion we have made a silent assumption: the notion of indices that data types can depend on is distinct from arbitrary terms of the computational language; furthermore, some logic for describing such properties is available. Still, our initial description of dependent types suggests that a type can depend on an arbitrary term. This style of full dependent types poses a number of problems:

e.g. what happens if the term contains a non-terminating function application, or a side-effecting operation? Non-termination would directly render type checking undecidable, as the compiler could be stuck for arbitrarily long trying to prove two terms equal. Side-effects would present other difficulties; for example, type checking would have to proceed in a well-specified order so that the threading order of side-effects is known to the user. Therefore all dependently-typed languages support restricted forms of dependent types. The only exception is Cayenne, one of the earliest proposals of dependently-typed languages, which was plagued by the problems suggested above. Languages such as Dependent ML and Haskell with GADTs choose to only allow dependency on a separate class of terms – the *indexes* as used throughout our discussion. Languages that can be used as type-theoretic logics (Epigram, Agda, Idris, Coq etc.) offer an infinite, stratified tower of universes, where values in one universe are typed through values in the next universe and dependency is only allowed from higher universes into lower ones. They also restrict computation to be pure.

The main idea behind dependent data types in languages such as Dependent ML and Haskell with GADTs is to allow other *kinds* of static data other than types; it is exactly these static data that are referred to as indexes. The natural number n representing the length of a vector in the example given above is such an index; its classifier *Nat* is now viewed as a kind instead of as a type of a runtime value. Indexes exist purely for type-checking purposes and can be erased prior to runtime. We thus understand that this style of dependently-typed programming preserves the phase distinction property: indices, together with the proofs about them, do not influence the runtime behavior of dependently-typed programs and can thus be erased prior to runtime. This is the same property that holds for polymorphic type abstraction and instantiation in System F: both forms can be erased.

In languages with a tower of universes we might say that a generalization of this property holds: evaluating a term of universe n is independent from all terms of higher universes contained in it; therefore these can be erased prior to evaluation. This is the intuition behind including erasure in definitional equality (as done, for example, in [Miquel, 2001]); doing program extraction from CIC terms [Paulin-Mohring, 1989, Paulin-Mohring and Werner, 1993]; and is also of central importance in the optimizations that Idris performs to yield efficient executable code from dependently-typed programs [Brady et al., 2004, Brady, 2005].

VeriML does not support such a phase distinction, as the indices are λHOL terms which might be pattern matched against and therefore influence runtime. Phase distinction holds for the proof objects used to discharge proof obligations though; this, in fact, is another way to describe the proof erasure property of VeriML. Yet, as the obligations are discharged during static evaluation, the indices are not actually pattern matched against: indices are metavariables, whose instantiations are only known at runtime, but the actual proof obligations that we discharge are collapsed versions, and so the metavariables are not part of them. The indices are only used to bring the proofs resulting from static evaluation into the appropriate context. Since the proofs themselves can be erased, it makes sense to also allow the indices to be erased.

This is possible by adding special support to the type system, in order to mark certain λHOL terms as erasable. The type system then checks that those terms are not pattern matched upon, making sure that it is safe to erase them prior to runtime. It is interesting to note that this special typing support can take the form of marking dependent λHOL abstractions or tuples as *static* where the associated λHOL terms are only allowed to be used during stage-one evaluation. For example, the type of append could be:

$$\mathsf{append}\ :$$

$$(n :^s Nat,\ m :^s Nat) \rightarrow \mathsf{vector}\ n \rightarrow \mathsf{vector}\ m \rightarrow (r :^s Nat) \times \mathsf{vector}\ r \times (r = n + m)^s$$

This reveals a relationship betwen staged programming and dependently-typed programming (as suggested for example in Sheard [2004]) which warrants further investigation.

Handling impossible branches. We mentioned earlier that the additional typing information might allow us to omit certain pattern matching cases, yet still cover all possible ones. For example, the code of the head function for lists is:

$$\mathsf{head}\ :\ \mathsf{vector}\ (succ\ n) \rightarrow \mathsf{vector}\ n$$

$$\mathsf{head}\ l\ =\ \mathsf{match}\ l\ \mathsf{with}\ \mathsf{cons}\ hd\ tl\ \mapsto\ hd$$

The compiler can indeed discover that the omitted nil branch is impossible, as it would mean that the false proposition $succ\ n = 0$ is provable based on the available typing information. Thus this check employs theorem proving as well, requiring us to show that the omitted branches lead to contradictions.

We need an additional construct in order to support such impossible branches in VeriML. Its typing rule is as follows:

$$\frac{\Psi \vdash t : \left([]\ \mathit{False}\right)}{\Psi;\ \mathcal{M};\ \Gamma \vdash \mathsf{absurd}\ t : \tau}$$

This construct turns a closed proof of the contradiction into a value of any type. It can be used in impossible branches in order to return a term of the required type. The required proof can be handled as above, through static proof expressions:

$$\begin{aligned} \mathsf{head}\ l\ =\ \mathsf{match}\ l\ \mathsf{with}\ \ & \mathsf{cons}\ hd\ tl\ \mapsto\ hd \\ & \mid \mathsf{nil}\ \mapsto\ \mathsf{absurd}\ \{\{\mathsf{Auto}\}\} \end{aligned}$$

The semantics of the construct are interesting: no operational semantics rule is added at all for the new construct! In order for the progress theorem to continue to hold we must make sure that a closed $\mathsf{absurd}\ t$ expression will never be encountered. Equivalently, we must make sure that no closed λHOL proof object for the proposition *False* exists. This is exactly the statement of soundness for λHOL. We therefore understand that the addition of the absurd construct requires soundness of λHOL to hold; in fact, it is the only construct we have presented that requires so.

Chapter 8

Prototype implementation

We have so far presented the VeriML language design from a formal standpoint. In this chapter we will present how VeriML can be practically implemented. Towards that effect I have completed a prototype implementation of VeriML that includes all the features we have seen so far as well as a number of other features that are practically essential, such as a type inference mechanism. The prototype is programmed in the OCaml programming language [Leroy et al., 2010] and consists of about 10k lines of code. It is freely available from `http://www.cs.yale.edu/homes/stampoulis/`.

8.1 Overview

The VeriML implementation can be understood as a series of components that roughly correspond to the phases that an input VeriML expression goes through: parsing; syntactic elaboration; type inferencing; type checking; and evaluation. We give an example of these phases in Figure 8.1. Parsing turns the input text of a VeriML expression into a *concrete syntax tree*; syntactic elaboration performs syntax-level transformations such as expansion of syntactic sugar and annotation of named variables with their corresponding hybrid deBruijn variables, yielding an *abstract syntax tree*. Type inferencing attempts to fill in missing parts of VeriML expressions such as implicit arguments and type annotations in constructs that need them (e.g. $\langle\, T,\ e \,\rangle$ and holmatch). Type checking then validates the completed expressions according to the VeriML typing rules. This is a redundant phase

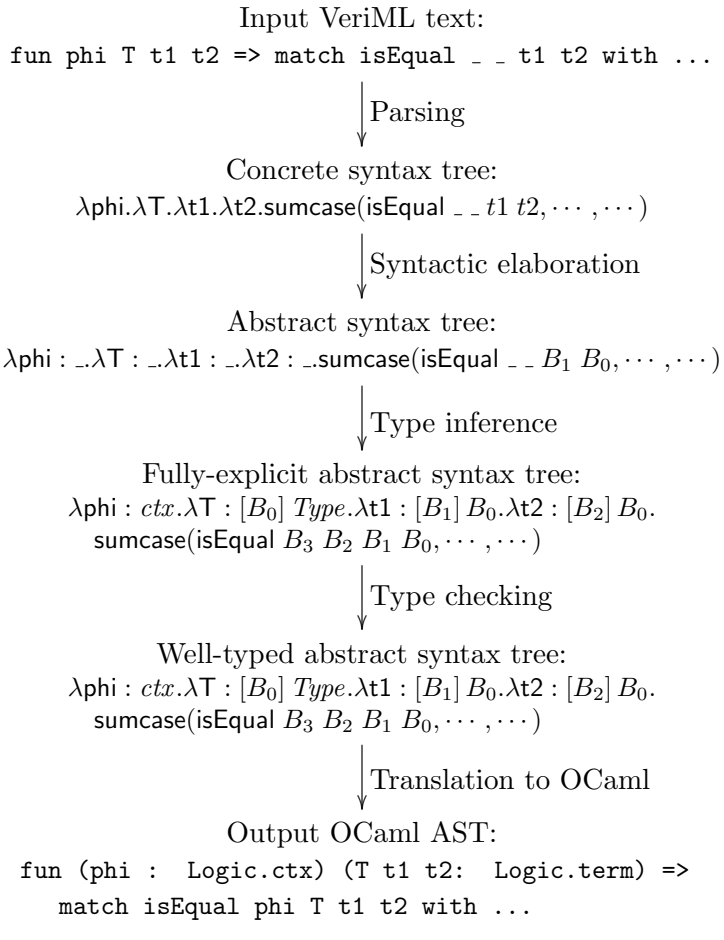

Figure 8.1: Components of the VeriML implementation

as type inferencing works according to the same typing rules; yet it ensures that the more complicated type inferencing mechanism did not introduce or admit any ill-typed terms.

The last phase is evaluation of VeriML expressions based on the operational semantics. We have implemented two separate evaluation mechanisms for VeriML: an interpreter, written as a recursive OCaml function (from VeriML ASTs to VeriML ASTs); and a translator, which translates well-typed VeriML expressions to OCaml expressions (a function from VeriML ASTs to OCaml ASTs). The resulting OCaml expressions can then be treated as normal OCaml programs and compiled using the normal OCaml compiler. We will only cover the latter approach as we have found evaluation through translation to OCaml to be at least one order of magnitude more efficient than direct interpretation.

Our prototype also includes an implementation of the λHOL logic. Since terms of the

λHOL logic are part of the VeriML expressions, the above computational components need to make calls to the corresponding logical components – for example, the VeriML expression parser calls the λHOL parser to parse logical terms. Our implementation of the λHOL logic is thus structured into components that perform parsing, syntactic elaboration, type inferencing and type checking for logical terms, following the structure of the computational language implementation. Last, the translation of logical terms into OCaml ASTs is done by reusing the λHOL implementation. The resulting OCaml code thus needs to be linked with (part of) the λHOL implementation.

In the rest, we will describe each component in more detail and also present further features of the prototype.

8.2 Logic implementation

The core of our implementation of λHOL that includes parsing, syntactic elaboration and typing consists of about 1.5k lines of code. It follows the presentation of the logic given in Section 4.2. The relevant files in our prototype are:

`syntax_trans_logic.ml`	Parsing for λHOL terms
`logic_cst.ml`	Concrete syntax trees; syntactic elaboration (CST to AST conversion)
`logic_ast.ml`	Abstract syntax trees and syntactic operations
`logic_core.ml`	Core auxiliary logic operations
`logic_typing.ml`	λHOL type checking

Concrete syntax. Parsing is implemented through Camlp4 grammar extensions; we present examples of the concrete syntax in Table 8.1. Our parser emits *concrete syntax trees*: these are values of a datatype that corresponds to λHOL terms at a level close to the user input. For example, concrete syntax trees use named variables, representing the term $\lambda x : Nat.x$ as `LLam("x", Var("Nat"), Var("x"))`. Also they allow a number of syntactic conveniences, such as being able to refer to a metavariable V without provid-

Concrete syntax	**Abstract syntax**
`fun x : t => t'`	$\lambda(t).t'$
`forall x : t, t'`	$\Pi(t).t'$
`t -> t'`	$\Pi(t).t'$
`t1 t2`	$t1\ t2$
`t1 = t2`	$t1 = t2$
`x`	v_L or b_i or X/σ or c/σ
`x/[σ]`	X/σ or c/σ
`id_phi, t1, t2, ..., tn`	$id_\phi,\ t1,\ t2,\ \cdots,\ t_n$
`[phi, x : Nat].x = x`	$[\phi,\ Nat]\, v_{\|\phi\|} = v_{\|\phi\|}$
`@x = x`	$[\phi,\ Nat]\, v_{\|\phi\|} = v_{\|\phi\|}$ when $@ = \phi,\ Nat$

Table 8.1: Concrete syntax for λHOL terms

ing an explicit substitution σ; referring to both normal variables v and meta-variables V without distinguishing between the two; and also maintaining and referring to an "ambient context" denoted as @ in order to simplify writing contextual terms – e.g. in order to write $[\phi,\ x : Nat]\, x = x$ as $@x = x$ when the ambient context has already been established as $\phi,\ x : Nat$. We will see more details of this latter feature in Subsection 8.3.1. We convert concrete syntax trees to *abstract syntax trees* which follow the λHOL grammar given in Section 4.2. The conversion keeps track of the available named variables and metavariables at each point and chooses the right kind of variable (free or bound normal variable, free or bound metavariable or constant variable) based on this information. We chose to have concrete syntax trees as an intermediate representation between parsing and abstract syntax, so that managing the information about the variables does not conflate the parsing procedure.

Abstract syntax. Abstract syntax trees as defined in our implementation follow closely the grammar of λHOL given in Section 4.2. We depart from that grammar in two ways: first, by allowing *holes* or *inferred terms* – placeholders for terms to be filled in by type inference, described in detail in Subsection 8.2.1; and second, by supporting constant schemata c/σ instead of simple constants c following our presentation in Section 4.3.

Syntactic operations. The syntactic operations presented in Section 4.2, such as freshening $\lceil\cdot\rceil$, binding $\lfloor\cdot\rfloor$ and substitution application $\cdot\cdot\sigma$ are part of the logic implementation,

as functions that operate on abstract syntax trees. These operations are defined over various datatypes – terms, substitutions, contexts, etc. Furthermore, other kinds of variables such as metavariables, context variables and even computational variables are represented in the implementation through hybrid deBruijn variables as well, as this representation makes for code that is easier to work with than deBruijn indices. In order to reuse the code of these basic operations, we have implemented them as higher-order functions. They expect a traversal function over the specific datatype that they work on which keeps track of the free and bound variables at each subterm. These generic operations are defined in the `binding.ml` file of the prototype implementation.

Type checking. The typing rules for λHOL are all *syntax-directed*: every λHOL term constructor corresponds exactly to a single typing rule. Thus we do not need a separate formulation of algorithmic typing rules; the typing rules given already suggest a simple type checking algorithm for λHOL. This simplicity is indeed reflected in our implementation: the main procedure that implements type checking of λHOL abstract syntax trees is just under 300 lines of OCaml code. As a side-effect, the type checker fills in the typing annotations that pattern matching requires and thus yields annotated λHOL terms, as presented in Section 5.1. For example, the Π-type former is annoted with the sort of the domain, yielding terms of the form $\Pi_s(t).t'$.

An important function that type checking relies on is `lterm_equal` which compares two logical terms up to definitional equality, which in the case of λHOL is simply syntactic equality. It is used often during type checking to check whether a type matches an expected one – e.g. we use it in the implementation of ΠELIM rule for $t_1\ t_2$, where $t_1 : \Pi(t).t'$ in order to check whether the type of t_2 matches the domain of the function type t. This equality checking function is one of the main parts of the implementation that would require changes in order to support a fixed conversion rule in the logic, yielding λHOL_C; as mentioned in Section 7.1, these changes become instead part of the standard rewriter and equality checker programmed in VeriML and do not need to be trusted.

8.2.1 Type inference for logical terms

One of the important additions in our implementation of λHOL compared to its formal description is allowing terms that include *holes* or *inference variables* – missing subterms that are filled in based on information from the context. This is an essential practical feature for writing logical terms, as the typed nature of λHOL results in many redundant occurrences of simple-to-infer terms. For example, the constructor for logical conjunction has the type:

$$andI : \forall P, Q : Prop, P \to Q \to P \wedge Q$$

The two first arguments are redundant, since the type of the second two arguments (the proof objects proving P and Q) uniquely determine what the propositions P and Q are. Given proofs π_1 and π_2, we can use inference variables denoted as ? to use the above constructor:

$$andI\ ?\ ?\ \pi_1\ \pi_2$$

Another example is using the elimination principle for natural numbers of type:

$$elimNat : [T : Type]\, T \to (Nat \to T \to T) \to Nat \to T$$

We can use this to write the addition function, as follows:

$$plus = \lambda x :?.\lambda y :?.elimNat/[?]\ y\ (\lambda p :?.\lambda r :?.succ\ r)\ x$$

or more succinctly, using syntactic sugar for functions and use of constants, as:

$$plus = \lambda x.\lambda y.elimNat\ y\ (\lambda p.\lambda r.succ\ r)\ x$$

Intuitively, we can view inference variables as unknowns for whom typing generates a set of constraints; we instantiate the inference variables so that the constraints are satisfied, if possible. In the case of λHOL, the constraints are requirements that the inference variables be syntactically equal to some other terms. For example consider the following typing derivation:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Psi;\ \Phi,\ x :?_1 \vdash t_1 : t \to t'} \qquad \dfrac{(\Phi,\ ?_1).x =?_1}{\Psi;\ \Phi,\ x :?_1 \vdash x : t}\ \textsc{Var}}{\Psi;\ \Phi,\ x :?_1 \vdash t_1\ x : t'}\ \Pi\textsc{Elim}}{\Psi;\ \Phi \vdash (\lambda x :?_1.t_1\ x) :?_1 \to t'}\ \Pi\textsc{Intro}$$

From the application of the VAR typing rule the constraint $?_1 \equiv t$ follows directly. More complicated constraints are also common, if we take into account the fact that our typing rules work involve operations such as freshening, binding and also application of substitutions. For example, in the following typing derivation we use a non-identity substitution with a metavariable:

$$\dfrac{\cdots \qquad \dfrac{(H : [\Phi']\,?_1) \in \Psi}{\Psi;\ \Phi \vdash H/[X,Y] :?_1 \cdot (X/a,\ Y/b)}\ \textsc{MetaVar}}{\Psi;\ f : (X = Y) \to (Y = X) \vdash f\ H/[X,Y] : Y = X}\ \Pi\textsc{Elim}$$

where

$$\Psi = \phi : ctx,\ X : [\phi]\,Nat,\ Y : [\phi]\,Nat,\ H : [a : Nat,\ b : Nat]\,?_1$$

resulting in the following constraint:

$$?_1 \cdot (X/a,\ Y/b) \equiv (X = Y)$$

which can be solved for

$$?_1 \equiv (a = b)$$

These equality constraints are not explicit in our formulation of the typing rules, yet an alternative presentation of the same typing rules with explicit constraints is possible, following the approach of Pottier and Rémy [2005]. Intuitively, the main idea is to replace cases where a type of a specific form is expected either in the premises or consequences of typing rules, with an appropriate constraint. The constraints might introduce further inference variables. For example, the typing rule ΠELIM can be formulated instead as:

$$\dfrac{\Psi;\ \Phi \vdash t_1 :?_1 \qquad \Psi;\ \Phi \vdash t_2 :?_2 \qquad ?_1 \equiv \Pi(?_3).?_4 \qquad ?_2 \equiv ?_3 \qquad ?_r \equiv \lceil ?_4 \rceil \cdot (id_\Phi,\ t_2)}{\Psi;\ \Phi \vdash t_1\ t_2 :?_r}\ \Pi\textsc{Elim-Infer}$$

We can view this as replacing the variables we use at the meta-level (the variables that we implicitly quantify over, used for writing down the typing rules as mathematical definitions), with the concrete notion of inference variables. We have implemented a modified type checking algorithm in the file `logic_typinf.ml` which works using rules of this form.

The presence of explicit substitutions as well as the fact that the constraints might equate inference variables with terms that contain further inference variables renders the

problem of solving these constraints equivalent to higher-order unification. Thus, solving the constraint set generated by typing derivations in the general case is undecidable. Our implementation does not handle the general case, aiming to solve the most frequent cases efficiently. Towards that effect, it solves constraints *eagerly*, as soon as they are generated, rather than running typing to completion and having a post-hoc phase of constraint solving.

Implementation details. The key point of our implementation that enables support for inference variables and the type inferencing algorithm is having the procedure `lterm_equal` that checks equality between terms be a *unification procedure* instead of a simple syntactic comparison. When an uninstantiated inference variable is compared to a term, the variable is instantiated accordingly so that the comparison is successful. The type inference algorithm thus uses `lterm_equal` in order to generate and solve constraints simultaneously as mentioned above.

We implement inference variables as mutable references of option type; they are initially empty and get assigned to a term when instantiated. Syntactic operations such as freshening, binding and substitution application cannot be applied to uninstantiated variables. Instead we record them into a *suspension* – a function gathering their effects so that they get applied once the inference variable is instantiated. Thus we represent inference variables as a pair:

$$(?_n, f)$$

where $?_n$ is understood as the mutable reference part and f as the composition of the applied syntactic operations. Thus the constraint $?_1 \cdot (X/a,\ Y/b) \equiv (X = Y)$ given as an example above, will instead by represented and solved by an equality checking call of the form:

$$\texttt{lterm\_equal}\ (?_1,\ f)\ (X = Y)$$

where the suspension f is defined as $f = - \cdot (X/a,\ Y/b)$. Different occurrences of the same inference variable – arising, for example, from an application like $t_1\ ?_2$ where $t_1 : \Pi(t).t'$ and t' has multiple occurrences of the bound variable b_0 – share the reference part but might have a different suspension, corresponding to the point that they are used. Because of the

use of mutable references, all occurrences of the same inference variable get instantiated at the same time.

The main subtlety of instantiating inference variables is exactly the presence of suspensions. In the case where we need to unify an inference variable with a term

$$(?_n,\ f) = t$$

we cannot simply set $?_n = t$: even if we ignore the suspension f, further uses of the same inference variable with a different suspension f' will not have the intended value. The reason is that t is the result of some applications of syntactic operations as well:

$$\begin{aligned}(?_n,\ f) &= t \text{ where } t = f(t'')\\ (?_n,\ f') &= t' \text{ where } t' = f'(t'')\end{aligned}$$

Instead of unifying $?_n$ with t, we should unify it with t''; applying the suspensions as first intended will result in the right value in both cases. In order to do this, we also maintain an *inverse suspension* f^{-1} as part of our representation of inference variables, leading to triples of the form:

$$(?_n,\ f,\ f^{-1})$$

where $f \circ f^{-1} = id$. Instantiating $?_n$ with $f^{-1}(t)$ yields the desired result in both occurrences of the variable.

Keeping in mind that f is a composition of syntactic operations, e.g. $f = \lceil - \rceil \circ (- \cdot \sigma)$, we maintain the inverse suspension by viewing it as a composition of the inverse syntactic operations, e.g. $f^{-1} = (- \cdot \sigma^{-1}) \cdot \lceil - \rceil$. In the case of freshening and binding, these inverse operations are trivial since $\lceil \cdot \rceil = \lfloor \cdot \rfloor^{-1}$. Yet in the case of applying a substitution σ, the case that an inverse substitution σ^{-1} exists is but a special case – essentially, when σ is a renaming of free variables – so we cannot always set $f^{-1} = (- \cdot \sigma^{-1})$. In the general case, σ might instantiate variables with specific terms. The inverse operation is then the application of a generalized notion of substitutions σ_G where arbitrary terms – not only variables – can be substituted for other terms. In our implementation, unification does not handle such cases in full generality. We only handle the case where the applied substitutions are renamings between normal variables and meta-variables. This case is especially useful for implementing static proof expressions as presented in Section 7.2, which yield constraints of the form:

$$?_1 \cdot (X/a,\ Y/b) \equiv (X = Y)$$

as seen above, where X and Y are metavariables and a and b are normal variables. In this case, our representation of inference variables yields:

$$(?_1,\ f = (- \cdot \sigma),\ f^{-1} = (- \cdot \sigma_G^{-1})) \equiv (X = Y)$$

with $\sigma = (X/a,\ Y/b)$ and $\sigma_G^{-1} = (a/X,\ b/Y)$, which leads to

$$?_1 \equiv f^{-1}(X = Y) \equiv (a = b)$$

as desired.

The implementation of these inference variables forms the main challenge in supporting type inference for logical terms. The actual type inference algorithm mimics the original type checking algorithm closely. The main departure is that it exclusively relies on the equality checking/unification procedure `lterm_equal` to impose constraints on types, even in cases where the original type checker uses pattern matching on the return type. An example is checking function application, where instead of code such as:

```
App(t1, t2) |-> let t1_t = type_of t1 in
                let t2_t = type_of t2 in
                match t1_t with
                  Pi(var, t, t') -> lterm_equal t t2_t
```

we have:

```
App(t1, t2) |-> let t1_t = type_of t1 in
                let t2_t = type_of t2 in
                let t = new_infer_var () in
                let t' = new_infer_var () in
                let t1_expected = Pi(var, t, t') in
                lterm_equal t1_t t1_expected && lterm_equal t t2_t
```

Essentially this corresponds to using typing rules similar to ΠElim-Infer, as described above. Our type inference algorithm also supports bi-directional type checking: it accepts an extra argument representing the expected type of the current term, and instantiates this

argument accordingly during recursive calls. The two directions of bi-directional type checking, synthesis (determining the type of an expression) and analysis (checking an expression against a type), correspond to the case where the expected type is simply an uninstantiated inference variable, and the case where it is at least partially instantiated, respectively. This feature is especially useful when the expected type of a term is known beforehand, as type-checking its subterms is informed from the extra knowledge and further typing constraints become solvable.

8.2.2 Inductive definitions

Our logic implementation supports inductive definitions in order to allow users to define and reason about types such as natural numbers, lists, etc. Also, it allows inductive definitions of logical predicates, which are used to define logical connectives such as conjunction and disjunction, and predicates between inductive types, such as the less-than-or-equal relation between natural numbers. Each inductive definition generates a number of constants added to the signature context Σ, including constants for constructors and elimination and induction principles. Support for inductive types is necessary so as to prescribe a 'safe' set of axioms that preserve logical soundness, as arbitrary additions to the constant signature can lead to an inconsistent axiom set. It is also necessary from a practical standpoint as it simplifies user definitions significantly because of the automatic generation of induction and elimination principles. Our support follows the approach of inductive definitions in CIC [Paulin-Mohring, 1993] and Martin-Löf type theory [Dybjer, 1991], adapted to the λHOL logic.

Let us proceed to give some examples of inductive definitions. First, the type of natural numbers is defined as:

$$\begin{aligned} \mathsf{Inductive}\ \mathit{Nat}\ &:\ \mathit{Type}\ := \\ \mathit{zero}\ &:\ \mathit{Nat} \\ \mid\ \mathit{succ}\ &:\ \mathit{Nat} \to \mathit{Nat} \end{aligned}$$

This generates the following set of constants:

$$
\begin{array}{lcl}
\mathit{Nat} & : & \mathit{Type} \\
\mathit{zero} & : & \mathit{Nat} \\
\mathit{succ} & : & \mathit{Nat} \to \mathit{Nat} \\
\mathit{elimNat} & : & [t : \mathit{Type}]\, t \to (\mathit{Nat} \to t \to t) \to \mathit{Nat} \to t \\
\mathit{elimNatZero} & : & [t : \mathit{Type}]\, \forall f_z f_s, \mathit{elimNat}\ f_z\ f_s\ \mathit{zero} = f_z \\
\mathit{elimNatSucc} & : & [t : \mathit{Type}]\, \forall f_z f_s p, \mathit{elimNat}\ f_z\ f_s\ (\mathit{succ}\ p) = f_s\ p\ (\mathit{elimNat}\ f_z\ f_s\ p) \\
\mathit{indNat} & : & \forall P : \mathit{Nat} \to \mathit{Prop}. P\ \mathit{zero} \to (\forall n, P\ n \to P\ (\mathit{succ}\ n)) \to \forall n, P\ n
\end{array}
$$

The elimination principle *elimNat* is used to define total functions over natural numbers through primitive recursion; the axioms *elimNatZero* and *elimNatSucc* correspond to the main computation steps associated with it. They correspond to one step of ι-reduction, similar to how the *beta* axiom corresponds to one step of β-reduction. An example of such a function definition is addition:

$$\mathit{plus} \ = \ \lambda x. \lambda y. \mathit{elimNat}\ y\ (\lambda p. \lambda r. \mathit{succ}\ r)\ x$$

Through the rewriting axioms we can prove:

$$
\begin{array}{lcl}
\mathit{plus}\ \mathit{zero}\ y & = & y \\
\mathit{plus}\ (\mathit{succ}\ x)\ y & = & \mathit{succ}(\mathit{plus}\ x\ y)
\end{array}
$$

The induction principle *indNat* is the standard formulation of natural number induction in higher-order logic. Lists are defined as follows, with t as a type parameter.

$$
\begin{array}{l}
\mathsf{Inductive}\ \mathit{List}\ :\ [t : \mathit{Type}]\ \mathit{Type}\ := \\
\qquad \mathit{nil}\ \ :\ \ \mathit{List} \\
\quad \mid\ \mathit{cons}\ \ :\ \ t \to \mathit{List} \to \mathit{List}
\end{array}
$$

We also use inductive definitions to define connectives and predicates at the level of propositions. For example, we define logical disjunction $\vee$ as:

$$
\begin{array}{l}
\mathsf{Inductive}\ \mathit{or}\ :\ [A : \mathit{Prop}, B : \mathit{Prop}]\ \mathit{Prop}\ := \\
\qquad \mathit{orIntroL}\ \ :\ \ A \to \mathit{or}/[A,\ B] \\
\quad \mid\ \mathit{orIntroR}\ \ :\ \ B \to \mathit{or}/[A,\ B]
\end{array}
$$

generating the following set of constants:

$$
\begin{array}{lcl}
\mathit{or} & : & [A : \mathit{Prop},\ B : \mathit{Prop}]\ \mathit{Prop} \\
\mathit{orIntroL} & : & [A : \mathit{Prop},\ B : \mathit{Prop}]\ A \to \mathit{or}/[A, B] \\
\mathit{orIntroR} & : & [A : \mathit{Prop},\ B : \mathit{Prop}]\ B \to \mathit{or}/[A, B] \\
\mathit{indOr} & : & [A : \mathit{Prop},\ B : \mathit{Prop}]\ \forall P : \mathit{Prop}, \\
 & & \qquad (A \to P) \to (B \to P) \to (\mathit{or}/[A, B] \to P)
\end{array}
$$

We have defined the two arguments to *or* as parameters, instead of defining *or* as having a kind $Prop \to Prop \to Prop$, as this leads to the above induction principle which is simpler to use.

An example of an inductively-defined predicate is less-than-or-equal for natural numbers:

$$\begin{aligned}&\mathsf{Inductive}\ le\ :\ Nat \to Nat \to Prop\ :=\\ &\quad leBase\ :\ \forall n, le\ n\ n\\ &\mid\ leStep\ :\ \forall nm, le\ n\ m \to le\ n\ (succ\ m)\end{aligned}$$

generating the induction principle:

$$\begin{aligned}indLe\ :\ &\forall P : Nat \to Nat \to Prop,\\ &(\forall n, P\ n\ n) \to\\ &(\forall nm, le\ n\ m \to P\ n\ m \to P\ n\ (succ\ m)) \to\\ &(\forall nm, le\ n\ m \to P\ n\ m)\end{aligned}$$

The types of constructors for inductive definitions need to obey the strict positivity condition [Paulin-Mohring, 1993] with respect to uses of the inductive type under definition. This is done to maintain logical soundness, which is jeopardized when arbitrary recursive definitions such as the following are allowed:

$$\mathsf{Inductive}\ D\ :\ Type\ :=\ mkD\ :\ (D \to D) \to D$$

This definition in combination with the generated elimination principle allows us to define non-terminating functions over the type D, which renders the standard model used for the semantics of λHOL unusable. In order to check the positivity condition, we extend the λHOL type system to include support for positivity types, following the approach of Abel [2010], instead of the traditional implementation using syntactic checks used in Coq [Barras et al., 2012] and other proof assistants. This results in a simple implementation that can directly account for traditionally tricky definitions, such as nested inductive types. The elimination principles are generated through standard techniques.

We give a formal sketch of the required extensions for positivity types to λHOL typing in Figure 8.2. The main idea is that contexts Φ carry information about whether a variable must occur positively (denoted as t^+) or not. We change the type judgement so that it includes the polarity of the current position: the positive polarity + for positive positions; the strictly positive polarity ++; and the negative polarity − where only normal unrestricted

Syntax

$$\begin{array}{rl} (\textit{Contexts}) & \Phi ::= \cdots \mid \Phi,\ t^{+} \\ (\textit{Polarities}) & p ::= - \mid {+}{+} \mid + \end{array}$$

$\boxed{\Psi;\ \Phi \vdash t :^{p} t'}$

$$\frac{\Phi.L = t^{+} \qquad p \neq -}{\Psi;\ \Phi \vdash v_L :^{p} t}\ \textsc{VarPos} \qquad \frac{\Phi.L = t}{\Psi;\ \Phi \vdash v_L :^{p} t}\ \textsc{VarAny}$$

$$\frac{\Psi;\ \Phi \vdash t_1 :^{p\downarrow} s \qquad \Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} :^{p} s' \qquad (s, s', s'') \in \mathcal{R}}{\Psi;\ \Phi \vdash \Pi(t_1).t_2 :^{p} s''}\ \Pi\textsc{Type}$$

$$\frac{\Psi;\ \Phi \vdash t_1 :^{-} s \qquad \Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} :^{-} t' \qquad \Psi;\ \Phi \vdash \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|\dot{+}1} :^{-} s'}{\Psi;\ \Phi \vdash \lambda(t_1).t_2 :^{p} \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|\dot{+}1}}\ \Pi\textsc{Intro}$$

$$\frac{\Psi;\ \Phi \vdash t_1 :^{p} \Pi(t).t' \qquad \Psi;\ \Phi \vdash t_2 :^{-} t}{\Psi;\ \Phi \vdash t_1\ t_2 :^{p} \lceil t' \rceil_{|\Phi|} \cdot (id_\Phi,\ t_2)}\ \Pi\textsc{Elim}$$

$$\frac{\Psi.i = T \qquad T = [\Phi']\, t' \qquad \Psi;\ \Phi \vdash \sigma :^{p} \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma :^{p} t' \cdot \sigma}\ \textsc{MetaVar}$$

$\boxed{\Psi;\ \Phi' \vdash \sigma :^{p} \Phi}$

$$\frac{\Psi;\ \Phi \vdash \sigma :^{++} \Phi' \qquad \Psi;\ \Phi \vdash t :^{++} t' \cdot \sigma}{\Psi;\ \Phi \vdash (\sigma,\ t) :^{++} (\Phi',\ t'^{+})}\ \textsc{SubstPosVarPos}$$

$$\frac{\Psi;\ \Phi \vdash \sigma :^{++} \Phi' \qquad \Psi;\ \Phi \vdash t :^{-} t' \cdot \sigma}{\Psi;\ \Phi \vdash (\sigma,\ t) :^{++} (\Phi',\ t')}\ \textsc{SubstAnyVarPos}$$

$$\frac{\Psi;\ \Phi \vdash \sigma :^{+} \Phi' \qquad \Psi;\ \Phi \vdash t :^{++} t' \cdot \sigma}{\Psi;\ \Phi \vdash (\sigma,\ t) :^{+} (\Phi',\ t'^{+})}\ \textsc{SubstPosVarAny}$$

$$\frac{\Psi;\ \Phi \vdash \sigma :^{p} \Phi' \qquad \Psi;\ \Phi \vdash t :^{p} t' \cdot \sigma}{\Psi;\ \Phi \vdash (\sigma,\ t) :^{p} (\Phi',\ t')}\ \textsc{SubstAnyVarAny}$$

$\boxed{p\downarrow}$

$$\begin{array}{lcl} +\downarrow & = & {+}{+} \\ {+}{+}\downarrow & = & - \\ -\downarrow & = & - \end{array}$$

Figure 8.2: Extension λHOL with positivity types

variables can be used. We demote polarities when in the domain of a function type; in this way we allow constructors with type $(Nat \to Ord) \to Ord$, where Ord occurs positively (the limit constructor of Brouwer ordinals); but we disallow constructors such as $(D \to D) \to D$, as the leftmost occurrence of D is in a negative position. We take polarities into account when checking substitutions used with metavariables. Thus if we know that a certain parametric type (e.g. lists) uses its parameter strictly positively, we can instantiate the parameter with a positive variable. This is used for defining rose trees – trees where each node has an arbitrary number of children:

$$\begin{array}{l}\mathsf{Inductive}\ \mathit{List}\ :\ [t : \mathit{Type}^+]\ \mathit{Type}\ :=\\ \quad \mathit{nil}\ \ :\ \ \mathit{List}\\ \mid\ \mathit{cons}\ :\ t \to \mathit{List} \to \mathit{List}\end{array} \qquad \begin{array}{l}\mathsf{Inductive}\ \mathit{Rose}\ :\ [t : \mathit{Type}^+]\ \mathit{Type}\ :=\\ \quad \mathit{node}\ :\ \mathit{List}/[\mathit{Rose}] \to \mathit{Rose}\end{array}$$

Using the new judgements, inductive definitions can be simply checked as follows (we use named variables for presentation purposes):

$$\frac{\bullet \vdash [\Phi]\, t : s \qquad t = \Pi\overrightarrow{x_{args} : t_{args}}.s' \qquad s' \in (\mathit{Prop}, \mathit{Type}) \qquad \bullet;\ \Phi,\ \mathit{name} : t^+ \vdash t_i :^+ s' \qquad t_i = \Pi\overrightarrow{x^1 : t_i^1}.\mathit{name}\ \overrightarrow{t_i^2}}{\vdash (\mathsf{Inductive}\ \mathit{name}\ :\ [\Phi]\, t\ := \overrightarrow{\mathit{cname}_i : t_i})\ \mathrm{wf}}$$

That is, the inductive definition must have an appropriate *arity* at one of the allowed sorts for definitions, the positivity condition must hold for the occurrences of the new type in each constructor, and each constructor must yield an inhabitant of the new type.

8.3 Computational language implementation

The implementation of the VeriML computational language follows a similar approach to the implementation of λHOL with similar components (parsing, syntactic elaboration, typing and type inference). Calls to the relevant λHOL functions are done at every stage in order to handle logical terms appearing inside computational expressions. Our implementation supports all the constructs presented in Chapter 6, including a pattern matching construct that combines all the features presented in Section 6.2. It also includes further constructs not covered in the formal definition, such as let and letrec constructs for (mutually recursive)

definitions, base types such as integers, booleans and strings, mutable arrays, generic hasing and printing functions and monadic-do notation for the option type. The support for these follows standard practice. Here we will cover the surface syntax extensions handled by syntactic elaboration and also aspects of the evaluation of VeriML expressions through translation to OCaml.

8.3.1 Surface syntax extensions

The surface syntax for the VeriML computational language supports a number of conveniences to the programmers. The most important of these are simplified syntax for contextual terms and surface syntax for tactic application. Other conveniences include abbreviated forms of the constructs that make use of inference variables in their expanded form – for example, eliding the return type in dependent tuples $\langle\, T,\ e \,\rangle_{(X:K)\times\tau}$.

Delphin-style syntax for contextual terms. The simplified syntax for contextual terms is based on the aforementioned notion of an *ambient context* Φ, denoted as @ in the surface syntax of our implementation. We denote it here as $\Phi_@$ for presentation purposes. The main idea is to use the ambient context instead of explicitly specifying the context part of contextual terms $[\Phi]\, t$, leading to contextual terms of the form $@t$ (denoting $[\Phi_@]\, t$). Uses of metavariables such as X/σ are also simplified, by taking σ to be the identity substitution for the ambient context (more precisely, the prefix of the identity substitution $id_{\Phi'} \subseteq id_{\Phi_@}$ when $X : [\Phi']\, t'$ with $\Phi' \subseteq \Phi_@$). Furthermore, we include constructs to manage the ambient context: a way to introduce a new variable, $\nu x : t$ in e, which changes the ambient context to $\Phi'_@ = \Phi_@,\ x : t$ inside e. Also, we have a way to reset the ambient context and specify it explicitly, denoted as let $@ = \Phi$ in e. Some existing constructs, such as abstraction over contexts $\lambda\phi : ctx.e$ also update the ambient context so as to include the newly-introduced context variable: $\Phi'_@ = \Phi_@,\ \phi$. Based on these, we can write the universal quantification branch of the tautology prover as:

$$
\begin{array}{lcl}
\mathsf{tautology} & : & (\phi : ctx) \to (P : @Prop) \to \mathsf{option}\ (@P) \\
\mathsf{tautology} & = & \lambda\phi : ctx.\lambda P : @Prop.\mathsf{holmatch}\ @P\ \mathsf{with} \\
 & & \quad @\forall x : Nat, Q \ \mapsto\ \mathsf{do}\ \langle\ pf'\ \rangle \leftarrow \nu x : Nat\ \mathsf{in}\ \mathsf{tautology}\ @\ @Q; \\
 & & \quad\quad \mathsf{return}\ \langle\ @\lambda y : Nat.pf'/[id_{@}, y]\ \rangle
\end{array}
$$

The syntax thus supported is close to the informal syntax we used in Section 2.3. Though it is a simple implementation change to completely eliminate mentioning the ambient context, we have maintained it in order to make the distinction between contextual terms of λHOL and computational expressions of VeriML more evident.

The idea of having an ambient context and of the fresh variable introduction construct $\nu x : t\ \mathsf{in}\ e$ comes from the Delphin programming language [Poswolsky, 2009]. Delphin is a similar language to VeriML, working over terms of the LF framework instead of λHOL; also, it does not include an explicit notion of contextual terms, supporting instead only terms inhabiting the ambient context as well as the constructs to manipulate it. Our implementation establishes that these constructs can be seen as syntactic sugar that gets expanded into full contextual terms of contextual modal type theory [Nanevski et al., 2008b], suggesting a syntax-directed translation from Delphin expressions into expressions over contextual terms. Instead of reasoning explicitly about contextual terms as we do in our metatheory, Delphin manages and reasons about the ambient context directly at the level of its type system and operational semantics. This complicates the formulation of the type system and semantics significantly, as well as the needed metatheoretic proofs. Furthermore, the syntax-directed translation we suggest elucidates the relationship between Delphin and Beluga [Pientka and Dunfield, 2008], a language for manipulating LF terms based on contextual terms, similar to VeriML.

Syntax for static proof expressions. Our implementation supports turning normal proof expressions $e : \big([\Phi]\, P\big)$ into statically-evaluated proof expressions as presented in Section 7.2 with a simple annotation $\{\{e\}\}$. Note that this is not the staging construct, as we assume that e might be typed under an arbitrary Ψ context containing extension variables instantiated at runtime. This construct turns e into a closed expression e_s that does not include any extension variables, following the ideas in Section 5.2; it then uses the

staging construct $\{e_s\}_{\text{static}}$ to evaluate the closed expression; and then performs the needed substitution of variables to metavariables in order to bring the result of the static evaluation into the expected Ψ context with the right type. A sketch of this transformation is:

$$\Psi;\ \Sigma;\ \Gamma|_{\text{static}} \vdash \{\{e\}\} : \big([\Phi]\, P\big)$$

$$\rightsquigarrow$$

$$\frac{\bullet;\ \Sigma;\ \Gamma|_{\text{static}} \vdash e_s : \big([\Phi']\, P'\big) \qquad \bullet \vdash \Phi' \text{ wf} \qquad \Psi;\ \Phi \vdash \sigma : \Phi' \qquad P' \cdot \sigma = P}{\Psi;\ \Sigma;\ \Gamma|_{\text{static}} \vdash \mathsf{let}\ \langle\ X\ \rangle\ = \{e_s\}_{\text{static}}\ \mathsf{in}\ \langle\ [\Phi]\, X/\sigma\ \rangle : \big([\Phi]\, P\big)}$$

The actual implementation of this construct is a combination of syntactic elements – most importantly, manipulation of the ambient context – and type inference. Let us first present an example of how this construct is used and expanded, arising from the plusRewriter1 example seen in Section 7.2. Consider the following code fragment:

$$\begin{array}{l}
\mathsf{plusRewriter1}\ = \\
\quad \lambda\phi : ctx.\lambda T : @\mathit{Type}.\lambda t : @T.\mathsf{holmatch}\ @t\ \mathsf{with} \\
\quad\quad @x + y \mapsto \mathsf{let}\ \langle\ y',\ H' : @y = y'\ \rangle\ = \cdots\ \mathsf{in} \\
\quad\quad\quad\quad \mathsf{let}\ \langle\ t',\ H'' : @x + y' = t'\ \rangle\ = \cdots\ \mathsf{in} \\
\quad\quad\quad\quad \Big\langle\ t',\ \{\{\mathsf{requireEqual}\ @\ @?_1\ @?_2\ @?_3\}\} : \big(@x + y = t'\big)\ \Big\rangle
\end{array}$$

where requireEqual has type:

$$\mathsf{requireEqual}\ :\ (\phi : ctx) \to (T : @\mathit{Type}) \to (t_1,\ t_2 : @T) \to \big(@t_1 = t_2\big)$$

At the point where requireEqual is called, the extension context is:

$$\Psi = \phi : ctx,\ T : @\mathit{Type},\ t, x, y, y' : @\mathit{Nat},\ H' : @y = y',\ t' : @\mathit{Nat},\ H'' : @x + y' = t'$$

Based on this, syntactic elaboration will determine the Φ' construct used in the expansion of $\{\{\cdots\}\}$ as (using $\widehat{\cdot}$ to signify normal variables and differentiate them from the metavariables they correspond to):

$$\Phi' = \widehat{T} :?_T,\ \widehat{t} :?_t,\ \widehat{x} :?_x,\ \widehat{y} :?_y,\ \widehat{y'} :?_{y'},\ \widehat{H'} :?_{H'},\ \widehat{t'} :?_{t'},\ \widehat{H''} :?_{H''}$$

and the substitution σ as:

$$\sigma = T/[id_\phi],\ t/[id_\phi],\ x/[id_\phi],\ y/[id_\phi],\ y'/[id_\phi],\ H'/[id_\phi],\ t'/[id_\phi],\ H''/[id_\phi]$$

The expansion then is:

$$
\begin{array}{l}
\{\{\mathsf{requireEqual}\ @\ @?_1\ @?_2\ @?_3\}\}\ \rightsquigarrow \\
\quad \mathsf{let}\ \langle\ S : [\Phi']\, ?_3\ \rangle\ = \\
\qquad \mathsf{let}\ @ =\ \Phi'\ \mathsf{in}\ \{\mathsf{requireEqual}\ @\ @?_1\ @?_2\ @?_3\}_{\text{static}} \\
\quad \mathsf{in} \\
\quad \langle\ [\phi]\, S/\sigma\ \rangle
\end{array}
$$

From type inference, we get that:

$$?_3 \cdot \sigma \equiv x + y = t'$$

From this, we get that $?_3 \equiv \widehat{x} + \widehat{y} = \widehat{t'}$. The rest of the inference variables are similarly determined.

The implementation thus works as follows. First, we transform e into a closed expression e_s by relying on the ambient context mechanism. We expect all contextual terms inside e to refer only to the ambient context. We change the ambient context to the Φ' context, as determined syntactically by introductions of metavariables and normal variables through the νx in e construct. Normal variables from Φ' shadow meta-variables leading to a closed expression. The exact types in Φ' are not determined syntactically; inference variables are used instead. The σ substitution is constructed together with Φ' and can be viewed as a renaming between normal variables and meta-variables. It is an invertible substitution, as described above in the type inference section; therefore even if e contains inference variables, these can still be uniquely determined.

Tactics syntax. Our implementation includes syntactic sugar for frequently-used tactics, in order to enable users to write concise and readable proof scripts. Each new syntactic form expands directly into a tactic call, potentially using the features described so far – implicit arguments through inference variables, manipulation of the ambient context and staging annotations. For example, we introduce the following syntax for conversion, hiding some implicit arguments and a static tactic call:

$$\mathsf{Exact}\ e\ \equiv\ \mathsf{eqElim}\ @\ @?\ @?\ e\ \{\{\mathsf{requireEqual}\ @\ @?\ @?\ @?\}\}$$

with the following type for eqElim:

$$\mathsf{eqElim} \quad : \quad (\phi : ctx,\ P : @Prop,\ P' : @Prop, H : @P, H' : @P = P') \to \left(@P'\right)$$

Intuitively, this syntax is used at a place where a proof of P' is required but we supply a proof of the equivalent proposition P. The proof of equivalence between P and P' is done through static evaluation, calling the requireEqual tactic. Using $\Uparrow$ for type synthesis and $\Downarrow$ for type analysis we get that the combined effect of syntactic elaboration and type inference is:

$$\frac{\vdash e \Uparrow \left(@P'\right)}{\vdash \mathsf{Exact}\ e \Downarrow \left(@P\right)}$$

$$\rightsquigarrow$$

$$\mathsf{eqElim}\ @\ @P\ @P'\ e\ \{\{\mathsf{requireEqual}\ @\ @Prop\ @P\ @P'\}\}$$

Another frequently-used example is the syntax for the cut principle – for proving a proposition P and introducing a name H for the proof to be used as a hypothesis in the rest of the proof script. This is especially useful for forward-style proofs.

$$\mathsf{Cut}\ H : P\ \mathsf{by}\ e\ \mathsf{in}\ e'$$

$$\equiv$$

$$(\mathsf{cut}\ :\ (\phi : ctx,\ P' : @Prop,\ P : @Prop, pf_1 : @P,\ pf_2 : (\nu H : P\ \mathsf{in}\ @P')) \to \left(@P'\right)\ )$$
$$@\ @?\ @?\ @P\ e\ (\nu H : P\ \mathsf{in}\ e')$$

An example of a full proof script for a property of addition that uses this style of tactic syntactic sugar follows:

$$\begin{array}{l}
\mathsf{let}\ plus_x_Sy\ :\ \left(@\forall x, y : Nat. x + (succ\ y) = succ\ (x + y)\right)\ = \\
\quad \mathsf{Intro}\ x\ \mathsf{in}\ \mathsf{Intro}\ y\ \mathsf{in} \\
\quad \mathsf{Instantiate} \\
\quad\quad (\mathsf{NatInduction\ for}\ z. z + (succ\ y) = succ(z + y) \\
\quad\quad\quad \mathsf{base\ case\ by\ Auto} \\
\quad\quad\quad \mathsf{inductive\ case\ by\ Auto}) \\
\quad \mathsf{with}\ @x
\end{array}$$

The tactic syntax we support is mostly meant as a proof-of-concept. It demonstrates the fact that proof scripts resembling the ones used in traditional proof assistants are possible with our approach, simply by providing specific syntax that combines the main features we

already support. We believe that this kind of syntax does not have to be built into the language itself, but can be provided automatically based on the available type information. For example, it can be determined which arguments to a function need to be concrete and which can be left unspecified as inference variables, as evidenced by existing implicit argument mechanisms (e.g. in Coq [Barras et al., 2012]). Also, obligations to be proved statically, such as $P = P'$ above, can be also handled, by registering a default prover for propositions of a specific form with the type inferencing mechanism. In this way, syntax for tactics defined by the user would not need to be explicitly added to the parser. We leave the details as future work.

Meta-generation of rewriters. When defining inductive types such as *Nat* and *List* as shown in Subsection 8.2.2, we get a number of axioms like *elimNatZero* that define the computational behavior of the associated elimination principles. The fact that we do not have any built-in conversion rule means that these axioms are not taken into account automatically when deciding term equality. Following the approach of Section 7.2, we need to define functions that rewrite terms based on these axioms. The situation is similar when proving theorems involving equality like:

$$\forall x.x + 0 = x$$

which is the example that prompted us to develop *plusRewriter1* in the same section. Writing such rewriters is tedious and soon becomes a hindrance: every inductive definition and many equality theorems require a specialized rewriter. Still the rewriters follow a simple template: they perform a number of nested pattern matches and recursive calls to determine whether the left-hand side of the equality matches the scrutinee and then rewrite to the right-hand side term if it does.

In order to simplify this process, we take advantage of the common template of rewriters and provide a syntactic construct that generates rewriters at the meta-level. This construct is implemented as an OCaml function that yields a VeriML term when given a theorem that proves an equality. That is, we add a construct GenRewriter c which is given a proof c of type $\forall \cdots, lhs = rhs$ and generates a VeriML term that corresponds to the rewriter a user would normally write by hand. The rewriter has similar structure as described earlier.

When programming the rewriter generator itself, we use the normal OCaml type system to construct an appropriate VeriML term. This term is then returned as the expansion of GenRewriter c and checked using the VeriML type system. Thus the rewriter generator does not need to be trusted.

8.3.2 Translation to OCaml

Our implementation evaluates VeriML programs by translating them into OCaml programs which are then run using the standard OCaml tools. Intuitively, the translation can be understood as translating dependently-typed VeriML expressions into their simply-typed ML counterparts. We follow this approach in order to achieve efficient execution of VeriML programs without the significant development cost of creating an optimizing compiler for a new language. We make use of the recently-developed Just-In-Time-based interpreter for OCaml [Meurer, 2010] which is suitable for the interactive workflow of a proof assistant, as well as of the high-quality OCaml compiler for frequently-used support code and tactics. The availability of these tools render the language a good choice for the described approach to VeriML evaluation. We have also developed a complete VeriML interpreter that does not make use of translation to OCaml. Its implementation follows standard practice corresponding directly to the VeriML semantics we have given already. Nevertheless, it has been largely obsoleted in favor of OCaml-based translation because of efficiency issues: typechecking and running the full code of our examples presented in the next chapter takes about 6 minutes with the interpreter-based evaluation, whereas it takes only 15 seconds in OCaml-based translation coupled with OCamlJIT, when using a 2.67 GHz Intel i7-620 CPU.

The main translation function accepts a VeriML computational language AST and emits an OCaml AST. The VeriML AST is assumed to be well-typed; the translation itself needs some type information for specific constructs and it can therefore be understood as a type-directed translation. We make use of the Camlp4 library of OCaml that provides quotation syntax for OCaml ASTs.

The ML core of VeriML presented in Figure 6.1 is translated directly into the corresponding OCaml forms. The λHOL-related constructs of Figure 6.2 are translated into

Description	**VeriML AST**	**OCaml AST**
Dependent functions over contextual terms		
type	$(X : [\Phi]\, t) \to \tau$	`Logic.term ->` $[\![\tau]\!]$
introduction	$\lambda X : [\Phi]\, t.e$	`fun X : Logic.term =>` $[\![e]\!]$
elimination	$e\ ([\Phi]\, t)$	$[\![e]\!]\ [\![t]\!]$
Dependent functions over contexts		
type	$(\phi : [\Phi]\ ctx) \to \tau$	`Logic.ctx ->` $[\![\tau]\!]$
introduction	$\lambda\phi : [\Phi]\ ctx.e$	`fun X : Logic.ctx =>` $[\![e]\!]$
elimination	$e\ ([\Phi]\, \Phi')$	$[\![e]\!]\ [\![\Phi']\!]$
Dependent tuples over contextual terms		
type	$(X : [\Phi]\, t) \times \tau$	`Logic.term *` $[\![\tau]\!]$
introduction	$\langle\ [\Phi]\, t,\ e\ \rangle_{(X:[\Phi]\, t') \times \tau}$	`(` $[\![t]\!]$`,` $[\![e]\!]$ `)`
elimination	$\mathsf{let}\ \langle\ X,\ x\ \rangle = e\ \mathsf{in}\ e'$	`let ( X, x ) =` $[\![e]\!]$ `in` $[\![e']\!]$
Type constructors over contextual terms		
kind	$\Pi V : K.k$	$[\![k]\!]$
introduction	$\lambda X : T.\tau$	$[\![\tau]\!]$
elimination	$\tau\ ([\Phi]\, t)$	$[\![\tau]\!]$

Table 8.2: Translation of λHOL-related VeriML constructs to simply-typed OCaml constructs

their simply-typed OCaml counterparts, making use of our existing λHOL implementation in OCaml – the same one used by the VeriML type checker. For example, VeriML dependent functions over contextual terms are translated into normal OCaml functions over the `Logic.term` datatype – the type of λHOL terms as encoded in OCaml. The fact that simply-typed versions of the constructs are used is not an issue as the type checker of VeriML has already been run at this point, providing the associated benefits discussed in previous chapters. We give a sketch of the translated forms of most λHOL-related constructs in Table 8.2. Note that type-level functions over extension terms are simply erased, as they are used purely for VeriML type checking purposes and do not affect the semantics of VeriML expressions.

Let us now present some important points of this translation. First, the context part of contextual terms is dropped – we translate a contextual term $[\Phi]\, t$ as the OCaml AST $[\![t]\!]$. For example, the term

$$[x : Nat]\, \forall y : Nat, x > 0 \rightarrow x + y > y$$

or, in concrete syntax:

$$[Nat]\, \Pi(Nat).\Pi(gt\ v_0\ zero).gt\ (plus\ v_0\ b_1)\ b_1$$

is translated as:

```
LPi(LConst("Nat"),
 LPi(LApp(LApp(LConst("gt"), LFVar(0)), LConst("zero")),
  LApp(LApp(LConst("gt"),
   (LApp(LApp(LConst("plus"), LFVar(0), LBVar(1)))),
   LBVar(1)))))
```

The reason why the context part is dropped is that it is not required during evaluation; it is only relevant during VeriML type checking. This is evident in our formal model from the fact that the λHOL-related operations used in the VeriML operational semantics – substitution and pattern matching – do not actually depend on the context part, as is clear from the practical version of pattern matching using annotated terms in Section 5.1.

Also, extension variables X and ϕ as used in the dependently-typed constructs are translated into *normal OCaml variables*. This reflects the fact that the operational semantics of

VeriML are defined only on closed VeriML expressions where the extension variables context Ψ is empty; therefore extension variables do not need a concrete runtime representation. By representing extension variables as OCaml variables, we get extension substitution for free, through OCaml-level substitution. This is evidenced in the following correspondence between the VeriML semantics of function application and the OCaml semantics of the translated expression:

$$(\lambda X : [\Phi]\, t'.e)\ ([\Phi]\, t) \longrightarrow_{VeriML} e \cdot ([\Phi]\, t/X)$$

$$\equiv$$

$$\texttt{(fun X : Logic.term => e) t} \longrightarrow_{OCaml} \texttt{e[t/X]}$$

Uses of extension variables inside logical terms are translated as function calls. As mentioned in Section 4.1, the usage form of meta-variables X/σ is understood as a way to describe a deferred substitution σ, which gets applied to the term t as soon as X is instantiated with the contextual term $[\Phi]\, t$. This is captured in our implementation by representing X/σ as a function call `subst_free_list X` σ, where the free variables of the term `X` (again, an OCaml-level variable) are substituted for the terms in σ. Similarly, uses of parametric contexts ϕ stand for deferred applications of weakening, shifting the free variables in a term by the length of the specified context, when ϕ is instantiated.

Pattern matching over λHOL terms is translated following the same approach into *simply-typed pattern matching* over the OCaml datatypes `Logic.term` and `Logic.ctx` that encode λHOL logical terms and contexts respectively. Patterns T_P as used in the VeriML construct become OCaml patterns over those data types; unification variables from Ψ_u become OCaml pattern variables. In this way we reuse pattern matching optimizations implemented in OCaml. The main challenge of this translation is maintaining the correspondence with the pattern matching rules described in Section 5.1. Most rules directly correspond to a constructor pattern for λHOL terms – for example, the rule for sorts: we can simply match a term like `LSort(LSet)` against a pattern like `LSort(s)`, where s is a pattern variable. Other rules require a combination of a constructor pattern and of guard conditions, such as the rule for application: matching the term `LApp(t1,t,s,t2)` corresponding to the term $(t_1 : t : s)\ t_2$ against a pattern `LApp(t1',t',s',t2')` requires checking that `s = s'`. Similarly, the exact pattern X/σ where X is not a unification metavariable, used match-

ing a term t against an already-instantiated metavariable, requires checking that $X \equiv t$, implemented as a guard condition `lterm_equal t t'`. Last, some rules such as function formation need to apply freshening after matching.

Though we reuse the λHOL implementation in order to translate the related parts of VeriML expressions, this is not a hard requirement. We can replace the OCaml representation of λHOL terms – for example, we can translate them into terms of the HOL-Light system, which is also implemented in OCaml and uses a similar logic. Translating a VeriML tactic would thus result in an HOL-Light tactic. In this way, the VeriML implementation could be seen as a richly-typed domain-specific language for writing HOL-Light tactics.

8.3.3 Staging

Supporting the staging construct of VeriML introduced in Section 6.3 under the interpreter-based evaluation model is straightforward: prior to interpreting an expression e, we call the interpreter for the staged expressions e_s it encloses; the interpreter yields values for them v_s (if evaluation terminates successfully); the staged expressions are replaced with the resulting values v_s yielding the residual expression.

When evaluating VeriML terms through translation to OCaml, this is not as straightforward. The translation yields an OCaml AST – a value of a specific datatype encoding OCaml expressions. We need to turn the AST into executable code and evaluate it; the result value is then reified back into an OCaml AST, yielding the translation of the staged expression. This description essentially corresponds to staging for OCaml. The VeriML translator could thus be viewed as a stage-1 OCaml function, which evaluates stage-2 expressions (the translations of staged VeriML expressions) and yields stage-3 executable code (the translation of the resulting residual expressions).

Still, staging is not part of Standard ML and most ML dialects do not support it; the OCaml language that we use does not support it by default. Various extensions of the language such as MetaOCaml [Calcagno et al., 2003] and BER MetaOCaml have been designed precisely for this purpose but do not support other features of the language which we rely on, such as support for Camlp4. We address this issue by evaluating OCaml ASTs directly through the use of the OCaml toplevel library, which can be viewed as a library

that provides access to the OCaml interpreter. The OCaml AST is passed directly to the OCaml interpreter; it yields an OCaml value which is reified back into an OCaml AST. Reification requires being able to look into the structure of the resulting value and is thus only possible for specific datatypes, such as λHOL tuples, integers and strings; we disallow using the staging construct for expressions of other types that cannot be reified, such as functions and mutable references.

There are thus two distinct interpreters involved in the evaluation of a VeriML program: one is the interpreter used during translation from VeriML to OCaml, whose purpose is to execute staged VeriML expressions; another is the interpreter (or compiler) used after the translation is complete, in order to evaluate the result of the translation – the translation of the residual VeriML program. An interpreter has to be used for the first stage, a consequence of how we are side-stepping the lack of staging support in OCaml. Still, we can make sure that the interpretation overhead is minimized by using the OCamlJIT interpreter, and also by making sure that the interpreter is mostly used to make calls to already-compiled code. We achieve this by splitting our code into separate files and using separate compilation.

From the two interpreters involved in evaluation, the first one corresponds to the static evaluation semantics $\longrightarrow_s$; the second to dynamic evaluation $\longrightarrow$, as described in Section 6.3. The two interpreters do not share any state between them, contrary to the semantics that we have given. We partially address the issues arising from this by repeating top-level definitions in both interpreters, so that an equivalent state is established; we assume that staged VeriML expressions do not alter the state in a way that alters the evaluation of dynamic expressions.

8.3.4 Proof erasure

In order to be able to control whether an expression is evaluated under proof-erasure semantics or not, we make normal semantics the default evaluation strategy and include a special construct denoted as $\{e\}_{trusted}$ to evaluate the expression e under proof-erasure. We implement the proof-erasure semantics as presented in Section 6.4, as erasure proof objects and evaluation under the normal semantics. We change the translation of λHOL terms to be a type-directed translation. When a term sorted under *Prop* is found, that is, a term

corresponding to a proof object, its translation into an OCaml AST is simply the translation of the *admit* constant. More precisely, the translation is a conditional expression branching on whether we are working under proof erasure or not: in the former case the translation is the trivial *admit* constant and in the latter it is the full translation of the proof object. The exact mode we are under is controlled through a run-time mutable reference, which is what the $\{e\}_{trusted}$ construct modifies. In this way, all enclosed proof objects in e are directly erased.

8.3.5 VeriML as a toplevel and VeriML as a translator

We include two binaries for VeriML: a VeriML toplevel and a VeriML translator. The former gives an interactive read-eval-print-loop (REPL) environment as offerred by most functional languages. It works by integrating the VeriML-to-OCaml translator with the OCaml interpreter and is suitable for experimentation with the language or as a development aid. The latter is used to simply record the OCaml code yielded by the VeriML-to-OCaml translation. When combined with the OCaml compiler, it can be viewed as a compiler for VeriML programs.

The VeriML toplevel is also suitable for use as a proof assistant. When defining a new VeriML expression – be it a tactic, a proof script or another kind of computational expression – the VeriML type checker informs us of any type errors. By leaving certain parts of such expressions undetermined, such as a part of a proof script that we have not completed yet, we can use the type information returned by the type checker in order to proceed. Furthermore, we are also informed if a statically-evaluated proof expression fails. We have created a wrapper for the toplevel for the popular Proof General frontend to proof assistants [Aspinall, 2000], which simplifies significantly this style of dialog-based development.

Chapter 9

Examples and Evaluation

We will now present a number of examples that we have implemented in VeriML. We will follow the ideas from Chapter 7 in building a conversion rule that supports β-equivalence, congruence closure and arithmetic simplifications in successive layers. Such extensions to the conversion rule have required significant engineering effort in the past [e.g. Strub, 2010]; their implementation in current proof assistants makes use of techniques such as proof by reflection and translation validation [Barras et al., 2012]. The VeriML design allows us to express them in a natural style. We will also present a first-order automated theorem prover which will be used to simplify some of the proofs involved. We will use syntax that is close to the actual syntax used by our prototype implementation, with slight changes for presentation purposes.

9.1 Basic support code

The basic VeriML library includes definitions of Standard ML datatypes, such as bool, option and list, along with auxiliary functions over these. For example, the type of lists and the associated map and fold functions are defined in Code Listing 9.1, as well as the `option` type with associated functions and syntax.

We also define a number of simple proof-producing functions to lift λHOL proof object constructors to the computational level. We provide tactic syntactic sugar for them, as described in Chapter 8. For example, the function `eqElim` in Code Listing 9.2 lifts the

```
type list = fun a : ⋆ ⇒ rec t : ⋆ ⇒ Unit + a * t ;;
let nil = fun a ⇒ fold( inl(unit), list a ) ;;
let cons = fun a hd tl ⇒ fold( inr( (hd, tl) )) ;;

letrec listmap = fun a : ⋆ ⇒ fun b : ⋆ ⇒ fun f : a → b ⇒ fun l : list a ⇒
    match unfold l with
       inl u ↦ nil _
     | inr (hd, tl) ↦ cons _ (f hd) (listmap _ _ f tl) ;;

letrec listfold = fun (a b : ⋆) (f : b → a → b) start ⇒ fun l : list a ⇒
    match unfold l with
       inl u ↦ start
     | inr (hd, tl) ↦ f (listfold _ _ f start tl) hd ;;

type option = fun a : ⋆ ⇒ a + Unit ;;
let optionalt = fun a : ⋆ ⇒ fun t1 : option a ⇒ fun t2 : unit → option a ⇒
     match t1 with
        inl s ⇒ s
      | inr u ⇒ t2 () ;;
"e1 || e2" := optionalt _ e1 (fun u ⇒ e2) ;;

let optionforce = fun a : ⋆ ⇒ fun t : option a ⇒
     optionalt _ t (fun u => error);;

let optionret = fun a : ⋆ ⇒ fun t : a ⇒ inl( t );
let optionbind = fun a b : ⋆ ⇒ fun t : option a ⇒ fun f : a → option b ⇒
     match t with
        inl s ↦ f s
      | inr u ↦ inr () ;;
"do x1 ← e1; ...; xn ← en; return e" :=
    optionbind _ _ e1 (fun x1 ⇒
      ...
      optionbind _ _ en (fun xn ⇒
        optionret e) ... ) ;;
```

Code Listing 9.1: Definition of basic datatypes and associated functions

```
let eqElim =
  fun ϕ : ctx, P1 : @Prop , P2 : @Prop ,
    pf1 : hol(@P1) , pf2 : hol(@P1 = P2) ⇒
  let < H1 > = pf1 in
  let < H2 > = pf2 in
  < @conv H1 H2 > : hol(@P2) ;;

let intro = fun ϕ : ctx, T : @Type, P : [@, x : T].Prop ,
                 pf : hol([@, x : T].P) ⇒
    let < [ @, x : T ].pf > = pf in
    < @fun x : T ⇒ pf > : hol(@∀x.P) ;;
"Intro x : T in e" ::= intro @ @T (ν x : T in @?) (ν x : T in e)
"Intro x in e" ::= intro @ @? (ν x : ? in @?) (ν x : ? in e)

let assume = fun ϕ : ctx, P : @Prop, P' : [ @, x : P ].Prop ,
                 pf : hol([ @, x : P].P') ⇒
    let < [@, x : P ].pf > = pf in
    < @fun x : P ⇒ pf > : hol(@P1 → P2) ;;
"Assume H : P in e" ::= assume @ @? (ν H : P in @?) (ν H : P in e)
"Assume H in e" ::= assume @ @? (ν H : ? in @?) (ν H : ? in e)

let cutpf = fun ϕ : ctx, P : @Prop, P' : @Prop ,
    pf1 : hol( [@ , H : P'].P ), pf2 : hol( @P' ) ⇒
    let < [@ , H : P'].pf1 > = pf1 in
    let < pf2 > = pf2 in
    < @pf1/[id_ϕ, pf2 ] > : hol( @P ) ;;
"Cut H : P' by e1 in e2" ::= cut @ @? @P' (ν H : P' in e2) e1
"Cut H by e1 in e2" ::= cut @ @? @? (ν H : ? in e2) e1
```

Code Listing 9.2: Examples of proof object constructors lifted into VeriML tactics

EqElim typing rule (in Figure 3.10) from the level of λHOL proof objects to the level of VeriML proof expressions of type `hol([Φ]P)`.

The code uses concrete syntax that we have not discussed so far. We remind the reader that VeriML proof expressions are expressions of the type $\left([\Phi]\, P\right) \equiv (G : [\Phi]\, P) \times \mathsf{unit}$ where P is a proposition; that is, dependent tuples where the first argument is a proof object upon successful evaluation and the second is the unit. We write this type as `hol([Φ]P)` in the concrete syntax. Also, we use @ to refer to the ambient context, denoted as @ in the previous chapter. Following the same notation we write contextual terms $[\Phi]\, t$ where Φ is exactly the ambient context as $@t$. In this notation, @ binds as far to the left as possible, capturing all the following logical terms until another computational term is encountered.

The code of the function itself is straightforward: it opens up the dependent tuples, making the meta-variables `H1` and `H2` hold the proof objects contained therein; using these it creates a new proof object, which it wraps inside a dependent tuple. We introduce the following syntactic sugar for this tactic, in order to omit all arguments but the proof expressions:

```
"Exact e1 by e2" ::= eqElim @ @? @? e1 e2
```

In the rest we will often omit definitions of such syntactic shorthands; we will note implicit arguments with brackets instead. It is understood that these are then instantiated implicitly. We give more examples of such proof constructor lifting functions along with their syntactic sugar in Code Listing 9.2. In all functions note that when we open up dependent tuples, we annotate metavariables with their context unless it is the ambient context.

A proof expression of the $\forall P, Q : \mathit{Prop}. P = Q \rightarrow (P \rightarrow Q)$ theorem of higher-order logic is given as follows:

```
Theorem propeq_impl : ∀P,Q : Prop.P = Q → (P → Q) :=
   Intro P : Prop in Intro Q : Prop in
   Assume H1 : P = Q in
   Assume H2 : P in
   Exact < H2 > by < H1 > ;;
```

9.2 Extensible rewriters

In Section 7.1, we mentioned that we have implemented some infrastructure for a user-extensible definitional equality checker. We develop this infrastructure before implementing any non-trivial notion of definitional equality such as β-equality, as it will allow us to simplify such implementations. The code is based around rewriters and equality checkers: rewriters perform one step of a relation such as $t \longrightarrow_R t'$, simplifying t into the form t'; equality checkers decide whether two terms are equal. A rewriter can be turned into an equality checker through a function similar to defEqual as seen before, which essentially computes the symmetric, transitive, reflexive and compatible closure $=_R$ of $\longrightarrow_R$. This assumes that $\longrightarrow_R$ is confluent, something that we do not enforce.

Rewriters. The type of rewriters is defined as:

```
type rewriter_t = ( {φ : ctx} , {T : @Type}, e : @T ) →
                  ( e' : @T ) × hol( @e = e' ) ;;
```

Some rewriting relations might proceed by structural recursion. For example, rewriting to weak head-normal form with respect to β-reduction is defined as follows:

$$t\ t' \longrightarrow_\beta t''[t'/x] \text{ when } t \longrightarrow_\beta \lambda x.t''$$

Furthermore, we build rewriters by composing many rewriters together:

$$\longrightarrow_R = \longrightarrow_{R_1} \cup \cdots \cup \longrightarrow_{R_n}$$

It makes sense to use $\longrightarrow_R$ for the recursive calls used by any one of the $\longrightarrow_{R_i}$ rewriters. In that way, progress made by one rewriter will trigger further progress in another. Consider the following example which makes use of an alternate definition of addition:

$$(elimNat\ \lambda y.y\ (\lambda p.\lambda r.\lambda y.succ\ (r\ y))\ 0)\ 5$$

Though the above rule for β-reduction does not apply, a different rewriter $\longrightarrow_\mathbb{N}$ can reduce the function term to $\lambda y.y$. Then the β-reduction rule can make progress.

In order to support this, we write each $\longrightarrow_{R_i}$ rewriter in open recursion style, expecting the current 'full' $\longrightarrow_R$ rewriter as an argument. We call each $\longrightarrow_{R_i}$ rewriter as a *rewriter module*. Their type is:

```
let build_rewriter =
  fun rewriters_list : list rewriter_module_t ⇒
  letrec rewriter : rewriter_t =
      fun {φ T} e ⇒
      (let test_progress =
         fun res : < e' : @T > hol( @e = e' ) ⇒
         let < e' > = res in
         holmatch @e' as x return bool with
             @e ↦ false
           | @e' ↦ true
       in
       let idelem = < @e , @refl e > in
       let first_successful =
         listfold
           (fun res currewritemodule ⇒
               (if test_progress res then
                   res
                else
                   currewritemodule rewriter @e))
           idelem
           rewriters_list
       in
       if test_progress first_successful then
          let < e' , pfe'  > = first_successful in
          let < e'' , pfe'' > = rewriter @e' in
          < @e'' , @trans pfe' pfe'' >
       else
         first_successful)
   in
   rewriter ;;
```

Code Listing 9.3: Building a rewriter from a list of rewriter modules

```
let global_rewriter_modules = ref nil : ref (list rewriter_module_t) ;;
let default_rewriter = fun {ϕ T} e ⇒
  build_rewriter !global_rewriter_modules @e ;;

let global_rewriter_add = fun r ⇒
      global_rewriter_modules := listsnoc !global_rewriter_modules r ;;
let global_rewriter_add_many = fun r ⇒
      global_rewriter_modules := listappend !global_rewriter_modules r ;;
let global_rewriter_add_top = fun r ⇒
      global_rewriter_modules := cons r !global_rewriter_modules ;;
let global_rewriter_remove_top = fun u ⇒
      match !global_rewriter_modules with
         nil ↦ unit
       | cons(hd,tl) ↦ global_rewriter_modules := tl ;;
"Temporary Rewriter e1 in e2" ::=
    global_rewriter_add_top e1 ;
    let res = e2 in
    global_rewriter_remove_top (); res
```

Code Listing 9.4: Global registering of rewriter modules

```
type rewriter_module_t =

  rewriter_t →

  ( {ϕ : ctx} , {T : @Type}, e : @T ) →

  ( e' : @T ) × hol( @e = e' ) ;;
```

The relationship between full rewriters and rewriter modules corresponds to the relationship between the functions rewriteBetaNat and rewriteBetaNatStep presented in Section 7.1. That is, rewriter modules are expected to perform a single rewriting step, whereas full rewriters are expected to proceed until a normal form is reached.

We build a full rewriter from a list of rewriter modules through the code presented in Code Listing 9.3. Intuitively, the full rewriter $\longrightarrow_R$ tries to make progress towards a normal form by trying each rewriter module successively. The next step is given by the first rewriter module which does make progress, yielding a term t' which is not syntactically identical to the original term t. The process continues until no further progress can be made.

We define a global mutable list of rewriter modules and a default rewriter that uses `build_rewriter` to combine them together, as shown in Code Listing 9.4. We also provide some management functions, adding rewriters as the first or last one to be used, as the

order in which they are executed is important: for example, normalizing $0 + x$ takes one step through a rewriter that evaluates addition directly, but multiple steps if going through more basic reductions (unfolding of the definition of $+$, multiple β-steps and a natural number elimination step). Also, we define a function to remove the top element of the list, which is mostly used in order to define temporary rewriters that are only used within one block of code.

Equality checkers. The type of equality checkers is identical to the type of defEqual as presented in Chapter 7:

```
type equalchecker_t = ( {φ : ctx}, {T : @Type} , e1 : @T , e2 : @T ) →
                              option (hol(@e1 = e2)) ;;
```

We will not use the open recursion style shown above for equality checkers, so each module we develop should have the above type. The simplest example is a checker that can only decide syntactic equality between two terms as presented in Code Listing 9.5, lines 1 through 5.

The next example we develop is the "contextual closure" equality tester, which compares two terms structurally, using the rewriter at each step. This is mostly identical to the checker presented in Section 7.1, where the rewriteBetaNat function is used as the rewriter inside the defEqual equality checking function. We thus present a sketch of this function in Code Listing 9.5 and refer the reader to Section 7.1 for the missing pattern matching cases.

Since no equality-related automation is built up at this point, we need to provide explicit detailed proofs for the missing proof obligations in this function. The fact that we perform these proofs at such an early stage in our development will allow us to take them for granted in what follows and never reason explicitly about steps related to contextual closure again. The full proof objects follow:

```
G1 = < @conv (conv (refl (a1 b1)) (subst (x.a1 b1 = a1 x) pfb))
              (subst (x.a1 b1 = x b2) pfa) >
G2 = < @congLam (x:T''.pfa/[id_φ, x]) >
G  = < @conv (conv pf (subst (x. x = e2s) (symm pfe1)))
              (subst (y. e1 = y) (symm pfe2)) >
```

```
let equalchecker_syntactic : equalchecker_t =
  fun {φ T} e1 e2 ⇒
  holmatch @e2 as e2' return option(hol(@e1 = e2')) with
      @e1 ↦ some < @refl e1 >
    | @e2' ↦ none ;;

let equalchecker_closure =
  fun rewriter : rewriter_t ⇒
  letrec check_equal =  fun {φ T} e1s e2s ⇒
    let < e1, pfe1 > = rewriter @e1s in
    let < e2, pfe2 > = rewriter @e2s in
    do < pf > ←
       (holmatch @T, @e1, @e2 with
       | @T, @(a1 : T' → T) b1, @(a2 : T' → T) b2 ↦
         do < pfa > ← check_equal @a1 @a2 ;
            < pfb > ← check_equal @b1 @b2 ;
            return G1 : hol( @a1 b1 = a2 b2 )

       | @T'' → T', @λx : T''.a1 , @λx : T''.a2 ↦
         do < [@, x : T'']pfa > ← νx:T.check_equal @a1 @a2 ;
            return G2 : hol( @λx : T''.a1 = λx : T''.a2 ) ;;

       | ... );
       return G : hol( @e1s = e2s )
  in
    check_equal ;;
```

Code Listing 9.5: Basic equality checkers: syntactic equality and compatible closure

```
let equalchecker_compose_many = fun equalcheckers : list equalchecker_t ⇒
  fun {ϕ T} e1 e2 ⇒
  listfold (fun prf checker ⇒
            prf || (do return checker @e1 @e2))
  (equalchecker_syntactic @e1 @e2)
  equalcheckers ;;

let build_equalchecker = fun rewriter equalcheckers_list ⇒
  equalchecker_closure rewriter
                       (equalchecker_compose_many equalcheckers_list) ;;

let global_equalcheckers = ref nil ;;

let default_isequal =
  build_equalchecker default_rewriter !global_equalcheckers ;;
let require_isequal =
  fun {ϕ T} e e' ⇒
    match default_isequal @e @e' with
       some(prf) ↦ prf
     | none ↦ error ;;
```

Code Listing 9.6: Global registration of equality checkers and default equality checking

Just as in the case of rewriters, we create a default equality checker which combines a list of others. We register available equality checkers in a global mutable list. We expect each checker in the list to be a 'local' checker – that is, it does not form the compatible closure seen above on its own. Instead, we take the compatible closure after all local checkers are combined into one and make the resulting checker the default one. These are shown in Code Listing 9.6.

We also define syntactic sugar for calls to `require_isequal` where all arguments are inferred; also a form where the call happens statically, following the insights from Chapter 7, and syntax for other tactics that require equality proofs. In this way, `require_equal` becomes similar to the conversion rule yet can be extended using more rewriters and checkers.

```
"ReqEqual" ::= require_isequal @ @? @? @?
"StaticEqual" ::= {{ require_isequal @ @? @? @? }}
"Exact e" ::= Exact e by StaticEqual
"Cut H : e1 = e2 in e3" ::= Cut H : e1 = e2 by StaticEqual in e3
```

9.3 Naive equality rewriting

After having set up the infrastructure, we are ready to develop non-trivial equality checking. Our first concern is to have a way to take existing hypotheses into account. For example, if we already know that $t_1 = t_2$, the two terms should be indistinguishable for the rest of our reasoning, eliminating the need for explicit rewriting by the user. This sort of reasoning is ubiquitous and by making it part of the default equality checking at an early point, we simplify the implementation of later extensions.

We first develop a naive way of supporting such reasoning: we add a term rewriter that substitutes all occurrences of t_1 to t_2 – or the inverse – when a hypothesis $t_1 = t_2$ exists. In this way, terms come into a 'normal' form where the equality hypotheses have been taken into account before being checked for syntactic equality. This strategy has numerous shortcomings. For example, the presence of a cycle in the equality hypotheses leads to an infinite loop – consider the case where $x = y$ and $y = x$. It is also very inefficient and can only prove simple cases of using the hypotheses. The obvious benefit is its simplicity. The code for this naive rewriter is presented in Code Listing 9.7. The auxiliary function `gather_eqhyps` isolates equality hypotheses from the current context and records them into a list.

In our actual implementation we try both directions for rewriting, as each direction is able to prove a different set of facts. We add this rewriter to the default one through the `global_rewriter_add` function seen earlier. We are then able to directly prove simple lemmas such as the following:

```
Lemma test1 : ∀x,y,z,f,g.x = y → g = f → y = z → f x = g z :=
 Intro x y z f g in Assume H1 H2 H3 in
 ReqEqual ;;
```

9.4 Equality with uninterpreted functions

As mentioned above, the naive strategy for incorporating hypothesis in equality checking suffers from a number of problems. In this section we will present the code for a sophisticated procedure that does the same thing in an efficient and scalable way. We will use

```
type eqhyps_t = fun {ϕ : ctx} ⇒ list ( ( {T : @Type}, t1 : @T, t2 : @T ) ×
                                       hol( @t1 = t2 )) ;;

letrec gather_eqhyps : ( ϕ : ctx ) → eqhyps_t = fun {ϕ} ⇒
    holmatch ϕ with
        [ ϕ0, H : t1 = t2 ] ↦
         let @ = ϕ0 in
         let res = gatherhyps @ in
         ν H : t1 = t2 in
         cons < @t1, @t2, @H > res
      | [ ϕ0, H : (P : Prop) ] ↦ gatherhyps [ ϕ0 ]
      | [ ϕ0, t : (T : Type) ] ↦ gatherhyps [ ϕ0 ]
      | [ ϕ0, t : Type] ↦ gatherhyps [ ϕ0 ] ;;
      | [] ↦ nil

let rewriter_naive_eq : rewriter_module_t =
  letrec subst : ( {ϕ : ctx} , hyps : eqhyps_t , {T : @Type} ) →
                 ( t : @T ) → ( t' : @T ) × hol( @t = t' ) =
    fun {ϕ} hyps {T} t ⇒
    let res =
      listfold
      (fun cur elm ⇒
         cur || (let < T' , t1' , t2' , pf  > = elm in
                 holcase @T, @t with
                   @T' , @t1' ↦ do return < @t2' , @pf >
                 | @T'' , @t'' ↦ none))
     none eqhyps
    in
    match res with
      some v ↦ v | non ↦ < @t , @refl t >
  in
  fun recursive {ϕ T} e ⇒
      let hyplist = gather_eqhyps @ in
      subst hyplist @e ;;
```

Code Listing 9.7: Naive rewriting for equality hypotheses

an imperative union-find algorithm to construct equivalence classes for terms based on the available hypotheses. It also takes the form of terms into account, having special provisions for function applications: for example, if we know that $f\ x = y$, then the extra knowledge that $f = g$ will make all three terms $f\ x$, y and $g\ x$ part of the same equivalence class. Thus the equality checker we will construct is essentially a decision procedure for *congruence closure* or the theory of *equality with uninterpreted functions*, as this theory is referred to in the SAT/SMT literature. In fact our implementation is a direct VeriML adaptation of the standard textbook algorithm for this procedure [e.g. Bradley and Manna, 2007, Kroening and Strichman, 2008]. This showcases the fact that VeriML allows us to implement sophisticated decision procedures without requiring significant changes to the underlying algorithms and data structures.

Formally, the theory of equality with uninterpreted functions is the combination of the following theorems:

```
refl  : ∀x.x = x
symm  : ∀x,y.x = y → y = x
trans : ∀x,y,z.x = y → y = z → x = z
congApp1 : ∀f,x,y.x = y → f x = f y
congApp2 : ∀f,g,x.f = g → f x = g x
```

Intuitively, the automation functions we have developed already include explicit proofs involving these theorems. Ideally we would thus like not to have to repeat such proofs in the development of our decision procedure. We will show that this is indeed the case.

The basic idea of the algorithm is the following: every term t has a parent term t' which it is equal to. If the parent term t' is identical to the term t itself, then the term t is the representative of the equivalence class it belongs to. Therefore equivalence classes are represented as a directed tree of terms with nodes pointing towards their parents; the root of the tree is the representative of the equivalence class and has a self link. The `find` operation for checking whether two terms t_1 and t_2 are equal involves finding the representatives $t_{1,rep}$ and $t_{2,rep}$ for both by following parent links and comparing the representatives. The `union` operation for incorporating new knowledge about equality between two terms $t_1 = t_2$ consists of merging their equivalence classes, making the representative of either term a

child of the representative of the other term.

We implement the above description quite literally in the `union` and `find` functions presented in Code Listing 9.10 and Code Listing 9.9. The main departure from the standard implementation is that we add proofs to all the associated data structures. Union then needs a proof of $t_1 = t_2$ and find produces a proof that the returned representative t_{rep} is indeed equal to the original term t. Another adjustment we need to do is to represent the tree as a hash table instead of an array as done usually; hashing a term gives us an easy way to map terms to hash table entries (see Code Listing 9.8). In order to handle function applications correctly, we add another kind of edge called *congruence edge*, represented as sets that are part of every hashtable entry. These edges associate a term t with all other terms of the form $t'\ t$ or $t\ t'$ – that is, all other function applications which include t. In this way, when we handle a new equality hypothesis involving either t or t', we can access the terms where it appears and make further updates to the equivalence classes through the use of the theorems `congApp1` and `congApp2`. This is the main functionality of `merge` as given in Code Listing 9.12. It is informed of the congruent terms whose equivalence classes also need updating through the `congruent_eq` function of Code Listing 9.11. The final code for the equality checking module that is then registered globally is given in Code Listing 9.13.

We will briefly comment on some interesting points of this code. First, the hashtable implementation in Code Listing 9.8 is entirely standard, save for the fact that keys are arbitrary λHOL terms and values can be *dependent* on them. This is one of the cases where we make use of the fact that VeriML allows mixing logic-related terms and normal imperative data structures. The type of values is kept parametric for the hashtable implementation in order to allow reuse; more precisely, instead of being a simple computational type of kind `*`, it is a *type constructor* reflecting the dependency on the keys. In fact, the type of keys is parametric too, so that any type of λHOL domain objects can become an entry in the hashtable – this is the reason for the implicit `T` parameter. The parametric type of hashtables is then instantiated with the concrete value type constructor `eufval_t` in Code Listing 9.9. Based on this definition we see that every term in the hash table is associated with two mutable fields: one representing the parent, along with the required proof of equality; and another one representing the set of terms corresponding to the congruence

```
type hash_t = fun {φ : ctx} ⇒ fun val_t : ({T : @Type}, key : @T)→⋆ ⇒
    array (option (({T : @Type}, key : @T) × val_t @key))

let hashget = fun {φ} {val_t} (h : hash_t val_t) {T} key ⇒
    let hn = arraylen h in
    letrec find = fun i : int ⇒ match h.(i) with
         some < key', val > ↦ holcase @key' with
              @key ↦ some value
            | @key'' ↦ find ((i+1)%hn)
       | none |-> none
    in
    find ((hash < @key >) % hn) ;;

let hashget_force = fun {φ val_t} h {T} key ⇒
    optionforce (hashget h key) ;;

let hashset = fun {φ val_t} h {T} key (val : val_t @key) ⇒
    let hn = arraylen h in
    letrec find = fun i : int ⇒
        match h.(i) with
            some < key', val' > ↦ holcase @key' with
                 @key ↦ h.(i) := some < @key , val >
               | @key'' ↦ find ((i+1)%hn)
          | none ↦ h.(i) := some < @key , val >
    in
      find ((hash < @key >) % hn) ;;

type set_t = fun {φ : ctx} ⇒
   (hash_t (fun _ _ ⇒ Unit)) × (list ({T : @Type} × hol(@T))) ;;

let setadd = fun {φ} ((sethash,setlist) as set) {T} t ⇒
   match hashget sethash @t with
     some _ ↦ set
   | none ↦ hashset sethash @t ();
             ( sethash, cons < @t > setlist ) ;;
let setfold = fun {a φ} (f : ( {T} t ) → a → a) (start : a) (_, setlist) ⇒
   listfold (fun cur < t > ⇒ f @t cur) start setlist ;;
let setiter = fun {φ} (f : (  t ) → Unit) set ⇒
   setfold (fun {T} t u ⇒ f @t) () set ;;
let setunion = fun φ set1 set2⇒
   setfold (fun {T} t curset ⇒ setadd curset @t) set1 set2 ;;
```

Code Listing 9.8: Implementation of hash table with dependent key-value pairs and of sets of terms as pairs of a hashtable and a list

```
type eufval_t = fun {φ T} key ⇒
  (ref ( parent : @T ) × hol( @key = parent ) ) ×
  (ref set_t ) ;;

type eufhash_t = fun {φ} ⇒ hash_t eufval_t ;;

let syntactic_booleq = fun {φ T} e1 e2 ⇒
  match equalchecker_syntactic @e1 @e with
      some _ ↦ true | none ↦ false ;;

letrec find
: ({φ}, h:eufhash_t, {T:@Type}, t:@T) → (rep:@T) × hol(@t = rep)
= fun {φ} h {T} t ⇒
   let add_to_congedges = fun {T1} t1 {T2} t2 ⇒
     let < t1rep, _ > = find h @t1 in
     let (_, congedges) = hashget_force h @t1rep in
     congedges := setadd !congedges @t2
   in
   let rep_of_funapp
   : ({T T'}, f : @T → T', a : @T) → (farep : @T')×hol(@f a = farep)
   = fun {T T'} f a ⇒
     let < frep, fprf > = find h @f in
     let < arep, aprf > = find h @a in
     if (syntactic_booleq @f @frep) && (syntactic_booleq @a @arep) then
          < @f a, StaticEqual >
     else let < farep, faprf > = find h (@frep arep) in
          < @farep, StaticEqual >
   in
   match hashget h @t with
      some (paredge, congedges) ↦
         let < parent, prf > = !paredge in
         if syntactic_booleq @t @parent then !paredge
         else let < rep , prf' > = find h @parent in
              < @rep , StaticEqual >
    | none ↦ let congedges = ref emptyset in
         holmatch @t with
            @(f : T' → T) a ↦ let paredge = rep_of_funapp @f @a in
                              hashset h @t (ref paredge, congedges);
                              add_to_congedges @a @t;
                              add_to_congedges @f @t;
                              paredge

          | @t'' ↦ let selfedge = < @t , StaticEqual > in
                   hashset h @t (ref selfedge, congedges);
                   selfedge ;;
```

Code Listing 9.9: `find` – Finding the equivalence class representative of a term

```
let union = fun {φ} h ⇒
  fun {T} t1 t2 (prf : @t1 = t2) ⇒
  let < n1rep, prf1 > = find h @t1 in
  let < n2rep, prf2 > = find h @t2 in
  if syntactic_booleq @t1rep @t2rep then ()
  else let (paredge1, congedges1) = hashget_force h @t1rep in
       let (_, congedges2) = hashget_force h @t2rep in
       let paredge1' =
            < @t2rep , StaticEqual >
            : (t1par : @T)×hol(@t1rep = t1par)
       in
       paredge1 := paredge1';
       congedges2 := setunion !congedges1 !congedges2;
       congedges1 := emptyset ;;
```

Code Listing 9.10: `union` – Merging equivalence classes

```
let simple_eq
: ({φ}) → eufhash_t → ({T}, t1, t2) → option hol(@t1 = t2)
= fun {φ} h {T} t1 t2 ⇒
   let < t1rep, prf1 > = find h @t1 in
   let < t2rep, prf2 > = find h @t2 in
   holmatch @t1rep with
       @t2rep ↦ some StaticEqual
     | @t1rep' ↦ none ;;

let congruent_eq
: ({φ}) → eufhash_t → ({T}, t1, t2) → option hol(@t1 = t2)
= fun {φ} h {T} t1 t2 ⇒
  (simple_eq h t1 t2) ||
  holmatch @t1, @t2 with
     @(f1 : T' → T) a1, @(f2 : T' → T) a2 ↦
        do < pff > ← simple_eq h @f1 @f2;
           < pfa > ← simple_eq h @a1 @a2;
           return StaticEqual
   | @t1', @t2' ↦ none
```

Code Listing 9.11: `simple_eq`, `congruent_eq` – Checking equality based on existing knowledge

```
letrec merge
: (ϕ) → eufhash_t -> ( T, t1 : @T, t2 : @T , prf : @n1 = n2) → Unit
= fun {ϕ} h {T} t1 t2 H ⇒
  let mergeifneeded =
    fun {T1} t1 {T2} t2 ⇒
    holmatch @T1 with
      @T2 ↦ let < t1rep, _ > = find h @t1 in
              let < t2rep, _ > = find h @t2 in
              if not (syntactic_booleq @t1rep @t2rep) then
              match congruent_eq h @t1 @t2 with
                some < prf2 > ↦ merge h @t1 @t2 @prf2
              | none ↦ ()
    | @T1 ↦ ()
  in
  let < t1rep, _ > = find h @t1 in
  let < t2rep, _ > = find h @t2 in
  if syntactic_iseq @t1rep @t2rep then ()
  else let (_, congedges1) = hashget_force h @t1rep in
       let (_, congedges2) = hashget_force h @t2rep in
       union h @t1 @t2 @prf;
       setiter (fun T1 t1 ⇒
         setiter (fun T2 t2 ⇒ mergeifneeded @t1 @t2)
         congedges2)
       congedges1 ;;
```

Code Listing 9.12: `merge` – Incorporating a new equality hypothesis

```
let build_eufhash : (ϕ : ctx) → eufhash_t @ = fun ϕ ⇒
  let hyplist = gather_eq_hyps @ in
  let eufhash = emptyhash in
  listfold
    (fun _ < {T}, t1, t2, prf : @t1 = t2 > ⇒
       merge eufhash @t1 @t2 @prf)
    () hyplist;
  eufhash ;;

let equalchecker_euf : equalchecker_t =
  fun {ϕ T} t1 t2 ⇒
    let eufhash = build_eufhash @ in
    simple_eq eufhash @t1 @t2 ;;
```

Code Listing 9.13: `equalchecker_euf` – Equality checking module for EUF theory

```
1  let rewriter_beta : rewriter_module_t =
2    fun recursive {φ T} e ⇒
3    holmatch @e with
4      @(t1 : T' → T) t2 ↦
5          let < t1', pf1 > = recursive @t1 in
6          let < e', pfe' > =
7              holmatch @t1' as t1''
8                 return ( e' : @T ) × hol( @t1'' t2 = e' )
9              with
10                @fun x : T' ⇒ t1body ↦
11                   < @t1body/[id_φ, t2], @beta (fun x⇒t1body) t2 >
12             | @t1'' ↦ < @t1' t2 , StaticEqual >
13          in < @e', StaticEqual >
14
15     | ( t : @T ). @t ↦ < @t , StaticEqual > ;;
```

Code Listing 9.14: Rewriter module for β-reduction

edges.

Perhaps the most surprising fact about the code is that it does not involve any manual proof. All the points where a proof is required are handled with `StaticEqual`, which performs a statically-evaluated call to the default equality checking function. For example, in Code Listing 9.9, line 27, a proof of `f a = farep` is discovered, given `f = frep`, `a = arep` and `frep arep = farep`. This is where the code we have developed so far pays off: all the functions involved in our EUF equality checker are verified, yet without any additional manual proof effort – save for setting up the data structures in the right way. We have effectively separated the 'proving' part of the decision procedure in the naive rewriter presented earlier, from the 'algorithmic' part of the decision procedure presented here. We believe that further surface syntax extensions can bring the VeriML code even closer to the simply typed version minimizing the additional user effort required. One such extension would be to automatically infer implicit introduction and elimination points for dependent tuples.

After registering the EUF equality checker so that it is always used, the implementation of further tactics and decision procedures is simplified. It is more robust than the naive rewriter presented earlier, so users can rely on the equality checker being able to take equality hypotheses into account in most cases where they would help. For example, we can

use it to write a rewriter module for β-reduction, as presented in Code Listing 9.14. The call to `StaticEqual` in line 13 relies on the EUF checker in order to discover the involved proof. The meta-generation of rewriter modules for arbitrary lemmas from Section 8.3.1, which is used to minimize the effort required to develop modules such as `rewrite_beta`, also emits code that relies on the EUF checker.

9.5 Automated proof search for first-order formulas

We will shift our attention to automatically proving first-order formulas involving logical connectives such as conjunction and disjunction. We will present a simple prover based on a brute-force strategy with backtracking. The main function of the prover is `autoprove` in Code Listing 9.15; it uses the function `findhyp` to find a proof of a formula using the current hypotheses as presented in Code Listing 9.16; and it handles implications of the form $P_1 \to P_2$ through the `autoprove_with_hyp` function, presented in Code Listing 9.17.

The `autoprove` function itself is straightforward. It proceeds by pattern matching on the current goal, proving conjunctions by attempting to prove both propositions recursively and disjunctions by attempting to solve the first and backtracking to solve the second if that fails. Universal quantification starting in line 18 uses the ν construct in order to introduce a new logical variable in the current ϕ context, so that we can proceed recursively. Equality is handled through the current equality checking function. The `autoprove_with_hyp` function solves goals of the form $P_1 \to P_2$. We separate it so that we can handle cases where conjunction or disjunction appear in P_1.

In the default case, we attempt to solve the goal using the current hypotheses directly, or by proving absurdity from the current hypotheses. We use the `findhyp` function to do this from Code Listing 9.16, which either tries to find a hypothesis that matches the goal up to the current equality checking relation, or a hypothesis that matches the goal save for some extra premises. These premises are then solved recursively using `autoprove`. In order to avoid infinite loops that could arise from this behavior, we maintain a set of propositions that we have already attempted to solve in the `visited` argument of `autoprove`. The `findhyp` function utilizes the auxiliary `gatherhyps` function, which separates hypotheses from the

```
let autoprove
: ( {φ}, P : @Prop ) → set_t → option (hol(@P))
= fun {φ} P visited ⇒
  if setmember visited @P then none
  else
  holmatch @P with
      @P1 ∧ P2 ↦ do < prf1 > ← autoprove @P1 visited ;
                    < prf2 > ← autoprove @P2 visited ;
                    return < @andIntro prf1 prf2 >

    | @P1 ∨ P2 ↦ (do < prf1 > ← autoprove @P1 visited ;
                     return < @orIntro1 prf1 > ) ||
                 (do < prf2 > ← autoprove @P2 visited ;
                     return < @orIntro2 prf2 > )

    | @P1 → P2 ↦ autoprove_with_hyp @P1 @P2

    | @∀x : A, P' ↦
        do < [ @ , x : A ].prf > ← ν x : A in autoprove @P visited ;
           return < @fun x : A ⇒ prf >

    | @True ↦ some < @trueIntro >

    | @t1 = t2 ↦ default_equalchecker @t1 @t2

    | @P' ↦ let visited' = setadd visited @P in
              (findhyp @P visited') ||
              (do < absurd > ← findhyp @False visited' ;
                  return < @FalseInd absurd >) ||
              (setremove visited @P; none) ;;

"Auto" := match autoprove @ @? emptyType with
            some prf → prf | none → error ;;
```

Code Listing 9.15: `autoprove` – Main function for automated proving of first-order formulas

```
type hyplist_t = fun {ϕ : ctx} ⇒ list (( H : @Prop ) × hol( @H )) ;;

let gatherhyps : (ϕ) → hyplist_t =
  fun ϕ ⇒
  holmatch @ with
      [] ↦ nil
    | [ ϕ0 , H : (P : Prop) ] ↦
        let @ = ϕ0 in
        let hyps = gatherhyps @ in
        let < P', prfeq > = default_rewriter @P in
        ν H : P in
        cons < @P' , {{ Exact H }} > hyps
    | [ ϕ0, t : (T : Type) ] ↦ gatherhyps ϕ0
    | [ ϕ0, t : Type] ↦ gatherhyps ϕ0 ;;

let findhyp
: ( {ϕ}, Goal : @Prop ) → set_t → option (hol( @Goal ))
= fun {ϕ} Goal visited ⇒
  let hyps = gatherhyps @ in
  listfold
  (fun curprf < Hyp : @Prop, hyp : @Hyp > ⇒
     (curprf) ||
     (do < prfeq > ← default_equalchecker @Hyp @Goal ;
         return (Exact @hyp)) ||
     (holmatch @Hyp with
         @P → Q ↦
           do < prfeq > ← default_equalchecker @P→Q @P→Goal ;
              < prfP > ← autoprove @P visited ;
              return (Exact @hyp prfP)) ||
     (holmatch @Hyp with
         @P1 → P2 → Q ↦
           do < prfeq > ← default_equalchecker @P1→P2→Q @P1→P2→Goal ;
              < prfP > ← autoprove @P1∧P2 visited ;
              return (Exact @hyp (andElim1 prfP) (andElim2 prfP))))
  none l
```

Code Listing 9.16: `findhyp` – Search for a proof of a goal in the current hypotheses

```
1  let autoprove_with_hyp
2  : ( {φ}, H : @Prop, G : @Prop ) → option (hol(@H → G))
3  = fun {φ} H G ⇒
4    holmatch @H with
5        @P1 ∧ P2 ↦ do < prf > ← autoprove_with_hyp @P1 @P2 → G ;
6                      return < @fun h ⇒ prf (andElim1 h) (andElim2 h) >
7
8      | @P1 ∨ P2 ↦ do < prf1 > ← autoprove_with_hyp @P1 @G ;
9                      < prf2 > ← autoprove_with_hyp @P2 @G ;
10                     return < @fun h ⇒ orInd prf1 prf2 h >
11
12     | @H' ↦
13         do < [@, h : H].g > ← ν h : H in autoprove @G emptyset ;
14            return < @fun h ⇒ g >
```

Code Listing 9.17: `autoprove_with_hyp` – Auxiliary function for proving a goal assuming an extra hypothesis

current ϕ context and stores them into a list so that they can be further manipulated. This is mostly similar to `gather_eqhyps` as seen earlier.

Using this automated prover we can solve goals that involve the basic first-order connectives. For example, the following lemma can be proved automatically thanks to the logical reasoning provided by `autoprove` and the equality reasoning provided through `equalchecker_euf`.

```
Lemma test1 : ∀x,y,z.(x=y∧y=z) ∨ (z=y∧y=x) → x = z := Auto ;;
```

9.6 Natural numbers

We will now present the VeriML implementation of the theory of natural numbers including the addition and multiplication functions, along with their algebraic properties. The type of natural numbers is defined as an inductive type in λHOL as shown in Code Listing 9.18, lines 1 through 3. This definition generates a number of constants to perform elimination and induction on natural numbers following the description of Subsection 8.2.2. Elimination is governed by the two generated rewriting lemmas `NatElimzero` and `NatElimsucc`; we add these to the default equality checking procedure through rewriter meta-generation as described in Section 8.3.1. In this way when using functions defined through elimination,

```
Inductive Nat : Type :=
  zero : Nat
| succ : Nat → Nat ;;

(* generated constants:
 NatElim : [T : Type]T → (Nat → T → T) → Nat → T;;
 NatElimzero : [T : Type]∀fz fs, NatElim fz fs zero = fz;;
 NatFoldsucc : [T : Type]∀fz fs n, NatElim fz fs (succ n) =
                                      fs n (NatElim fz fs n);;
 NatInd : ∀P : Nat → Prop, P zero → (∀n,P n → P (succ n)) → ∀n,P n;; *)

global_rewriter_add (GenerateRewriter NatElimzero);;
global_rewriter_add (GenerateRewriter NatElimsucc);;

let nat_induction =
   fun φ (P : [φ, x : Nat].Prop)
       (pf1 : hol(@P/[id_φ,0]))
       (pf2 : hol(@∀x : Nat, P/[id_φ,x] → P/[id_φ, succ x]))
       (spf1 : hol(@?)) (spf2 : hol(@?)) (spf3 : hol(@?)) ⇒
       let < pf1' > = Exact pf1 by spf1 in
       let < pf2' > = Exact pf2 by spf2 in
       (Exact < @NatInd (fun x ⇒ P) pf1' pf2' > by spf3)
       : hol( @∀x : Nat, P ) ;;

"NatInduction for x.P base case by e1 inductive case by e2" :=
  nat_induction @ (νx in @P) e1 e2 StaticEqual StaticEqual StaticEqual ;;
```

Code Listing 9.18: Definition of natural numbers and natural induction tactic

such as addition and multiplication, their result will be computed during equality checking. These are shown in lines 5 through 13.

We next define a tactic that simplifies the use of the natural number induction principle. We need this so that the types of the base and inductive cases do not include β-redeces in the common case. Such redeces arise in most cases of using natural number induction because the induction principle is represented as a *Nat* $\rightarrow$ *Prop* predicate, usually of the form `fun x` $\Rightarrow$ `P`. The lack of β-reduction support in the definitional equality of λHOL means that an application of this predicate such as `(fun x` $\Rightarrow$ `P) 0` has to be explicitly proved to be equal to `P/[id, 0]`. The induction tactic uses the default equality checking function in order to prove the equality between such forms, allowing the user to consider the reduced forms for the base and inductive case.

A surprising thing about `nat_induction` is that it does not directly use the `StaticEqual` tactic to solve the involved equalities in its code. The essential reason is that the positions where this tactic would be used, in all applications of the `Exact` tactic, do not fall under the constraints imposed by the collapsing transformation described in Section 5.2: they include non-identity substitutions for the metavariable P. Thus this transformation cannot be directly used so that `ReqEqual` is called statically inside the code of `nat_induction`. Yet we expect that in most cases where the induction tactic is used the constraints of the collapsing transformation will indeed be satisfied and the involved proofs can be statically discovered at those points. We therefore transfer the requirement for the equality proofs from the place where the induction tactic is defined to all the places where it is used. Through suitable syntactic sugar, this additional requirement is hidden from the user.

In Code Listing 9.19, we present the definition and algebraic properties of the natural number addition function. The proofs of the properties are mostly automated save for having to specify the induction principle used. The fact that the induction principle has to be explicitly stated in full stems from a limitation in the type inference support of the VeriML prototype. The automation comes from the extensible equality checking support: as soon as a property is proved it is added to the global default rewriter, which enables equality checking to prove the required obligations for further properties. Thus by proving the lemmas in the right order, we can limit the amount of the required manual effort.

```
Definition plus : Nat → Nat → Nat := fun x : Nat ⇒ fun y : Nat ⇒
  NatElim y (fun p r ⇒ succ r) x ;;

Temporary Rewriter (Unfolder plus) in {
Theorem plus_Z_y  : ∀y,0 + y = y := Auto ;;
Theorem plus_Sx_y : ∀x y,(succ x) + y = succ (x + y) := Auto ;;
}

global_rewriter_add (GenerateRewriter plus_Z_y);;
global_rewriter_add (GenerateRewriter plus_Sx_y);;

Theorem plus_x_Z  : ∀x : Nat, x + 0 = x :=
  NatInduction for x. x + 0 = x
   base case by Auto
   inductive case by Auto ;;

Theorem plus_x_Sy : ∀(x y : Nat), x + (succ y) = succ (x + y) :=
  Intro x y in
  Instantiate
     (NatInduction for z. z + (succ y) = succ (z + y)
                 base case by Auto
                 inductive case by Auto)
  with @x ;;

global_rewriter_add (GenerateRewriter plus_x_Z);;
global_rewriter_add (GenerateRewriter plus_x_Sy);;

Theorem plus_comm : ∀(x y : Nat), x + y = y + x :=
  Intro x y in
  NatInduction for y. x + y = y + x
     base case by Auto
     inductive case by Auto ;;

Theorem plus_assoc : ∀(x y z : Nat), x + (y + z) = (x + y) + z :=
  Intro x y in
  NatInduction for z.x + (y + z) = (x + y) + z
     base case by Auto
     inductive case by Auto ;;

global_equalchecker_add (equalchecker_comm_assoc plus_comm plus_assoc) ;;
```

Code Listing 9.19: Addition for natural numbers

After proving the commutativity and associativity properties of addition, we use a generic commutativity and associativity equality module to add them to the global equality checking procedure. The type of the generic module is:

```
equalchecker_comm_assoc : ({T : @Type}, {op : @T → T → T}) →
            (op_comm : @∀x y, op x y = op y x) →
            (op_assoc : @∀x y z,op x (op y z) = op (op x y) z) →
            equalchecker_t ;;
```

We do not give the code of this function here. It proceeds by pattern matching on terms, ordering applications of the `op` argument (here instantiated as the addition function `plus`), based on the hash values of their arguments. It yields a normalized form for terms that are equivalent up to associativity and commutativity. This function resembles the naive equality rewriter of Section 9.3 and suffers from similar shortcomings: it can only prove simple cases where associativity and commutativity apply and it implements a very inefficient strategy.

The theory of multiplication is similar and is presented in Code Listing 9.20. We have elided the global rewriter registration calls for presentation purposes; they directly follow the addition case. The main difference with addition is that not all proofs are automatic: in the theorems `mult_x_Sy` and `mult_plus_distr`, the equality checking support is not enough to prove the inductive case. We have to specify explicitly that terms should be rewritten based on the induction hypothesis instead – using a tactic similar to the naive rewriter of Section 9.3, which only rewrites terms based on a given equality proof. To see why this is necessary, consider the following proof obligation from `mult_x_Sy`, line 17:

$$\frac{IH : z * (succ\ y) = z + (z * y)}{G : (succ\ z) * (succ\ y) = (succ\ z) + ((succ\ z) * y)}$$

Equality checking proceeds as follows. First we rewrite the left-hand side and right-hand

```
Definition mult : Nat → Nat → Nat := fun x : Nat ⇒ fun y : Nat ⇒
   NatElim 0 (fun p r ⇒ y + r) x ;;

Theorem mult_z_y  : ∀y : Nat, 0 * y = 0 := Auto ;;
Theorem mult_Sx_y : ∀(x y : Nat), (succ x) * y = y + (x * y) := Auto ;;

Theorem mult_x_z : ∀x : Nat, x * 0 = 0 :=
 NatInduction for x. x * 0 = 0
     base case by Auto
     inductive case by Auto ;;

Theorem mult_x_Sy : ∀x y, x * (succ y) = x + (x * y) :=
 Intro x y in
 Instantiate
 (NatInduction for z. z * (succ y) = z + (z * y)
     base case by Auto
     inductive case by ( Intro z in Assume IH in Rewrite @IH in
                         Auto ) )
 with @x ;;

Theorem mult_comm : ∀x y, x * y = y * x :=
  Intro x in
  NatInduction for y. x * y = y * x
  base case by Auto
  inductive case by Auto ;;

Theorem mult_plus_distr : ∀x y z, x * (y + z) = (x * y) + (x * z) :=
  Intro x y z in
  Instantiate
    (NatInduction for a. a * (y + z) = (a * y) + (a * z)
     base case by Auto
     inductive case by ( Intro a in Assume IH in Rewrite @IH in
                         Auto ) )
  with @x ;;

Theorem mult_assoc : ∀x y z, x * (y * z) = (x * y) * z :=
  Intro x y in
  NatInduction for z.x * (y * z) = (x * y) * z
  base case by Auto
  inductive case by Auto ;;
```

Code Listing 9.20: Multiplication for natural numbers

side of the equality to normal form based on the current default rewriter:

$$\begin{aligned}
\text{(LHS) } & (succ\ z) * (succ\ y) \longrightarrow && \texttt{(mult_Sx_y)} \\
& (succ\ y) + z * (succ\ y) \longrightarrow && \texttt{(plus_Sx_y)} \\
& succ\ (y + z * (succ\ y)) &&
\end{aligned}$$

$$\begin{aligned}
\text{(RHS) } & (succ\ z) + ((succ\ z) * y) \longrightarrow && \texttt{(plus_Sx_y)} \\
& succ\ (z + (succ\ z) * y) \longrightarrow && \texttt{(mult_Sx_y)} \\
& succ\ (z + y + z * y) &&
\end{aligned}$$

The current equality checking cannot decide these two normal forms to be equal. In order for the EUF procedure to take the induction hypothesis into account, there would need to be an explicit $z + z * y$ subterm in the RHS. Such a term would only appear if we rewrite the RHS into $y + z + z * y$ based on commutativity, yet the naive commutativity support does not have any sophistication so as to perform this commutativity step; rewriting into this form would be just a coincidence based on the hash values of the terms and we should not therefore depend on it. What we do instead is to force a rewriting step in the LHS based on the inductive hypothesis, resulting in the equivalent term $succ\ (y + z + z * y)$. Now the naive commutativity support will rewrite both the LHS and RHS into the same normal form and syntactic equality yields the desired equality proof.

The distributivity property of multiplication with regards to addition requires no special provision in terms of equality checking support; the default meta-generated rewriter for it is enough to handle cases where it needs to be used.

9.7 Normalizing natural number expressions

We will now develop a more sophisticated rewriter that normalizes arithmetic terms up to associativity and commutativity of addition and multiplication. The rewriter works as follows: given an arithmetic term, it constructs a list of monomials of the form $a \cdot x$ where a is a constant expression and x is a variable or a product of variables (`nat_to_monolist` in Code Listing 9.23); the list is sorted so that monomials on the same variable become adjacent

```
Inductive List [ T : Type ] : Type :=
   nil : List
 | cons : T → List → List ;;

Definition append [T : Type] : List/[T] → List/[T] → List/[T] :=
   fun x y ⇒ ListElim y (fun hd tl r ⇒ cons hd r) x ;;

Definition sumlist : List/[Nat] → Nat :=
   ListElim 0 (fun hd tl r ⇒ hd + res) ;;

Lemma sum_append : ∀l1 l2,
    sumlist (append l1 l2) = (sumlist l1) + (sumlist l2)

Inductive removed [T : Type] : List/[T] → T → List/[T] → Prop :=
   removedHead : ∀hd tl, removed (cons hd tl) hd tl
 | removedTail : ∀elm hd tl tl', removed tl elm tl' →
                                  removed (cons hd tl) elm (cons hd tl') ;;

Lemma removed_sum : ∀l elm l',
    removed l elm l' → sumlist l = elm + (sumlist l')

Inductive permutation [T : Type] : List/[T] → List/[T] → Prop :=
   permutationNil  : permutation nil nil
 | permutationCons : ∀hd l1' l2 l2', removed l2 hd l2' →
                                      permutation l1' l2' →
                                      permutation (cons hd l1') l2 ;;

Theorem permutation_sum : ∀l1 l2,
    permutation l1 l2 → sumlist l1 = sumlist l2

Lemma permutation_removed [T : Type] : ∀l1 elm l1' l2',
    removed l1 elm l1' → permutation l1' l2' →
                         permutation l1 (cons elm l2')

Theorem permutation_symm [T : Type] : ∀l1 l2,
    permutation l1 l2 → permutation l2 l1
```

Code Listing 9.21: Definition of lists and permutations in λHOL, with associated theorems

(`sort_list` in Code Listing 9.22); consecutive monomials on the same variable are factorized (`factorize_monolist` in Code Listing 9.24); and the list of monomials is converted back into an arithmetic expression. These auxiliary functions need to be partially correct so that the resulting rewriter can prove that rewriting a term t to t' based on this sequence of steps is actually valid. For example, the `nat_to_monolist` function has to return a monomials list that, when summed, is equivalent to the original number; and sorting a list of monomials needs to return the same list, albeit permuted. Note that the second property of sorting, namely the fact that the returned list is ordered, is not important for our purposes.

In order to be able to specify and reason about the behavior of these functions, we need a definition of lists at the level of the logic. This is shown in Code Listing 9.21. Contrast this definition with the definition of lists at the computational level of VeriML, as seen earlier in Code Listing 9.1: the logic definition allows us to reason about lists, describing the behavior of list-related functions at the logic or computational level; whereas the computational definition allows us to define lists of arbitrary VeriML values, including mutable references and contextual terms but precludes reasoning about their behavior. The effort associated with having to duplicate definitions can be mitigated by having logic definitions implicitly define the equivalent computational versions as well.

In Code Listing 9.21 we also give the definitions of the `append` function at the logical level and of `sumlist` which accumulates together the members of lists of natural numbers. We also define the predicate of a list being a permutation of another, as well as the auxiliary predicate `removed l1 elm l1'`. This stands for the property "if the element `elm` is removed from the list `l1`, the resulting list is `l1'`". Last, we give statements of lemmas about the interaction of these functions. We do not give the proofs here; these follow the ideas presented so far. The proof that presents the most difficulty is the `permutation_symm` theorem, taking 50 lines in total. It requires proving an inversion lemma for permutations and a number of manual proof steps. This proof could be significantly improved by meta-generation of inversion principles, as well as the development of automation for user-defined predicates following the ideas presented for the equality predicate; we leave these matters to future work.

In Code Listing 9.22 we present the sorting function for lists. This is a *computational*

```
letrec min_list
: ({ϕ : ctx}, {T : @Type}, cmp : (@T) → (@T) → bool, l : @List) →
  (min : @T) × (rest : @List) × hol(@removed l min rest)
= fun {ϕ T} cmp l ⇒
   let < @l' , @pfl' > = default_rewriter @l in
   let < @min, @rest, @pf > =
     holmatch @l' with
       @nil ↦ error
     | @cons hd nil ↦ < @hd , @nil , @removedHead ? ? >
     | @cons hd tl ↦
       let < min', rem, pf > = min_list cmp @tl in
       if (cmp @hd @min' ) then
          < @hd , @tl, Exact @removedHead hd tl >
       else
          < @min', @cons hd rem, @removedTail pf >
   in < @min , @rest , {{ Auto }} > ;;


let sort_list
:  ({ϕ : ctx}, {T : @Type}, cmp : (@T) → (@T) → bool, l : @List) →
   (l' : @List) × hol (@permutation l l')
= fun {ϕ T} cmp l ⇒
   let < lnorm , pflnorm > = default_rewriter @l in
   let < l' , pfl' , u > =
    holmatch @lnorm with
      @nil ↦ < @nil , @permutationNil >
    | @cons hd tl ↦
       let < newhd, newtl, prf1 > = min_list cmp (@cons hd tl) in
       let < newtl', prf2  > = sort_list cmp @newtl in
       let < prf2' > = < @permutation_symm prf2 > in
       < @cons newhd newtl' ,
         @permutation_symm (permutationCons prf1 prf2') >
   in < @l', {{ Auto }} > ;;
```

Code Listing 9.22: Selection sorting for lists

```
let is_const = fun {phi} n ⇒
  holmatch @n with @const/[] ↦ true | @n'' ↦ false ;;

let nat_to_monolist
: ({ϕ : ctx} , n : @Nat) → ( l : @List ) × hol( @n = sumlist l ) ;;
= fun {ϕ} n ⇒ holmatch @n with
      @a + b ↦
         let < l1, pf1 > = nat_to_monolist @a in
         let < l2, pf2 > = nat_to_monolist @b in
         < @append l1 l2 , {{ Cut @sum_append l1 l2 in Auto }} >

    | @a * b ↦
         if is_const @a && is_const @b then
            < @cons ((a * b) * 1) nil , {{ Auto }} >
         else if is_const @a then
            < @cons (a * b) nil , {{ Auto }} >
         else if is_const @b then
            < @cons (b * a) nil , {{ Auto }} >
         else
            < @cons (1 * (a * b)) nil , {{ Auto }} >

    | @e ↦ if is_const @e then
                < @cons (e * 1) nil , {{ Auto }} >
             else
                < @cons (1 * e) nil , {{ Auto }} > ;;

let cmp = fun ϕ e1 e2 ⇒
  holmatch @e1, @e2 with
      @a1 * b1, @a2 * b2 ↦ ( hash @b1 ) < ( hash @b2 )
    | @e1 , @e2 ↦ ( hash @e1 ) < ( hash @e2 )
```

Code Listing 9.23: Rewriter module for arithmetic simplification (1/2)

```
let factorize_monolist
: ({ϕ : ctx}, l : @List) → (l' : @List) × hol( @sumlist l = sumlist l' )
= fun {ϕ n} l ⇒ holmatch @l with
    @cons (hda * hdb) tl ↦
       let < tl' , pftl' > = factorize_monolist @tl in
       holmatch @tl' with
           @cons (hdc * hdb) tl'' ↦
                < @cons ((hda + hdc) * hdb) tl'' ,
                  {{ Rewrite @pftl' in ReqEqual }} >
         | @tl' ↦ < @cons (hda * hdb) tl' , StaticEqual >
  | @l ↦ < @l , StaticEqual > ;;

let rewriter_arithsimpl : rewriter_module_t =
  fun recursive {ϕ T} e ⇒
  holmatch @e with
    @e1 + e2 ↦
        let < l0, pf1 > = nat_to_monolist @e1+e2 in
        let < l1, pf2 > = sort_list cmp @l0 in
        let < l2, pf3 > = factorize_monolist @l1 in
        < @sumlist l2, {{ Cut @permutation_sum l0 l1 in Auto }} >
  | @e ↦ < @e , StaticEqual > ;;
```

Code Listing 9.24: Rewriter module for arithmetic simplification (2/2)

function that manipulates *logical* lists. The reason behind this choice is that we want to sort lists based on the hash of their elements – that is, based on an extra-logical operation –, yet we also need a proof that sorting returns a permutation of the original list – therefore the choice of logical lists. We leave the comparison operation as a parameter to the sorting function. The sorting algorithm we implement is selection sort; we use the auxiliary function `min_list` to remove the minimum element of a list. In both functions we provide manual proofs of most obligations, using the definitions and lemmas of Code Listing 9.21. A point worth mentioning is that we need to explicitly rewrite the input lists to head-normal form before pattern matching is performed, as matching happens only up to syntax and not up to β-equivalence or evaluation of logical functions such as append.

The rest of the code for arithmetic simplifications is presented in Code Listing 9.23 and 9.24, following the description given earlier. Sorting of monomial lists is done using the comparison function `cmp`. This function only uses the hash value of the variable component x, thus making monomials on the same variable adjacent in the sorted list. It is worth

	Total lines of code	Logic definitions, proofs, hints
Basic library	131	16
Basic connectives	36	36
Basic rewriting support	191	19
Naive equality rewriter	79	2
Hash tables and sets (Prop and Type)	69x2=138	0
EUF rewriter	200	0
β-rewriting	21	1
Auto prover	178	14
Naive comm./assoc. rewriting	111	17
Force rewriting on hypothesis	56	7
Theory of natural numbers	128	128
Lists, permutations and sorting	200	158
Arithmetic simplification	84	3
Full code involved	**1581**	**401**

Table 9.1: Line counts for code implemented in VeriML

	Total code	Manual proof
VeriML		
EUF conversion	279	2
Arithmetic simplification	395	178
Natural numbers theory	128	128
Coq		
Congruence	1437	62
Ring simplification	1997	846
Abstract ring theory	259	259

Table 9.2: Comparison of line counts for EUF rewriting and arithmetic simplification between VeriML and Coq

noting that most of the proof obligations for these functions are solved automatically for the most part –an additional hint is required in some cases– and are validated statically following the ideas we have seen earlier.

9.8 Evaluation

This concludes our presentation of the examples we have implemented in VeriML. It is worth noting that the full code of the examples mentioned above is about 1.5k lines of VeriML code, including all the infrastructure and basic library code we describe. We offer

a breakdown of the line counts for the different modules in Table 9.1. In the same table we note the line counts for the proof-related aspects of the code, which include all logic definitions, proofs, hints (as additions of meta-generated rewriters) and induction tactics, in order to give an accurate portrayal of the λHOL-related reasoning that the programmer has to do. It is worth noting that the vast majority of the proof-related code corresponds to defining and proving the theory of natural numbers, lists and permutations. The actual conversion implementation functions have minimal manual proof involved.

It is worth contrasting these line counts with the two tactics in Coq that are roughly equivalent in functionality to our implementation of the EUF and arithmetic simplification conversion rules, namely `congruence` and `ring_simplify` respectively. This comparison is shown in Table 9.2. For EUF conversion in VeriML, we have included the naive equality rewriter as well as the main EUF module; for arithmetic simplification, we have included the naive commutativity and associativity rewriter, the theory of lists and the arithmetic simplification procedure. In both of these cases some of the code is reusable: e.g. the theory of lists consists of theorems that are useful independently of arithmetic simplification, and the code of the naive rewriter is mostly reused for implementing the forced rewriting procedure. In computing the line counts for the Coq versions, we have manually isolated the proof-related lines, such as calls to tactics and CIC constructors inside ML code in the case of the congruence tactic, and the lemmas proved using LTac proof scripts in the case of ring simplification. In the same table, we have also included the counts for our development of the theory of natural numbers, as well as the theory of abstract rings used by `ring_simplify` which are composed of roughly the same theorems.

The VeriML versions of the tactics and proofs compare favorably to their Coq counterparts; it is telling that the manual proofs required for all of our VeriML code, including basic logic definitions and tactics, is half the size of the proofs required just for arithmetic simplification in Coq. In arithmetic simplification, the Coq proofs are about 5 times larger than their VeriML equivalent; in the EUF procedure, the ratio is 30:1, or 3:1 if we include the basic rewriting support code of VeriML (which is what accounts for the congruence properties). The main reason behind the significant reduction in manual proof size for the VeriML versions is the way we implement our proofs and proof-producing functions, making

use of the extensible conversion support. As for the overall reduction in code size, the main reason is that we do not have to go through encodings of logical terms such as the ones used by proof-by-reflection or translation validation, as required by the Coq versions, but can rather work on the exact terms themselves. Of course the comparison of line counts for the programming aspect of the procedures cannot provide accurate grounds for drawing conclusions. This is especially true since the compared tactics are for different logics, use more realistic and full-featured implementations of the algorithms we describe (e.g. congruence takes injectivity of constructors into account) and the performance is not yet comparable, owing largely to our prototype VeriML implementation. Still, the difference in manual proof effort required indicates the savings that VeriML can offer.

Chapter 10

Related work

There is a large body of existing related literature that the work in this dissertation rests upon. The design of VeriML is such that it can be viewed as either a programming language tailored to tactic programming or as a complete proof assistant. We will therefore compare VeriML to other tactic development methodologies as well as existing architectures of proof assistants. When viewed as a programming language, the core departure of VeriML from traditional functional programming languages such as ML and Haskell is the availability of constructs for type-safe manipulation of an object language; we will present a number of existing language designs that contain similar constructs. Last, VeriML represents a particular way of combining a logic with a side-effectful programming language, where a strong separation between the two languages is maintained. We will compare this approach with other potential combinations.

10.1 Development of tactics and proof automation

Tactic development in ML. Most modern proof assistants, such as HOL [Slind and Norrish, 2008, Harrison, 1996], Isabelle [Paulson, 1994], Coq [Barras et al., 2012] and NuPRL [Allen et al., 2000], allow users to extend the set of available tactics, by programming new tactics inside the implementation language of the proof assistant. In all cases this language is a dialect of Standard ML [Milner, 1997]. In fact, ML was designed precisely for this purpose in the context of the original LCF system [Gordon et al., 1979] – as the **M**eta

Language of the system, used to manipulate the terms of the logic – the object language. Developing tactics in ML is thus the canonical methodology used in the direct descendants of this system such as HOL.

The programming constructs available in ML were chosen specifically with tactic development in mind, making the language well-suited to that purpose. For example, higher-order functions are available so that we can define tacticals – functions that combine other tactics (which are functions themselves). The programming model available in ML is very expressive, allowing general recursion, mutable references and other side-effectful operations; this enables the development of efficient tactics that make use of sophisticated algorithms and data structures such as discrimination nets [Gordon, 2000].

The main problem with tactic development in ML is that it is essentially *untyped*. Logical terms and proofs are encoded within ML as algebraic data types. Data types are simply-typed in ML, effectively identifying all proofs and all terms at the level of types and precluding any extra information – such as the context of free variables they depend on, or the kinds of terms, or the consequence of proofs – from being available statically. This hurts the composability of tactics and their maintainability as well, as we cannot give precise signatures to the kinds of arguments they expect, the results they produce and the relationships between them. VeriML can be viewed as a dependently-typed extension of ML that addresses this shortcoming, by making the type of proofs and logical terms dependent on the information produced by the type system of the logic. The relationship between programming tactics in VeriML instead of ML is therefore similar to the relationship of developing functional programs in ML instead of Lisp.

For example, the VeriML type we gave to rewriters is the following:

$$(\phi : ctx, T : Type, t : T) \rightarrow (t' : T) \times \left(t = t'\right)$$

In ML, the equivalent type would be:

$$\texttt{holterm} \rightarrow \texttt{holterm} \times \texttt{holproof}$$

In this type, the fact that the input and output terms share the same type and variable context, as well as the fact that the output proof witnesses the equivalence between the input and output terms is lost. Still, the types `holterm` and `holproof` have equivalent

types in VeriML, constructed by hiding the extra context and typing information through existential packages: the types $(\phi : \mathit{ctx}, T : \mathit{Type}) \times \big(t : T\big)$ and $(\phi : \mathit{ctx}, P : \mathit{Prop}) \times \big(p : P\big)$, respectively. This shows that, in principle, any tactic written in ML for an LCF-style proof assistant can be written as a VeriML function instead.

Another shortcoming of ML in terms of tactic development is that users have to master the internal representation of logical terms and proofs in the proof assistant. For examples, the user needs to know the details of how logical variables are encoded through deBruijn indices in order to develop tactics that deal with open terms. This can be somewhat mitigated through proper syntactic extensions to ML as done in HOL-Light [Harrison, 1996]. The first-class constructs for pattern matching over open logical terms and contexts make tactic development in VeriML more convenient, hiding the implementation details of the proof assistant.

Tactic development in LTac. The Coq proof assistant offers the higher-level **LTac** language [Delahaye, 2000, 2002], which is specifically designed for tactic development. This is a significant feature, as it enables users to extend the provided automation with their own tactics through convenient pattern-matching constructs, addressing the above-mentioned shortcoming of tactic development in ML. LTac has been put to good use in various developments where the manual proof effort required is reduced to a minimum [e.g. Chlipala, 2011, Morrisett et al., 2012]. The key construct of LTac is the support for pattern matching on proof states with backtracking. It is used to examine the forms of the current hypotheses and the current goal and to issue the corresponding tactics; upon failure of such tactics, LTac implicitly backtracks to an alternative pattern matching case allowing further progress. The pattern matching construct of VeriML has been influenced by LTac; it is similar in features, save for the backtracking support which we have chosen to leave as a user-defined feature instead.

Other than the pattern matching construct, LTac is an untyped functional language without support for ML-style data types or side-effects such as mutability. This hurts expressivity, prompting users to use ad-hoc idioms to overcome the limited programming model of LTac. In order to write more sophisticated tactics such as congruence closure,

one has to revert to ML. Also, LTac is interpreted, leading to long running times for large tactics. Last, the untyped nature of the language makes LTac tactics notoriously hard to maintain and debug and are sometimes avoided precisely for this reason [e.g. Nanevski et al., 2010]. VeriML addresses these shortcomings by offering an expressive type system, a full programming model and compilation through translation to ML; it can thus be seen as a 'typed LTac'.

Proof by reflection. Coq offers another methodology for writing user-defined tactics, enabled by the inclusion of the conversion rule in its logic. This rule renders the logic (referred to as *Gallina*) into a functional programming language. For reasons of soundness, programs in this language need to be terminating. Through the technique of *proof by (computational) reflection* [Boutin, 1997], we can program the decision procedures directly in the logic language. The technique consists of reflecting part of the propositions of the logic as an inductive type within the logic itself, so that they can be manipulated as a normal datatype. Also, we need to define encoding and decoding functions from propositions to the inductive type and back. The decision procedure can then be programmed as a function at the level of the logic and its correctness can be stated and proved as a normal theorem, using the interactivity support of Coq. A sketch of the different components involved in the proof-by-reflection approach would be as follows using VeriML-style types:

$$\mathsf{Inductive}\ \mathit{IProp} : \mathit{Type} := \mathit{IAnd}\ \mathit{ip1}\ \mathit{ip2} \mid \mathit{IOr}\ \mathit{ip1}\ \mathit{ip2} \mid \cdots .$$

$$\mathsf{encode} : \mathit{Prop} \rightarrow \mathit{IProp}\ \text{(programmed in LTac)}$$

$$\mathsf{decode} : \mathit{IProp} \rightarrow \mathit{Prop}\ \text{(programmed in Gallina)}$$

$$\mathsf{decision\_procedure} : \mathit{IProp} \rightarrow \mathit{Bool}\ \text{(programmed in Gallina)}$$

$$\mathsf{correctness} : \forall\ \mathit{ip} : \mathit{IProp}.\mathsf{decision\_procedure}\ \mathit{ip} = \mathit{true} \rightarrow \mathsf{decode}\ \mathit{ip}$$

In VeriML, the first three components would not be needed because we can manipulate propositions directly; and we can write a single function that combines the last two components:

$$\mathsf{decision\_procedure} : (P : \mathit{Prop}) \rightarrow \mathsf{option}\ (P)$$

The fact that the decision procedures written in proof-by-reflection style are certified means that the procedures do not need to generate proof objects; thus procedures written in

this style are very efficient. Together with the fact that Gallina functions are total, decision procedures that are complete for certain theories do not even need to be evaluated; their definition and soundness proof is enough to prove all related theorems directly. This fact has prompted Pollack [1995] to propose total and certified decision procedures, developed using a mechanism similar to proof-by-reflection, as the main extensibility mechanism for proof assistants; yet this proposal has not been followed through.

The main problem with this approach is that it has a high up-front cost for setting up the reflection, especially if we want our decision procedure to manipulate open terms: a concrete deBruijn representation must be used for representing variables and the user must also develop functions to manipulate this representation. Pattern matching over such encodings is cumbersome compared to the first-class constructs offered by LTac or VeriML. Also, the programming model of Gallina does not support side-effects in the code of the decision procedures. In VeriML we can code similar procedures which are also certified, yet the total correctness requirement is relaxed: VeriML procedures can still fail to terminate successfully, so they need to be evaluated to completion. This is a limitation compared to proof-by-reflection, as it increases the required run time; but it can also be viewed as an advantage, as we do not have to prove termination of the decision procedures. Non-terminating behavior is not necessarily a problem and can be used towards good effect in our developments (e.g. consider the naive equality rewriter of Section 9.3, which is non-terminating in some cases). In other cases, a procedure is terminating yet its termination criterion might be difficult to prove. In proof-by-reflection, the totality requirement has to be circumvented through the use of an argument that limits the amount of steps that are performed. Also, the certification of VeriML procedures is more light-weight than proof by reflection: the type system of VeriML can be viewed itself as a light-weight certification mechanism. Coupled with the support for static proof expressions to solve the required proof obligations inside procedures, this significantly limits the manual proof effort required.

Still, the proof-by-reflection technique can be used in the context of VeriML, which could potentially be useful in large developments. In fact VeriML offers benefits for programming the LTac components of the proof-by-reflection procedures. For example, we could assign the following rich type to the encoding function:

$$\mathsf{encode} : (P : \mathit{Prop}) \to (I : \mathit{IProp}) \times \big(\mathit{decode}\ I = P\big)$$

Also, the small-scale automation mechanism used to implement proof-by-reflection, namely the conversion rule, is programmed directly in VeriML as well, leading to a smaller trusted base.

Yet the unique characteristic of the proof-by-reflection approach is that the tactics are themselves logical functions. Therefore we can reason about them using the full features of the logic, allowing us to prove deep theorems about their behavior. Together with the fact that Gallina has an infinite type hierarchy with computation allowed at any level, this would allow us to give very precise signatures to tactics – e.g. capturing precisely the set of cases where an automation tactic can be applied successfully. In contrast, VeriML maintains a two-layer architecture, where tactics are written in the computational language and cannot be reasoned about in the logic. Thus the signatures that we can give to tactics are limited to what is allowed by the VeriML type system. In the future we would like to explore the potential of offering VeriML as a surface language in a proof assistant such as Coq, where VeriML-defined tactics are compiled down to Gallina tactics through proof-by-reflection. This is an interesting alternative that offers some of the benefits of both approaches: the up-front cost of setting up proof-by-reflection would be significantly reduced, but we could also reason about VeriML tactics in the logic. Still, the tactics would not be able to use side-effects and an efficient evaluation strategy for the conversion rule (e.g. compilation-based) would need to be part of the trusted base of the system.

Unification-based automation. Coq includes another small-scale automation mechanism other than the conversion rule: a *unification-based type inference engine* used to elaborate surface-level terms to concrete CIC terms. By issuing hints to this engine, users can perform automation steps as part of type inferencing. The hints take the form of Prolog-like logic programs. Gonthier et al. [2011] show how these hints can be encoded through the existing Coq mechanism of canonical structures (a generalization of type classes); also, the experimental Matita proof assistant includes explicit support for these unification hints [Asperti et al., 2009]. This technique offers various advantages over untyped LTac-style automation tactics: Automation procedures are essentially lemmas whose premises are

transparently resolved through unification. Therefore we can specify and develop the main part of the automation lemmas by using the existing infrastructure of the proof assistant for proving lemmas. Also, since the technique is based on a small-scale automation mechanism, automation developed in this way applies transparently to the user, allowing us to omit unnecessary details.

The limitations of this approach are that automation procedures need to be written in the logic programming style instead of a more natural functional programming model. Also, procedures written in this way are hard to debug, as missing or erroneous hints can result in unspecified behavior depending on how the type inferencing engine works. Also the support for writing this style of procedures in Coq is still rudimentary and requires ad-hoc usage patterns and a high up-front cost. Our approach to extensible small-scale automation allows for more sophisticated procedures that use a full programming model. Still, the unification-based approach is useful in cases other than proof automation, such as automatic instantiation of generic mathematical operators. We have not yet programmed a unification algorithm as a VeriML procedure; doing so in the future will allow us to replicate this technique in the context of VeriML.

A calculus of tactics. An alternative tactic development methodology is described by Jojgov and Geuvers [2004], where a calculus of basic building blocks for writing tactics is presented. The calculus is based around an extension of λHOL with metavariables presented in Geuvers and Jojgov [2002], similar to our own extensions of λHOL. Though the provided building blocks do not allow the same generality as the full ML programming model we support, they include an explicit unification construct between terms that both contain metavariables. This is a more general construct than the pattern matching supported in VeriML, where only the patterns can contain metavariables and the scrutinees must be closed. Extending our language so that such a unification construct can be built-in or programmed within the language is a high priority for future development of VeriML.

Automation tactics with user-defined hints. A last way that user-defined automation is added in current proof assistants is to use automated tactics that implement a fixed proof

search strategy, but can be informed through user-defined hints. Though this is not a tactic development technique per se, it is relevant here since adding user-defined automation is one of the main reasons why tactic development is important. There are multiple instances of this technique in a wide variety of current proof assistants.

First, the Isabelle simplifier [Paulson, 1994, Nipkow et al., 2002] is one such case: the simplifier is a powerful rewriting-based prover that is used in most tactics and can be informed through user-defined rewriting lemmas. Users can register first-order lemmas they prove directly with the simplifier, increasing the available automation. This mechanism is quite similar in function to our extensible equality checking support, though it applies more widely – owing both to properties of the logic (propositional equality coincides with propositional equivalence) and to more sophisticated infrastructure for the simplifier (e.g. rewriting lemmas can include premises). Still, our equality checking allows more sophisticated extensions that are not possible for the Isabelle simplifier: we can extend it not only with first-order lemmas, but with arbitrary decision procedures as well, written in a general-purpose programming model. For example, the congruence closure function for equality hypotheses we support cannot be directly added to the simplifier.

Another instance of this technique that is often used as part of large proof developments is to use a powerful user-defined automation tactic that is informed of domain-specific knowledge through user-provided hints [e.g. Chlipala, 2008, 2011, Morrisett et al., 2012]. The hints can be as simple as a rewriting lemma and as complicated as an LTac tactic that is to be applied for specific propositions. We refer to this technique as **adaptive proof scripts**. By replacing proof scripts with a single call to the automation tactic, our proofs can adapt to changes and additions in our definitions: if the changes are small, automation will still be able to find a proof and the proof scripts will go through; larger changes will make automation fail, but users can respond to the points of failure by adding appropriate lemmas and tactics as hints. This approach has inspired the way that we program and use equality checking to provide automation to other tactics. The extensions that we add to the equality checking procedure can be seen as a form of adding sophisticated user hints.

A last instance of this technique is the standard way of developing proofs in fully automated provers such as ACL2 [Kaufmann and Strother Moore, 1996] and SMT solvers

[e.g. De Moura and Bjørner, 2008]. Lemmas that have already been proved play implicitly the role of hints. By offering very sophisticated and efficient proof search algorithms, such provers can handle large theorems automatically, with a small number of additional auxiliary lemmas. Still, these provers offer poor or no support for extensibility with user-defined tactics; this limits their applicability in cases where the fixed proof search algorithms available are not well-suited to the user domain.

In all such cases, the basic automation tactic can be viewed as a small-scale automation mechanism, as it is used ubiquitously and transparently throughout the proof. It is also extensible, owing to the support for user-defined hints. This extensibility is key to the success of such techniques, allowing users to omit a large amount of details; it has directly inspired our focus on offering extensible small-scale automation mechanisms. VeriML is a good candidate as the implementation language of such automation tactics, especially compared to LTac or proof by reflection: other than the benefits we have discussed already, it allows the use of hash tables and other efficient data structures in order to represent the database of user-supplied hints.

Overall, VeriML offers a single language that combines many of the benefits of these existing approaches. Developing tactics in VeriML is the only methodology among the ones presented above that combines three characteristics:

1. *expressivity*, where a full, side-effectful programming model is available for writing sophisticated tactics;

2. *convenience*, where first-class constructs for pattern matching on arbitrary open logical terms and proof states are available; and

3. *safety*, where the behavior of tactics can be specified and checked through a type system, enabling increased static guarantees as well as better maintainability and composability.

Also, we have shown how many of the techniques mentioned above can be programmed directly within VeriML. Last, it is important to stress the fact that users only have to

master a single language and a single programming model. In many current developments, the above techniques are combined in order to get the automation behavior required as well as good efficiency. The fact that VeriML is compiled and does not need to produce proof objects shows that similar efficiency can be achieved in the future using a single language.

10.2 Proof assistant architecture

The **LCF family of proof assistants** has had a profound influence on the design of VeriML. The main shared characteristic with VeriML is that these proof assistants follow the language-based approach: all aspects of the proof assistant, including the basic tactics, decision procedures and equality checking functions, as well as user-developed proofs and proof-producing functions, are implemented within the same language, namely ML. In the LCF family, the core logic implementation is offered as an ML module that introduces two abstract data types: a type of logical terms such as propositions and domain objects (the `holterm` type as mentioned above) and a type of proofs (`holproof`). The only way to produce values of such types is through the constructor functions offered by this module. These constructors are in one-to-one correspondence with the logical rules, guaranteeing that all values of the proof type correspond to a particular derivation in the logic. Also, basic destructor functions for looking into the structure of logical terms are offered. This design leads to an extensible proof assistant that is conceptually clear and easy to master. It allows the use of the full ML programming model when writing proof-producing functions. Thanks to the LCF design and to the type safety property of ML, full proof objects do not need to be produced to guarantee validity. The same property is true in VeriML, as shown in the proof erasure corollary of type safety. VeriML can be viewed as following precisely the same architecture, albeit with a significant departure: the basic constructor and destructor functions are dependently-typed instead of simply-typed, capturing the context and typing information from the logic type system. The correspondence is direct if we consider the pattern matching support as a principled way to offer typed destructors over logical terms.

More recently, proof assistants based on a **two-layer architecture** such as Isabelle and Coq have become popular. In these, a higher-level language is made available to users

for developing proofs, different from the implementation language of the proof assistant. This allows increased interactivity support, convenient syntax for writing proofs and in the case of Coq/LTac, a higher-level language for tactic development. Despite the benefits that this architecture offers, it has a significant downside: user-defined tactic development is de-emphasized and the programming model available for user-defined tactics is significantly limited compared to ML. The Coq architecture departs from LCF in another significant way: the LCF approach to hiding the data types of the logic is not followed, and tactics need to construct full proof objects to be checked by a proof checker. This leads to significant scalability issues in large developments, which are usually mitigated through increased use of proof-by-reflection. The VeriML design allows increased interactivity through queries to the type checking engine; supports convenient tactic syntax through syntactic sugar extensions as presented in Chapter 9; maintains the language-centric approach of the original LCF design; and allows users to control the size of proof objects through selective use of the proof erasure semantics.

The **ACL2 theorem prover** [Kaufmann and Strother Moore, 1996] is an interactive proof assistant that follows a very different style than the LCF-based or LCF-derived proof assistants we have presented so far. It is based on a simple first-order logic and eschews the use of tactics, opting instead to use a powerful automated theorem prover based on rewriting that takes into account user-defined rewriting lemmas. The increased automation is in large made possible by the simplicity of the logic language and the use of domain-specific decision procedures. ACL2 has successfully been used for large-scale verification of software [e.g. Bevier et al., 1987, Young, 1989] and hardware [e.g. Brock et al., 1996], which shows that rewriting-based proving can be expressive and powerful. We would like to investigate the addition of ACL2-like proving in our extensible rewriting infrastructure in the future. Extensions to the existing rewriters in ACL2 are made in the form of metafunctions; yet the untyped nature of ACL2 does not allow metafunctions to use facts that can be discovered through the existing rewriters [Hunt et al., 2005]. The VeriML type system addresses this shortcoming.

In the traditional ACL2 system the proofs are not foundational and one needs to trust the entirety of the system in order to trust their validity. **Milawa** [Davis, 2010] is a theorem

prover that addresses this shortcoming of ACL2. It is based on a similar first-order logic which includes a notion of proof objects. A checker for this logic constitutes the first level of the Milawa system. More sophisticated checkers are then developed and proved sound within the same logic using a technique similar to proof-by-reflection as presented above. In this way, a large number of the original ACL2 features and decision procedures are built in successive layers and verified through the checker of the previous layer, yielding a powerful certified theorem prover. The layered implementation of the conversion rule to yield extensible proof checking in VeriML shares a similar idea; we apply it to a particular fragment of the logic (equality proofs) and not to the full logic checker instead, but also we use a more expressive logic and a richer programming language. Milawa has also been connected to a verified implementation of its runtime system [Myreen and Davis, 2011], yielding perhaps the most high-assurance proof assistant in existence, as it has been certified all the way down to the machine code that executes it. We leave the investigation of similar ideas for the VeriML runtime to future work.

10.3 Extensibility of the conversion rule

CoqMT [Strub, 2010] is an extension of Coq that adds decision procedures for first-order theories such as congruence closure and arithmetic to the conversion rule. This increases the available small-scale automation support, reducing the required manual proof effort in many cases; also, it makes more dependently typed terms typeable. Still the metatheory of CIC needs significant new additions in order to account for the extended conversion rule [Blanqui et al., 2007]. Also, the added decision procedures are not verified, increasing the trusted base of the system.

NuPRL [Constable et al., 1986, Allen et al., 2000] supports an extensional type theory instead of an intensional as in the case of Coq. This translates to a very powerful conversion rule, where all provably equal terms are made definitionally equal. The extensional conversion rule destroys decidability for proof checking, but allows a large number of decision procedures to be integrated within the conversion rule, resulting in good small-scale automation support. Still, these decision procedures become part of the trusted base of the

system.

The approach to the conversion rule we have described in this dissertation amounts to replacing the conversion rule with explicit proof witnesses and re-implementing the computational aspect of the conversion rule outside the logic, as a VeriML function. This allows us to extend conversion with arbitrary user-defined decision procedures, without metatheoretic additions to the logic and without increasing the trusted base. It also allows the implementation of different evaluation strategies for the existing conversion rule. We have also implemented the decision procedures used in CoqMT, arriving at a conversion rule of similar expressivity. We believe that our ideas can be extended to a full type-theoretic logic instead of the simple higher-order logic we support, but have left this as future work.

Geuvers and Wiedijk [2004] argue about why replacing the conversion rule with explicit witnesses is desirable for foundational proof objects. The logic they present is in fact very similar to λHOL_E. They leave the question of how to reconcile the space savings of the implicit conversion approach with the increased trust of their explicit witnesses approach as future work; we believe our approach addresses this question successfully.

Grégoire [2003] presents a different sort of extension to the conversion rule: an alternative evaluation strategy for the $\beta\iota$-conversion rule in Coq, based on compilation of CIC terms into an abstract machine. This evaluation strategy significantly speeds up conversion in many situations and is especially useful when evaluating decision procedures implemented through proof-by-reflection. Still, this implementation of the conversion rule represents a significant increase in the trusted base of the system; this is possibly the reason why the independent Coq proof checker `coqchk` does not use this implementation of conversion by default. Though in our examples we have implemented a very simple call-by-name interpretation-based evaluation strategy for $\beta\iota$-conversion, our approach to conversion allows for much more sophisticated implementations, without requiring any increase in the overall trusted base.

10.4 Type-safe manipulation of languages with binding

The **Beluga programming language** [Pientka, 2008, Pientka and Dunfield, 2008] is perhaps the most closely related language to VeriML. It is a language designed for type-safe manipulation of LF terms with a similar two-layer architecture: LF comprises the logical level (corresponding to λHOL in our system), and Beluga the computational level (corresponding to VeriML). LF itself is a logical framework that can be used to represent the terms and derivations of various logics, including the λHOL logic we describe [Harper et al., 1993]. Beluga uses contextual terms to represent open LF terms and their types; it provides dependently typed constructs to manipulate such terms, including a dependently-typed pattern matching construct for contexts and contextual terms. The ML-style universe supported is limited compared to VeriML, as it does not include mutable references. The use of contextual type theory [Nanevski et al., 2008b] in Beluga has been of crucial importance in the design of VeriML. Through VeriML, we show how a language based on similar ideas can be used to write small-scale and large-scale automation procedures and can serve as an alternative architecture for proof assistants.

Perhaps the biggest difference between the two languages is that Beluga aims to be useful as a meta-logic of the LF system with Beluga functions representing meta-proofs about LF-encoded logics. We have chosen to focus on proofs in the well-established HOL logic instead. Therefore VeriML functions are not seen as meta-proofs, but merely as functions that produce foundational proofs. This allows us to add features such as mutability and non-covering pattern matching whose presence in a meta-logic would be problematic due to soundness issues. Also, our metatheoretic proofs about λHOL and VeriML present improvements over the related Beluga proofs, such as the use of hybrid deBruijn variables to elucidate the various substitution operations we define, and our novel handling of pattern matching. A recent extension to Beluga [Cave and Pientka, 2012] employs a metatheoretic study with similar orthogonality characteristics between the logic and the computational language as does our own development. This extension also supports recursive types and type constraints. The former are already part of the ML universe we support; the latter are similar to the equality constraints we use for λHOL terms but are also applicable in the

case of contexts. Still, the constraints are solved using a fixed procedure that is built into the Beluga type checker. We have shown how to solve such constraints through user-defined procedures, using a combination of staging and the collapsing transformation.

Delphin [Poswolsky and Schürmann, 2008, Poswolsky, 2009] is a similar language to Beluga, used for type-safe manipulation of LF terms. The main difference is that manipulation of open terms does not make use of contextual types but makes use of an ambient context instead. This leads to simpler code compared to Beluga in the most common case, as contexts do not have to be mentioned. Still this solution to open terms is not as general as contextual types, as we cannot maintain terms that inhabit entirely different contexts. We show how the ambient context can be supported through extensions at the syntactic level, offering the same benefits as Delphin while maintaining the generality of the contextual types approach. Also, our use of contextual types allows us to use the normal typing rules for mutable references; the use of ambient contexts would require seemingly ad-hoc restrictions as side conditions in order to maintain type safety.

A number of languages support embedding open terms of a typed object language within a meta-language in a type-safe way; some examples are MetaML [Taha and Sheard, 2000], FreshML [Shinwell et al., 2003], and MetaHaskell [Mainland, 2012]. With the exception of FreshML, these languages do not allow pattern matching on such embedded open terms. MetaML is especially interesting as its primary motivation is to support staging, leading the object and meta language to coincide. While VeriML supports staging, it only does so for closed expressions and thus does not require special treatment for quoted VeriML expressions. MetaML uses environment classifiers in order to differentiate between open terms inhabiting different contexts at the typing level; this enables hygienic treatment of bound variables – that is, bound variables cannot escape the scope of their binder. The solution based on classifiers can be viewed as a restriction of the contextual approach we follow, where all open terms must inhabit fully polymorphic contexts. FreshML and MetaHaskell do not support hygienic treatment of variables; this would destroy type-safety in our setting. This can be remedied in the case of FreshML through the use of static obligations that resemble a logic [Pottier, 2007]; we believe that the contextual type theory solution is much simpler. The solution used in MetaHaskell also employs polymorphic

contexts. Overall, the contextual type theory-based solution of VeriML offers a number of advantages: type-safe and hygienic embedding of open terms, even when the object language is dependently-typed; variable contexts that combine a polymorphic part with concrete variables; and type-safe manipulation of the embedded terms as well. We believe that the metatheoretic treatment we have presented presents a recipe for combining a typed object language with a typed meta language, yielding these benefits. As future work, we would like to apply this recipe to languages with full (heterogeneous) staging support such as MetaML and MetaHaskell.

10.5 Mixing logic with computation

The **YNot framework** [Nanevski et al., 2008a] is similar to VeriML as it mixes a logical language – specifically CIC – with side-effectful computation, such as the use of mutable references. Yet the approach taken is profoundly different, as side-effectful computations become integrated within the existing logical language through a monadic type system. Combined with proof-by-reflection, this approach could lead to type-safe decision procedures that make use of side effects, yet we are not aware of any such usage of the YNot framework. Also, this approach requires a significant addition both to the metatheory of the logic language as well as its implementation.

The **Trellys project** [Casinghino et al., 2011] presents another potential combination between a logic and a side-effectful computational language. The logic is a sub-language of the computational language, where non-termination and other side-effects are not allowed. Still, logical terms can reason about computational terms, even side-effectful ones. Compared to VeriML, this approach successfully addresses the duplication of data structures and functions between the two levels (consider, for example, the two definitions of lists in Chapter 9), as well as the inability to reason about functions in the computational language. Still, the resulting logic is non-standard. The apparent duplication of definitions in VeriML can be addressed in the future by a process similar to type promotion in Haskell [Yorgey et al., 2012]: the same surface-level definition results in definitions at both the logical and the computational level.

Chapter 11

Conclusion and future work

The work presented in this dissertation started from a simple observation: the language support for writing new automation procedures in current proof assistants is lacking. This leads to automation that is error-prone, difficult to maintain and extend; or requires mastery of the proof assistant to a level that only advanced users possess. Seeing how important it is to automate trivial and non-trivial details in large proofs, this is a shortcoming of profound importance. It ultimately leads to a proof development process that requires significant manual effort, even when the actual proof at hand is well-understood or not particularly deep in a mathematical sense – as is often the case in software certification tasks.

A second key observation is that the boundary between trivial and non-trivial details is thin and not constant. The non-trivial details of a particular domain become trivial details as soon as we move into a more complicated domain. Therefore the same automation procedures should be used in either case, with minimal user effort. Also, when using automation to handle trivial details in a way that is transparent to the user, these details should not necessarily be expanded out to their full formal counterpart. The reason is that trivial details occur pervasively throughout proofs, and full expansion leads to prohibitively large formal proofs that take a long time to check. This suggests that the automation procedures themselves should be *verified* upon definition, instead of verifying their results upon evaluation. In this way, we can be certain that automation does not make false claims, even when formal proofs are not produced.

These two observations combined result in the basic design idea behind VeriML: a pro-

gramming language that enables the development of verified automation procedures, using an expressive programming model and convenient constructs tailored to this task. We achieve expressiveness by including the side-effectful, general-purpose programming model of ML as part of the language; convenience by adding first-class support for introducing proofs, propositions and other logical terms as well as pattern matching over them – the exact constructs that form the main building blocks of automation procedures. The verification of automation procedures is done in a light-weight manner, so that it does not become a bottleneck in itself. It is the direct product of two mechanisms. The first is a rich type system that keeps information about the logical terms that are manipulated and enables user to assign detailed signatures to their automation procedures. The second is a staging mechanism which is informed by the typing information and allows the use of existing automation in order to minimize the proof burden associated with developing new automation procedures. The use of the type system also leads to a principled programming style for automation procedures, making them more composable and maintainable.

In summary, VeriML is therefore a "programming language combining typed manipulation of logical terms with a general-purpose side-effectful programming model", as is stated in my thesis. In this dissertation, I have described the overall design of the language; I have shown that the language is type-safe and that automation procedures do not need to produce full proofs, through an extensive metatheoretic study; I presented an implementation of the language, including a type inferencing engine and a compiler for it; and I described a number of examples written in the language, which have been previously hard to implement using the existing approaches. Also, I have shown that various features of the architecture of modern proof assistants can be directly recovered and generalized through the VeriML design, rendering the VeriML language as a proof assistant in itself. Proofs, as well as automation procedures and other tactics can be written as VeriML programs. Interactive support for writing proofs is recovered as queries to the VeriML type checker; it is generalized to the case of tactics as well, allowing them to be developed interactively as well. Also, the conversion rule that is part of the trusted core of modern proof assistants, can be programmed instead in VeriML, thus allowing sophisticated user extensions while maintaining logical soundness.

The VeriML design therefore constitutes a viable alternative to the architecture of proof assistants, one that represents a return to form: a single meta-language is used to implement all aspects of the proof assistant, allowing maximum extensibility. This was the rationale behind the original LCF system and the original design of ML [Milner, 1972]. Our work essentially updates this language-centric design with the modern discoveries in the domain of dependently typed languages, resulting in a meta-language that can be used for conducting proofs and for programming verified proof procedures with reduced manual effort.

Still, a long way lies ahead of us until an actual VeriML implementation can be a realistic alternative to existing proof assistants. A number of features that are essential for practical purposes are not yet covered by our design and its corresponding metatheory and implementation. We will first focus on features where the main challenge spans across the theory and the implementation; features that are mostly related to engineering issues of the implementation will follow.

Metatheory

Meta-N-terms. In order to be able to manipulate open logical terms, we closed them over the variable environment they depend on. This led to the introduction of contextual terms or meta-terms, along with their corresponding notions of meta-variables, contexts and substitutions. The computational language of VeriML allows us to introduce contextual terms, bind them to meta-variables and pattern match to look into their structure. In many situations, it would be useful to be able to close the contextual terms once more, over the meta-variables they depend on. These would then be characterized as *meta-2-terms*, as they would contain not only normal logical variables but meta-variables as well. By repeating this process we would be able to manipulate such meta-2-terms directly inside VeriML using similar constructs such as the pattern matching we already support.

In this dissertation we have already presented a situation where this would be useful, namely the static evaluation of proof expressions within tactics – as exemplified by the `StaticEqual` expression we often use throughout our examples. Such proof

expressions contain meta-terms, usually instantiated through type inference; these meta-terms contain meta-variables, standing for the input arguments of the tactic we are defining. In order to evaluate these proof expressions, we would therefore need to perform pattern matching where the scrutinee is an open meta-term. Yet our pattern matching theorem and algorithm only covers the case where the scrutinee is closed and the pattern is open. This is for good reason too, as generalizing to the case where both the scrutinee and the pattern contain meta-variables, together with the fact that the scrutinee might contain non-identity substitutions, changes the operation from pattern matching to unification which would destroy the decidability and determinism properties of the operation. The way we have addressed this is the use of the collapsing transformation, which turns the open meta-terms into equivalent closed ones which pattern matching can already handle. Yet this transformation is only applicable under certain limitations. If we had support for manipulating meta-2-terms instead, this transformation would not be necessary and we would be able to lift such limitations.

Another situation where meta-2-terms show up is the notion of proof state, that is, the type of an incomplete proof. We can view the current context and the current goal, as captured through type inference, as the notion of proof state that VeriML supports. This is limited compared to current proof assistants: most proof assistants allow multiple hypotheses-goals pairs to be part of the current proof state, as well as unification variables – variables to be implicitly instantiated based on their usage. Such variables are often useful when solving goals involving existentials, or when using universally quantified lemmas: we can delay the instantiation of the existential witness, or of the arguments of the lemmas, until their identity is trivial to deduce from the context. We could encode such notions of proof state as meta-2-terms, and we would therefore need pattern matching over meta-2-terms as well as their associated meta-2-contexts in order to work with such proof states and incomplete proofs.

A last situation where meta-2-terms and potentially even higher-order meta-N-terms would be useful is in programming a well-typed unification procedure in VeriML:

namely, a procedure that takes two open terms with meta-variables (representing unification variables), and decides whether a substitution exists that makes the two terms match. As mentioned earlier, the problem of unification of λHOL terms is undecidable, as it is equivalent to higher-order unification in the λ-calculus. Yet undecidability can be overcome by enabling programmatic control. We would therefore like to follow a similar approach as we did for the equality proofs of the conversion rule to handle unification: by programming unification inside VeriML instead of making it a fixed procedure, we would be able to develop further extensions to it, to handle more cases. We believe that being able to manipulate meta-N-terms plus their accompanying notions of contexts and substitutions, would allow us to program this procedure inside VeriML. Such an extensible unification procedure would allow us to treat the remaining small-scale automation mechanisms in current proof assistants, which are all unification-based, in a similar manner to the conversion rule, with similar benefits.

Extending our metatheory to meta-2-terms, as well as higher meta-N-terms, should be possible by repeating the process we described in Chapter 4 for deriving meta-1-terms and the associated proofs. In fact, we have done our proofs with such an extension in mind, making sure that the operations and theorems of the base level of normal logical terms exactly match up with the operations and theorems of the meta-1-level of contextual terms. Therefore repeating this process once more, or inductively up to N, should in principle give us the desired. Still, further extensions would be needed for meta-2- or meta-N-terms to be truly useful, such as being able to write code that is polymorphic over meta-levels and also to be able to pattern match on substitutions in the same way that we pattern match on terms and contexts. Managing this complexity in order to arrive at a conceptually clear description of the metatheory and the language would be a process in itself; it would be an even more significant challenge to find ways to mask this complexity at the level of the end user of the language.

Inductive definitions. VeriML needs better support for inductive types and inductive predicates. For the time being, inductive definitions are opaque – there is no pattern

matching support for them, which precludes writing VeriML functions that operate directly on such definitions. We have so far addressed this shortcoming through ad-hoc means, such as the meta-generation of rewriters for the elimination principles. We should be able to write such code in a generic manner within VeriML supporting all inductive definitions, while retaining the increased typing information that VeriML offers. This would allow the definition of typed generic induction and inversion tactics, which have proved to be one of the main hindrances of using inductive definitions in our current developments. The inversion tactic is especially notorious even in current proof assistants, as its behavior is hard to describe formally; being able to give it a precise signature through typing would be an interesting potential.

The main challenge in supporting inductive definitions comes from the fact that their well-formedness conditions are syntactic in nature, instead of being based on the judgemental approach. We have partly addressed this in the case of the positivity condition and in the handling of parameters, yet syntactic side-conditions remain. This precludes us from applying the techniques we used in our metatheory in order to derive well-formed patterns over such definitions, as our techniques assume that the typing of the object language is described as a pure judgement.

Controlling representations. We use a generic representation for logical terms, but we believe that specialized representations for use under specific circumstances could improve the efficiency of VeriML programs significantly. For example, closed natural numbers up to a certain number could be represented as machine integers and their operations such as addition and multiplication could be implemented through machine instructions. A more ambitious use case for the same idea has to do with the equality checking function described in Section 9.4. This function needs to construct a hash table representing equivalence classes anew for every context. We could instead add an additional representation to contexts that keep track of the hash table as well, calling the right function when a new element enters the current context. Furthermore, the proof erasure procedure we describe for proof objects could be seen as choosing a unit representation for proof objects; the same representation could be used in order to

erase other terms that do not influence the runtime behavior of programs and are only useful during type checking. It is interesting to see if user-defined representations for existing classes of logical terms can be maintained and manipulated while maintaining type safety. The main challenge would be to coordinate between the set of patterns allowed over terms and the information that the actual representation of such terms contains.

Implementation

Our prototype implementation has served well as an experimentation platform for the work presented in this dissertation. Still it is far from a language implementation ready for the end user of VeriML. The ML core of the language does not support many of the conveniences present in modern functional languages like OCaml and Haskell, such as Hindley-Milner type inference, first-class modules, type classes and effect encapsulation through monads. The syntax and constructs of the language are more verbose in many situations, such as pattern matching over normal datatypes, and error reporting is rudimentary. Last, our compilation methodology goes through translation to OCaml, which introduces the unnecessary overhead of re-typechecking the resulting OCaml AST. Our implementation of the λHOL logic has similar shortcomings.

In order to address these issues, we would like to work on a new implementation of VeriML that extends an existing programming language implementation with the VeriML-specific constructs for an existing logic implementation, so as to inherit the engineering effort that has gone into them. The OCaml implementation together with the CIC logic implementation used in Coq are natural choices for this purpose, as they would allow us to use VeriML as an additional language in Coq, without enlarging its trusted base. In this way we would also be able to reuse the large library and proof developments that are already implemented in Coq, and ease adoption of VeriML in situations where it can be of benefit. One challenge with this approach is to extend the design and metatheory of VeriML so as to incorporate the CIC logic, which contains many features not in λHOL. We are confident that the main techniques used in our current metatheory are still applicable in the case of CIC.

The VeriML extension should be as isolated as possible from the main OCaml implementation, reflecting the orthogonality between the λHOL-related constructs and the ML core that characterizes our metatheory. This would allow development of further extensions to VeriML without requiring changes to the core OCaml implementation; it would also minimize the cost associated with updating to newer versions of the OCaml implementation as they become available. We expect the main challenge to be inserting hooks into the main OCaml modules, such as the AST definitions, the type checker and the compiler to bytecode, so that they make calls to corresponding extension modules. Such changes are a particular instance of the expression problem; we might be able to make use of the existing solutions to it [e.g. Swierstra, 2008].

One important aspect of any implementation of VeriML is the capabilities of the type inferencing engine. Our current support for type inference of logical terms, as described in Chapter 8, is based around greedy mutable reference-based unification, using invertible syntactic operations on terms. This approach is both versatile and efficient, yet we would like to explore further enhancements to it and use the same approach for computational expressions and types, and all the other kinds of terms in our language. The main extension to this approach would be to allow some backtracking when more than one typing alternatives are possible and also a way to interoperate with constraint-based type inference as described by Pottier and Rémy [2005]. The inference engine in recent versions of OCaml is based on this mechanism, where type inference is formulated as a stage of constraint generation followed by a stage of constraint solving. This formulation is well-suited to our purposes of extending the OCaml implementation, as VeriML-related constraints could be presented as an additional kind of constraints to be solved by an independent constraint solver based around unification of logical terms.

Another issue we would like to address is the duplication of definitions of types and functions between the logic and the computational language. In many cases the same definition is directly applicable both in the logical and in the computational level – e.g. definition of recursive types without negative occurrences or mutable references, or of recursive functions that are pure and total. We are confident that such cases can be handled by offering surface-level syntax that performs both definitions at the logical and at the computational level,

allowing users to treat each definition at either level transparently. This correspondence is not as direct in cases where the boundary between the logical and the computational level is crossed, such as tactics that pattern match on logical terms and data types that use ML-only types such as mutable references. Still, it is interesting to explore whether surface syntax can be offered in such situations as well that hides some of the complexities associated with having two strongly isolated levels in the language.

Translating to reflective CIC procedures would be an alternative implementation strategy for the pure subset of VeriML. This option might be worth exploring in parallel to the above described strategy. The main benefit of this approach would be that we would be able to reason about VeriML functions using the full expressivity of the CIC logic, instead of the comparatively more limited expressivity of the VeriML type system. The challenge of this approach would be to reify a large subset of the CIC logic as an inductive datatype within CIC, as well as the corresponding typing relation. Even though considerable effort has gone into similar considerations [e.g. Chapman, 2009, Chapman et al., 2010], this is still largely an open problem.

User interface

When viewed as a proof assistant, the user interface that VeriML offers becomes very important. Exploring ways that the VeriML design can enhance the user experience becomes an interesting problem in itself. For example, traditional proof assistants usually have two different languages for writing proofs: one that follows the imperative style, where the focus is placed on the tactics applied; and one that follows the declarative style, where the intermediate steps of the proof are the main focus. We believe that the increased typing information present in VeriML would allow us to mix these two styles, by writing two sets of strongly-typed tactics tailored to each style and using type inference to mediate the proof state between them.

Supporting interactive development of VeriML proofs and tactics presents an interesting user-interface design issue, especially in the presence of our staging support. The proof script languages used in current proof assistants have clear breakpoints, allowing partial proof scripts to be evaluated up to a certain point. Interactive development environments

such as Proof General [Aspinall, 2000] have a notion of "locked region" that follows such breakpoints, and moves as subsequent steps of proof scripts are performed. Proof state information is presented to the user, as it is determined for the currently active breakpoint. In VeriML, proof expressions do not necessarily have a similar linear structure with well-defined break points. Partial proof expressions can also have missing parts at various points. The presence of staged expressions complicates the evaluation order of expressions further; such expressions also need to be able to provide information about their result to the user, e.g. in order to allow them to fix erroneous applications of the conversion rule. These issues are even more serious in the case of tactics, whose structure is based around pattern matching and is therefore certainly not linear. Yet, proof state and further typing information is available at all program points, even for incomplete proof expressions and incomplete tactics, by querying the type checking engine of VeriML. In order to address the above issues and make use of such information to offer a better interface to the user, we expect that increased coupling between the interactive development environment and the VeriML implementation will be essential, departing further from the traditional REPL-based interface to proof assistants.

Other uses of the VeriML design

We believe that VeriML could be used to investigate a number of interesting possibilities, other than the examples seen so far and the traditional usage scenario of proof assistants. First, we believe that VeriML could be used to realize projects such as the extensible low-level programming language and software certification framework Bedrock [Chlipala, 2011]. The additional expressivity, extensibility and efficiency that a realistic VeriML implementation would be able to offer to tactic programming would address the main issues of this framework. Also, we believe that the first-class support for proofs in VeriML could render it an appealing platform for implementing a number of certifying tools, such as compilers, static analysis tools and SMT solvers. The combination of such tools with a software certification framework such as Bedrock could result in an intriguing implementation language for certifiable systems code with significant reductions in the required certification effort.

Another interesting potential is to relax the pattern matching restriction in VeriML,

where proof objects cannot be inspected, in order to develop procedures that extract *knowledge* by looking into the structure of existing proofs. For example, one procedure could generalize the steps used in a specific proof, in order to create a tactic that can handle proofs that are similar in structure. Designing a tactic that corresponds to the word "similarly" as used in informal pen-and-paper proofs is a very motivating research problem for future proof assistants. We believe that VeriML constitutes a good framework to support experimentation with such ideas, because of its language-centric nature, its treatment of trivial steps in proofs and its pattern matching capabilities.

Last, we believe that the design of VeriML and especially our treatment of combining a typed object language and a typed meta language could be useful in the context of normal functional programming languages. One such example would be to staging support in languages such as OCaml and Haskell, with the ability to pattern match and manipulate the code of later stages. In this way we would be able to write optimization stages programmatically, something that would be especially useful in the context of embedded domain-specific languages. Ideas similar to the extensible conversion rule would be useful in this situation, in order to provide extensible type checking for the embedded languages.

Bibliography

A. Abel. MiniAgda: Integrating sized and dependent types. *Arxiv preprint arXiv:1012.4896*, 2010.

S. Allen, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The NuPRL open logical environment. *Automated Deduction-CADE-17*, pages 170–176, 2000.

A. Asperti and C. Sacerdoti Coen. Some Considerations on the Usability of Interactive Provers. *Intelligent Computer Mathematics*, pages 147–156, 2010.

A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. Hints in unification. *Theorem Proving in Higher Order Logics*, pages 84–98, 2009.

D. Aspinall. Proof General: A generic tool for proof development. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 38–43, 2000.

L. Augustsson. Cayennea language with dependent types. *Advanced Functional Programming*, pages 240–267, 1999.

B. Aydemir, A. Charguéraud, B.C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. *ACM SIGPLAN Notices*, 43(1):3–15, 2008.

H. Barendregt. Lambda calculus with types. *Handbook of logic in computer science*, 2: 118–310, 1992.

H.P. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Sci. Pub. B.V., 1999.

B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual (version 8.4), 2012.

Y. Bertot, P. Casteran, G. Huet, and C. Paulin-Mohring. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Inc, 2004.

W.R. Bevier et al. *A verified operating system kernel*. PhD thesis, University of Texas at Austin, 1987.

M. Blair, S. Obenski, and P. Bridickas. Gao report b-247094, 1992.

F. Blanqui, J.P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In *Rewriting Techniques and Applications*, pages 671–671. Springer, 1999.

F. Blanqui, J.P. Jouannaud, and P.Y. Strub. A calculus of congruent constructions. *Unpublished draft*, 2005.

F. Blanqui, J.P. Jouannaud, and P.Y. Strub. Building decision procedures in the calculus of inductive constructions. In *Computer Science Logic*, pages 328–342. Springer, 2007.

F. Blanqui, J.P. Jouannaud, and P.Y. Strub. From formal proofs to mathematical proofs: a safe, incremental way for building in first-order decision procedures. In *Fifth Ifip International Conference On Theoretical Computer Science–Tcs 2008*, pages 349–365. Springer, 2010.

S. Boutin. Using reflection to build efficient and certified decision procedures. *Lecture Notes in Computer Science*, 1281:515–529, 1997.

A.R. Bradley and Z. Manna. *The calculus of computation: decision procedures with applications to verification*. Springer-Verlag New York Inc, 2007.

E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. *Types for Proofs and Programs*, pages 115–129, 2004.

E.C. Brady. IDRIS: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, pages 43–54. ACM, 2011.

B. Brock, M. Kaufmann, and J. Moore. ACL2 theorems about commercial microprocessors. In *Formal Methods in Computer-Aided Design*, pages 275–293. Springer, 1996.

C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering*, pages 57–76. Springer, 2003.

C. Casinghino, H.D. Eades, G. Kimmell, V. Sjöberg, T. Sheard, A. Stump, and S. Weirich. The preliminary design of the Trellys core language. In *Programming Languages meet Program Verification*, 2011.

A. Cave and B. Pientka. Programming with binders and indexed data-types. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 413–424. ACM, 2012.

J. Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009.

J. Chapman, P.É. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM.

A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 54–65. ACM, 2007.

A. Chlipala. Certified Programming with Dependent Types, 2008. URL `http://adam.chlipala.net/cpdt/`.

A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2011.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming*, pages 79–90. ACM, 2009.

R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, et al. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.

T. Coquand. An Analysis of Girard's Paradox. In *Symposium on Logic in Computer Science*, pages 227–236. IEEE Computer Society Press, 1986.

T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3), 1988.

K. Crary and S. Weirich. Flexible Type Analysis. In *In 1999 ACM International Conference on Functional Programming*, 1999.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

J.C. Davis. *A "self-verifying" theorem prover*. PhD thesis, University of Texas at Austin, 2010.

N.G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75,5, pages 381–392. Elsevier, 1972.

L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

D. Delahaye. A tactic language for the system Coq. *Lecture notes in computer science*, pages 85–95, 2000.

D. Delahaye. A proof dedicated meta-language. *Electronic Notes in Theoretical Computer Science*, 70(2):96–109, 2002.

G. Dowek. Higher-order unification and matching. *Handbook of automated reasoning*, 2: 1009–1062, 2001.

P. Dybjer. Inductive sets and families in Martin-Löfs type theory and their set-theoretic semantics. *Logical Frameworks*, pages 280–306, 1991.

M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

H. Geuvers and G. Jojgov. Open proofs and open terms: A basis for interactive logic. In *Computer Science Logic*, pages 183–209. Springer, 2002.

H. Geuvers and F. Wiedijk. A logical framework with explicit conversions. In *C. Schürmann (ed.), Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, page 32. Elsevier, 2004.

J.Y. Girard. *Interprétation fonctionelle et élimination des coupures de larithmétique dordre supérieur*. PhD thesis, Université Paris VII, 1972.

H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. *Algebra, Meaning, and Computation*, pages 521–540, 2006.

G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 163–175. ACM, 2011.

M. Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

M.J. Gordon, R. Milner, and C.P. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *Springer-Verlag Berlin*, 10:11–25, 1979.

B. Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml.* Thése de doctorat, spécialité informatique, Université Paris 7, École Polytechnique, France, December 2003.

B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *ACM SIGPLAN Notices*, volume 37, pages 235–246. ACM, 2002.

R. Harper. *Practical Foundations for Programming Languages.* Carnegie Mellon University, 2011. URL `http://www.cs.cmu.edu/~rwh/plbook/book.pdf`.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

J. Harrison. HOL Light: A tutorial introduction. *Lecture Notes in Computer Science*, pages 265–269, 1996.

W. Hunt, M. Kaufmann, R. Krug, J. Moore, and E. Smith. Meta reasoning in ACL2. *Theorem Proving in Higher Order Logics*, pages 163–178, 2005.

A. Hurkens. A simplification of Girard's paradox. *Typed Lambda Calculi and Applications*, pages 266–278, 1995.

G. Jojgov and H. Geuvers. A calculus of tactics and its operational semantics. *Electronic Notes in Theoretical Computer Science*, 93:118–137, 2004.

M. Kaufmann and J. Strother Moore. ACL2: An industrial strength version of Nqthm. In *Computer Assurance, 1996. COMPASS'96,'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pages 23–34. IEEE, 1996.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

D. Kroening and O. Strichman. *Decision procedures: an algorithmic point of view*. Springer-Verlag New York Inc, 2008.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7): 107–115, 2009.

X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Remy, J. Vouillon, et al. The OCAML language (version 3.12. 0), 2010.

J.L. Lions et al. Ariane 5 flight 501 failure, 1996.

G. Mainland. Explicitly heterogeneous metaprogramming with MetaHaskell. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 311–322. ACM, 2012.

P. Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin of a Series of Lectures given in Padua, June 1980*. Studies in Proof Theory, Lecture Notes, 1, 1984.

C. McBride. Epigram: Practical programming with dependent types. *Advanced Functional Programming*, pages 130–170, 2005.

J. McKinna and R. Pollack. Pure type systems formalized. *Typed Lambda Calculi and Applications*, pages 289–305, 1993.

B. Meurer. Just-In-Time compilation of OCaml byte-code. *CoRR*, abs/1011.6223, 2010.

R. Milner. Implementation and applications of Scott's logic for computable functions. In *Proceedings of ACM conference on Proving assertions about programs*, pages 1–6, New York, NY, USA, 1972. ACM.

R. Milner. *The definition of Standard ML: revised*. The MIT press, 1997.

A. Miquel. The Implicit Calculus of Constructions: Extending Pure Type Systems with an Intersection Type Binder and Subtyping. *Typed Lambda Calculi and Applications*, pages 344–359, 2001.

G. Morrisett, G. Tan, J. Tassarotti, and J.B. Tristan. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

M. Myreen and J. Davis. A verified runtime for a verified theorem prover. *Interactive Theorem Proving*, pages 265–280, 2011.

A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of International Conference on Functional Programming*, volume 8, 2008a.

A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3), 2008b.

A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. In *Proceedings of the 37th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 261–274. ACM, 2010.

T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

U. Norell. Towards a practical programming language based on dependent type theory. Technical report, Goteborg University, 2007.

S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Computer Aided Verification*, pages 411–414. Springer, 1996.

C. Paulin-Mohring. Extracting programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 89–104. ACM, 1989.

C. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. *Lecture Notes in Computer Science*, pages 328–328, 1993.

C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, 1993.

L.C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer, 1994.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61. ACM, 2006.

F. Pfenning and C. Schürmann. System description: Twelf-a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, pages 202–206, 1999.

B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Proceedings of the 35th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 371–382. ACM, 2008.

B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, pages 163–173. ACM New York, NY, USA, 2008.

B.C. Pierce. *Types and programming languages*. The MIT Press, 2002.

R. Pollack. On extensibility of proof checkers. *Types for Proofs and Programs*, pages 140–161, 1995.

R. Pollack, M. Sato, and W. Ricciotti. A canonical locally named representation of binding. *Journal of Automated Reasoning*, pages 1–23, 2011.

A. Poswolsky. *Functional programming with logical frameworks: The Delphin project*. PhD thesis, Yale University, 2009.

A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. *Lecture Notes in Computer Science*, 4960:93, 2008.

F. Pottier. Static name control for FreshML. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 356–365, 2007.

F. Pottier and D. Rémy. The Essence of ML Type Inference. *Advanced Topics in Types and Programming Languages*, pages 389–489, 2005. URL `http://cristal.inria.fr/attapl/`.

N. Pouillard. Nameless, painless. In *ACM SIGPLAN Notices*, volume 46,9, pages 320–332. ACM, 2011.

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, 2009.

Z. Shao. Certified software. *Communications of the ACM*, 53(12):56–66, 2010.

T. Sheard. Languages of the future. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119. ACM, 2004.

M.R. Shinwell, A.M. Pitts, and M.J. Gabbay. FreshML: Programming with binders made simple. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 263–274. ACM New York, NY, USA, 2003.

K. Slind and M. Norrish. A brief overview of HOL4. *Theorem Proving in Higher Order Logics*, pages 28–32, 2008.

M. Sozeau. Subset coercions in Coq. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs*, pages 237–252. Springer-Verlag, 2006.

A. Stampoulis and Z. Shao. VeriML: Typed computation of logical terms inside a language with effects. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 333–344. ACM, 2010.

A. Stampoulis and Z. Shao. Static and user-extensible proof checking. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284. ACM, 2012a.

A. Stampoulis and Z. Shao. Static and user-extensible proof checking (extended version). Available in the ACM Digital Library, 2012b.

P.Y. Strub. Coq modulo theory. In *Proceedings of the 24th International Conference on Computer Science Logic*, pages 529–543. Springer-Verlag, 2010.

M. Sulzmann, M.M.T. Chakravarty, S.P. Jones, and K. Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

W. Swierstra. Functional Pearl: Data types a la carte. *Journal of Functional Programming*, 18(4):423, 2008.

W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1-2):211–242, 2000.

D. Vytiniotis, S.P. Jones, and J.P. Magalhaes. Equality proofs and deferred type errors: a compiler pearl. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP*, 2012.

B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.

A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 214–227. ACM, 1999.

H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN symposium on Principles of programming languages*, pages 224–235. ACM, 2003. ISBN 1581136285.

B.A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J.P. Magalhães.

Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012.

W.D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5 (4):493–518, 1989.

M. Zhivich and R.K. Cunningham. The real cost of software errors. *Security & Privacy, IEEE*, 7(2):87–90, 2009.