Incomplete proofs and terms and their use in interactive theorem proving

Citation for published version (APA):

DOI:
10.6100/IR573767

Document status and date:
Published: 01/01/2004

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. R.A. van Santen, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op maandag 5 april 2004 om 16.00 uur

door

Gueorgui Ivanov Jojgov

geboren te Karlovo, Bulgarije
Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. J.C.M. Baeten
en
prof.dr. H.P. Barendregt

Copromotor:
dr. J.H. Geuvers

© 2004 G. I. Jojgov
IPA Dissertation Series 2004-02
Printed by University Press Facilities, Eindhoven

Cover design by Paul Verspaget & Carin Bruinink
Typeset using LATEX.

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Jojgov, Gueorgui I.
Incomplete proofs and terms and their use in interactive theorem proving
Proefschrift. - ISBN 90-386-0862-4
NUR 993
Subject headings: lambda calculus/formal logic/automated theorem proving
GR Subject Classification (1998) : F.4.1, D.3.1, F.3.2, F.1.1

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) at the Eindhoven University of Technology, supported by the Dutch Organization for Scientific Research (NWO) under contract 612-010-002.
На моите родители
Acknowledgments

There are many people who directly or indirectly have helped me and in doing so made this thesis possible. First of all, I would like to thank the members of the reading committee for the thesis: Jos Baeten, Henk Barendregt, Gilles Dowek, Herman Geuvers and Rob Nederpelt. They provided a valuable feedback that helped improve the original manuscript. I am honored to have them as well as Jan Friso Groote and Jan Willem Klop as members of the promotion committee.

I especially want to thank my supervisors Herman Geuvers and Rob Nederpelt. I learned a lot from the discussions with Herman and our work together. As my day-to-day supervisor he was always open for my ideas and did not fail to warn me for a few hidden traps along the road. I would like to thank Rob for his support during the last four years. In key moments he provided much appreciated advice and just the right viewpoint to the problems.

My work was done in the Formal Methods group headed by prof. Jos Baeten. I would like to thank Jos for creating the pleasant work atmosphere that I experienced in Eindhoven. I also enjoyed the hospitality of the group of Henk Barendregt in Nijmegen where I was a frequent visitor. I thank my present and past officemates and all colleagues from both groups. The discussions with Henk Barendregt, Roel Bloo, Mirna Bognar, Tijn Borghuis, Luis Cruz-Filipe, Herman Geuvers, Fairouz Kamareddine, Marko Luther, Conor McBride, Rob Nederpelt, Mladen Niqui, Martijn Oostdijk, Ana Sokolova, Martin Strecke, Dan Synek, Freek Wiedijk, Tim Willemsen, Jan Zwanenburg and others as well as the reports of the anonymous referees of the papers I have submitted have been most helpful.

I am grateful to Ivan Soskov and all the members of the Faculty of Mathematics and Informatics of Sofia University for introducing me to the field of theoretical computer science.

Special thanks go to my good friend Ana, but also to Драга, Сузана, Marcela, Tim, Martijn and Victor for the nice RISK evenings and their help in difficult moments. I thank Анi and Кирил, Вероника and Нан, Виола, Габи and Алаин, Дечка, Ени and Иван, Ени and Жори Ка Лок, Кирило, Крис and Сашо, Мария and Никола, Мариян, Мариян and Камен, Марк and Войчо, Мая, Мирю, Надя, Петко, Самуил, Сашо, Снежка and Митко, Светлана and Сотир for their help and the nice moments we spent together. I thank Гери and Иван, Анн and Живко, Желязко, Румен and Вероника for keeping in touch despite the distance.

Last, but most importantly, I would like to thank my parents Елена and Иван and my brother Николай for their love and unconditional support through the years.

1
# Contents

Acknowledgments i

Chapter 1. Introduction 1
  1.1. Object of Study and Motivation 1
  1.2. Overview of the Thesis 7

Chapter 2. Related Work: Incomplete Terms and Applications 9
  2.1. Incomplete Terms in Type Theory: Preliminaries 9
  2.2. Overview of Systems of Open Terms in Type Theory 11
  2.3. Tactic-Based Interactive Proof Construction 24
  2.4. Higher-Order Unification 25

Chapter 3. Incomplete Terms and Proofs in Higher-Order Logic 27
  3.1. Introduction 27
  3.2. Higher-Order Predicate Logic 28
  3.3. Open Proofs in Higher-Order Logic 32
  3.4. Open Proofs with Binding Holes 45
  3.5. Refinement and Completeness 53

Chapter 4. Typing for Incomplete Terms and Proofs 63
  4.1. Typing for HOL: λHOL 63
  4.2. Typing for oHOL: λoHOL 68
  4.3. Typing for HOL with binding holes: λbHOL 73
  4.4. Pure Type Systems with Hereditarily Parameterized Variables, Constants and Definitions 85

Chapter 5. The Formulas-as-Types Embedding 105
  5.1. Introduction 105
  5.2. Embedding bHOL into λbHOL 106
  5.3. Proof Reduction and β-Reduction 115
  5.4. Are Terms with Free Variables Open Terms? 117

Chapter 6. Applications to Tactics 121
  6.1. Introduction 121
  6.2. An Integrated Model of an Interactive Theorem Prover 122
  6.3. Encoding Proof-Search Procedures: Pre-tactics and Tactics 126
  6.4. Definability of Some Basic Tactics and Tacticals 131
  6.5. Semantics 134
6.6. Properties 141
6.7. Example: A Tactic for Deciding Minimal IPC 145
6.8. Related and Future Work 149

Appendix A. λoHOL with Global Definitions 153
Appendix B. Rules for Evaluation of Tactics and Pre-tactics 155

Bibliography 161
Samenvatting 165

Curriculum Vitae 167
CHAPTER 1

Introduction

This thesis is a result of the author’s work as a Ph.D. student (Onderzoeker in Opleiding) at the Technische Universiteit Eindhoven on the project “Use and Meaning of Open Terms in Interactive Formal Problem Solving” funded by the Netherlands Organisation for Scientific Research (NWO). It is based on revised versions of the papers [Joj01, GJ02, Joj03a, Joj03b, GJ03].

1.1. Object of Study and Motivation

The application of computers in automating routine or computation-intensive tasks has revolutionized many industries in the past few decades. Since 1967 when De Bruijn started his pioneering work on Automath [dB80] there has been increasing interest in doing formal reasoning with computers. The application areas range from foundations of mathematics to verification of computer hardware and software. One branch in the constellation of tools for formal reasoning is formed by programs that allow the user to create formal proofs in interaction with the machine. These programs are often called interactive theorem provers or more generally interactive proof assistants. They are mainly targeting problems of complexity which prohibits full automation and the only feasible role for the computer is to assist a human operator in solving the problem. The assistance can be in the form of bookkeeping, proof-checking, providing hints or solving ‘easy’ subproblems.

Standing on the formulas-as-types embedding, type theory provides a robust theoretical framework in which these activities can be expressed and reasoned about. There are a number of proof assistants based on variants of type theory or logical frameworks encoded in type theory like Coq [CDT03], ALF [Mag95], LEGO [Pol], etc. In this thesis we will study some aspects of the meta-theory of type-theory based proof assistants that result from the need to work in an interactive manner.

Let us illustrate with an example how a tactic-based proof assistant works and which questions we are trying to answer in this thesis. In Figure 1.1 we see a sample transcript of proof session in an idealized proof assistant in the style of Coq and LEGO. The transcript is a sequence containing the representations of the current goal the user is focused on at the particular moment and commands that he/she issues (the so-called tactics). For readability the tactic calls are underlined. A goal is a statement that we want to prove from the list of assumptions of the form $x:A$. The statement is separated by four dashes from its assumptions.
\[ U : \text{Type} \]
\[ ? : \forall x, y^U. (x =_U y) \rightarrow (y =_U x) \]

**Unfold \(=_U\).**

\[ U : \text{Type} \]
\[ ? : \forall x, y^U. (\forall \mathcal{P}^U \rightarrow \text{Prop}. \mathcal{P}_x \rightarrow \mathcal{P}_y) \rightarrow (\forall \mathcal{P}^U \rightarrow \text{Prop}. \mathcal{P}_y \rightarrow \mathcal{P}_x) \]

**Intros.**

\[ \begin{array}{ll}
U : \text{Type} & x, y : U \\
h : \forall \mathcal{P}^U \rightarrow \text{Prop}. \mathcal{P}_x \rightarrow \mathcal{P}_y & P : U \rightarrow \text{Prop} \\
h_0 : \mathcal{P}_y & h_0 : \mathcal{P}_y \\
? : \mathcal{P}_x \rightarrow \mathcal{P}_x & ? : \mathcal{P}_y \\
\end{array} \]

**Apply \( h (\lambda z^U. \mathcal{P}_z \rightarrow \mathcal{P}_x) \).**

\[ \begin{array}{ll}
U : \text{Type} & U : \text{Type} \\
 x, y : U & x, y : U \\
h : \forall \mathcal{P}^U \rightarrow \text{Prop}. \mathcal{P}_x \rightarrow \mathcal{P}_y & h : \forall \mathcal{P}^U \rightarrow \text{Prop}. \mathcal{P}_x \rightarrow \mathcal{P}_y \\
P : U \rightarrow \text{Prop} & P : U \rightarrow \text{Prop} \\
h_0 : \mathcal{P}_y & h_0 : \mathcal{P}_y \\
? : \mathcal{P}_x \rightarrow \mathcal{P}_x & ? : \mathcal{P}_y \\
\end{array} \]

**Intros; Assumption.**

**Assumption.**

**Figure 1.1.** Constructing a proof of the symmetry of Leibniz equality.

In this sample session we construct a proof of the symmetry of the Leibniz equality defined as \( x = y \) if and only if \( \forall \mathcal{P}^U \rightarrow \text{Prop}. \mathcal{P}_x \rightarrow \mathcal{P}_y \). After unfolding the definition of \( =\), we call the tactic **Intros** that applies introduction rules as far as possible. This causes all antecedents of the formula to become assumptions. Next, we use the assumption \( h \) which is a proof of \( \forall \mathcal{P}^U \rightarrow \text{Prop}. \mathcal{P}_x \rightarrow \mathcal{P}_y \) specialized by the predicate \( \lambda z. \mathcal{P}_z \rightarrow \mathcal{P}_x \). To do that we call the tactic **Apply\( (h (\lambda z^U. \mathcal{P}_z \rightarrow \mathcal{P}_x)) \).** This generates two goals, namely \( \mathcal{P}_x \rightarrow \mathcal{P}_x \) and \( \mathcal{P}_y \) which are easily solved using **Intros** and the tactic **Assumption** which uses an assumption to solve the goal.

To make this style of development possible, the proof assistant needs to:

- be able to encode the theorem we want to prove in its language
- keep track of the open goals and the goal the user is focused on
• translate tactic calls into proof-constructs
• perform type proof checks
• correctly handle the incomplete proofs at the intermediate steps in order to ensure soundness.

The path from a logical formula to be proven to its proof is as follows. Using the formulas as-types embedding the formula is represented by a type. This type is given as a goal to the proof assistant and with the help from the user a term of this type is created. Using the correspondence between \( \lambda \)-terms and proofs, the result is translated back into a logical proof. For example, the term constructed by the development in Figure 1.1 is presented in Figure 1.2. The natural deduction

\[\lambda x, y^U, \lambda h^1 (\forall P^U \rightarrow \text{Prop}. P_x \rightarrow P_y). \lambda h^0_P y. (h (\lambda z^U. P_z \rightarrow P_x) (\lambda h^1_P x. h_0))\]

Figure 1.2. The proof-object constructed by the development in Figure 1.1.

Because of the incremental nature of the term construction process we would like to be able to define correctness criteria for the intermediate steps that can be automatically checked by the computer and such that conformance to those criteria automatically ensures correctness of the whole development. This has been done in a number of type systems where the so-called incomplete terms are introduced. These incomplete terms (also called open terms) are explicit (i.e. first-class) representations of terms in which some parts are missing. In such terms we introduce special variables called \( \text{meta-variables} \) that stand for a not yet known term and therefore a term containing meta-variables represents a partially constructed term.

In systems without binding (e.g. in Term Rewriting Systems [Ter03]) it is not very difficult to introduce first-class incomplete terms. Let us consider for example
1. INTRODUCTION

the case of Combinatory Logic (CL) where the terms are made up of variables and the combinators \( I, S \) and \( K \) using application and where the reduction is defined by

\[
\begin{align*}
Ix & \rightarrow x \\
Kxy & \rightarrow x \\
Sxyz & \rightarrow (xz)(yz)
\end{align*}
\]

If under 'incomplete term' we understand one with some of its subterms missing, it suffices to introduce meta-variables that stand for unknown terms: \( m, n, \text{etc.} \). These meta-variables behave like 'normal' variables except that on the meta-level we know that they stand for CL-terms and not objects. One can see that instantiation of such meta-variables will commute with the reduction relation which is the main requirement for the handling of meta-variables. Meta-variables introduced in such a way can be called first-order meta-variables.

However, we may want to have a more general notion of incomplete term where not only a whole subterm is missing. For example, if one tries to construct a CL-term in a forward manner (i.e. from the leaves of its parse-tree to the root), we have a situation where a subterm is unknown, in the sense that we do not know all of it, but it has some known subterms. To represent such 'unknowns' we can introduce 'second-order' meta-variables that allow us to view the unknown as a meta-function of its known parts. Let us write such meta-variables as \( m[x_1 \ldots x_n] \) where the arguments \( x_i \) stand for the known subterms of the unknown part \( m \).

To illustrate this approach, consider the term \( SIK \) and assume that the subterm \( S \) together with the two applications is missing. If we denote this missing part by \( m \), then \( m \) must have two parameters for the two known subterms \( I \) and \( K \) and therefore \( m[I, K] \) represents a term that can potentially be refined to \( SIK \) by instantiating \( m[x, y] \) with \( Sxy \).

The second order meta-variables seem to be suitable also for representing incomplete terms in the untyped \( \lambda \)-calculus. Consider for example the \( \lambda \)-term corresponding to the \( S \) combinator from above: \( S = \lambda xyz.(xz)(yz) \). If we remove the subterm \( xz \) we get an incomplete term that can be represented by \( \lambda xyz.m[x, y, z](yz) \). The parameters of \( m \) now are used to capture the fact that the unknown part may depend on \( x, y \) and \( z \). In the setting of the untyped \( \lambda \)-calculus, the second order meta-variables can be seen as variables ranging over contexts as defined in [Bar84]. There (see p.375) the set of contexts is defined by the following grammar:

\[
C := x \mid [\ ]_n \mid CC \mid \lambda x.C
\]

and the constructors \([\ ]_n\) stand for missing subterms. To make it clear why a meta-variable can be as standing for such a context, consider again the term \( \lambda xyz.m[x, y, z](yz) \). If in it we 'instantiate' \( m \) by \([\ ]_1[\ ]_2\), we get the complete term \( S \).

As in the case of CL, we can pose ourselves the question of constructing untyped \( \lambda \)-term stepwise in a forward manner. Then we would need to have an incomplete term in which a bound variable appears, but its corresponding \( \lambda \)-binder is not yet constructed (see Figure 1.4). For example, think of the incomplete term obtained from \( S \) by deleting the binders \( \lambda x \) and \( \lambda z \). One way to depict this term
would be

\[ A[\lambda y. B((xz)(yz))] \]

where the context \( A \) is intended to contain the binder \( \lambda x \) and \( B \) the binder \( \lambda z \). If \( A \) and \( B \) are these specific contexts, we see that the bindings are respected and \( x \) and \( z \) are bound variables. If however the specific context is unknown we cannot 'see' the bindings for \( x \) and \( z \). This leads us to the third-order meta-variables that we call binding meta-variables later in the thesis. To correctly specify the term \( S \) with \( \lambda x \) and \( \lambda z \) missing, we use the following notation:

\[ A[(x)\lambda y. B((z)(xz)(yz))] \]

The expression \( B[(z)M] \) stands for unknown term containing a binder for \( z \) and a known subterm \( M \) in its scope.

Using typed open terms one can model the interactive term construction process presented in Figure 1.5. We start with a completely unknown term with type given by the statement that we want to prove. When a tactic is applied, the meta-variable is solved with a term containing new meta-variables that represent the subgoals generated by the tactic. If a state is reached in which the term contains no more meta-variables, we actually have a term of the required type.

Open terms in type theory have been studied for a variety of systems (see Chapter 2 for an overview). Having in mind the formulas-as-types embedding, we ask the question *What are the logical counterparts of open terms in type theory?* To answer this question we start from a formalization of logic with first-class incomplete proofs that correspond to the usual representations of open terms in type theory. We can see that with this formalization we can represent incomplete proofs that are occurring in the usual backwards goal-oriented reasoning. Motivated by the somewhat unexpected negative answer to the question whether this also holds for forward reasoning, we propose a refined logic with incomplete proofs in which the holes can bind. This is necessary in order to represent forward proof-steps
in which a hypothesis that is supposed to be discharged in the complete proof (or a variable, supposed to be bound) needs to be used before its corresponding introduction rule is constructed.

The next step is the study of the typing systems that correspond to these formalizations of incomplete terms. The questions we are interested in here are *What are the open terms that correspond to incomplete terms and proofs with binding holes?* and *Can the formulas-as-types embedding be lifted to a sound and complete embedding into this typing system with open terms?* We answer the first one by introducing the binding meta-variables in the typing system and show that then the second question has an affirmative answer.

After having looked at the objects proof assistants work with, we turn our attention to the dynamic side of the process. As already mentioned, the user guides the proof assistant by means of tactics. By executing tactics the proof assistant transforms one incomplete proof into another. Since the tactics can be of varying complexity ranging from simple applications of deduction rules to complex decision procedures, they are often implemented as part of the proof assistant in its implementation language. This is a practical and efficient way to implement tactics, but it has its disadvantages. For example, writing a new tactic requires knowledge of implementation details and internal data representation of the proof assistant. Apart from that, the tactics written in this way are barely portable or backwards compatible with older versions. From the theoretical viewpoint however, the more serious problem is that the semantics of the tactics is given in terms of the semantics of the general-purpose programming language in which they are implemented. In practice this is not a big problem since the result a tactic produces can always
be checked for correctness, but it does make reasoning about tactics and proving properties like completeness practically impossible.

To address this issue specialized tactic languages are developed (like \( \mathcal{L}_{\text{tac}} \) for Coq). In such a language one is given a number of primitive tactics implemented by the proof assistant and means to construct more complex tactics from them. In the last part of the thesis we show that one can improve this result by presenting a calculus based on open terms in which one can encode both the tactics normally provided as primitive and the combinators on tactics.

By providing a typed operational semantics for this tactic calculus we can provide an integrated model of a hypothetical tactics-based interactive theorem prover. The proof-states of this prover are modelled with open terms. The tactics are terms in the tactic calculus and its semantics prescribes how to execute them in a given proof-state.

1.2. Overview of the Thesis

The contents of the thesis is distributed as follows:

In Chapter 2 we present a brief overview of existing systems of open terms in type theory and also of systems dealing with tactics. Our goal is not a complete presentation of the systems, but rather to present those of their features that are relevant for the treatment of incomplete terms so that we can put the developments in the following chapters into perspective.

In Chapter 3 we are interested in the question how to formalize the notions of incomplete proofs and terms in logic, and in particular in the case of higher-order logic (HOL). By means of several examples we introduce some typical uscases for incomplete proofs and terms in constructing proofs. We propose an extension (called oHOL) in which we can represent explicitly missing parts of proofs and terms. In this system every proof can be constructed backwards. Next, we continue with some more examples, mainly using forward reasoning which cannot be represented in oHOL and that call for an extension with unknowns that can also act as binders (holes with binding power). We present a second extension of HOL called bHOL with binding holes and show that it is complete with respect to forward constructions of proofs.

In Chapter 4 we present two systems extending the \( \lambda \)-calculus \( \lambda \text{HOL} \) corresponding to HOL. The two extensions are intended to correspond to the logics oHOL and bHOL introduced in Chapter 3. We present the systems and show that they preserve most meta-theoretical properties of \( \lambda \text{HOL} \). Inspired by the binding holes in bHOL we introduce similar constructions in \( \lambda \text{bHOL} \) – namely meta-variables whose parameters can also be parameterized themselves. In the last section of the chapter we take this concept one step further by considering Pure Type Systems (PTSs) with hereditary parametrization of arbitrary level for variables, constants and definitions.

In Chapter 5 we make explicit the intended correspondence between the extensions of higher-order logic with open terms from Chapter 3 and the extensions of the typing system \( \lambda \text{HOL} \) from Chapter 4. We do that by showing that the formulas-as-types embedding of HOL into \( \lambda \text{HOL} \) can be lifted to the level of the systems with open terms. Despite this result, in Section 5.3 we see that there are
differences between the approach to open terms from the viewpoint of logic and type theory. We discuss this mismatch and the possibilities to mend it. At the end of the chapter we compare our approach to open terms with the functional approach and conclude that if we commit ourselves to first-class open terms, we need to have a way to distinguish object- and meta-level functional dependencies and that parametrization is a good tool to do that.

In Chapter 6 we look at the next meta-level – the operations on open terms which in the context of interactive theorem provers are often called tactics. We present a calculus based on λHOL in which we are able to encode many common tactics and tacticals (meta-operations on tactics). We define typed operational semantics for this calculus and show that it is deterministic (if it terminates), sound and complete with respect to λHOL. The chapter ends with a case study defining a tactic for deciding minimal propositional intuitionistic logic and proving its correctness and completeness using the semantics of the tactic calculus.
CHAPTER 2

Related Work: Incomplete Terms and Applications

In this chapter we present an overview of existing systems of open (incomplete) terms for theorem proving. We restrict ourselves only to systems with dependent typing and first-class representation of unknowns. We also discuss related work on tactics and unification.

There are a number of $\lambda$-calculi with explicit, first-class representation of unknowns. Those include untyped calculi, calculi with simple types and dependently typed ones. Since the primary application we are interested in is interactive theorem proving, we will restrict ourselves to dependently typed systems. Our goal in this chapter is to present those features and properties of the systems that are relevant to the further discussions; their full description is beyond the scope of the present thesis. Interested readers may consult the provided references for a comprehensive reading on each of the systems.

After some preliminaries in Section 2.1, we will make an overview of existing systems of incomplete terms in type theory in Section 2.2. Then in Sections 2.3 and 2.4 we will discuss related work on tactics and unification.

2.1. Incomplete Terms in Type Theory: Preliminaries

2.1.1. Some Issues of Interest. When presenting each of the systems, we will be interested (among others) in the following notions:

Representation of incomplete terms. By definition, a system of open terms is one in which the unknowns have an explicit representation. This representation may vary and may have more than one component. For example, some systems use meta-variables and explicit substitutions which together form the representation of the unknown objects.

Computations with unknowns are usually the first problem one is confronted with. For soundness reasons one can not ignore the computations being executed over an unknown and should keep in some form the information about them. On the other hand, this should not block the computations, because the idea of introducing unknowns in the first place is that we want to reason and compute with them while delaying the moment of their solving. The computational mechanism strongly depends on the representation of the unknowns.

The solutions for the unknowns are the objects that we fill in for the placeholders. The questions here are what is a legal solution for an unknown and how to propagate the solution (instantiation) through the open term.

One of the goals that one tries to achieve in a calculus of open terms is a high level of integration of the open terms. Ideally the introduction of open terms
in a calculus should change a minimal number of its meta-theoretical properties. Hence one can measure the degree of integration by studying and comparing the properties of the original system and its extension with open terms.

We can speak informally of the expressive power of a system for open terms as the ‘ratio’ between the components of the system that can be left open and the components of the original system. For example, a system that allows us to work with unknown object terms and proofs in first-order predicate logic is less expressive than a system that allows also unknown formulas. This means that the user of a system with higher expressive power has more choices and and is more flexible by being able to delay more decisions.

2.1.2. The Conversion Rule and Open Terms. When we embed a logic into type theory we can convert a proof-construction problem into a problem of term-construction. In order to do that for systems with quantifiers like predicate logics, one needs dependent function types. In dependently typed systems usually we have in some form a conversion rule:

\[
\begin{align*}
M & \text{ is of type } A \\
B & \text{ is a type } A =_\beta B \\
\hline
M & \text{ is of type } B
\end{align*}
\]

This rule allows us to perform \(\beta\)-reductions and expansions on the level of types and because these are dependent types we may need to deal with computations with open terms in the typing system. Since we would like that instantiation of unknowns preserves typing, it is necessary that computation and instantiation commute.

2.1.3. The Transitivity Example. Suppose we have a type \(U\) and a binary relation \(R\) on it. Suppose also that \(a\) and \(c\) are of type \(U\) and we have to prove \((R \ a \ c)\). To do that using a proof of \(\forall x, y, z:U. (R \ x \ y) \rightarrow (R \ y \ z) \rightarrow (R \ x \ z)\) we need to eliminate the quantifiers for \(x, y\) and \(z\) and then provide proofs of \((R \ x \ y)\) and \((R \ y \ z)\). This will provide a proof of \((R \ a \ c)\) if we have eliminated \(x\) by \(a\) and \(z\) by \(c\).

It is however not clear what term to use to eliminate \(y\). This is a typical situation when we would like to delay the choice for a term and continue with the proof. Hopefully at a later stage we will know more about the unknown term and that can help us find a (partial) solution for it.

The same holds for the proofs of the antecedents \((R \ x \ y)\) and \((R \ y \ z)\). We would like to leave their proofs open. Furthermore, we would like to have the freedom to solve the parts left open in any order we like. For example, if next to the transitivity of \(R\), we know that \(\forall x:U. (R \ a \ x)\), we could solve the first of the proof obligations before we have decided what the value of the unknown \(y\) should be.

This is a classic example that demonstrates the need to introduce meta-variables. For each of the systems below we will show how they can deal with the situation above in order to illustrate the handling of open terms in that particular system.
2.2. Overview of Systems of Open Terms in Type Theory

2.2.1. ALF. ALF [Mag95] is an interactive proof editor based on Martin-Löf’s monomorphic type theory. It was the first system with explicit representations of incomplete terms. The terms of its calculus are given by the following grammar:

\[ M ::= x \mid c \mid [x]M \mid (M M) \mid M\sigma \mid \gamma \]\n
where \([x]M\) represents abstraction of \(x\) and \(\sigma\) is an explicit substitution. Note that in an abstraction term \([x]M\) the abstracted variable \(x\) is not given a type. ALF supports incomplete terms by introducing the so-called placeholders. These correspond to metavariables in other systems. The placeholders are denoted by identifiers starting with '?. As we see, in a term we may have explicitly given substitutions which are represented as lists of assignments. Substitutions can be fully composed as follows:

\[ \{\}\beta = \beta \]
\[ \{\gamma, x := e\}\beta = \{\gamma\beta, x := (e\beta)\} \]

On the level of types we distinguish between types and families of types which are given by:

\[ T ::= \text{Set} \mid T \rightarrow F \mid FM \mid T\sigma \]
\[ F ::= \text{El} \mid [x]T \mid F\sigma \]

Incomplete types are types containing incomplete terms and incomplete contexts are contexts containing incomplete types.

Each placeholder has a unique expected type (possibly incomplete) and a unique local context (also possibly incomplete). This means that the placeholder may only be refined by a term whose free variables are from its local context and in this context the term should have as type the expected type of the placeholder.

The typechecking in this system poses specific problems, some of which are not encountered in other systems because of the nature of the underlying type system. One of them is caused by the lack of type annotations of the bound variables in \(\lambda\)-abstraction. Consider for example the term

\([x]x\)

It can be inferred using the derivation rules that in the empty context this term is both of type \(\text{Set} \rightarrow \text{Set}\) and \(\text{Bool} \rightarrow \text{Bool}\) for example. Therefore the following problem

\[ f(\gamma_1, \gamma_1) : \text{Set} \]

where

\[ f : (\text{Set} \rightarrow \text{Set}) \rightarrow (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Set} \]

can be solved by instantiating \(\gamma_1\) with the term \([x]x\). But then we see that the first occurrence of \(?\) has expected type \(\text{Set} \rightarrow \text{Set}\) and the second \(\text{Bool} \rightarrow \text{Bool}\) which contradicts the condition for uniqueness of the expected type imposed on the placeholders.

A similar problem occurs if we are not careful about the local contexts. Consider for example

\[ g : (A \rightarrow A) \rightarrow A \rightarrow \text{Set} \]
and the problem

\[ g([x]?_2, ?_2) : \text{Set} \]

in the empty context. Considering the first occurrence of \(?_2\) which is in context \([x : A]\), we may decide to solve \(?_2\) by \(x\) obtaining

\[ g([x]x, x) : \text{Set} \]

which is not typable in the empty context, because \(x\) occurs as a free variable.

ALF solves these problems by disallowing the introduction of multiple occurrences of the same placeholder by the user. Possible duplications that may appear in the process of unification will then refer to the “major” occurrence when the placeholder was introduced and must therefore have the same type and same context (or extension of it).

The typechecking of complete terms is done by a modular algorithm which has two main phases [Mag95]. Using the typing rules, the algorithm produces a list of constraints that must hold in order to have a valid typing derivation of the judgment that we check. These constraints constitute a typed higher-order unification problem which has a solution if and only if the typing judgment holds.

Since typed higher-order unification is undecidable, we cannot check whether the system of constraints has a solution in general. Instead, it is simplified by decomposition and eliminating trivial constraints. During the simplification we may find out that some constraints cannot possibly have a solution and this means that the unification problem has no solution and hence the typing judgment does not hold. However, if this does not happen, we are left with a number of equations involving placeholders. Therefore the validity of the typing judgement involving placeholders is parameterized by a unification problem. When at a later stage we want to instantiate one of the placeholders, we will need not only to check its type, but also to check whether or not it satisfies the constraints left from the previous typecheck. It may be the case that the instantiation invalidates them and then it will be rejected, but in general, we will get a new set of constraints that will be kept by the system to restrict the future instantiations of the remaining placeholders.

Let us consider two examples. Suppose that \(f\) is of type \(\text{Nat} \to [n](Pn \to \text{Set})\) where \(P\) is a predicate on \(\text{Nat}\) and let \(M\) be a proof of \(P0\). Consider the typechecking problem whether \(f(?_1, M)\) is of type \(\text{Set}\). The constraints generated will be

\[ ?_1: \text{Nat} \\
\ P0 = P?_1 : \text{Set} \]

that after simplification yield \(?_1 = 0\). Therefore, the only possible instantiation for \(?_1\) is 0.

On the other hand, consider \(f : \text{Nat} \to \text{Bool} \to \text{Set}\) and the typechecking problem \(f(?_1, ?_1) : \text{Set}\). The constraints will be \(?_1 : \text{Nat} \) and \(\text{Nat} = \text{Bool}\) which are obviously unsatisfiable and the typecheck will fail.

The metatheoretical properties of the system are not studied in [Mag95] and the results obtained are relative to some meta-theory assumptions such as that constructors are one-to-one, that equal ground terms have the same outermost
2.2. Overview of Systems of Open Terms in Type Theory

constructor and that computation on provably equal typable terms terminates and leads to equal normal forms.

Let us consider how the transitivity example can be represented in ALF. Suppose we have defined $U$, $R$, $a$ and $c$ of their appropriate types and that we have a term $\text{trans}$ of type $(x, y, z: U) \rightarrow (R x y) \rightarrow (R y z) \rightarrow (R x z)$. Then our goal would be to create a term of type $(R a c)$:

$$\text{theorem-aRc:: } (R a c) = ?$$

We specify that we want to use $\text{trans}$:

$$\text{theorem-aRc:: } (R a c) = \text{trans } ?1 \ ?2 \ ?3 \ ?4 \ ?5$$

The system typechecks $\text{trans } ?1 \ ?2 \ ?3 \ ?4 \ ?5$ and finds out that the typing constraints are $?1 = a$ and $?3 = c$ and automatically solves $?1$ and $?3$, leaving

$$\text{theorem-aRc:: } (R a c) = \text{trans } a \ ?2 \ c \ ?4 \ ?5$$

Now we have 3 goals: to find a witness $?2$ of type $U$ and to prove that $(R a ?2)$ and $(R ?2 c)$. At this point we may choose to provide the witness directly, or try to prove the two other goals. This may be a good idea if we cannot directly see a suitable candidate for $?2$. Imagine for example that somehow we manage to find a proof $M$ of $(R a b)$ for some $b$. Then we can instantiate $?4$ by $M$ which automatically will result in the constraint $?2 = b$ and only the goal $?4: (R b c)$ will remain.

On the level of the theory of ALF the first state is represented by a list of equations which is roughly

$$\ldots \ [?0 \ : \ (R a c) \ \Gamma]$$

where $\Gamma$ is a context declaring the appropriate variables. When we refine $?0$ by $\text{trans}$ and use the witnesses $a$ and $c$, we get approximately

$$\ldots \ [?2 \ : \ U \ \Gamma]$$
$$[?4 \ : \ (R a ?2) \ \Gamma]$$
$$[?5 \ : \ (R ?2 c) \ \Gamma]$$
$$[?0 = \text{trans } a ?2 c \ ?4 \ ?5 \ \Gamma]$$

At a later stage, if we succeed in proving $M: (R a b)$ for some $b$, we can solve $?4$ by $M$:

$$\ldots \ [?2 \ : \ U \ \Gamma]$$
$$[?5 \ : \ (R ?2 c) \ \Gamma]$$
$$[(R a ?2) = (R a b) \ \Gamma]$$
$$[?0 = \text{trans } a ?2 c \ M \ ?5 \ \Gamma]$$

from which by simplifying we can (automatically) infer $?2 = b$. 

Summary: ALF

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>MTA(^1)</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>?</td>
</tr>
<tr>
<td>Typechecking</td>
<td>decidable</td>
</tr>
<tr>
<td>Type inference</td>
<td>not decidable</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>MTA</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>no</td>
</tr>
</tbody>
</table>

\(^1\)MTA – meta-theoretical assumption.

2.2.2. The \(CC_L\)-calculus. The \(CC_L\)-calculus [Mn97] is a calculus of explicit substitutions using de Bruijn indices. It provides a formal framework for representing proof-terms with placeholders.

The calculus is an extension of a presentation of the Calculus of Constructions [CH88] with de Bruijn indices and explicit substitutions in the style of the \(\lambda\sigma\) calculus [ACCL91].

The well-formed expressions in \(CC_L\) are given by the following grammar:

- natural numbers \(n\) ::= \(0\)\(|\)\(n + 1\)
- meta-variables \(M\) ::= \(X\)\(|\)\(Y\)\(\ldots\)
- terms \(A, B, M, N\) ::= \(\text{Kind}[Type]|\(1\)|\(\lambda_A.M|\Pi_A.M|(MN)|[S].M|M\)
- substitutions \(S, T\) ::= \(\uparrow^n[M.A]|S \circ T\)

As we can see, substitutions are represented explicitly in the calculus and instead of meta-level substitution, \(\beta\)-reduction uses the built-in substitutions. The reduction rules for the calculus are presented in Figure 2.1. Using these rules, substitutions that are generated by \(\beta\)-reductions in open terms can be propagated through the structure of the term until they hit upon a meta-variable where they are suspended until an instantiation of the meta-variable is found.

\[\begin{align*}
\text{(Beta)} & \quad (\lambda_A.M)N \rightarrow [\uparrow^0 \cdot N.A]M \\
\text{(Lambda)} & \quad [S](\lambda_A.M) \rightarrow \lambda_{[S].A}(([\uparrow^1 \circ S] \cdot 1.A)M) \\
\text{(Pi)} & \quad [S](\Pi_A.B) \rightarrow \Pi_{[S].A}(([\uparrow^1 \circ S] \cdot 1.A)B) \\
\text{(App)} & \quad [S](MN) \rightarrow ([S]M[N]) \\
\text{(Clos)} & \quad [T]\cdot S.M \rightarrow [T \circ S]M \\
\text{(VarCons)} & \quad [S \cdot M.A]1 \rightarrow M \\
\text{(Id)} & \quad [\uparrow^0]M \rightarrow M \\
\text{(Map)} & \quad T \circ (S \cdot M.A) \rightarrow [T \circ S] \cdot [T].M.A \\
\text{(IdS)} & \quad S \circ [\uparrow^0] \rightarrow S \\
\text{(ShiftCons)} & \quad (S \cdot M.A) \circ \uparrow^{n+1} \rightarrow S \circ \uparrow^n \\
\text{(ShiftShift)} & \quad \uparrow^m \circ \uparrow^{n+1} \rightarrow \uparrow^{m+1} \circ \uparrow^n \\
\text{(Shift1)} & \quad \uparrow^1 \cdot 1.A \rightarrow \uparrow^0 \\
\text{(ShiftS)} & \quad \uparrow^{n+1} \cdot [\uparrow^n].1.A \rightarrow \uparrow^n \\
\text{(Type)} & \quad [S]Type \rightarrow Type
\end{align*}\]

Figure 2.1. The \(\lambda_{IL}\) rewrite system.
Proof-term construction is done in the following way. We translate the formula to be proven to a $CC_L$-type and the language and the global assumption to a $CC_L$-context. In this context we introduce a meta-variable representing the unknown proof-term. Using well-typed instantiations we can partially solve this meta-variable and then iterate this process for the new meta-variables that may have been introduced.

In order to be able to type meta-variables, judgments in the system are extended to contain signatures (denoted by $\Sigma$) which are lists of meta-variable declarations. There are three sorts of judgments. The first one is of the form $\Sigma; \Gamma \vdash M : A$ and states that in signature $\Sigma$ and context $\Gamma$ the term $M$ has a type $A$. The judgments expressing the validity of a signature have the form $\vdash \Sigma$, the judgment expressing that a context $\Gamma$ is valid in a signature $\Sigma$ is $\vdash \Sigma; \Gamma$. The type $A$, and the context $\Gamma_i$ of a meta-variable $X_i$ in a signature $\Sigma = (X_1 : \Gamma_1, \ldots, X_n : \Gamma_n, A_n)$ may depend only on meta-variables with index $j$ where $j < i$. The rules for the meta-variables are the following:

$$(\text{MV-dec}) \quad \vdash \Sigma; \Gamma \vdash A : s \quad \text{X-fresh}$$

$$(\text{MV}) \quad \Sigma; \Gamma \vdash A \equiv_{\mu \nu L} \Gamma \quad \Sigma; \Gamma \vdash X : A \quad X \in \Sigma$$

An occurrence of a meta-variable $X$ is always typed in the same local context that corresponds to its declaration in the signature. The full set of rules and a study of the properties of $CC_L$ can be found in [Mn97]. There, Muñoz also presents an application of the calculus to proof-synthesis. This requires an extension of the calculus with the so-called constrained signatures that may contain unification constraints of the form $M \equiv_{\Gamma} N$ expressing the assumption that in context $\Gamma$ the terms $M$ and $N$ (which have the same type) are equal. This results in a presentation of the proof-synthesis method of Dowek [Dow93] using explicit substitutions and typed meta-variables.

One specific feature of $CC_L$ is that the substitutions are annotated with types. These annotations are created by the rules (Beta), (Lambda) and (Pi) and are preserved during the life-cycle of the substitution. This is necessary in order to obtain the Subject Reduction property. Allowing full composition of the explicit substitutions leads to loss of Strong Normalization (see [Mei94]). This is also the case with $CC_L$. The rest of the meta-theoretic properties of the calculus include confluence on well-typed terms, subject reduction, type uniqueness, weak normalization, etc.

In $CC_L$ we have the needed functionality to represent the transitivity example. Let $\Gamma$ be the context

$$U : Type.R:A \to A \to Type, \ \trans : \Pi x, y, z:U.(Ryx) \to (Ryz) \to (Rxz).a:U.c:U$$

Then the goal of proving $\trans$ can be expressed by the following judgment:

$$X : \Gamma; \trans \vdash X : \Gamma$$

$^1$Using names for variables
refining by \textit{trans} results in
\[ Y;\Gamma \vdash U.P_1;\Gamma(Yc).P_2;\Gamma(RYc); \Gamma \vdash (\text{trans} \ a \ Y \ c \ P_1 P_2); (\text{Rac}) \]
which shows that the transitivity example can be encoded in \( CC_c \).

\underline{Classification Summary:} \( CC_c \)

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>yes</td>
</tr>
<tr>
<td>Typechecking</td>
<td>decidable</td>
</tr>
<tr>
<td>Type inference</td>
<td>decidable</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>no</td>
</tr>
</tbody>
</table>

\( \text{Technical Issues:} \)
- Uses meta-vars.: yes
- Explicit Subst.: yes
- Based on: \( CC \)
- de Bruijn Ind.: yes

\textbf{2.2.3. Typelab.} The Typelab system [SLvH98, Str99] is a tool aimed at the specification and the refinement of functional programs. Specifications are represented in type theory as dependently typed records (\( \Sigma \)-types) and finding programs meeting these specifications is translated to finding inhabitants of the corresponding \( \Sigma \)-type.

The basis of Typelab is the Extended Calculus of Constructions (ECC, see [Luo99]) with cumulative universes. ECC is an extension of the Calculus of Constructions [CH88] with dependent \( \Sigma \)-types and universes. Typelab employs a dependently-typed language with meta-variables, a very restricted form of explicit substitutions and dependent \( \Pi \)- and \( \Sigma \)-types. It also features a cumulative hierarchy of type universes:

\[ \text{Prop} : \text{Type}_0 : \text{Type}_1 : \ldots : \text{Type}_n : \ldots \]

The terms of the language are given by the following grammar:

\[ T ::= x \mid \text{Prop} \mid \text{Type} \mid m \sigma \mid \Pi x : T. T \mid \lambda x : T. T | \Sigma x : T. T \mid TT \mid \text{pair}(T, T) \mid \pi_1(T) \mid \pi_2(T) | \]

\[ \sigma ::= [] \mid [x := T] :: \sigma \]

where \( x \) is a member of the set of variables \( \text{Var} \), \( m \) is from the set of meta-variables \( MVar \) and \( :: \) denotes concatenation. Since we work in an extension of the calculus of constructions, we do not distinguish between terms and types on syntactic level.

In Typelab open terms are represented by means of terms which contain meta-variables (denoted by identifiers starting with \( ? \)). These meta-variables are intended to be placeholders or “holes” for terms that will be constructed at a later stage. Each meta-variable has attached to it an explicit substitution. Meta-variables without substitutions are not valid terms, but one can always identify a meta-variable with an empty substitution with the meta-variable itself.

It is assumed that a function \( \text{svars} : MVar \to \text{List}(\text{Var}) \) is given that assigns a list of variables to each meta-variable. The intended meaning is that \( \text{svars}(?n) \) is the list of variables which may occur free in \(?n\) in a term that will instantiate the meta-variable. So, if \( \text{svars}(?n) = (x, y) \), then \( \lambda z : \text{xx} \) is eligible to be considered as a candidate to instantiate \(?n\), but \( \text{xx} \) is not.

Typelab solves the problems of commutation of computation and instantiation using the substitutions attached to the meta-variables. Meta-level substitution is
extended to terms with meta-variables. For a meta-variable $?n$ and $x \in \text{svars(?n)}$ when we substitute $N$ for $x$ in $?n[y := M]$ we get $?n[x := N, y := M\{x := N\}]$ (up to the order of the variables in the explicit substitution) When a meta-variable is instantiated, the substitutions attached to the meta-variables are executed on the term that instantiates it. Therefore, if in the term $?n[x := M]$ we instantiate $?n$ by $x$ we get $M$.

Suppose we have a meta-variable $?n$ and consider the term $(\lambda y : A.?n)x$. Assume also that $y \in \text{svars(?n)}$. Then we have commuting instantiation and $\beta$-reduction:

$$(\lambda y : A.?n)x \xrightarrow{\beta} (\lambda y : A.y)x \xrightarrow{\beta} x$$

$$(?n[y := x]) \xrightarrow{\beta} ?n[y := x] \xrightarrow{\beta} x$$

Naturally not all terms of the calculus represent ‘correctly’ constructed incomplete terms. The notion of correctness is given by the typing rules of the calculus. We have the usual rules of the ECC extended with rules for typing meta-variables. When typing meta-variables, we always have in mind some proof problem. This proof problem in effect parameterizes the derivation rules by globally describing which meta-variables may occur in the terms that we want to type and also gives their contexts and types.

Formally, a proof problem $P$ is a triple $\langle M_P, ctxt_P, type_P \rangle$ where $M_P$ is a finite set of meta-variables, $ctxt_P$ is a finite function which for every $?n$ in $M_P$ produces a context such that $\text{dom(ctxt(?n))} = \text{svars(?n)}$ and the function $\text{type}_P$ assigns a type to each meta-variable in $M_P$.

Since $M_P$ is a finite set, we can think of the proof problem $P$ to be a triple $P = \langle \langle ?_1, \ldots, ?_k \rangle, \langle \Gamma_1, \ldots, \Gamma_k \rangle, \langle T_1, \ldots, T_k \rangle \rangle$ expressing the fact that we are looking for $k$ terms $?_1, \ldots, ?_k$ and each $?_i$ in context $\Gamma_i$ should have type $T_i$. It is allowed for some $T_i$ or $\Gamma_i$ to contain $?_j$ in case this does not lead to circular dependencies between the meta-variables. More precisely, the relation defined by $?_i < ?_j$ if and only if $?_i$ occurs in $T_j$ or $\Gamma_j$ should be an irreflexive partial order. For a fixed proof problem $P$ this is a finite relation and consequently this question is decidable.

The typing rules involving meta-variables in Typelab are:

$$\frac{ctxt(?n) + \text{type(?n)} : Type_j}{ctxt(?n) + ?n} : \text{type(?n)} \quad \text{(MV-base)}$$

If $z \notin \text{FV}(\Gamma) \cup \text{FV}(\Delta) \cup \text{dom}(\sigma)$:

$$\frac{\Gamma \vdash T : Type_j \quad \Gamma, \Delta \vdash ?n \sigma : N}{\Gamma, z : T, \Delta \vdash ?n \sigma : N} \quad \text{(MV-weak)}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma, x : T, \Delta \vdash ?n \sigma : N}{\Gamma, \Delta \{x := t\} \vdash \{?n \sigma\} \{x := t\} : N \{x := t\}} \quad \text{(MV-\beta-Red)}$$

The typing system of Typelab inherits all the usual meta-theoretical properties of the original ECC. One can prove for example that the typable terms have
principle types, that the subject reduction property and the confluence are preserved and that the calculus is strongly normalizing. This shows that open terms are very well integrated in Typelab.

The transitivity example that we described in Section 2.1.3 can be expressed in Typelab. Suppose \( \Gamma \) is the following context containing the declarations for a set \( U \), a binary relation \( R \), two elements \( a \) and \( c \) of \( U \) and the assumption \( \text{trans} \) that \( R \) is transitive:

\[
U : \text{Type}, R : U \to U \to \text{Prop}, a : U, c : U, \forall x, y, z : U \ xRy \to yRz \to xRz
\]

In this context we want to prove \( a Rc \) which can be expressed by \( \Gamma \vdash ?_1 : a Rc \). Using \( \text{trans} \) we instantiate \( ?_1 \) by the term \( (\text{trans} a ?_2 c ?_3 ?_4) \), obtaining a new proof problem with these three goals\(^2\)

\[
\begin{align*}
\Gamma & \vdash ?_2 : U \\
\Gamma & \vdash ?_3 : a R ?_2 \\
\Gamma & \vdash ?_4 : ?_2 R c
\end{align*}
\]

which we are free to solve in any order we want.

**Summary: Typelab**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>Uses meta-vars. yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>Explicit Subst. yes</td>
</tr>
<tr>
<td>Typechecking</td>
<td>MV only</td>
</tr>
<tr>
<td>Type inference</td>
<td>Based on Extended Calculus</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>de Bruijn Ind. no</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td></td>
</tr>
</tbody>
</table>

\(^2\)To be precise these three goals form the following proof problem: \( P = (\langle ?_2, ?_3, ?_4 \rangle, (\Gamma, \Gamma), (U, a R ?_2, ?_2 R c)) \).

2.2.4. OLEG. OLEG [McB99] is a calculus of open terms for the ECC with local definitions but without \( \Sigma \)-types. It consists of two layers - OLEG core and OLEG development calculus. The core is essentially ECC with definitions and the development calculus is wrapped around it adding meta-variable support.

OLEG is developed as a tool to study dependently typed functional programs and is partly implemented as an extension of the proof-assistant LEGO [Pol].

The OLEG calculus has several specific features. The first one is the introduction of explicit binders for unknowns that allow a meta-variable to be treated as a normal variable in many respects. The second is the way OLEG handles reductions involving meta-variables. Due to the locality of the declarations, during computation some meta-variable binders may get copied and it is not possible to express that there should be a link between two different copies of the declaration. For that reason some restrictions are being imposed. Finally, it differs from other systems also in the way meta-variables are solved. This is done not by meta-level instantiation, but by converting meta-variable declarations into local definitions.

The syntax and typing of OLEG are given by a number of rules. We will not present all of them here, for a complete list we refer the reader to [McB99].
terms of OLEG are defined by:
\[
T := x \mid \text{Type} \mid (TT) \mid \forall x:T.T \mid \lambda x:T.T \mid ?x:T.T \\
\mid !x := T:T.T \mid ?x \approx T:T.T
\]
The idea is that dependent types are constructed by the \(\forall\)-binder, \(\lambda\)-abstraction is introduced by the \(\lambda\)-binder, local definitions by the \(!\)-binder and meta-variables by the \(?\)-binder. If from the grammar we remove the clauses for \(?\)-binding we obtain the core OLEG terms.

A context in OLEG is a list of bindings of different variables (including the binders): \(\lambda x:Type, \lambda y:A, !y := x:A, ?z:A \rightarrow A\)

An unknown in OLEG is represented by a variable introduced by a \(?\)-binder. Therefore, meta-variables can be introduced globally by a binding in a context or locally by a local binder in the term of a typing judgment.

The computation in OLEG is defined by means of special judgments that allow context-dependent reductions. The reduction and conversion relations are defined as the usual closure of the following three basic reduction steps:

\[
\begin{align*}
(\beta) \quad & \Gamma \vdash (\lambda x:T.N)M \rightarrow_\beta !x = M:T.N \\
(\delta) \quad & \Gamma, !x = M:T, \Delta \vdash x \rightarrow_\delta M \\
(\lambda) \quad & \Gamma \vdash !x = M:T.N \rightarrow_\lambda N \text{ if } x \not\in FV(N)
\end{align*}
\]

Note that \((\beta)\) does not involve externally defined substitution, it just converts a \(\beta\)-redex into a \(\delta\)-redex:
\[
(\lambda x:T.N)M \rightarrow_\beta !x = M:T.N \rightarrow_\delta \ldots \rightarrow_\delta !x = M:T.N[x/M] \rightarrow_\lambda N[x/M]
\]

The typing rules for the calculus mimic those for ECC, with substitution replaced by appropriate use of local definitions. For example the rule for typing applications looks like this:

\[
(\text{app}) \quad \frac{\Gamma \vdash F:T.S \quad \Gamma \vdash t : S}{\Gamma \vdash Ft !x = t:S.T}
\]

The part that is relevant to our exposition is the extension of the core calculus with open terms. As we already mentioned, the term language has a binder for meta-variables “?”. So, if \(M\) is a term and \(T\) is a type, then \(?x:T.M\) is also a term. The binder declares a meta-variable and this meta-variable can be used as a normal variable in its scope: \(\lambda f:A. ?x:B.fx\). Another feature is the introduction of “guesses” which represent a term that may possibly be a solution of a meta-variable: \(?x \approx P:S.M\). The “\(\approx\)” sign is used to emphasize the fact that this expression cannot be treated as a local definition and \(x\) cannot be replaced by \(P\) in \(M\). This allows the user of the system to attach and remove guesses at any time. This is appropriate when one has to try several possible instantiations, but does not know in advance which of them will turn out to be the correct one.

The term construction process goes like this: first we introduce a meta-variable of the appropriate type \(?x:A.M\), then we try to solve \(x\) by \(P_1\) resulting in \(?x \approx P_1.M\). However, this may be a wrong guess, so we can change it to \(P_2\) and so on, until we have found the right one \(?x \approx P.M\). Once this has happened, we can turn the guess into local definition \(!x = P.M\). All these operations can be shown to be sound by means of appropriate admissible rules.

For every meta-variable we have to be able to uniquely identify the context in which it is to be solved and its type. The type is explicitly given in its declaration
and the context can be determined by the position of its declaration (i.e. the context is formed by all variables in whose scope the declaration of the meta-variable is).

The treatment of meta-variables in such a way is not enough to avoid problems. For example we can form terms like \( \lambda f : (\forall x:A.x)f \) whose typability is problematic, especially since we want instantiation to preserve typability, to be able to solve meta-variables in arbitrary order and to have commuting hole filling and computation.

McBride in [McB99] relates those problems to meta-variable declarations leaking into types and imposes restrictions on the system which guarantee that declarations of meta-variables will not occur in the types involved in a derivable judgment.

The rules involving meta-variable declarations are given below:

\[
\begin{align*}
\text{(declare)} & \quad \frac{\Delta \vdash \text{valid}}{\Delta, ?x:T \vdash \text{valid}} \quad \frac{\Delta \vdash T : \text{Type}}{\Delta, ?x:T \vdash \text{valid}} \quad \frac{\Delta \vdash p:T}{\Delta, ?x \approx p \vdash T \vdash \text{valid}} \\
\text{(hole)} & \quad \frac{\Delta \vdash ?x:S \vdash p:T \quad x \notin \text{FV}(T)}{\Delta \vdash ?x:S : p : T} \quad \frac{\Delta \vdash ?x \approx q:S \vdash p : T \quad x \notin \text{FV}(T)}{\Delta \vdash ?x \approx q : T}
\end{align*}
\]

Note the side conditions in (hole) and (guess). They disallow a meta-variable bounded in a term to occur in its type. If this condition were not present, we would have to accept meta-variable declarations in types.

The side conditions in these two typing rules however are not enough to guarantee that ?-binders do not appear in types, because they may still be put there by a reduction using the conversion rule. To prevent this, applications and \( ! - \) bounded values are not allowed to contain ?-binders (i.e. if \( !x = s:S.M \) and \( fs \) are terms, then \( s \) is not allowed to contain ?-binders) and \( \beta \) - and \( \delta \) - reduction are removed from the contraction schemes for partial constructions. Using the embedding of core terms into the development calculus the system is still allowed to do computations on expressions with meta-variables (declared in the context), but computations on terms with ?-binders are disallowed.

The development calculus has most of the meta-properties of the core system. McBride [McB99] shows that among others, the development calculus enjoys confluence, Subject Reduction, Cut, Strong Normalization and a number of other properties. In that sense we can say that the open terms are well integrated into the development calculus, but this comes with the price of the somewhat reduced expressive power.

To represent the transitivity example in OLEG we need the following declarations:

\[
\begin{align*}
\lambda U & : \text{Type}, \\
\lambda R & : U \rightarrow U \rightarrow \text{Type}, \\
\lambda a,c & : U, \\
\lambda \text{trans} & : \forall x, y, z:U. (Rx) \rightarrow (Ry) \rightarrow (Rx), \\
?x_1 & : (Rac)
\end{align*}
\]
Then refining \( x_1 \) by trans and filling in \( a \) and \( c \) for \( x \) and \( z \) we get

\[
\begin{align*}
\ldots & \\
?x_1 \approx \ & ?y:U \\
?r_1: & (R \ a \ y) \\
?r_2: & (R \ y \ c) \\
\text{trans a y c} & r_1 \ r_2 \\
: & (R \ a \ c)
\end{align*}
\]

with three meta-variables to solve: \( y, r_1 \) and \( r_2 \).

**Summary: OLEG**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>yes</td>
</tr>
<tr>
<td>Typechecking</td>
<td>decidable</td>
</tr>
<tr>
<td>Type inference</td>
<td>decidable</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>yes</td>
</tr>
<tr>
<td>Uses meta-vars.</td>
<td>yes</td>
</tr>
<tr>
<td>Explicit Subst.</td>
<td>no</td>
</tr>
<tr>
<td>Based on</td>
<td>Extended Calculus of Constructions</td>
</tr>
<tr>
<td>de Bruijn Ind.</td>
<td>no</td>
</tr>
</tbody>
</table>

### 2.2.5. The \( \lambda[|\)cube

In [dB78] de Bruijn introduced the notion of segment. A segment is a \( \lambda \)-term with exactly one hole that is placed in the leftmost leaf in the tree representation of \( \lambda \)-terms. Later in [Bal86, Bal94] Balsters formalized this notion and provided a typing system for segments. The notion of segment is a special case of the notion of context. A context in the (untyped) \( \lambda \)-calculus is introduced in [Bar84] as a term with special places called holes, where other expressions can be placed. The notion of context\(^3\) is used in the proofs of several theorems but is left on meta-level as a notational convention. Examples for contexts are

\[
C = \lambda x. x(\lambda y.y\Box) \quad y\Box
\]

Motivated by applications from programming, proof checking and linguistics, Bogner formalizes the untyped contexts in the \( \lambda c \)-calculus [BdV, Bog02] The formalization takes into consideration the effect of variable capturing which occurs when the term filling a hole has a free variable which becomes bound by a binder in whose scope the hole is situated. As an example, suppose in the context \( C \) above we want to put the term \( M = xy \). Then the result is

\[
C[M] = \lambda x. x(\lambda y.y(\lambda y.(xy)))
\]

and the variables \( x \) and \( y \) of \( M \) become bound.

Such treatment however has as disadvantage that contexts cannot be identified up to \( \alpha \)-conversion (renaming of bound variables) and hence \( \beta \)-reduction is also problematic.

The systems of the \( \lambda[|\)cube are the corresponding systems of the \( \lambda \)-cube extended with a formalization of dependently-typed contexts.

---

\(^3\)not to be confused with the term context as a list of declarations of variables in other systems
In $\lambda[\cdot]$ we have a few new term constructors: multiple abstraction $\Lambda_n \bar{x} : \bar{T}.M$, multiple dependent products $\Lambda_n \bar{x} : \bar{T}.B$ and multiple applications $M(N)_n$, a binder to introduce hole variables $\delta h : T.M$, a binder to type hole abstractions $\overline{\delta h} : T.B$ and hole-filling application $M[N]$.

Using them we can represent open terms in the following way. Each hole has to be supplied with a list of variables $\bar{x}$ with their types $\bar{T}$. In the context of these variables the hole should have some type $B$ (thus $\bar{x}$ may occur in $B$). This is then translated into a declaration $\delta h : (\Lambda \bar{x} : \bar{T}.B).M$ that introduces a hole $h$ of type $B$ parameterized by $\bar{x}$ into scope and thus $h$ can be used in $M$. Typically this is done by supplying a number of terms $\bar{t}$ corresponding in type to $\bar{x}$ and then forming the term $h(\bar{t})$. We illustrate this by showing how to obtain commuting hole filling and $\beta$-reduction in the systems of $\lambda[\cdot]$:

$$
\delta h : (\Lambda x : A.A).(\lambda y : A.h(y))z \xrightarrow{\beta} \delta h : (\Lambda x : A.A).h(z)
$$

fill $h(x)$ by $x$

$$
(\delta h : (\Lambda x : A.A).(\lambda y : A.h(y))z)[\Lambda x : A.x] \xrightarrow{\beta} (\delta h : (\Lambda x : A.A).h(z))[\Lambda x : A.x]
$$

Note how the distinction between the ‘formal’ parameter $x$ of the hole and the ‘actual’ parameters $y$ and $z$ creates a flexible coupling between them which makes it possible to rename one of them (achieving $\alpha$-conversion) without losing the connection. The computational mechanism takes care of replacing the formal parameters by the actual ones in order to obtain a correct final result.

Note also that the only way a variable in the solution can get ‘captured’ during instantiation is if it is one of the formal parameters that has as a corresponding actual parameter a bound variable. This representation of holes is compatible with the variable convention for renaming of bound variables while admitting variable capture.

The diagram above uses three kinds of reduction. The first one is the usual $\beta$-reduction:

$$
(\lambda x : A.M)N \rightarrow_{\beta} M[x := N]
$$

and we have the context-related reductions:

$$
(\Lambda \bar{x} : A.M)(\bar{N}) \rightarrow_{\mu} M[\bar{x} := \bar{N}]
$$

$$
(\delta \bar{x} : A.M)[N] \rightarrow_{fill} M[x := N]
$$

with the usual assumption for renaming of conflicting variable names. With resolved computational problems, the rewriting rules of $\lambda[\cdot]$ are shown to be confluent.

Next comes the problem of giving a typing system for the $\lambda[\cdot]$-calculus. The system of the $\lambda[\cdot]$-cube is an extension of the $\lambda$-cube where $\lambda$-abstractions are given $\Pi$-types as usual, but $\Lambda$-abstractions have $\Lambda$-types and $\delta$-abstractions have $\delta$-types. The rules for forming the extra types are also restricted by the set of pairs of sorts $\mathcal{R}$ in a way similar to the $\lambda$-cube.
This means that $\vec{x} : A.B$ can be formed and will be of sort $s$ if there are $s^1, \ldots, s^n$ such that $A_i$ can be shown to be of sort $s^i$ in a context extended with the assumptions $x_1 : A_1, \ldots, x_{i-1} : A_{i-1}$. $B$ is of sort $s$ in a context extended with $\vec{x} : A$ and for each $i$ the pair $(s^i, s)$ is in $\mathcal{R}$.

The rules for forming $\delta$-types are a bit more complicated. Two new sorts $\triangle_\ast$ and $\triangle_\square$ are introduced. $\triangle_\ast$ is, roughly speaking, used to type $\delta x : A.B$ where normally $\Pi x : A.B$ would be of sort $s$.

The formation rules for $\delta$-types are also restricted by the set $\mathcal{R}$. As a consequence, it is not possible to form a $\delta$-type whose arguments are in the sort of holes. This means that it is not allowed to have 'nested' declarations of meta-variables where the types of an argument of a meta-variable is again a $\delta$-type.

The calculus of the $\lambda[\ ]$-cube is shown to be confluent on all terms and it has good meta-theoretical properties as Subject reduction, Strong normalization, etc.

The introduction of binders for meta-variables in $\lambda[\ ]$ comes together with the introduction of an explicit hole-filling operation. This means that it is unclear how to solve meta-variables introduced by binders that are embedded deep in the term because the subterm introducing the meta-variable will change its type after instantiation.

The transitivity example can be formulated in systems of the $\lambda[\ ]$-cube that are strong enough to express the first order property of transitivity. In those systems we can formalize the problem by the following judgment

$$\Gamma \vdash \delta p : \text{Rac}.p : \delta p : \Delta.\text{Rac}.$$

where $\Gamma = U : s, R : U \to U \to s, tr : \forall x, y, z : U. Rxy \to Ryz \to Rxz, a, c : U$. When we decide to use the transitivity, we use hole-application to fill in the hole $p$ by the term $t r a y c p_1 p_2$

$$\Gamma \vdash \delta y : U.\delta p_1.\text{Ray.}\delta p_2 : \text{Ryc.}(\delta p : \text{Rac}.p)[t r a y c p_1 p_2] : \text{Rac}.$$

A common feature in the systems of the $\lambda[\ ]$-cube and OLEG are the binders for locally introducing meta-variables. In $\lambda[\ ]$ hole abstractions are typed by $\delta$-types and in this way meta-variable binders are allowed in the types. Note that because $\delta$-terms have no computational power, there is no problem that the binders appear in the types. OLEG however lacks an analog of the $\delta$- binder and to prevent meta-variable binders from entering the types has to impose restrictions on the calculus. The systems of the $\lambda[\ ]$-cube demonstrate that these restrictions could be lifted by adding a binder to type $\vec{x} : ?^n$-abstractions. The price however is the abovementioned problem with filling of such holes.

**Summary: $\lambda[\ ]$**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Yes</th>
<th>Technical Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluence</td>
<td>Yes</td>
<td>Uses meta-vars. yes</td>
</tr>
<tr>
<td>Subject Reduction</td>
<td>Yes</td>
<td>Explicit Subst. $\delta$-abstractions with mult. application</td>
</tr>
<tr>
<td>Typechecking</td>
<td>Decidable</td>
<td>Based on $\lambda$-cube</td>
</tr>
<tr>
<td>Type inference</td>
<td>Decidable</td>
<td>de Bruijn Ind. no</td>
</tr>
<tr>
<td>Weak Normalization</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Strong Normalization</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
2.2.6. Summary. We see that all systems that we considered here, unknown objects are thought of as functionally dependent on the meta-level upon the variables that may appear free in them. The implementation mechanisms vary from de Bruijn indices with explicit substitutions to introducing binders for meta-variables. Strong normalization is not always necessary, but weak normalization and decidable typechecking are a commonplace.

2.3. Tactic-Based Interactive Proof Construction

2.3.1. The LCF Approach. In the 1970s Milner and his group developed a system for proof checking called Edinburgh LCF [GMW79]. In it a proof can be developed in a backwards manner starting from a goal, which is the conjecture we want to prove. The backwards proof is created by applying tactics to a goal and as a result we may get new subgoals. The process is iterated until no more subgoals are left.

In order to guarantee correctness, proofs are only created using a fixed number of constructors that are supposed to be sound. This allows a tactic to be represented by an arbitrary computation that uses these constructors. Since all proofs are created through a controlled interface, the soundness of the proof construction will not be affected by errors in the tactic code.

The LCF approach has influenced or has been taken up in a number of systems like HOL [Gor88], Isabelle [Pau93], Coq [CDT03], etc.

2.3.2. The Tactic Language of Coq. In [Del00, CDT03] Delahaye presents a language $\mathcal{L}_{tac}$ of tactics for the proof assistant Coq [CDT03]. This is a dedicated proof meta-language used to build up tactics from a set of basic tactics using combinators in the LCF tradition called tacticals, plus operators for matching on terms and proof contexts. The basic tactics are the built-in tactics of Coq like Apply, Intros, etc. They are taken as atomic objects and their semantics is left unspecified.

The calculus gives to the user the following tacticals

- $tac_1; tac_2$ Applies $tac_1$ and $tac_2$ to all the subgoals
- $tac; [tac_1|\ldots|tac_n]$ Applies $tac$ and $tac_i$ to its $i$th subgoal
- $tac_1\ Or else\ tac_2$ Applies $tac_1$ or $tac_2$ if $tac_1$ fails
- $Do\ n\ tac$ Applies $tac$ $n$ times
- $Repeat\ tac$ Applies $tac$ until it fails
- $Try\ tac$ Applies $tac$ and does not fail if $tac$ fails
- $First\ [tac_1|\ldots|tac_n]$ Applies the first $tac_i$ which does not fail
- $Solve\ [tac_1|\ldots|tac_n]$ Applies the first $tac_i$ which solves
- $Idtac$ Leaves the goal unchanged
- $Fail$ Always fails

plus the possibility to introduce (recursive) definitions. The central feature however is the powerful context-matching operator which allows pattern-matching on
the current context in combination with backtracking. For example the expression

\[
\text{Match Context With} \\
\text{[id}_1 : ?!; id}_2 : ?!\rightarrow ?2| ?2 \rightarrow \text{Apply id}_2; \text{Exact id}_1| \\
\_\_ \rightarrow \text{Idtac}
\]

searches the current context for two assumptions \(id_1\) and \(id_2\) respectively of type \(?!\) and \(?!\rightarrow ?2\) such that \(?2\) is the type of the current goal. If found, the tactic expression \(\text{Apply id}_2; \text{Exact id}_1\) is executed. If not, the current goal is left unchanged with \(\text{Idtac}\).

Proof-context matching combined with recursive tactic definitions allows natural encoding of a number of non-trivial tactics. For example, [CDT03] presents an implementation of a decision procedure for intuitionistic propositional logic based on the contraction-free calculus of Dyckhoff [Dyc92] (see also Section 6.7).

2.3.3. The Proof Language \(\mathcal{L}_{ptl}\). In [Del02] Delahaye presents a proof language combining three different proof styles – procedural, declarative and term-oriented. The idea is to combine elements of the tactic language of Coq with Let-constructions allowing declarative proofs and possibility to directly use proof terms as in ALF. This makes it possible to freely mix the three styles in a manner suitable for the specific proof one is constructing.

In the core of the language is the calculus of Coq terms extended with possibilities to create terms by calls to tactics (\(<\text{by tac}>\)) and with declarative proof constructs (Let). Around it we have layers of sentences and scripts.

The tactics allowed in a procedural term are built up from the basic tactics \(\text{Intro}, \text{Intros}, \text{Cut}, \text{Apply}, \text{Assumption and Pattern}\), using the standard tacitacs of Coq. In contrast to \(\mathcal{L}_{tac}\), there is no possibility to define new tactics and to perform matching on the context.

The language \(\mathcal{L}_{ptl}\) has a formal semantics that allows proof scripts written in it to be evaluated to pure terms. This language is related to the calculus of tactics that we describe in Chapter 6 (see Section 6.8).

2.4. Higher-Order Unification

Higher-order logic [Chu40] was presented in 1940 by Church in relation to the simple type theory. Informally speaking, higher-order unification is the problem of finding a substitution that makes two \(\lambda\)-terms equal. In [Hue75] Huet presents a complete semi-decision procedure for higher-order unification for the simply-typed \(\lambda\)-calculus. Despite its undecidability [Hue73], this algorithm has been successfully used in a range of applications. The main ones include proof search in interactive theorem proving, logical frameworks and higher-order logical programming. For references and background on higher-order unification the reader may consult [Dow01, DHK00].

Since it is related to our further discussion, we want to single out the idea presented by Miller [Mil92] on unification under a mixed prefix. Because during unification we look for instantiations of a number of variables and they may occur under binders, Miller proposes to consider these unknowns as functions on the variables that may occur free in a solution for that unknown. Then the occurrences
of these variables are replaced by function applications. The function symbol representing the unknown is applied to the respective bound variables.

This idea of Miller is in the basis of our approach to open terms. In the chapters that follow we will extend it since we will have to consider two different kinds of functional dependencies – one on the meta-level and one on the object level. For a comparison of the two approaches, see Section 5.4.
CHAPTER 3

Incomplete Terms and Proofs in Higher-Order Logic

In this chapter we formalize the notions of incomplete term and incomplete proof in logic. As a basis we take a natural deduction formulation of higher-order logic (HOL). We discuss the different kinds of incompleteness that may occur in a logical proof or term and extend the formal framework of HOL in order to support them.

3.1. Introduction

Logic is a means to formalize, express and prove statements about a system. A logician looks at the subject of study from the outside, from the so-called meta-level. Using observations and deduction represented respectively as axioms and derivation rules in logic, one can reason about the system by stating and proving certain properties.

A subject of such a study can be a mathematical system like the Peano Arithmetic, a physical system like a chip with digital circuitry or a computer program that needs to be verified. All these systems are very complex and so are the proofs that one needs to create when studying them. This raises the need for tool support for formal reasoning. The main difficulty in some of the problems is the computational resources needed to efficiently solve them, others are provably impossible to automate.

To address this situation, many people have turned to interactive systems where the work of solving the difficult problem is shared between a computer doing the routine tasks and a human that guides it by providing the non-automatable creative thinking. Due to the complexity of the problems, in an interactive setting we are forced to use the step-by-step approach where we start with a goal to prove a statement and by successive refinements we construct hopefully better and better ‘approximations’ of the proof that we are looking for.

But what is an ‘approximation’ of a proof? How do we know that a certain object is an ‘approximation’ of the problem that we wanted to solve? Are there rules that tell us when an ‘approximation’ is well-formed? Are we sure that if we manage to refine an approximation to a full proof, that will be a proof of the right statement?

In this chapter we will discuss these questions for the case of higher-order logic (HOL). Our approximation will be an incomplete proof, a proof in which some parts are omitted. We will give a formal system describing the incomplete HOL-proofs and describe the operations that allow us to refine one incomplete proof to another.
The chapter is organized as follows: In Section 3.2 we introduce the version of HOL to be used as a basis for the discussion. In Section 3.3 by means of a number of examples we introduce some typical situations where incomplete terms and proofs are involved. Then we present the system $\sigma$HOL in which the incomplete HOL-proofs and terms are first-class citizens. In Section 3.4 we extend this system by ‘binding holes’ motivated by examples from forward reasoning. In Section 3.5.1 we discuss the instantiating operations that fill in a value for an unknown object. Section 3.5.2 is devoted to showing that $\sigma$HOL is complete with respect to backward reasoning and that its extension by binding holes is also complete with respect to forward reasoning.

### 3.2. Higher-Order Predicate Logic

In this section we briefly present the formalization of higher-order logic that we will use as a basis for our discussion.

#### 3.2.1. The Language of Higher-Order Logic.

**Definition 3.1 (Domains).** Let $\mathcal{B}$ be a set whose elements we will call basic domains. Let $\text{Prop}$ be a separate domain that we will call the domain of propositions. The set of the domains $\mathcal{D}$ of HOL is defined by:

$$\mathcal{D} ::= \mathcal{B} \mid \text{Prop} \mid \mathcal{D} \to \mathcal{D}$$

The set $\mathcal{B}$ contains the domains (in logic also known as sorts\(^1\)) of our language. New domains can be made using the function-space operator $\to$. A domain $A \to B$ represents the functions from $A$ to $B$. The special domain $\text{Prop}$ contains all formulas (i.e. propositions). As $\text{Prop}$ is one of the domains, we can form domains like $A \to \text{Prop}$ that is the domain of the predicates on the set $A$.

A signature describes the basic domains and the constant, predicate and function symbols in the particular language that we use. As $\mathcal{D}$ contains function and predicate domains, giving a signature is equivalent to giving a set of constants with their domains. The fact that the constant $c$ is of domain $\sigma$ will be denoted by $c^{\sigma}$. In traditional presentations of first-order predicate logic every predicate and function symbol has arity describing what number of arguments it expects and their domains. First order constants can be viewed as functions of zero arity.

In the presence of higher-order domains we have a similar situation – the function and predicate symbols can be seen as constants of functional domains. A ‘traditional’ constant has a basic domain, a (first-order) function symbol is a constant of domain $\sigma_1 \to \ldots \to \sigma_n \to \tau$ where $\sigma_i$ and $\tau$ are basic domains. A (first-order) predicate symbol is a constant of domain $\sigma_1 \to \ldots \to \sigma_n \to \text{Prop}$ with $\sigma_i$ basic domains. In general every domain can be written as $\sigma_1 \to \ldots \to \sigma_n \to \tau$ where $\tau$ is either $\text{Prop}$ or a basic domain.

Given a fixed signature $\mathcal{L} = \langle \mathcal{B}, c^{\sigma_1} \rangle_i$ (not necessarily finite, but decidable) we can define the set of the $\mathcal{L}$-terms. Every term represents an object of certain domain that is also called the domain of the term. We assume that for every domain $\sigma$ there is an infinite set of variables of that domain.

---

\(^1\)We will use the term sort only in its meaning in a typing system [See Chapter 4]. Logical 'sorts' will be referred to as 'domains'.
3.2. Higher-Order Predicate Logic

Definition 3.2 (Higher-Order Terms). *The set of terms of the signature $\mathcal{L} = \langle c_i^{\sigma_i} \rangle_i$ and their domains are defined inductively as follows:*

- Every constant $c_i$ is a term of domain $\sigma_i$.
- Every variable $x^\sigma$ of domain $\sigma$ is a term of domain $\sigma$.
- If $f$ is a term of domain $\sigma \to \tau$ and $t$ is of domain $\sigma$ then $ft$ is a term of domain $\tau$.
- If $t$ is a term of domain $\tau$ and $x^\sigma$ is a variable of domain $\sigma$ then the term $\lambda x^\sigma.t$ is of domain $\sigma \to \tau$.
- If $A$, $B$ are terms of domain Prop and $x^\sigma$ is a variable of domain $\sigma$, then the following terms are of domain Prop:

  $$A \to B, \ A \land B, \ A \lor B, \ \neg A, \ \forall x^\sigma.A, \ \exists x^\sigma.A$$

Remark 3.3. *Please note the ‘overloading’ of the symbol $\to$. Between domains it designates the function space operator and between propositions it represents the logical implication.*

We will refer to the terms of domain Prop as *formulas* or *propositions*. As Prop is a domain, we can have variables over formulas and we can even quantify over formulas and predicates. This allows us to formulate higher-order statements like for example the induction principle on the natural numbers.

$$\text{Ind} \equiv \forall P^{\text{nat} \to \text{Prop}}. (P \ 0) \land \forall k^{\text{nat}}. ((P \ k) \to (P \ (Sk))) \to \forall n^{\text{nat}}. (P \ n)$$

where $0$ is a constant of the basic domain nat and $S$ is the successor function of domain nat $\to$ nat.

Using the $\lambda$-abstraction we can build functions and predicates. For example, the predicate "$n$ is even" can be defined by the term

$$\lambda n^{\text{nat}}. \exists x^{\text{nat}}. (n = x + x)$$

that is of domain nat $\to$ Prop.

The term constructors $\lambda, \exists$ and $\forall$ abstract a variable over a term. The variable being abstracted then becomes *bound* in the term. Variables that are not bound are called *free*.

Definition 3.4 (Free Variables). *The set of the variables with free occurrences of a term $M$ (notation $\text{FV}(M)$) is defined inductively on the structure of the term $M$:*

- $\text{FV}(c) = \emptyset$
- $\text{FV}(x) = \{x\}$
- $\text{FV}(\lambda x^\sigma.M) = \text{FV}(M) \setminus \{x\}$
- $\text{FV}(\forall x^\sigma.A) = \text{FV}(A) \setminus \{x\}$
- $\text{FV}(\exists x^\sigma.A) = \text{FV}(A) \setminus \{x\}$

We will identify terms up to renaming of bound variables. Hence we adopt Barendregt’s ‘Variable Convention’ [Bar84] stating that the bound variables are always suitably renamed so that they are different from the free variables and from each other. This spares us the problems with terms like $(\lambda x)x$ that after renaming the bound variable could be read ambiguously as either $(\lambda y)y$ or
(λy.x)x. The convention is also very useful in technical definitions like the one of substitution:

**Definition 3.5 (Substitution).** The result (denoted by M[t/x]) of the substitution of t for the free occurrences of x in a term M is defined as follows:

\[
c[t/x] = c
\]

\[
z[t/x] = z \quad (x \neq z)
\]

\[
x[t/x] = t
\]

\[
(\βy.M)[t/x] = \βy.(M[t/x])
\]

\[
(\∀y.M)[t/x] = \∀y.(M[t/x])
\]

\[
(\∃y.M)[t/x] = \∃y.(M[t/x])
\]

\[
(A \land B)[t/x] = A[t/x] \land B[t/x]
\]

\[
(A \lor B)[t/x] = A[t/x] \lor B[t/x]
\]

\[
(f \circ g)[t/x] = (f[t/x]) \circ (g[t/x])
\]

\[
(A \rightarrow B)[t/x] = A[t/x] \rightarrow B[t/x]
\]

\[
(\neg A)[t/x] = \neg(A[t/x])
\]

Because of the variable convention we need not explicitly state that the bound variables y above are different from x.

Using the substitution operation we define β-reduction (notation →β) between terms as the least binary relation on terms that is reflexive, transitive and compatible with the term constructors and that contains the relation

\[
(\lambda x. M)t \rightarrow_β M[t/x]
\]

The β-reduction can be seen as an evaluation procedure computing the result of the application of the function \(\lambda x. M\) to the argument t.

The least compatible equivalence relation containing \(\rightarrow_β\) is called β-conversion and is denoted by \(\equiv_β\). Using basic λ-calculus theory it can be shown that \(\rightarrow_β\) and more importantly, \(\equiv_β\) are decidable on well-formed HOL-terms. This is important because we use \(\equiv_β\) in the definition of the logical rules (see the conversion rule in Definition 3.6 below). We refer the reader to [Bar84, Bar92] for background reading on the λ-calculus.

### 3.2.2. Proofs in Higher-Order Logic

Above we defined the language of HOL with its terms and (among them) formulas. In this section we will give the rules of inference that allow us to prove statements in the language of HOL. We will adopt Gentzen’s natural deduction style of presentation of proofs.

A proof in natural deduction is a finitely branching tree whose nodes are labelled by formulas, built up according to a set of rules. We will often need to refer to formulas at the leaves and the root of a proof tree. A tree Σ with a root labelled by B and formulas A among the labels of the leaves of Σ will be depicted by

\[
\begin{array}{c}
\Sigma \\
\end{array}
\]

\[
\begin{array}{c}
A \\
\end{array}
\]

\[
\begin{array}{c}
B \\
\end{array}
\]

Some leaves that are discharged will be additionally labelled. A discharged leaf A marked by i will be depicted by [A]i. The set of the well-formed proof-trees (also called logical derivations) is given by the following definition.

**Definition 3.6 (Logical Proof, Derivation Rules).** The set of the logical derivations is defined inductively by the following rules:
(axiom): A tree consisting of a single root node labelled by \( A \) is a proof of \( A \) from assumption \( \{ A \} \).

(elimination of \( \rightarrow \)): If \( \Sigma_1 \) is a derivation of \( A \rightarrow B \) from assumptions \( \bar{C}_1 \) and \( \Sigma_2 \) is a derivation of \( A \) from assumptions \( \bar{C}_2 \) then the following is a derivation of \( B \) from assumptions \( \bar{C}_1, \bar{C}_2 \):

\[
\begin{array}{c}
\Sigma_1 \\
\downarrow \\
A \rightarrow B
\end{array}
\quad
\begin{array}{c}
\Sigma_2 \\
\downarrow \\
A
\end{array}
\quad
\begin{array}{c}
\downarrow \\
B
\end{array}
\quad
(\rightarrow\text{-elim})
\]

(introduction of \( \rightarrow \)): Let \( \Sigma \) be a derivation of \( B \) from assumptions among which \( A \) possibly occurs. Then

\[
\begin{array}{c}
\{A\}^k \\
\downarrow \\
B
\end{array}
\quad
\begin{array}{c}
\downarrow \\
A \rightarrow B
\end{array}
\quad
(\rightarrow\text{-intro})
\]

where some (not necessarily all or any) assumptions \( A \) of \( \Sigma \) are discharged, is a derivation of \( A \rightarrow B \) from assumptions the same as those of \( \Sigma \) but without the discharged ones. (Note how the label \( k \) designates which node in the proof tree has discharged a particular assumption).

(elimination of \( \forall \)): Let \( \Sigma \) be a derivation of \( \forall x^\sigma A \) from assumptions \( \bar{C} \). If \( t \) is a term of domain \( \sigma \) then the following is a derivation of \( A[t/x] \) from assumptions \( \bar{C} \):

\[
\begin{array}{c}
\Sigma \\
\downarrow \\
\forall x^\sigma A
\end{array}
\quad
\begin{array}{c}
\forall x^\sigma A \\
\downarrow \\
A[t/x]
\end{array}
\quad
(\forall\text{-elim})
\]

(introduction of \( \forall \)): Let \( \Sigma \) be a derivation of \( A \) from assumptions \( \bar{C} \). Provided that \( x^\sigma \) does not occur free in any of the formulas \( \bar{C} \), the following is a derivation of \( \forall x^\sigma A \) from assumptions \( \bar{C} \):

\[
\begin{array}{c}
\Sigma \\
\downarrow \\
A
\end{array}
\quad
\begin{array}{c}
\forall x^\sigma A \\
\downarrow \\
\forall x^\sigma A
\end{array}
\quad
(\forall\text{-intro})
\]

\( x \notin \text{FV}(\bar{C}) \)
**Conversion:** Let $\Sigma$ be a derivation of $A$ from assumptions $\vec{C}$. If $B$ is a formula (i.e. a term of domain $\text{Prop}$) such that $A \equiv B$ then

$$
\begin{array}{c}
\vec{C} \\
\Sigma \\
A & \frac{(conv)}{B} & A \equiv B
\end{array}
$$

is a derivation of $B$ from assumptions $\vec{C}$.

The above definition gives the rules of the *minimal intuitionistic higher-order logic*. It is a well-known fact that the rest of the connectives are second-order definable as shown in Figure 3.1. One can show that these definitions comply with the standard introduction and elimination rules for the connectives shown in Figure 3.2. For this reason we will focus our attention to the minimal version of HOL, but use the other connectives freely when suitable.

A notion that will play an important role in our further discussion is derivability introduced below:

**Definition 3.7 (Derivability in HOL).** A formula $A$ is derivable from a set of assumptions $\Gamma$, notation $\Gamma \vdash A$, if there is a derivation tree with conclusion $A$ and assumptions among $\Gamma$.

Derivability in HOL is preserved under substitutions:

**Proposition 3.8.** Let $x$ and $t$ be from the same domain. If $\Gamma \vdash A$ with derivation $\Sigma$, then $\Gamma[t/x] \vdash A[t/x]$ with derivation $\Sigma[t/x]$.

### 3.3. Open Proofs in Higher-Order Logic

We introduce the notions of incomplete proof and incomplete term. We do that first informally by means of several examples in Section 3.3.1. The formal definitions follow in Section 3.3.2.
3.3. Examples of Incomplete Terms and Proofs. Using an interactive theorem prover one can incrementally build up proofs and terms. The computer does the housekeeping tasks by tracking the open proof obligations and making sure that only legal operations are applied. The human introduces the creative ideas needed to find the proof. In this process we necessarily need to work with unfinished proofs and terms. These are proofs and terms parts of which are left open and are to be filled before the proof-construction is finished. Below we will look at several typical scenarios that illustrate different aspects of the manipulation and the use of incomplete terms.
3.3.1.1. Unfinished proofs in backward proof construction. This method of
proof construction is known as backward reasoning, goal-oriented and also goal-
driven proof construction because one constructs the proof in a bottom-up manner
starting from the root of the proof tree as the original goal to prove. Applying the
logical rules in an backward manner one reduces a goal expressed by the conclusion
of the rule to the goals given by the premisses. Decomposing the goal in this way
one hopes that the new goals would be ‘easier’ to solve than the original goal.
When a goal becomes trivial (i.e. the goal is among the assumptions) we can close
the branch by means of the axiom rule.

In the example of Figure 3.3 we start at (1) with the goal of proving \( A \rightarrow C \)
from hypotheses \( A \rightarrow B \rightarrow C \) and \( A \rightarrow B \). The question mark symbol stands for
the missing part of the proof that should be a derivation of \( A \rightarrow C \) from the two
assumptions above it.

In (2) we partially solve the goal by reducing the goal to the problem of finding
a derivation of \( C \) under a list of hypotheses extended by \( A \). Note how the old goal
is solved using the new goal and the introduction rule for implication. We also see
that the assumption \( A \) that is added to the hypotheses list has been discharged by
the introduction rule. The label \( i \) is an arbitrary but distinctive symbol.

In (3) we have already reduced the goal \( C \) to the goals \( A \) and \( B \). As \( A \) is in
the list of assumptions, we may close this branch of the tree. To deduce \( B \) we need
one more backward step that results in (4). Solving the two goals in (4) is easy
as both of them appear in the assumptions list. The final stage (5) represents the
complete proof.

3.3.1.2. Unfinished proofs in a forward proof construction. When constructing
a proof forwards one uses the known facts in order to deduce new ones from them.
This process is iterated until the goal that we need to prove has been deduced.
This kind of reasoning is often called forward reasoning.

In the example depicted on Figure 3.4 we start with the initial goal of proving
\( C \) from \( A \), \( A \rightarrow B \) and \( B \rightarrow B \rightarrow C \). We proceed forwards by using elimination
rules on the assumptions. In (2) we have used \( A \) and \( A \rightarrow B \) to obtain \( B \) which
is used in (3) to deduce \( B \rightarrow C \). Then we must infer \( B \) again and use it to derive
\( C \) at step (4). Note that in step (4) we would like to be able to reuse the already
proven result \( B \) instead of having to derive it again, but natural deduction does
not allow this.

3.3.1.3. An unfinished proof with open terms. In this example we have a transitive relation \( R(x, y) \) and we want to prove \( R(a, c) \). By a simple matching we see
that to obtain \( R(a, c) \) from \( \forall x y z. R(x, y) \rightarrow R(y, z) \rightarrow R(x, z) \), we need to eliminate
\( x \) by \( a \) and \( z \) by \( c \). Then the new goals would be to prove \( R(a, y) \) and \( R(y, c) \). In
order to do that however, we need to eliminate the universal quantifier for \( y \). The
question is what to take for \( y \)? We don’t know (yet), so we would want to leave \( y \)
open. To do that we need to be able to denote unknown terms for which we need
to find an instantiation.

The ‘open place’ \( y \) in the example has a different role than a variable: we seek
an value for it and we will not abstract over it. We will call these open places
3.3. OPEN PROOFS IN HIGHER-ORDER LOGIC

| (1) | A → B → C  A → B | (2) | A → B → C  A → B [A]^i |
|     |                 |     |                 |
|     |                ?|     |                ?|
|     | A → C          |     | A → C          |

|     | A → B → C       |     | A → B → C       |
|     |                 ?|     |                 ?|
|     | A → B           |     | A → B           |
|     | B → C           |     | C               |
|     |                 ?|     |                 ?|
|     | B → C           |     | B → C           |
|     |                 ?|     |                 ?|
|     | A → C          |     | A → C          |

(5)

| A → B → C  [A]^i  A → B  [A]^i |
|------------|------------|
| B → C      | B         |
| C          | A → C     |

**Figure 3.3.** Example of incomplete proofs in backward proof construction.

| (1) | B → B → C  A → B  A |
|     |                 |
|     |                ?|
|     | C               |

| (2) | B → B → C  A → B  A |
|     |                 |
|     |                ?|
|     | C               |

| (3) | B → B → C  A → B  A |
|     |                 |
|     |                ?|
|     | C               |

| (4) | B → B → C  A → B  A |
|     |                 |
|     |                ?|
|     | C               |

**Figure 3.4.** Example of incomplete proofs in forward proof construction.

**meta-variables.** A term containing meta-variables will be called an open term or incomplete term.

To clearly distinguish variables from meta-variables, in this chapter we will underline the occurrences of meta-variables, so $\underline{y}$ denotes a meta-variable and $y$ is different from $\underline{y}$.

3.3.1.4. Delaying the choice for a witness. Computing with open terms. In the previous example we saw how a meta-variable can be used to delay the choice for
a term that is used to eliminate a universal quantifier. A similar situation occurs with the existential quantifier: In order to prove an existential formula \( \exists x. A(x) \) constructively one usually needs to find a term \( t \) (called also a witness) and prove \( A(t) \). Often the choice of the term is not obvious and one may want to leave it open while continuing with the proof. At a later stage of the proof more information about the witness may become available that provides an easier or even automatic way (e.g., by unification) to find the witnessing term. The example below again uses a meta-variable to represent the unknown witness. It also illustrates some of the problems one may get into if one is not careful with the handling of the local context of an unknown and with the definition of the operation that fills in values for meta-variables.

In Figure 3.6 in four steps we try to construct the proof that an identity function exists. In the second proof we introduce a meta-variable \( \lambda y.x \) representing

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (fx) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]

\[
\exists y. \lambda x. (fx) = x
\]

\[
\forall x. (\lambda y.x) = x
\]
in the last proof becomes
\[
\begin{align*}
? & \quad \forall x. y = x \\
? & \quad \forall x. (\lambda z. y) x = x \\
\forall \forall \forall \forall & \quad \exists f \forall x. f x = x
\end{align*}
\]

Note the renaming of the bound variable \( y \) to \( z \) as expected by the variable convention. As substitution always avoids capture of free variables, we need to give a different operation that would allow us to do that. The second problem that becomes evident in Figure 3.6 is the management of the context of the meta-variables. Note that in \( \forall x. (\lambda y. n) x \) the meta-variable is in the context of the variables \( x \) and \( y \), while only \( y \) may occur free in a solution for \( n \). Furthermore, in the \( \beta \)-equal term \( \forall x. n = x \) it is in the context of \( x \) only, but it does not depend on it. Where does the variable \( y \) that we want to use to instantiate \( n \) then come from?

The solution to this problem is to consider the unknown term as a meta-level function depending on the variables that may occur in it. Any bound variables that the unknown may depend on can be given as arguments of this meta-function. Then instead of representing the function with unknown body by \( \lambda y. n \), we will explicitly denote the dependency of \( n \) on \( y \) by writing \( \lambda y. n[y] \). Applying this method to the proofs in Figure 3.6, we get the proofs in Figure 3.7. If we now

<table>
<thead>
<tr>
<th>( \exists f \forall x. f x = x )</th>
<th>( \forall x. (f x) = x )</th>
<th>( \exists f \forall x. (f x) = x )</th>
<th>( \forall x. (\lambda y. n(y)) x = x )</th>
<th>( \exists f \forall x. (f x) = x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists f \forall x. (f x) = x )</td>
<td>( \forall x. (\lambda y. n(y)) x = x )</td>
<td>( \exists f \forall x. (f x) = x )</td>
<td>( \forall x. (\lambda y. n(y)) x = x )</td>
<td>( \exists f \forall x. (f x) = x )</td>
</tr>
</tbody>
</table>

Figure 3.7. Meta-variables with parameters allow proper management of the context of the unknown.

want to instantiate the meta-variable \( n[y] \) by \( y \), we know that the variable \( y \) refers to the parameter of the meta-variable. This allows us to give a proper instantiating operation. For example, instantiating \( n[y] \) by \( y \) in \( \forall x. (\lambda y. n(y)) x = x \) gives us \( \forall x. (\lambda y. y) x = x \) and in \( \forall x. n x = x \) gives \( \forall x. x = x \). A formal definition of the instantiation can be found in Section 3.3.2. Here we will only note that instantiation can be used to achieve variable capture and that the only way to have a variable in the instantiating term captured is it to be declared as a parameter of the meta-variable.

This formalization of unknown terms allows us to show the soundness of their use – the instantiation of unknowns commutes with the derivation rules (see Proposition 3.18 in Section 3.3.2).

Using parameters to denote the dependencies of an unknown term on some variables is in fact a variant of the Miller’s approach [Mil92] to represent unknowns by functions. However here we use meta-level functional dependencies represented by the parameters. Other possible approaches are possible using for example the explicit substitutions in calculi of Muñoz [Mn97], ALF [Mag95] or Typelab [Str99]. The need to move the functional approach to the meta-level is discussed in Section 5.4.
3.3.1.5. Using meta-variables to represent unknown formulas and predicates. As we work in higher-order logic, we have the possibility to quantify over higher-order domains including predicates and formulas. Inventing witnesses for quantifiers over such domains is not always easy and often we would like to delay such choices by using meta-variables as described in the previous sections.

Let us consider the setting of (Peano) arithmetic. The ‘usual’ induction principle is expressed by the formula

\[ \text{Ind}_1 : = \forall n. P^{\Lnot-\text{Prop}}(0) \land (\forall n. P(n) \rightarrow P(n + 1)) \rightarrow \forall n. P(n) \]

The ‘course-of-value’ induction principle is expressed by the formula

\[ \text{Ind}_2 : = \forall n. P^{\Lnot-\text{Prop}}. (\forall n. (\forall k. k < n \rightarrow P(k)) \rightarrow P(n)) \rightarrow \forall n. P(n) \]

Suppose we want to prove the well-known fact \text{Ind}_1 implies \text{Ind}_2 (the opposite also holds). We will show how we can construct a proof of this implication without having to make all difficult choices in the very beginning. Instead, we will introduce a meta-variable representing the unknown and refine it ‘on-the-fly’ as we collect more information about its properties during the development of the proof.

After two obvious backward steps we have the open proof shown below \((P_<(n))\) abbreviates \(\forall k \cdot k < n \rightarrow P(k)\).

\[
\begin{array}{c}
\text{Ind}_1 \quad [\forall n. P_< (n) \rightarrow P(n)]^i \\
\hline
\forall n.P(n) \\
\hline
(\forall n. P_< (n) \rightarrow P(n)) \rightarrow \forall n. P(n) \\
\end{array} \quad \text{Ind}_2
\]

It is clear that we need to use the hypothesis \text{Ind}_1. To do that we have to eliminate the universal quantifier \(\forall n. P^{\Lnot-\text{Prop}}\) in \text{Ind}_1. Since we do not want to make guesses, we delay the choice and in one forward step we introduce a meta-variable \(B\) for the unknown predicate.

\[
\begin{array}{c}
\text{Ind}_1 \\
B[0] \land (\forall n. B[n] \rightarrow B[n + 1]) \rightarrow \forall n. B[n] \\
\hline
[\forall n. P_< (n) \rightarrow P(n)]^i \\
\hline
\forall n.P(n) \\
\hline
(\forall n. P_< (n) \rightarrow P(n)) \rightarrow \forall n. P(n) \\
\end{array} \quad \text{Ind}_2
\]

Our strategy now is to show that the conjunction \(B[0] \land (\forall n. B[n] \rightarrow B[n + 1])\) follows from \(\forall n. P_< (n) \rightarrow P(n)\). This will give us \(\forall n. B[n]\). To finish the proof we need to show that that implies \(\forall n. P(n)\). This gives rise to the three new goals given below:

\[
\begin{align*}
(1) \quad & \forall n. P_< (n) \rightarrow P(n) \\
(2) \quad & \forall n. B[n] \rightarrow B[n + 1] \\
(3) \quad & \forall n. B[n] \rightarrow P(n)
\end{align*}
\]

To discard goal (3), it is sufficient to have \(B[n]\) in such a way that it implies \(P(n)\). Therefore we define \(B[n] := P(n) \land C[n]\) where \(C[n]\) is a fresh meta-variable of
type Prop. After the instantiation of $B$, goals (1) and (2) look like this:

\[
\forall n. P_<(n) \rightarrow P(n) \\
\frac{?}{P(0) \land C[0]} \quad (1) \\
\forall n. P_<(n) \rightarrow P(n) \\
\frac{?}{\forall n. (P(n) \land C[n]) \rightarrow (P(n+1) \land C[n+1])} \quad (2)
\]

Goal (2) is the hardest to solve. We notice that the conclusion is of the form \( \forall n. A \rightarrow (C \land B) \). We can solve it by for example showing \( \forall n. A \rightarrow B \) and \( \forall n. B \rightarrow C \). This gives us the following two goals: (2a) \( \forall n. P(n) \land C[n] \rightarrow C[n+1] \) and (2b) \( \forall m. C[m] \rightarrow P(m) \). Developing further goal (2b) we get to the following situation:

\[
\forall n. P_<(n) \rightarrow P(n) \\
\frac{P_<(m) \rightarrow P(m)}{P(m)} \quad \frac{?}{C[m] \rightarrow P(m)} \quad \frac{?}{\forall m. C[m] \rightarrow P(m)} \\
\frac{\forall n. P_<(n) \rightarrow P(n)}{P(0) \land P_<(0)} \\
\frac{\forall n. P_<(n) \rightarrow P_<(n+1)}{P(n) \land P_<(n)}
\]

and it is now not difficult to see that this goal can be solved by taking \( C[m] \) to be the formula \( P_<(n) \). The two remaining goals (1) and (2a) now look like this

\[
\forall n. P_<(n) \rightarrow P(n) \\
\frac{?}{P(0) \land P_<(0)} \quad (1) \\
\forall n. P_<(n) \rightarrow P(n) \\
\frac{?}{\forall n. P(n) \land P_<(n) \rightarrow P_<(n+1)} \quad (2a)
\]

and are easily provable after unfolding the definition of \( P_<(n) \). Hence the final solution for the predicate \( B[n] \) is \( P(n) \land P_<(n) \) or equivalently, \( \forall k \leq n. P(k) \).

In this example we see how the use of unknowns gives us more flexibility in the proof-construction by providing a mechanism to partially solve the unknown, explore the consequences of this partial choice, make next approximation, etc. Of course this does not provide an algorithm for finding the proof we are looking for—we do not claim that the proof construction above represents a general procedure for solving proof-search problems—but it gives us one more tool that helps us in finding it.

3.3.1.6. Unknown Domains. If we want to be complete, we could think of an example where we use unknowns over domains. This however is not very exciting because the domains are rather static structures (we cannot compute with them) and there is no interference between the unknowns and outside factors. Although theoretically acceptable, such unknowns have little practical applicability and we will not consider them further.

3.3.2. Open Higher-Order Predicate Logic (oHOL).

We now give a formal definition of higher order predicate logic with open terms and open proofs, oHOL. We first define the language, then the derivation rules and then the notion of derivability. We show that oHOL is conservative over HOL, ordinary higher order predicate logic [Bar92, Geo93].
3.3.2.1. The Language of oHOL. As in HOL the domains of oHOL are given by

\[ D ::= \text{Prop} \mid B \mid D \rightarrow D \]

Variables like \( \sigma, \tau \) will range over domains. The main addition to the language in oHOL is the introduction of the so-called meta-variables. These are special variables that stand for the unknown terms. A meta-variable has a domain and zero or more parameters. We will denote meta-variables by \( m^\tau[x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}] \), underlining them to emphasize the difference from the normal variables. The need for parametrization of the meta-variables was demonstrated in the example of Section 3.3.1.4 where we needed a mechanism to record the substitutions for bound variables passing through a meta-variable in order to ensure commuting instantiation and computation.

**Definition 3.9 (Terms of oHOL).** The set of terms of the signature \( \mathcal{L} = \langle c_i^\sigma \rangle_i \) and their domains are defined inductively as follows:

- Every constant \( c_i \) is a term of domain \( \sigma_i \).
- Every variable \( x^\sigma \) of domain \( \sigma \) is a term of domain \( \sigma \).
- If \( f \) is a term of domain \( \sigma \rightarrow \tau \) and \( t \) is of domain \( \sigma \) then \( ft \) is a term of domain \( \tau \).
- If \( t \) is a term of domain \( \tau \) and \( x^\sigma \) is a variable of domain \( \sigma \) then the term \( \lambda x^\sigma.t \) is of domain \( \sigma \rightarrow \tau \).
- If \( A, B \) are terms of domain \( \text{Prop} \) and \( x^\sigma \) is a variable of domain \( \sigma \), then the following terms are of domain \( \text{Prop} \):

  \[ A \rightarrow B, \ A \land B, \ A \lor B, \ \neg A, \ \forall x^\sigma.A, \ \exists x^\sigma.A \]

- If \( m^\tau[x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}] \) is a meta-variable and if for every \( i \), \( t_i \) is a term of domain \( \sigma_i \), then \( m[t_1, \ldots, t_n] \) is a term of domain \( \tau \).

The fact that the term \( t \) is of domain \( \sigma \) will be denoted by \( t^\sigma \). If the domains that we quantify over are irrelevant, we will write \( \forall x.A \) instead of \( \forall x^\sigma.A \). Similarly we will often write \( m[x_1, \ldots, x_n] \) for \( m^\tau[x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}] \). The abbreviated forms \( m[\overline{x}] \) and \( m[\overline{x}^\sigma] \) will also be used.

As in HOL, we call ‘formula’ any term from the domain \( \text{Prop} \). Note that the definition above allows also meta-variables standing for formulas or functions producing formulas.

Meta-variables by themselves are not terms. They have to be given appropriate actual arguments corresponding to the formal parameters in their declaration to become terms. Therefore one should distinguish between a meta-variable declaration \( m^\tau[x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}] \) (not being a term) and a meta-variable instance \( m[t_1, \ldots, t_n] \) (being a term). We assume countably many meta-variables for every \( \sigma_1, \ldots, \sigma_n, \tau \). We view the ‘assignment’ \([x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}]\) and the domain \( \tau \) as being part of the meta-variable. Thus for example \( m^\tau[y^\sigma] \) and \( m^\sigma[y^\tau] \) are different meta-variables (but of course we will use different names as much as possible). Furthermore, \( \alpha \)-convertible assignments are considered identical: e.g. \( m^\tau[x^\sigma] \) and \( m^\tau[y^\tau] \) denote the same meta-variable.

As terms with meta-variables are ordinary terms, meta-variables can occur in the arguments of another (or the same) meta-variable. For example, if \( m^\tau[y^\sigma, z^\tau] \)
is a meta-variable and $f^α$ and $a^α$ then e.g. $m[(f a), m[a, (f a)]]$ is a well-formed term.

The definitions of the set of the variables with free occurrences in a term and
of substitution on the terms of oHOL are immediate extensions of the respective
HOL-counterparts with the following clauses for meta-variables:

\[
\begin{align*}
\text{FV}(m[t_1, \ldots, t_n]) &= \bigcup_{i=1}^{n} \text{FV}(t_i) \\
(m[t_1, \ldots, t_n])[t/x] &= m[t_1[t/x], \ldots, t_n[t/x]]
\end{align*}
\]

3.3.2.2. Logical Rules of oHOL. After extending the language of HOL with
incomplete terms, we present the logical rules of oHOL.

**Definition 3.10 (Derivation Rules of oHOL).** The logical rules of oHOL are
obtained from the rules of HOL (see Definition 3.6) plus the extra (claim) rule
representing missing parts of proofs:

\[
\begin{array}{c}
\Sigma_1 \\ \vdots \\ \Sigma_n \\
\hline \\
B_1 \\ \vdots \\ B_n \\
\hline
A
\end{array}
\]

(claim)

The rule (claim) represents an unknown part of the derivation of $A$ from
known derivations of $B_1, \ldots, B_n$. In this rule we explicitly write the hypotheses
of the unknown part because, for example, we need to check the side conditions
for $\forall$-introduction rules used later in the proof. These side conditions refer to
the leaves of the proof-tree and hence we need to keep the whole derivations for
the hypotheses. The explicit representation of the hypotheses also allows us to
represent any forward steps that one may want to do. Sometimes in derivations
we will use the symbol ‘??’ to denote the (claim) rule as shown below

\[
\begin{array}{c}
\Sigma_1 \\ \vdots \\ \Sigma_n \\
\hline \\
B_1 \\ \vdots \\ B_n \\
\hline
??
\end{array}
\]

A proof in HOL is very much like a proof in HOL, except for the fact that
we can now also have (claim) nodes in the tree and meta-variables occurring in
the formulas. We would like to introduce a notion of derivability in oHOL, but
we cannot use directly Definition 3.7 because in the presence of the (claim) rule
it will be the universal relation. Therefore we have to take the ‘open parts’ of
the derivation tree (the (claim) nodes) into account. We will call the instances of
the (claim) rule goals. Hence, a goal in our setting is a formula to be proven in a
given context (see Definition 3.12). Before giving precise definitions, we need to
introduce some technical notions.

A variable may occur free in a specific formula in a derivation $Σ$. Later in
$Σ$ this variable may get bound by for example a $\forall$-introduction rule. We will
consider such variables as free in the formula but bound in $\Sigma$. We define this notion explicitly, as it is important for our interpretation of goals.

**Definition 3.11** (Bound occurrences of a variable in a derivation). Let $\Sigma$ be a derivation and $A$ a formula occurring in $\Sigma$ with $x \in \operatorname{FV}(A)$. We say that $x \in \operatorname{FV}(A)$ is bound in $\Sigma$ in one of the following two cases

\[
\begin{array}{c}
\hline
G \\
\hline
\forall x^\sigma.B \\
\hline
\end{array}
\quad \forall \text{-} I
\]

\[
\begin{array}{c}
\hline
G \\
\hline
\exists x^\sigma.C \\
\hline
\end{array}
\quad \exists \text{-} E
\]

So, the notion of ‘the free variable $x$ of $A$ is bound in $\Sigma$’ is about a specific occurrence of $A$ in the derivation $\Sigma$. It is defined by induction on $\Sigma$. Note that a free occurrence $x$ may be bound for one occurrence of $A$ and free for another.

**Definition 3.12** (Goals in a derivation).

(1) A goal in $\text{oHOL}$ is a judgement of the form

\[
\begin{array}{c}
\hline
x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}, A_1, \ldots, A_n \vdash B,
\hline
\end{array}
\]

where $A_1, \ldots, A_n, B$ are formulas. The goal binds the occurrences of $x_1, \ldots, x_n$ in its formulas.

(2) A goal $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}, A_1, \ldots, A_n \vdash B$ is a goal of the derivation $\Sigma$ if $\Sigma$ contains an instance of the claim rule of the following form

\[
\begin{array}{c}
\hline
\Sigma_1 \ldots \Sigma_n \\
\hline
A_1 \ldots A_n
\end{array}
\]

(claim)

with $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ containing all variables free in $\Sigma_1, \ldots, \Sigma_n, B$ but bound in $\Sigma$.

We can now define the notion of derivability in $\text{oHOL}$:

**Definition 3.13** (Derivability in $\text{oHOL}$). Given a set of formulas $\Gamma$, a set of goals $G$ and a formula $B$, we say that $B$ is derivable from $\Gamma; G$ in $\text{oHOL}$, notation

\[
\Gamma; G \vdash B,
\]

if there is a derivation $\Sigma$ with conclusion $B$, (non-discharged) assumptions in $\Gamma$ and goals in $G$.

As in HOL (see Proposition 3.8), we have that derivability and substitution are compatible. However we have to take note that in a goal $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}, A_1, \ldots, A_n \vdash B$, the occurrences of $x_1, \ldots, x_n$ in $A_1, \ldots, A_n, B$ are bound. Hence, we do not substitute for these variables but rename them appropriately.

**Lemma 3.14** (Compatibility of derivability and substitution in $\text{oHOL}$). Let $x$ and $t$ be of the same domain. If $\Gamma; G \vdash A$, then $\Gamma[t/x]; G[t/x] \vdash A[t/x]$. 

3.3. OPEN PROOFS IN HIGHER-ORDER LOGIC

PROOF. By induction on the derivation tree $\Sigma$, one proves that, if $\Sigma$ has conclusion $A$, assumptions $\Gamma$ and goals $G$, then $\Sigma[t/x]$ is a well-formed derivation with conclusion $A[t/x]$, assumptions $\Gamma[t/x]$ and goals $G[t/x]$. \hfill \Box

EXAMPLE 3.15. Consider the following two derivations

\[
\begin{array}{c}
A \rightarrow \quad B(x) \\
\hline
C \rightarrow \quad \forall x^\sigma. D(x)
\end{array}
\quad
\begin{array}{c}
A \rightarrow \quad B(x) \\
\hline
C \rightarrow \quad B(x) \rightarrow D(x)
\end{array}
\quad
\begin{array}{c}
A \rightarrow \quad B(x) \\
\hline
C \rightarrow \quad B(x) \rightarrow D(x)
\end{array}
\quad
\begin{array}{c}
A \rightarrow \quad B(x) \\
\hline
C \rightarrow \quad B(x) \rightarrow D(x)
\end{array}
\]

In the first $x$ occurs bound and in the second, $x$ occurs free. The judgements associated with these two derivations are

$A, C; (y^\sigma, A \vdash B(y)), (z^\sigma, C \vdash B(z) \rightarrow D(z)) \vdash_i \forall x^\sigma. D(x)$

for the first and

$A, C; (A \vdash B(x)), (C \vdash B(x) \rightarrow D(x)) \vdash_i D(x)$

for the second. Note that if we substitute a term $t$ for $x$ in the first derivation it does not change, while the second one becomes

\[
\begin{array}{c}
A \rightarrow \quad B(t) \\
\hline
C \rightarrow \quad B(t) \rightarrow D(t)
\end{array}
\]

with the corresponding derivability judgment

$A, C; (A \vdash B(t)), (C \vdash B(t) \rightarrow D(t)) \vdash_i D(t)$

\hfill \Box

An important operation on derivations is instantiation (choosing a value for a meta-variable). Therefore, an equally important property for oHOL is the compatibility of the derivation rules with instantiation of meta-variables. We first give a precise definition of instantiation.

DEFINITION 3.16. For $n^\tau[y^\sigma]$ a meta-variable and $t$ a term of domain $\tau$, we call

$\{n[y^\sigma] := t\}$

an instantiation (of $n[y]$ by $t$). The instantiation binds the occurrences of $y$ in $t$ and $t$ may contain also variables different from those in $y$.

Since the variables $y$ are considered bound, the following two instantiations by the bound variable convention are considered identical:

$\{n[x^\sigma, y^\sigma] := xy\}$

$\{n[z^\sigma, x^\sigma] := zx\}$

The application of instantiation is defined immediately for all terms. The only interesting cases are the meta-variable instances.

$\{n[q]\} \{n[y] := t\} := t[q\{n[y] := t\}] / y$, $\{m[q]\} \{n[y] := t\} := m[q\{n[y] := t\}]$ for $m, n$ different meta-variables.
3. INCOMPLETE TERMS AND PROOFS IN HIGHER-ORDER LOGIC

Note that the instantiations have to be applied hereditarily (also to \( \vec{q} \) in the first case), because \( \vec{q} \) may contain \( \vec{n} \), so for example

\[
\vec{n}(f \ a), \vec{q}(a, (f \ a)) \mid \{n[x, y] := g \ x \ y\} = g (f \ a)(g \ a (f \ a)).
\]

The well-foundedness of the instantiation can easily be proven by induction on the structure of the term in which we instantiate. Informally, we can think of the instantiation \( M\{\vec{n}|g := t\} \) as (a reduct of) \( (\lambda n.M)(\lambda g.t) \), of the declaration \( \vec{q}^*\{\vec{y}|g\}\ ) as a meta-level skolem function from \( \vec{q} \) to \( \vec{y} \) and of the instance of a meta-variable \( \vec{n}(\vec{r}) \) as a fully applied skolem function.

Sometimes we have to rename bound variables in derivations before performing an instantiation. This problem is not really new for oHOL, because it already appears in HOL (when performing a substitution, see Example 3.15). To make our point clear we treat the following example.

**Example 3.17.** Consider a derivation \( \Sigma \) of \( (P \vec{n}|x) \) and a derivation \( \Theta \) of \( (P \vec{n}|x) \), where \( \Theta \) and \( \Sigma \) do not contain a free \( x \) in their assumptions. We can do a \( \forall \)-introduction and we can perform an instantiation, \( \{\vec{n}|x := x + y\} \) on \( \Sigma \), respectively \( \{\vec{n}|x := x + y\} \) on \( \Theta \).

In the first derivation, to perform the instantiation, we first have to rename the bound variable \( x \).

\[
\begin{array}{c}
\Sigma \\
\forall x. P(\vec{n}|x) \\
\end{array} \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \q
3.4. Open Proofs with Binding Holes

In this section we will consider a few more examples of incomplete proofs and see that in some situations (e.g. forward reasoning) we may want a more sophisticated representation of unknowns that is capable of simulating the binding effects of the introduction rules that are intended to occur in the still unknown part of the proof. We will refine our formalization from Section 3.3.2 to accommodate these features.

3.4.1. More Examples: Holes and Introduction Rules. An implication introduction rule may discharge one or more occurrences of the hypothesis being abstracted. It creates a binding for this discharged hypothesis. An introduction of a universal quantifier also creates a binding (in the sense of Definition 3.11). We may ask ourselves the question: How do we account for the binding of hypotheses or variables being used in the known parts of a proof that are supposed to be bound by an introduction rule in the unknown part of the proof? The answer to this question will lead us to a refined formalization of incomplete proofs in HOL.

3.4.1.1. Forward reasoning and introduction rules. When doing forward reasoning we naturally obtain unknown proofs with known subproofs. The introduction rules intended to be in the unknown part however influence the known part of the proof. This is illustrated in the following two examples.

Example 3.20. An incomplete proof with an implication introduction rule in the missing part.

Suppose we want to give a forward construction of a proof of \( A \rightarrow C \) from assumptions \( A \rightarrow B \) and \( A \rightarrow B \rightarrow C \).

The incomplete proof (1) in Figure 3.8 reflects the initial situation where we have the two assumptions \( A \rightarrow B \) and \( A \rightarrow B \rightarrow C \) and we are looking for a proof of \( A \rightarrow C \). In (2) we introduce an extra hypothesis \( A \) that is used in (3) to derive the formulas \( B \) and \( B \rightarrow C \) (in two forward steps). Finally in (4) we finish the proof (in two more forward steps) by deriving \( C \) and using the \( \neg \)-introduction rule to discharge the extra hypothesis \( A \) that was introduced in step (2). Note that instead of (2), one may transform (1) into the problem of proving \( C \) from assumptions \( A, A \rightarrow B \) and \( A \rightarrow B \rightarrow C \), but this would be a backward step as it amounts to backward application of the implication introduction rule.

Since we construct the natural deduction tree from the leaves to the root, in any forward proof of the above tautology we always will have to first introduce an extra hypothesis \( A \) and discharge it at a later stage. For this reason (1) and (4) represent (incomplete) derivations of \( A \rightarrow C \) from two assumptions, but (2) and (3) represent incomplete derivations from three assumptions. On the other hand, we would like to view the three transitions as refinements which means that we should be able to explain the result of a given step by a (partial) solution of the unknowns from the previous step. The problem is the binding for the extra hypothesis \( A \) introduced at the second step. In a sense this is not a 'normal' hypothesis because when we introduced it we ‘promised’ that it will be discharged. This promise is however only implicit and we have no means of enforcing it.
(1) \[
\begin{array}{c}
\frac{A \rightarrow B}{A \rightarrow C} \\
\frac{A \rightarrow B}{A \rightarrow C} \\
\frac{A \rightarrow B}{A \rightarrow C}
\end{array}
\]

(2) \[
\begin{array}{c}
\frac{A \rightarrow B}{A \rightarrow C} \\
\frac{A \rightarrow B}{A \rightarrow C} \\
\frac{A \rightarrow B}{A \rightarrow C}
\end{array}
\]

(3) \[
\begin{array}{c}
\frac{A \rightarrow B}{A \rightarrow C} \\
\frac{A \rightarrow B}{A \rightarrow C} \\
\frac{A \rightarrow B}{A \rightarrow C}
\end{array}
\]

(4) \[
\begin{array}{c}
\frac{[A]^i \rightarrow B}{[A]^i \rightarrow C} \\
\frac{[A]^i \rightarrow B}{[A]^i \rightarrow C} \\
\frac{[A]^i \rightarrow B}{[A]^i \rightarrow C}
\end{array}
\]

Figure 3.8. Forward construction of a proof of \( A \rightarrow C \) from \( A \rightarrow B \) and \( A \rightarrow B \rightarrow C \).

Formally this means that (5) could also be accepted as a valid last step, although it is obviously not a derivation of \( A \rightarrow C \) from the original two assumptions (but from three instead because one of the \( A \)s is not discharged). Therefore, we

(5) \[
\begin{array}{c}
\frac{A \rightarrow B}{A \rightarrow C} \\
\frac{[A]^i \rightarrow B}{[A]^i \rightarrow C} \\
\frac{[A]^i \rightarrow B}{[A]^i \rightarrow C}
\end{array}
\]

Figure 3.9. A different last step leads to inconsistency.

need a mechanism that allows us to state that a hypothesis will be discharged by the rules that will be filled in when solving the (claim) rule. To do that we will assign temporary bindings to the instances of the (claim) rule where the binding is supposed to occur. Then (2) and (3) become (6) and (7) in Figure 3.10. In (6) and

(6) \[
\begin{array}{c}
\frac{(A^i) \rightarrow B}{(A^i) \rightarrow C} \\
\frac{(A^i) \rightarrow B}{(A^i) \rightarrow C} \\
\frac{(A^i) \rightarrow B}{(A^i) \rightarrow C}
\end{array}
\]

(7) \[
\begin{array}{c}
\frac{[A]^i \rightarrow B}{[A]^i \rightarrow C} \\
\frac{[A]^i \rightarrow B}{[A]^i \rightarrow C} \\
\frac{[A]^i \rightarrow B}{[A]^i \rightarrow C}
\end{array}
\]

Figure 3.10. The hole can discharge hypotheses.
(7) we have added an extra assumption $A$ to each of the two subderivations. The notation $\{A^i\}_\Sigma$ expresses that $A$ is discharged in the derivation $\Sigma$. This is clearly visible in (7) where we see the discharged hypotheses used. Now the (claim) rule contains information that ensures that any of its solutions provides binding for the two discharged hypotheses.

**Example 3.21.** An incomplete proof with a $\forall$-introduction rule in the missing part.

Let $P$ be a unary predicate and $A$ and $B$ be formulas. Consider the incomplete derivation (8) and one of its possible forward completions (9) in Figure 3.11. Looking at the variable $x$ we notice that in (8) it occurs free while in (9) it has been bound. This does not create the consistency problems of Example 3.20, but we clearly see another side of the problem. The unknown part (10) in (8) is replaced by the tree (11) in (9). As we can see (11) is not a well-formed deduction

<table>
<thead>
<tr>
<th>(8)</th>
<th>$\forall z. A \to P(z)$</th>
<th>$A \to P(x)$</th>
<th>$P(x)$</th>
<th>$(\forall y. P(y)) \to B$</th>
<th>$\frac{}{B}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(9)</td>
<td>$\forall z. A \to P(z)$</td>
<td>$A \to P(x)$</td>
<td>$P(x)$</td>
<td>$(\forall y. P(y)) \to B$</td>
<td>$\frac{}{B}$</td>
</tr>
</tbody>
</table>

**Figure 3.11.** An incomplete proof with unknown part that is intended to contain an instance of the rule $(\forall\!\!-\!\!I)$. |

| (10) | $P(x)$ | $(\forall y. P(y)) \to B$ | $\frac{}{B}$ |
| (11) | $\forall x. P(x)$ | $(\forall y. P(y)) \to B$ | $B$ |

**Figure 3.12.** The ‘hole’ and its intended meaning.

of $B$ because the variable $x$ that is bound by the $\forall$-introduction rule occurs free in the assumption $P(x)$ and we are not allowed to bind variables that occur free in hypotheses (see Definition 3.6). Therefore, we could not express the transition
from (8) to (9) by a meta-variable refinement. If in (10) we explicitly state that the
variable \( x \) occurring in \( P(x) \) is supposed to be bound, this would imply that
it should not occur free in the assumptions of the subderivation of \( P(x) \). If the
binding of \( x \) were made explicit, then we would know that \( P(x) \) in (11) will be
given a 'correct' derivation with \( x \) not occurring in the leaves and thus we can be
sure that the instantiation would produce a correct derivation tree. The solution

\[
\frac{\forall z. A \rightarrow P(z)}{A \rightarrow P(x)} \quad \frac{\{x\} P(x)}{(\forall y. P(y)) \rightarrow B} \quad \frac{\gamma}{B}
\]

**Figure 3.13.** The binding hole makes explicit the intended binding of \( x \) in the unknown part of the proof.

that we propose is shown in Figure 3.13. The notation \( \{x\} \) there denotes the fact
that the variable \( x \) is intended to be bound by an introduction rule in the missing
part.

\[ \square \]

### 3.4.2. Open Proofs with Binding Holes (bHOL)

In this section we will extend HOL with unknowns that are capable of acting as temporary binders for
variables that are intended to be bound (or for assumptions intended to be discharged), but whose corresponding introduction rule is not constructed yet.

#### 3.4.2.1. The Syntax

The domains of the extension are the domains of HOL. As in HOL and oHOL, Greek letters \( \sigma, \tau, \ldots \) will range over the set of domains \( \mathcal{D} \). We assume that for each domain \( \sigma \) there is an infinite set \( \mathcal{V}^\sigma \) of variables of
that domain. These variables will be denoted by \( x^\sigma, y^\sigma, z^\sigma \), etc. and the domain
may be omitted if it is irrelevant. We also assume that for each domain \( \sigma \) we are
given an infinite set \( \mathcal{M}^\sigma \) of meta-variable names.

We will refer to any sequence of different variables as a variable list. Variable
lists will be denoted by \( \Theta \) and \( \varepsilon \) will denote the empty list.

**Definition 3.22** (Meta-variables). Given a domain \( \sigma \), any expression of the
form \( m^\tau[\Delta] \) is called a meta-variable declaration. The declaration defines the
domain \( \sigma \) and the parameter list \( \Delta \) of the meta-variable \( m \). The parameter lists
are generated by the following grammar:

\[ \Delta ::= \varepsilon \mid \Delta, x^\tau \mid \Delta, y^\tau[\Theta] \]

with \( \tau \) a domain, \( \varepsilon \) being the empty list, \( x^\tau \in \mathcal{V}^\tau \) and \( y^\tau \in \mathcal{M}^\tau \).

We will use \( m, n \), etc. for meta-variable names and \( \Delta \) will range over parameter
lists. Examples of meta-variables are:

\[ m^\sigma[\varepsilon], \ m^\tau[x^\sigma, y^\tau], \ n^\sigma[m^\tau_1[x^{\sigma_1}, y^{\sigma_2}], z^{\sigma_2}] \]
We will consider the (meta-)variable declarations appearing in the list of parameters of a meta-variable as bound by that meta-variable (thus \(x^\sigma\) and \(y^\tau\) are bound in \(\overline{\sigma^m[\tau^\sigma][x^\sigma, y^\tau]}\)) and we will apply the usual variable convention for them. Therefore, parameter lists that differ only in the names of the declared variables will be considered identical. Furthermore, we will assume that the parameter list of a meta-variable is uniquely determined by the name of the meta-variable and for this reason we will identify meta-variables with their names.

Intuitively, \(\overline{\sigma^m[\Delta]}\) represents an unknown term of domain \(\sigma\) parameterized by \(\Delta\). The use of the parameters in \(\Delta\) is illustrated in Example 3.25 below.

We now introduce the term language of HOL with binding unknowns. In addition to the usual syntactic category of terms, we have a new category of argument lists. As suggested by their name, the argument lists represent the actual arguments of meta-variables.

**Definition 3.23 (Terms).** The terms and argument lists are constructed by simultaneous induction as described below:

**Terms:**

- **Variable:** Every variable \(x^\sigma\) is a term of domain \(\sigma\).
- **Meta-variable instance:** If \(\overline{\sigma^m[\Delta]}\) is a meta-variable and \(A\) is a \(\Delta\)-argument list then \(\overline{\sigma^m[A]}\) is a term of domain \(\sigma\).
- **Application:** If \(M\) is a term of domain \(\sigma \rightarrow \tau\) and \(N\) is a term of domain \(\sigma\) then \(MN\) is a term of domain \(\tau\).
- **Abstraction:** If \(x\) is a variable of domain \(\sigma\) and \(M\) is a term of domain \(\tau\) then \(\lambda x^\sigma M\) is a term of domain \(\sigma \rightarrow \tau\).
- **Formulas:** If \(A\) and \(B\) are terms of domain \(\text{Prop}\) and \(x^\sigma\) is a variable of domain \(\sigma\) then \(A \land B\), \(A \lor B\), \(\neg A\), \(\forall x^\sigma A\), \(\exists x^\sigma A\) are terms of domain \(\text{Prop}\).

**Argument Lists:**

- The empty list \(\varepsilon\) is an \(\varepsilon\)-argument list.
- If \(A\) is a \(\Delta\)-argument list and \(M\) is a term of domain \(\sigma\) then \(A, \langle \varepsilon \rangle M\) is a \((\Delta, x^\sigma)\)-argument list.
- If \(A\) is a \(\Delta\)-argument list and \(M\) is a term of domain \(\sigma\) and \(\overline{\sigma^m[\Theta]}\) is a meta-variable then \(A, \langle \Theta \rangle M\) is a \((\Delta, \overline{\sigma^m[\Theta]}\))-argument list.

**Remark 3.24.** Note that argument lists are not terms, in particular \(\langle \Theta \rangle M\) is not a term.

**Example 3.25 (Well-formed terms).** Examples of well-formed terms are \(x^\sigma\) and \(\lambda x^\sigma x\). We can have more complicated terms though, like for example

\[
\lambda y^\tau \overline{\sigma^m[(x^\sigma) \ f \ x \ \lambda z^\tau \ y^\tau]}
\]

is a term of domain \(\tau \rightarrow \sigma \rightarrow \tau\) if \(m\) has parameter list \(\overline{\sigma^m[\tau^\sigma \rightarrow \tau^\sigma][x^\sigma, y^\tau]}\) and \(f\) is of domain \(\sigma \rightarrow \sigma \rightarrow \tau\). This term represents a function of \(y^\tau\) with body that is unknown (the meta-variable \(\overline{\sigma^m}\)). When constructing a solution for this unknown we are allowed to use the terms \(\lambda z^\tau y^\tau\) and \(f x\). The prefix \((x^\sigma)\) tells us that the free variable \(x\) of \(f x\) should be bound by the solution. For example, a possible solution would be to instantiate \(\overline{\sigma^m}\) with \(\lambda x_0^\sigma \ n[x_0] g\) (where \(n\) and \(g\) are
the parameters of \( n \) to obtain the term:
\[
\lambda y^\tau. \lambda x^\sigma. \ f x \ (\lambda z^\tau. y)
\]

How this result is computed can be seen in Example 3.36 in Section 3.5.1 where we give the definition of instantiation.

In general, a meta-variable instance has the shape \( \overline{m}^\sigma[\langle x_1^\tau \rangle M_1 \ldots \langle x_n^\tau \rangle M_n] \) and each variable list \( \overline{x}_i \) introduces variables that may appear free in \( M_i \), but a solution for \( n \) must bind them. Intuitively, the terms \( M_i \) are the parts of the solution that have been constructed forwards and \( \overline{x}_i \) contains variables whose binders have not been constructed yet.

**Definition 3.26 (Free and Bound Variables of Terms).** The set of variables \( \text{FV}(M) \) with free occurrences in a term \( M \) is defined as usual by induction on the structure of \( M \). In the nonstandard case when \( M \) is a meta-variable instance we define:
\[
\text{FV}(\overline{m}[\langle x_1^\tau \rangle M_1 \ldots \langle x_n^\tau \rangle M_n]) = \bigcup_{i=1}^n (\text{FV}(M_i) \setminus \overline{x}_i)
\]

Bound variables are variables that do not occur free.

As usual, we adopt the variable convention and assume that all bound variables are chosen to be different from each other and the free variables. From the definition above we clearly see that the meta-variables play the role of binders and can introduce variables in the scope of their arguments. This motivates the use of the term ‘binding holes’.

\[
\begin{align*}
\text{FV}(f x) &= \{f, x\} \\
\text{FV}(\lambda x^\tau. m^\sigma[y^\tau[f x, \lambda z^\tau. y]]) &= \{f, y\} \\
\text{FV}(\mu^\tau[[x^\sigma]f x, \forall z^\tau. (f z = f x)]) &= \{f, x\}
\end{align*}
\]

**Figure 3.14.** Computing the free variables of some terms.

Next, we define the operation of substituting a term for a variable in a term.

**Definition 3.27.** Let \( N \) be a term and \( x \) be a variable. The result of the capture-avoiding substitution of \( x \) by \( N \) in a term \( M \) (notation \( M[N/x] \)) is defined as usual by induction on \( M \) with an extra clause for meta-variables:
\[
\begin{align*}
\overline{m}[(\langle x_1^\tau \rangle M_1 \ldots \langle x_n^\tau \rangle M_n)N/x] &= \overline{m}[(\langle x_1^\tau \rangle M_1N/x, \ldots, \langle x_n^\tau \rangle M_nN/x)] \\
x[N/x] &= N \\
y[N/x] &= y \\
(M_1M_2)[N/x] &= M_1[N/x]M_2[N/x] \\
(Qy.M)[N/x] &= Qy.M[N/x] \\
(M_1\sigma M_2)[N/x] &= M_1[N/x]\sigma M_2[N/x] \\
(\neg M)[N/x] &= \neg M[N/x]
\end{align*}
\]

where \( x \neq y \), \( Q \in \{\lambda, \forall, \exists\} \) and \( \sigma \in \{\land, \lor, \neg\} \).
3.4. Open Proofs with Binding Holes

Note that due to the variable convention we can assume that $x$ is different from each variable in $\vec{x}_i$ and no renaming is necessary.

3.4.2. The Deduction Rules. The difference in the formulation of oHOL and its extension so far is the introduction of hereditarily parameterized meta-variables on the level of terms. On the proofs level we take the deduction rules of HOL plus a modified version of the (claim) rule that allows the hole to act as a binder for some of the variables occurring in its known subproofs.

**Definition 3.28** (The rule (claim) with binding).

\[
\begin{array}{c}
\Sigma_1 \\
\vdots \\
\Sigma_n \\
\end{array}
\begin{array}{c}
\vec{A}_1 \\
\vdots \\
\vec{A}_n \\
\end{array}
\begin{array}{c}
\{\vec{x}_1, \vec{A}_1\} B_1 \\
\{\vec{x}_n, \vec{A}_n\} B_n \\
\end{array}
\frac{}{C} \text{(claim)}
\]

For each $k$, the hypotheses $\vec{A}_k$ are discharged in $\Sigma_k$. After discharging $\vec{A}_k$, $\vec{x}_k$ may not occur in an undischarged hypothesis of $\Sigma_k$.

When applying the (claim) rule we need to check that the variables $\vec{x}$ do not occur free in the assumptions of the subderivations $\Sigma$ after discharging the assumptions $\vec{A}$. Examples of a valid and invalid instances of the (claim) rule are given in Figure 3.15.

\[\begin{array}{c}
\{x, A(x)^r\} [A(x)]^r \\
\{x, A(x)^r\} B(x) \\
\end{array}
\frac{}{\forall y. (A(y) \rightarrow A(y))}
\]

\[\begin{array}{c}
\forall y. (A(y) \rightarrow A(y)) \\
\end{array}
\frac{}{\forall y. (A(y) \rightarrow A(y))}
\]

**Figure 3.15.** A valid (a) and invalid (b) instance of the (claim) rule.

The intuition behind the discharging of hypotheses in the (claim) rule is that it provides a temporary discharging of hypotheses that need to be discharged, but the rule that should do so is intended to be in the not yet constructed part of the proof represented by the (claim) rule. We have seen this situation in Example 3.20 where one needs to use a hypothesis in step (2) before the corresponding ($\rightarrow$-I) rule that should discharge the hypothesis has been constructed in step (4). This allows us to enforce that all temporarily introduced hypotheses are properly discharged.

The binding of object variables $x_i$ by the (claim) rule plays a similar role. It allows us to give a temporary binding for a bound variable that needs to be
used before its corresponding (\(\forall\)-I) rule has been created. The binding of object variables allows us to capture the refinements in Example 3.21.

After having defined the deduction rules we would like to define the notion of derivability. As in the case of oHOL, we need to take into consideration the instances of the (claim) rule occurring in a proof. This requires changes in the definitions of a goal and a variable bound in a proof.

**Definition 3.29** (Bound variable at a given position in a derivation). Let \(\Sigma\) be a derivation and \(A\) be a formula occurrence in it such that \(x \in \text{FV}(A)\). Then the variable \(x\) is bound at position \(A\) in \(\Sigma\) in one of the following situations

\[
\frac{A}{\forall x^\sigma. B} (\forall\text{-I}) \quad \frac{\exists x^\sigma. C}{B} (\exists\text{-E}) \quad \frac{A}{\text{claim}}
\]

Recall that the notation \(\{x, \ldots\}\) introduces a (temporary) binder for \(x\).

We will often identify a node in a derivation tree with its label when it does not cause confusion. Thus when talking about a bound occurrence of \(x\) in \(A\) we have in mind a specific occurrence of \(A\) (i.e. a node \(\pi\) in the deduction tree labelled by \(A\)) in the derivation. We will also adopt the variable convention for variables that are bound in a derivation.

**Definition 3.30.** Let \(\vec{A}_i, B_i\) and \(A\) be formulas and \(\vec{x}\) and \(\vec{x}_i\) be variables. A goal is any expression of the following shape:

\[
\vec{x}, \{\vec{x}_1, \vec{A}_1\} B_1, \ldots, \{\vec{x}_n, \vec{A}_n\} B_n \vdash A
\]

The occurrences of \(\vec{x}\) are bound and in scope in all formulas. The occurrences of \(\vec{x}_i\) are also bound, but their scope is only limited to \(\vec{A}_i\) and \(B_i\).

A goal \(G = \vec{x}, \{\vec{x}_1, \vec{A}_1\} B_1, \ldots, \{\vec{x}_n, \vec{A}_n\} B_n \vdash A\) is a goal of a derivation \(\Sigma\) if \(\Sigma\) contains an occurrence of the claim rule

\[
\frac{\begin{array}{c}
\vec{x}_1, \vec{A}_1 \\
\vdots
\end{array}}{\Sigma_1} \quad \frac{\begin{array}{c}
\vec{x}_n, \vec{A}_n \\
\vdots
\end{array}}{\Sigma_n} \quad \frac{\begin{array}{c}
\vec{x}, \vec{A} \\
\vdots
\end{array}}{\Sigma} (\text{claim})
\]

and \(\vec{x}\) contains the list of variables that occur free in \(\Sigma\) and \(A\) but are bound in \(\Sigma\).

**Example 3.31.** The occurrences of the (claim) rule in (7) in Figure 3.10 and in Figure 3.13 give rise to the following two goals:

\[
\{A\} B, \{A\} B \rightarrow C \vdash A \rightarrow C \\
\{x^2\} P x, \{\forall y^\omega. P y\} \rightarrow B \vdash \cdots B
\]
3.5. Refinement and Completeness

3.5.1. Solving the Unknowns. In this section we will define the two instantiation operations that allow us to fill in a value for an unknown object.

3.5.1.1. Instantiating meta-variables. One of the main properties of the substitution of a term for the free occurrences of a variable in another term is that no free variables get ‘captured’. This is guaranteed by the adopted variable convention that ensures that bound variables are always different from the free ones. When one works with incomplete terms, it often happens that the holes of the open term are under one or more binders and they should be filled in with terms that may contain variables bound by them. Therefore, we need a mechanism allowing us to have variable capturing when instantiating a meta-variable. At the same time this mechanism should behave ‘properly’ in the sense that it should allow α-conversion and retaining of the variable convention.

We will present the instantiation operations for bHOL. As oHOL is contained in bHOL, we can obtain the instantiations in it as restrictions of the operations in bHOL. We start with the definition of the instantiation of meta-variables.

Definition 3.34 (Meta-variable instantiation). Let $m^\sigma[\Delta]$ be a (meta-)variable where $\Delta = x_1^\sigma_1[\Theta_1], \ldots, x_n^\sigma_n[\Theta_n]$, $n \geq 0$. The instantiation of $m$ by an arbitrary term $N$ of sort $\sigma$ in the term $M$ (notation $M\{m[\Delta] := N\}$) is defined as follows:
\[
m_\tau[(\Theta_1)_{u_1}, \ldots, (\Theta_n)_{u_n}]{m[\Delta]} := N \quad = \quad N\{x_1[\Theta_1] := u_1^* \ldots x_n[\Theta_n] := u_n^* \}
\]
\[
\overline{\tau}[(\Theta'_1)_{u'_1}, \ldots, (\Theta'_k)_{u'_k}]{m[\Delta]} := N \quad = \quad \overline{\tau}[(\Theta'_1)_{u'_1}, \ldots, (\Theta'_k)_{u'_k}]
\]
\[
(M_1M_2){m[\Delta]} := N \quad = \quad M_1^* M_2^*
\]
\[
(\lambda y: \sigma. M){m[\Delta]} := N \quad = \quad \lambda y: \sigma. M^*
\]
\[
(\forall y: \sigma. B){m[\Delta]} := N \quad = \quad \forall y: \sigma. B^*
\]
\[
(\exists y: \sigma. B){m[\Delta]} := N \quad = \quad \exists y: \sigma. B^*
\]
\[
(\neg B){m[\Delta]} := N \quad = \quad \neg B^*
\]
\[
(A \rightarrow B){m[\Delta]} := N \quad = \quad A^* \rightarrow B^*
\]
\[
(A \lor B){m[\Delta]} := N \quad = \quad A^* \lor B^*
\]
\[
(A \land B){m[\Delta]} := N \quad = \quad A^* \land B^*
\]

where on the right-hand side for readability we have abbreviated \(M{m[\Delta]} := N\) by \(M^*\) for every term \(M\) (so \(A^*\) means \(A{m[\Delta]} := N\)).

The interesting clause in this definition is the first one, where we instantiate the meta-variable instances. The term \(N\) by which we instantiate is in the context of the parameters in \(\Delta\). The values for these parameters are \(u_i\). Therefore, after propagating the instantiation of \(m\) to \(u_i\), we need to initiate recursively instantiations \(\{x_i[\Theta_i] := u_i^*\}\) for the parameters in the body \(N\) of the term instantiating \(m\). These recursive calls terminate because \(\Theta_i\) contains only variable declarations.

Note that the variable convention is used to resolve name conflicts. We extend the convention to instantiations by considering the variables of \(\Delta\) as being bound by the instantiation and identifying instantiations obtainable from each other by renaming. For example \(\{m[x:A] := \lambda y:B. x\}\) is the same instantiation as \(\{m[y:A] := \lambda z:y.B. y\}\).

To streamline the definition of instantiation, we treated the variables as meta-variables with empty parameter lists. Thus instantiation is defined also for variables. The following observation states that on variables instantiation and substitution coincide:

**Proposition 3.35.** Let \(x\) be a variable and \(N\) be a term. Then for every \(M\)

\[M[N/x] \equiv M\{x[\_] := N\}\]

The instantiation on terms extends immediately to contexts, goals and derivations in the obvious manner. We will use \(\Gamma{m[\Delta]} := N\), \(G{m[\Delta]} := N\) and \(\Sigma{m[\Delta]} := N\) respectively to denote the extended instantiation.

**Example 3.36** (Variable Capture). Let for a term \(M\), \(M^*\) denote the result of performing the instantiation

\[
\{m^* \varphi_n[n^* \varphi_n[x^*], g^*] := \lambda x_0^* \cdot n[x_0\ g]\}
\]

Then (see Example 3.25) we have

\[
(m^*\varphi_n[(x^*)f\ x, (\lambda z^*\ y)^*])^* = (\lambda x_0^* \cdot n[x_0\ g] \{n[x^*] := (f\ x)^*\} \{g[\_] := (\lambda z^*\ y)^*\})
\]
\[
= (\lambda x_0^* \cdot n[x_0\ g] \{n[x^*] := (f\ x)^*\} \{g[\_] := (\lambda z^*\ y)^*\})
\]
\[
= \lambda x^* \cdot (f\ x)^* (\lambda z^*\ y)^*
\]
\[
= \lambda x^* \cdot f\ x (\lambda z^*\ y)
\]

Please note that the variable \(x\) in the term \(f\ x\) is bound after the instantiation.
As we now know how to represent unknowns by meta-variables and how to solve them by instantiation, one needs to show that instantiation preserves the structure of proofs. This would provide the soundness for the delayed choices for terms. When the moment comes in a proof construction that we know what should be filled in, we can instantiate the meta-variable and the proof would look like as if we knew the term from the beginning.

**Proposition 3.37** (Instantiation preserves the domains of the terms). Let $M$ be a term of domain $\sigma$. If $m^\tau[\Delta]$ is a meta-variable of domain $\tau$ and $t$ is a term of domain $\tau$, then the term $M\{m[\Delta] := t\}$ is of domain $\sigma$.

**Proof.** Define the depth of the empty context to be zero and the depth of $\Delta, n^\sigma[\Sigma]$ to be the maximum of the depth of $\Delta$ and the depth of $\Sigma$ plus one.

The proposition is proven by induction on the depth of the context $\Delta$. For a fixed $\Delta$ and assuming the proposition holds for all contexts of lesser depth, we proceed by induction on $M$. In the case of $M$ being an instance of $m$, we apply the first induction hypothesis (note that instantiation preserves the depth of contexts) and in the other cases we use the second. \qed

**Proposition 3.38.** Let $m^\sigma[\Delta_1]$ and $n^\tau[\Delta_2]$ be meta-variables. Assume that $m$ does not occur in $N$. Then for all $M$ and $P$

\[ P\{m^\sigma[\Delta_1] := M\}\{n^\tau[\Delta_2] := N\} \equiv P\{n^\tau[\Delta_2] := N\}\{m^\sigma[\Delta_1] := M\{n^\tau[\Delta_2] := N\}\} \]

**Proposition 3.39** (Instantiation preserves $\beta$-reductions). For every meta-variable $m[\Delta]$ and every term $N$, if $P \rightarrow^\beta Q$ then

\[ P\{m[\Delta] := N\} \rightarrow^\beta Q\{m[\Delta] := N\} \]

**Theorem 3.40** (Instantiation preserves derivability). Let $m^\sigma[\Delta]$ be a meta-variable of domain $\sigma$ and $t$ be a term from the same domain. If $\Gamma; G \vdash A$ then $\Gamma\{m[\Delta] := t\}; G\{m[\Delta] := t\} \vdash A\{m[\Delta] := t\}$.

**Proof.** Induction on the structure of derivation trees. Using the two propositions above we can check that the side conditions of the derivation rules making up the tree would be preserved. \qed

**3.5.1.2. Instantiating the (claim)-rule.** The (claim) rule stands for a number of unknown inference rules. Its premises stand for (possibly incomplete) subderivations that may be used when filling in the unknown proof steps. For reasons of readability, we will show how to instantiate a (claim) rule with one assumption.

Let the instance of the rule that we want to instantiate be the one given at (12a). An instantiator for it is any derivation of $B$ (see (12b)) together with a list of subderivations $\pi_1, \ldots, \pi_n$ in it that have the shape given in (12c).
Then the result of the instantiation is obtained by replacing the subderivations \( \pi_i \) with the derivations \( \pi'_i \) at (12d).

**Example 3.41** (See also Example 3.20). Consider the following incomplete proof

\[
\Sigma = \frac{[A]^i}{\{A_i\}} \frac{A \rightarrow B}{(A_i)B} \frac{A \rightarrow C}{\{A_i\}B \rightarrow C}
\]

An instantiator for it is the derivation

\[
\mathcal{D} = \frac{[A]^i}{\frac{B \rightarrow C}{\{A_i\}}} \frac{A \rightarrow B}{\frac{B \rightarrow C}{\{A_i\}}} \frac{A \rightarrow C}{\{A_i\}}
\]

The derivations \( \pi_1 \) and \( \pi_2 \) corresponding to the two premises of the claim rule are respectively

\[
\pi_1 = \frac{A}{B} \frac{B \rightarrow C}{A \rightarrow C}
\]

\[
\pi_2 = \frac{A}{B} \frac{B \rightarrow C}{A \rightarrow C}
\]

They get instantiated by the two derivations in the premises obtaining:

\[
\pi_1' = \frac{A}{B} \frac{B \rightarrow A}{A \rightarrow B} \frac{A \rightarrow C}{B \rightarrow C}
\]

\[
\pi_2' = \frac{A}{B} \frac{B \rightarrow A}{A \rightarrow B} \frac{A \rightarrow C}{B \rightarrow C}
\]

To obtain the final result we replace the claim rule in \( \Sigma \) by the result of replacing \( \pi_1 \) by \( \pi_1' \) and \( \pi_2 \) by \( \pi_2' \) in \( \mathcal{D} \):

\[
\Sigma' = \frac{[A]^i}{B} \frac{A \rightarrow B}{B \rightarrow C} \frac{A \rightarrow B \rightarrow C}{B \rightarrow C}
\]

**Example 3.42** (See also Example 3.21). Consider the incomplete derivation

\[
\Sigma = \frac{\forall z. A \rightarrow P(z)}{A} \frac{A \rightarrow P(x)}{P(x)} \frac{(\forall y. P(y)) \rightarrow B}{\frac{B}{}}
\]
and instantiator

\[
D = \frac{\pi}{P(x)} \frac{\forall z. A \rightarrow P(z)}{A \rightarrow P(x)} \frac{(\forall z. A \rightarrow P(z)) \rightarrow B}{B}
\]

The derivation \( \pi \) corresponding to the non-trivial first hypothesis is

\[
\pi = \frac{\pi}{P(x)} \frac{A}{\forall z. A \rightarrow P(z)}
\]

it get instantiated by the derivation at the first premise

\[
\pi' = A \frac{\forall z. A \rightarrow P(z)}{A \rightarrow P(x)} \frac{A \rightarrow P(x)}{P(x)}
\]

and then the final result is as expected

\[
\frac{\forall z. A \rightarrow P(z)}{A \rightarrow P(x)} \frac{A \rightarrow P(x)}{P(x)} \frac{\forall x. P(x)}{(\forall y. P(y)) \rightarrow B}
\]

Similarly to the instantiation of meta-variables, one can show the following result:

**Proposition 3.43.** If \( \Sigma \) is a derivation with a goal \( G \) and if \( D \) is an instanstial for \( G \), then the result of instantiating the goal \( G \) by \( D \) in \( \Sigma \) is a valid derivation.

We will omit the proof of this proposition here, but its validity will follow from the faithfulness of the extended Curry-Howard embedding (see Chapter5) and the relevant instantiation lemmas for the typing system presented in the next chapters of the thesis.

We also want to note that finding the derivations \( \pi_i \) for forming an instantiation may involve matching.

**3.5.2. Completeness.** The method of proof construction adopted in our model of incomplete proofs is to start with an unknown representing the proof to be constructed and then iterate what we will call refinement steps: choose an unknown, find a partial solution for it, possibly introducing new unknowns and instantiate the original unknown by the solution found.

We will refer to this process as successive refinement. In the examples earlier in his chapter we introduced the two major modes of successive refinement – forward and backward reasoning. Given the formalization of incomplete proofs presented in the previous sections, we will define formally the notions of forward and backward refinement step. We will show that the formalization in Section 3.3.2 is enough to
accommodate the incomplete proofs needed to construct any HOL-proof by means of backward refinement steps (backward completeness).

It turns out however that this formalization is not complete with respect to the forward refinement steps as it does not support binding for variables and assumptions whose corresponding introduction rule is not yet constructed. Fortunately we do have precisely this feature in our extended formalization with binding holes from Section 3.4.1, for which we can prove completeness of the forward reasoning.

3.5.2.1. Backward and forward refinement. Completeness.

**Definition 3.44** (Refinement step). A refinement step is a triple \((\Sigma_1, i, \Sigma_2)\), denoted by \(\Sigma_1 \overset{i}{\rightarrow} \Sigma_2\), such that the derivation \(\Sigma_2\) is obtained by applying the instantiation \(i\) to the derivation \(\Sigma_1\) and furthermore, the conclusion and the sets of the free variables and the sets of the undischarged hypotheses of the two derivations are the same.

Then a successive refinement is a sequence of derivations starting with a derivation containing a single claim rule instance and such that each next derivation is obtained from the previous one by means of a refinement step.

**Definition 3.45** (Elementary backward refinement step). The backward application of every rule of HOL generates a refinement step as shown in Figure 3.16. We will call these elementary backward refinement steps.

An example of two backward refinement steps is given below:

\[
\frac{\varphi}{A \rightarrow A} \quad \frac{[A]^i}{?} \quad \frac{[A]^i}{A \rightarrow A}
\]

By induction on the subderivations one can show the following completeness result:

**Proposition 3.46.** Let \(\Sigma\) be a complete HOL derivation of \(\varphi\) with undischarged assumptions \(\varphi_1 \ldots \varphi_n\). Let \(\Sigma_0\) be the incomplete proof

\[
\frac{\varphi_1 \ldots \varphi_n}{?} \quad \frac{\varphi}{?}
\]

Then there is a successive refinement containing only elementary backward refinement steps starting with \(\Sigma_0\) and ending with \(\Sigma\).

Since the backward refinement steps do not use the binding features of the claim rule this shows that not only bHOL, but also oHOL is backward complete with respect to HOL, i.e. that in both systems every HOL-derivation can be created using backward refinement.

Unfortunately, the examples from Section 3.4.1 show that oHOL is not forward complete. The reason for that was that in purely forward constructions we need to use bound variables and discharged hypotheses before their corresponding introduction rules have been constructed. As bHOL does have the possibilities
3.5. Refinement and Completeness

| \( \varphi_1 \ldots \varphi_n \) | \( \varphi_i \) |
| \( \varphi \rightarrow \psi \) | \( \varphi \rightarrow \psi \) |
| \( \varphi_1 \ldots \varphi_n \rightarrow \varphi_1 \ldots \varphi_n \) | \( \varphi \rightarrow \psi \) |
| \( \forall x. \varphi \) | \( \varphi \rightarrow \psi \) |
| \( \varphi \rightarrow \psi \) | \( \varphi = \beta \psi \) |

**Figure 3.16.** Backward refinement steps. The premises of the (claim) rule contain only (possibly discharged) assumptions.

to represent those kinds of incomplete proofs, we would expect that bHOL is forward complete. Indeed, below we will define what a forward refinement step in bHOL is and show that in the presence of binding holes we can obtain forward completeness.

**Definition 3.47 (Elementary forward refinement step).** The forward applications of the rules of HOL give rise to the refinement steps shown in Figure 3.17. We will call them elementary forward refinement steps.

The forward elementary steps use essentially the binding features provided by bHOL. For example, a purely forward construction of the proof of \( A \rightarrow A \) can be done as follows:

\[
\frac{\varphi}{A \rightarrow A} \quad \rightarrow \quad \frac{(\varphi') [A]^i}{A \rightarrow A} \quad \rightarrow \quad \frac{[A]^i}{A \rightarrow A} \quad \rightarrow \quad \frac{[A]^i}{A \rightarrow A}
\]
Figure 3.17. Forward refinement steps.
With the notion of forward refinement defined we can show that bHOL is complete with respect to HOL.

**Proposition 3.48.** Let $\Sigma$ be a complete HOL derivation of $\varphi$ with undischarged assumptions $\varphi_1 \ldots \varphi_n$. Let $\Sigma_0$ be the incomplete proof

$$\varphi_1 \ldots \varphi_n \vdash \varphi$$

Then there is a successive refinement containing only elementary forward refinement steps starting with $\Sigma_0$ and ending with $\Sigma$.

**Proof.** (Sketch) Call an approximation any set of non-overlapping subderivations of $\Sigma$ that cover all its leaves (including the ones containing discharged assumptions). Every such approximation induces an incomplete proof in the following way — the proof consists of a claim rule with premises given by the subderivations in the approximation. The variables that occur free in a subderivation, but are bound in $\Sigma$ are bound by the claim rule. Similarly, the assumptions discharged by an introduction rule outside the approximation are discharged also by the claim rule.

Define the size of an approximation as the total number of nodes of $\Sigma$ in it. Let the size of $\Sigma$ be $N$. Then by induction on $k$ we can show that every incomplete proof induced by an approximation of size $N - k$ can be refined to $\Sigma$ using only elementary forward steps.

Then if $l$ is the total number of leaves of $\Sigma$, the unique approximation of size $l$ is forward refinable to $\Sigma$. Now it is not difficult to see that to obtain the proof induced by this unique approximation, we only need to add discharged hypotheses to the premises of the initial goal. This of course can be done using the first refinement step in Figure 3.17. \qed

3.5.2.2. **Forward Reasoning and Local Definitions.** In our discussion so far we have taken the notion ‘forward reasoning’ as meaning the creation of a proof-tree from the leaves to the root. In the theorem proving community however the term is sometimes given a slightly different meaning. Often forward reasoning is described as using already given or proven results to deduce new ones. To implement this mode of reasoning in type-theory-based provers, one often uses some sort of definition mechanism (local or global definitions). When a new fact can be deduced in a given context, one introduces a local definition, giving a name for the proof term. In the body of the definition then the new name can be used to continue the proof. The problem with this approach is however that if one wants to introduce new assumptions, one has to introduce a cut in the proof and use a backward step in order to get in a context containing the new assumption (see Figure 3.18). This makes it impossible to create some proofs (e.g. cut free ones) in a purely forward manner. Contrast this with the proof in Figure 3.19 where because we do not fix the place where $A$ is discharged, we do not need to create a new goal either. Of course, the obligation to discharge $A$ remains, but the proof does not have cuts and we have more flexibility in choosing where to discharge $A$. 

3. INCOMPLETE TERMS AND PROOFS IN HIGHER-ORDER LOGIC

\[
\frac{\varphi}{\psi} \Rightarrow \frac{[A]^i}{\varphi} \frac{\psi}{[A]} \frac{A \rightarrow \psi}{\psi}
\]

Figure 3.18. Introducing a fresh assumption requires a cut and a second backward step and it introduces a second goal.

\[
\frac{\varphi}{\psi} \Rightarrow \frac{[A]^i}{\varphi} \frac{\psi}{[A]}
\]

Figure 3.19. Using a binding hole one does not introduce cuts and no backward steps are needed. We also do not have to fix the place where \( A \) is discharged.

3.5.2.3. Other Notions of Completeness. The notion of completeness that we discuss in this chapter is not the only one that one can be interested in. Here we were asking the question \textit{can we create all proofs} in a specific way. There are however many ways to create the same proof. Here we have basically limited ourselves to the strategy of stepwise refinement, where one proof is created from the other by filling in some of the unknowns. The steps that we make can be described as \textit{derivable rules} of the logic. A rule \((r)\) of the form

\[
\frac{A_1, \ldots, A_n}{B}
\]

is called derivable if \( A_1, \ldots, A_n \vdash B \). A more generic approach would be to base the transformations on the set of \textit{admissible rules} that strictly extend the set of the derivable ones. A rule \((r)\) is called admissible if from \( \vdash A_i \) for each \( i \) follows \( \vdash B \).

The use of the admissible rules in proof-state transformations is related to the subject of tactics that we will discuss in Chapter 6.
CHAPTER 4

Typing for Incomplete Terms and Proofs

In this chapter we continue our discussion on incomplete terms and proofs from a type-theoretic perspective. The connection between intuitionistic propositional logic and simply-typed \( \lambda \)-calculus (known as the Curry-Howard isomorphism) and its extensions to more expressive logics has allowed wide use of typed \( \lambda \)-calculi in machine implementations of logical systems. Therefore, after extending higher-order logic (HOL) with incomplete terms and proofs in Chapter 3 we can ask ourselves what are the typed \( \lambda \)-calculi corresponding to the extensions with incomplete terms and proofs and can we extend the Curry-Howard embedding for the standard higher-order logic to embedding of the logics into the corresponding typed \( \lambda \)-calculi. In the present chapter we will give an answer to the first question by presenting two typing systems that extend the system \( \lambda \)HOL – the standard calculus representing higher-order logic. These two systems will correspond to the two extensions of HOL with incomplete terms and proofs. The second question will be the topic of discussion of Chapter 5.

Not surprisingly, the structure of the chapter follows closely the structure of Chapter 3 where we introduced the logics. We start by presenting the basic system \( \lambda \)HOL that corresponds to HOL and that will be used as a basis for the two extensions that we present in Section 4.2 and Section 4.3. At the end of the chapter, in Section 4.4 we present an extension of standard pure types systems (PTSs) inspired by the technology that we will develop to describe the typing system for HOL with binding holes. The extension allows us to work with hereditarily parameterized variables, constants and definitions.

Due to the incremental nature of the discussions in this chapter there is a lot of overlap between the systems that we describe. To keep the presentation focused we will frequently refer to definitions/statements/proofs already given for previous systems pointing out only the essential differences.

4.1. Typing for HOL: \( \lambda \)HOL

In this section we will present a formulation of the standard typing system for higher-order logic. We present the syntax, the reduction and typing rules of the system and state some of its important meta-properties.

\( \lambda \)HOL (see e.g. [Geu93, Bar92]) is a dependently-typed \( \lambda \)-calculus corresponding to higher-order logic. The language of the pseudo-terms of \( \lambda \)HOL is given in Figure 4.1. Some examples and non-examples of terms are shown in Figure 4.2. The terms are built up from the variables \( \forall \) and the sorts Prop, Type and Kind using application, \( \lambda \)-abstraction and \( \Pi \)-abstraction. The interpretations of \( \lambda \)
and application are the same as in HOL: the term $\lambda A: M$ represents a function mapping its argument $x$ of type $A$ to the term $M$ representing the body of the function and a term $f$ representing a function can be applied to an argument $t$ by means of application $f t$. The binder $\Pi$ is new. The term $\Pi x:A.B$ represents a dependent function space, i.e. the type of all functions taking an argument of type $A$ and returning a result of type $B$. Note that the variable $x$ introduced by the binder $\Pi$ can occur in the body $B$ (hence the dependency). Using the $\Pi$ binder one can encode both the connectives $\rightarrow$ and $\forall$ of HOL.

As a notational convention, we will use the letters $x, y, z$ for denoting variables, $M, N, P, \ldots$ and $f, t$ for denoting general terms, $A, B, \ldots$ for terms in type positions (e.g. $\lambda x:A.M$). $s$ will be used exclusively for sorts.

Both $\lambda$ and $\Pi$ are binders - they bind the variable written after them. The effect of binding is that the variable gets encapsulated and is not 'visible' outside of the scope of the binder. To make this more precise we need to define when a variable occurs free in a term.

**Definition 4.1 (Free Variable Occurrence).** The set $\text{FV}(M)$ of the variables with free occurrences in the term $M$ is defined as follows:

$$
\begin{align*}
\text{FV}(s) &= \emptyset \\
\text{FV}(x) &= \{x\} \\
\text{FV}(MN) &= \text{FV}(M) \cup \text{FV}(N) \\
\text{FV}(\lambda x:A.M) &= \text{FV}(A) \cup (\text{FV}(M) \setminus \{x\}) \\
\text{FV}(\Pi x:A.B) &= \text{FV}(A) \cup (\text{FV}(B) \setminus \{x\})
\end{align*}
$$

Variables that do occur in a term, but have no free occurrences are called bound. For technical reasons we adopt the following convention:

**Definition 4.2 (Barendregt’s Variable Convention).** We assume that the names of all bound variables are chosen to be different from each other and from the names of the free variables.

Of course the adoption of this convention presupposes that we identify terms up to $\alpha$-equivalence (i.e. up to renaming of bound variables).
Remark 4.3. In our further discussions we will also have to work with variables bound by other binders. It will go without further mentioning that the above convention applies to those variables too.

As in HOL, one of the basic operations on terms is the capture-avoiding substitution of a term for the free occurrences of a variable in a term:

Definition 4.4. The result of substituting \( t \) for the free occurrences of \( x \) in \( M \) (denoted \( M[t/x] \)) is defined as follows:

\[
\begin{align*}
    s[t/x] &= s \\
    x[t/x] &= t \\
    y[t/x] &= y \quad (x \neq y)
\end{align*}
\]

\[
(MN)[t/x] = M[t/x]N[t/x]
\]

\[
(\lambda z : A.M)[t/x] = \lambda z : A[t/x].M[t/x]
\]

\[
(\Pi z : A.B)[t/x] = \Pi z : A[t/x].B[t/x]
\]

Note that because of the variable convention we do not need to consider the cases \( (\lambda x : A.M)[t/x] \) and \( (\Pi x : A.B)[t/x] \). Capture-avoiding substitution is used to compute the result of applying a function represented by a \( \lambda \)-abstraction to an argument. The relation between this function application and the computed result is called as usual \( \beta \)-reduction:

Definition 4.5 (\( \beta \)-reduction and conversion).

- One-step \( \beta \)-reduction \( \rightarrow_\beta \) is the smallest relation that is compatible with the term constructors and contains the relation

\[
(\lambda x : A.M)N \rightarrow_\beta M[N/x]
\]

- Multiple-step \( \beta \)-reduction \( \rightarrow_\beta \) is the reflexive and transitive closure of the relation \( \rightarrow_\beta \).

- The smallest equivalence relation containing \( \rightarrow_\beta \) is called \( \beta \)-conversion (notation \( =_\beta \)).

Some important properties of substitution and \( \beta \)-reduction are listed below:

Property 4.6.

1. If \( x \notin \text{FV}(Q) \) then \( M[P/x][Q/y] \equiv M[Q/y][P/Q/y]/x \).
2. If \( M \rightarrow_\beta N \) and \( P \rightarrow_\beta Q \) then \( M[P/x] \rightarrow_\beta N[Q/x] \)
3. If \( M \rightarrow_\beta P \) and \( M \rightarrow_\beta Q \) then there is a term \( N \) such that \( P \rightarrow_\beta N \) and \( Q \rightarrow_\beta N \).

Now we are ready to present the typing system of \( \lambda \text{HOL} \). The system describes an inductively defined relation \( \vdash \) called typing relation (or derivability relation). It is a ternary relation \( \Gamma \vdash M : A \) between a context \( \Gamma \) and two terms \( M \) and \( A \). A context is a list of variable declarations: \( x_1 : A_1, \ldots, x_n : A_n \). It declares a variable at most once - i.e. \( x_i \neq x_j \) for \( i \neq j \). The intended meaning of a context is to assign types to the free variables of the term \( M \). The term \( A \) represents the type of \( M \). Therefore \( \Gamma \vdash M : A \) should be read as "in context \( \Gamma \) the term \( M \) has type \( A \)."

The rules defining the typing relation are given in Figure 4.3. The rule (ax) states that in the empty context \( \text{Prop} \) is of type \( \text{Type} \) and that \( \text{Type} \) is of type \( \text{Kind} \). The rule (start) states that if \( A \) is a type in a context, then a freshly
introduced variable \( x \) is of type \( A \) in the context extended by the declaration \( x:A \). The weakening rule (weak) states that a term does not change its type if we extend the context with a declaration of a variable of a well-formed type. The rules (\( \Pi \)) govern the formation of \( \Pi \)-types. Without going into details we will just mention that (\( \Pi_1 \)) allows formation of implications, (\( \Pi_2 \)) gives us universal quantification and (\( \Pi_3 \)) allows formation of types representing functional domains of HOL. The rule (\( \lambda \)) types \( \lambda \)-abstractions. Its has two premises – one is that the type of the abstraction should be valid. The other is that the body of the abstraction should be typable in a context extended with a declaration of the variable being abstracted. The application rule (app) takes as premises two derivations showing that \( M \) is of a functional type \( \Pi x:A.B \) and \( N \) is of type \( A \). Note that the type of the argument \( N \) matches the argument type of the \( \Pi \)-type. The resulting type of the application is obtained by substituting \( N \) for the free occurrences of \( x \) in \( B \). Finally, the conversion rule allows us to conclude that a term that is shown to be of type \( A \) is also of type \( B \) provided that \( A \) and \( B \) are \( \beta \)-convertible and that \( B \) is a well-formed type.

One may notice that the rules of \( \lambda \text{HOL} \) in Figure 4.3 fit the general scheme of a Pure Type System (PTS) [Ber89, Ter89, Bar92]. In fact \( \lambda \text{HOL} \) can be given as a PTS specification as shown in Figure 4.4. As a result one obtains many properties of \( \lambda \text{HOL} \) just by specializing the general results for PTSs. Below we have listed some of them:

**Proposition 4.7** (Generation Lemma for \( \lambda \text{HOL} \)).

- if \( \Gamma \vdash \text{Prop} : C \) then \( C \equiv \beta \text{ Type} \).
- if \( \Gamma \vdash \text{Type} : C \) then \( C \equiv \text{Kind} \).
\[ \lambda \text{HOL} \]

- \( S = \{ \text{Prop, Type, Kind} \} \)
- \( A = \{ \text{Prop, Type, Kind} \} \)
- \( R = \{ (\text{Prop, Prop, Prop}), (\text{Type, Type, Type}), (\text{Type, Prop, Prop}) \} \)

**Figure 4.4.** \( \lambda \text{HOL} \) as a PTS.

- if \( \Gamma \vdash x : C \) then there is a term \( A \) such that \( C \equiv_\beta A \) and \( x : A \in \Gamma \).
- if \( \Gamma \vdash \Pi x : A. B : C \) then we have one of the following three cases:
  - \( \Gamma \vdash A : \text{Prop}, \Gamma, x : A \vdash B : \text{Prop} \) and \( C \equiv_\beta \text{Prop} \).
  - \( \Gamma \vdash A : \text{Type}, \Gamma, x : A \vdash B : \text{Prop} \) and \( C \equiv_\beta \text{Prop} \).
  - \( \Gamma \vdash A : \text{Type}, \Gamma, x : A \vdash B : \text{Type} \) and \( C \equiv \text{Type} \).
- if \( \Gamma \vdash MN : C \) then there are terms \( A \) and \( B \) such that \( \Gamma \vdash M : \Pi x : A. B \), \( \Gamma \vdash N : A \) and \( C \equiv_\beta B[N/x] \).
- if \( \Gamma_1, x : A, \Gamma_2 \vdash M : C \) then there is a sort \( s \) such that \( \Gamma_1 \vdash A : s \).

**Proposition 4.8 (Weakening).** Let \( x \) be a fresh variable. If \( \Gamma_1, \Gamma_2 \vdash M : B \) and \( \Gamma_1 \vdash A : s \) then \( \Gamma_1, x : A, \Gamma_2 \vdash M : B \)

**Proposition 4.9 (Correctness of types).** If \( \Gamma \vdash M : A \) then either \( A \equiv \text{Kind} \) or there is a sort \( s \) such that \( \Gamma \vdash A : s \).

**Proposition 4.10 (Cut Lemma).** If \( \Gamma_1, x : A, \Gamma_2 \vdash M : B \) and \( \Gamma_1 \vdash N : A \) then

\[
\Gamma_1, \Gamma_2[N/x] \vdash M[N/x] : B[N/x]
\]

**Proposition 4.11 (Subject Reduction).** If \( \Gamma \vdash M : A \) and \( M \equiv_\beta N \) then

\[
\Gamma \vdash N : A
\]

**Proposition 4.12 (Decidability results).**
- There is a total recursive function \( \sigma \) such that for every \( \Gamma \) and \( M \) the result \( \sigma(\Gamma, M) \) is either a term \( A \) such that \( \Gamma \vdash A : A \) or \( \bot \) if no such term exists.
- There is a total recursive boolean function \( \tau \) such that \( \tau(\Gamma, M, A) = \text{true} \) if and only if \( \Gamma \vdash M : A \).
- The type inhabitation problem (i.e. given \( \Gamma \) and \( A \) deciding whether there exists a term \( M \) such that \( \Gamma \vdash M : A \)) is not decidable.

**Proposition 4.13 (Unicity of Types).** If \( \Gamma \vdash M : A \) and \( \Gamma \vdash M : B \) then \( A \equiv_\beta B \).

There are no general normalization results for PTSs as there are non-terminating PTSs. For \( \lambda \text{HOL} \) however normalization follows from the normalization result for the Extended Calculus of Constructions (ECC) [Luo90].

**Proposition 4.14 (Weak Normalization).** Every typable term has a \( \beta \)-normal form

**Proposition 4.15 (Strong Normalization).** Every reduction sequence of well-typed terms is finite.
4.2. Typing for oHOL: λoHOL

In this section we present λoHOL – an extension of λHOL with meta-variables for object terms and proof-terms. We show that the properties of HOL are transferred to λoHOL. By means of several examples we show how incomplete oHOL proofs can be represented as λoHOL terms.

The extension of HOL to the system with first-class incomplete proofs oHOL we presented in Chapter 3 involved adding meta-variables to the term language of HOL and adding an extra derivation rule (claim).

The meta-variables were parameterized variables representing an unknown object term. The parameters were needed to correctly handle the context of the unknowns under binders during computation (see Section 3.3).

The (claim) rule represents unknowns on the level of proofs. It has multiple derivations as premises and a formula as a conclusion. It can also be viewed as a parametric object taking the derivations in the premises as input and producing a derivation of the conclusion as output. As HOL proofs are encoded as terms of propositional types in λHOL, it is not surprising that we can represent the instances of the (claim) rule as meta-variables of propositional types.

This is exactly what we will do in λoHOL by adding meta-variables to λHOL and allowing them to have types that correspond to HOL domains or propositions. Below we present the technical details:

4.2.1. λoHOL. We assume that there is an infinite set \( \mathcal{M} \) whose elements will be called meta-variables. The syntax of the terms is extended (see Figure 4.5) with terms of the form \( m[t_1, \ldots, t_n] \) where \( m \) is a meta-variable and \( t \) are terms. As

\[
\begin{align*}
S & ::= \text{Prop} \mid \text{Type} \mid \text{Kind} \\
T & ::= S \mid \forall \mid M[T \ldots T] \mid TT \mid \lambda V : T.T \mid \Pi V : T.T
\end{align*}
\]

**Figure 4.5. Pseudo-terms of λoHOL**

a convention we assume that name of the meta-variable uniquely determines the number of arguments the meta-variable takes as arguments. Thus if \( m[t_1, \ldots, t_n] \) and \( m[u_1, \ldots, u_k] \) are well formed terms, then \( n = k \).

A meta-variable instance \( m[t] \) represents an occurrence of an unknown object. Whether this is an object-level or a proof-level term will be determined by the typing rules. Before we present them, we need to lift the notions of free variable occurrence and substitution from λHOL to λoHOL.

**Definition 4.16 (Free Variable Occurrence).** The set \( \text{FV}(M) \) of the variables with free occurrences in the term \( M \) is defined as in λHOL (Definition 4.1) with the following extra clause:

\[
\text{FV}(m[t_1 \ldots t_n]) = \bigcup_{i=1}^{n} \text{FV}(t_i)
\]

Similarly for the definition of substitution:
Definition 4.17. The result of substituting \( t \) for the free occurrences of \( x \) in \( M \) (denoted \( M[t/x] \)) is defined in \( \lambda \text{HOL} \) (Definition 4.4) with the following extra clause:

\[
(m[u_1 \ldots u_n])[t/x] = m[u_1[t/x] \ldots u_n[t/x]]
\]

As we mentioned, the typing system for \( \lambda \text{HOL} \) will have to assign types to the occurrences of the meta-variables. To do this we need some extra information about the expected types of the parameters and the type of the meta-variable. This is similar to the information stored in the context for the free variables. Therefore we will extend our notion of context by allowing meta-variable declarations in it. Such a declarations has the following shape:

\[
m[x_1:B_1 \ldots x_n:B_n] : A
\]

where \( A \) and \( B_i \) are types and \( x_i \) are different variables. A variable \( x_i \) may occur free in the types \( A \) and \( B_j \) when \( j > i \). A meta-variable declaration acts therefore as a binder for the parameters (see Convention 4.2) and the scope of a parameter begins after it has been declared and extends to the end of the meta-variable declaration. A context containing a meta-variable declaration also acts as a binder for the meta-variable. Its scope begins immediately after the declaration and extends as far to the right as possible. For example, in the context

\[
x:A, m[y:A, p;\varphi;\psi, z:C] \]

\( x \) may occur in \( A, \varphi, \psi \) and \( C \); \( m \) may occur only in \( C \); \( y \) may occur in \( \varphi \) and \( \psi \).

A typing judgment of \( \lambda \text{HOL} \) still has the form \( \Gamma \vdash M : A \), but now \( \Gamma \) is a context containing both variable and meta-variable declarations. The typing rules for \( \lambda \text{HOL} \) are the typing rules for \( \lambda \text{HOL} \) (see Figure 4.3) plus the four extra rules in Figure 4.6 for introducing meta-variable declarations (\( \text{weak}_M \)) and typing meta-variable instances (\( \text{start}_M \)). Each of the rules (\( \text{weak}_M \) and \( \text{start}_M \) comes

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash M:C \quad \Gamma \vdash B_i;\text{Type} \quad \Gamma \vdash A;\text{Type} )</td>
<td>( \text{weak}_M^{\text{Type}} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M[C] \quad \Gamma \vdash B_i;\text{Type} \quad \Gamma \vdash A;\text{Type} )</td>
<td>( \text{start}_M^{\text{Type}} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M:C \quad \Gamma \vdash B_i;\text{Type} \quad \Gamma, \vec{x};\text{Prop} )</td>
<td>( \text{weak}_M^{\text{Prop}} )</td>
</tr>
<tr>
<td>( \Gamma \vdash M[C] \quad \Gamma \vdash B_i;\text{Type} \quad \Gamma, \vec{x};\text{Prop} )</td>
<td>( \text{start}_M^{\text{Prop}} )</td>
</tr>
</tbody>
</table>

Figure 4.6. Rules for typing meta-variables in \( \lambda \text{HOL} \).

in two variants. The variants with superscript \( \text{Type} \) treat meta-variables corresponding to the object-level meta-variables of \( \text{oHOL} \). The rules with superscript \( \text{Prop} \) correspond to the instances of the (claim) rules.
The rule \((\text{weak})^\text{Type}_M\) introduces a declaration of a fresh meta-variable in the context. The parameters and the type of the meta-variable should be of type Type that represents the domains of HOL.

The rule \((\text{start})^\text{Type}_M\) types instances of object-level meta-variables. To type \(m[t]\) we need to check that \(m\) is a meta-variable of type \(A\) with parameters of type \(\vec{B}\) declared in the context and that each of the arguments \(t_i\) is a term of the right domain \(B_i\). The second premise is used to ensure \(\Gamma\) is a valid context in the cases when the number of arguments is zero.

The rule \((\text{weak})^\text{Prop}_M\) is more complicated than \((\text{weak})^\text{Type}_M\) because we may have dependencies between the arguments and between the arguments and the type of the meta-variable. We divide the arguments into two groups — object-level parameters and proof-level parameters. The object-level parameters \(\vec{x}\) stand for terms of HOL domains \(\vec{B}\) and as domains do not depend on objects, there are no internal dependencies in the group. However, the types \(\vec{p}\) of proof-level parameters \(\vec{p}\) may depend on \(\vec{x}\). The same holds for the type \(A\) of the meta-variable.

To strengthen the intuition of the reader we will note that the declarations for the proof-level meta-variables will correspond to the goals of the incomplete derivations (see Definition 3.12).

The rule \((\text{start})^\text{Prop}_M\) types the occurrences of the proof-level meta-variables. Note that as we provide actual arguments \(t_i\) for the formal parameters \(x_i\), we need to update both the expected types of the arguments \(\vec{p},\vec{q}\) and the type \(A\) by means of substitution. The last premise of the rule is again used to check the validity of the context in case there are no arguments.

**Definition 4.18 (Instantiation).** Let \(N\) be a term and let \(m[\vec{x}:\vec{A}]\) be a meta-variable. The instantiation of \(m[\Delta]\) by \(N\) (denoted by \(\{m[\vec{x}:\vec{A}] := N\}\)) is a function from terms to terms defined inductively as follows:

\[
\begin{align*}
s\{m[\vec{x}:\vec{A}] := N\} &= s \\
x\{m[\vec{x}:\vec{A}] := N\} &= x \\
m[\vec{t}]\{m[\vec{x}:\vec{A}] := N\} &= N\{\vec{t}\{m[\vec{x}:\vec{A}] := N\}/\vec{x}\} \\
n[\vec{t}]\{m[\vec{x}:\vec{A}] := N\} &= n[\vec{t}\{m[\vec{x}:\vec{A}] := N\}] \\
(MN)\{m[\vec{x}:\vec{A}] := N\} &= M\{m[\vec{x}:\vec{A}] := N\}N\{m[\vec{x}:\vec{A}] := N\} \\
(\lambda x:A.M)\{m[\vec{x}:\vec{A}] := N\} &= \lambda x:A\{m[\vec{x}:\vec{A}] := N\}M\{m[\vec{x}:\vec{A}] := N\} \\
(\Pi x:A.M)\{m[\vec{x}:\vec{A}] := N\} &= \Pi x:A\{m[\vec{x}:\vec{A}] := N\}.M\{m[\vec{x}:\vec{A}] := N\}
\end{align*}
\]

Here are several examples of instantiations:

\[
\begin{align*}
(\lambda x:A.m[[]])\{m[] := (f y)\} &= \lambda x:A.(f y) \\
(\lambda x:A.m[x])\{m[y:A] := (f y)\} &= \lambda x:A.(f x)
\end{align*}
\]

Please note that free variables of \(N\) that are among \(\vec{x}\) may get bound after the instantiation \(\{m[\vec{x}:\vec{A}] := N\}\). In that sense it is better to regard the variables \(\vec{x}\) as bound in the instantiation. We will identify \(\alpha\)-convertible instantiations as \(\{m[y:A] := (f y)\}\) and \(\{m[x:A] := (f x)\}\).

The typing relation of \(\lambda\text{HOL}\) enjoys all the properties of \(\lambda\text{HOL}\). For completeness we list them below with updated formulations where necessary. Those listed without proofs can be proven using standard techniques.
Proposition 4.19 (Confluence). If $M \rightarrow_{\beta} P$ and $M \rightarrow_{\beta} Q$ then there is a term $N$ such that $P \rightarrow_{\beta} N$ and $Q \rightarrow_{\beta} N$.

Proposition 4.20 (Generation Lemma for $\lambda$HOL). For $\lambda$HOL, the Generation Lemma for $\lambda$HOL (Proposition 4.7) holds with the following extra clauses:

- if $\Gamma \vdash m[\bar{t}]; C$ then there is a meta-variable declaration $m[\bar{x}]:A \in \Gamma$ such that $A[\bar{t}/\bar{x}] =_{\beta} C$ and $\Gamma \vdash t_i : A_i[t_j/x_j]_{j<i}$.
- if $\Gamma_1, m[\bar{x}]:A, \Gamma_2 \vdash C$ then there is a sort $s$ such that $\Gamma_1, \bar{x} : A : s$.

Proposition 4.21 (Weakening). Let $\Gamma, \Gamma_2 \vdash M : B$ and let $x, \bar{y}, \bar{p}$ and $m$ be fresh. Then

1. If $\Gamma_1 \vdash A : s$ then $\Gamma_1, x : A, \Gamma_2 \vdash M : B$.
2. If $\Gamma_1, \bar{y} : A \vdash \text{Type}$ and $\Gamma_1 \vdash A_i : \text{Type}$ for all $i$, then $\Gamma_1, m[\bar{y}]:A, \Gamma_2 \vdash M : B$.
3. If $\Gamma_1 \vdash A_i : \text{Type}$ and $\Gamma_1, \bar{x} : A \vdash \varphi_i : \text{Prop}$ and $\Gamma_1, \bar{x} : A, \bar{p} : \varphi \vdash \varphi : \text{Prop}$ then $\Gamma_1, m[\bar{x}]:A, \bar{p} : \varphi \vdash \varphi : \text{Prop}$.

Proposition 4.22 (Correctness of types). If $\Gamma \vdash M : A$ then either $A \equiv \text{Kind}$ or there is a sort $s$ such that $\Gamma \vdash A : s$.

Proposition 4.23 (Cut Lemma).

1. If $\Gamma_1, x : A, \Gamma_2 \vdash M : B$ and $\Gamma_1 \vdash N : A$ then $\Gamma_1, \Gamma_2[N/x] \vdash M[N/x] : B[N/x]$.
2. If $\Gamma_1, m[\bar{x}]:A, \Gamma_2 \vdash M : B$ and $\Gamma_1, \bar{x} : A \vdash N : A$ then $\Gamma_1, \Gamma_2 \{m[\bar{x}]:=N\} \vdash m[\bar{x}]:=N : B[m[\bar{x}]:=N]$.

Proposition 4.24 (Subject Reduction). If $\Gamma \vdash M : A$ and $M \rightarrow_{\beta} N$ then $\Gamma \vdash N : A$.

Proposition 4.25 (Unicity of Types). If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$ then $A =_{\beta} B$.

Proposition 4.26 (Strong Normalization). Every reduction sequence of well-typed terms is finite.

Proof. Let $\Gamma \vdash M : A$. Let $m$ be a meta-variable from $\Gamma$. By $\pi\nu$ we will denote a fresh variable that does not occur in $\Gamma$. We assume that if $m$ and $n$ are different meta-variables in $\Gamma$, then $\pi\nu \neq \pi\nu$. Now we define a translation $\mid - \mid$ from the terms of $\lambda$HOL to the terms of $\lambda$HOL and its extension to contexts as follows:

$\mid s \mid = s$
$\mid x \mid = x$
$\mid m[t_1 \ldots t_n] \mid = \pi\nu \mid t_1 \ldots t_n \mid$
$\mid \lambda x : A. M \mid = \lambda x : \mid A \mid, \mid M \mid$
$\mid \Pi x : A. M \mid = \Pi x : \mid A \mid, \mid M \mid$
$\mid M \mid = \mid M \mid$

The parameters of a meta-variable of sort Type can be only of sort Type. Similarly, the parameters of a meta-variable of sort Prop can be both of sort Type and Prop.
This corresponds to the three (\(\Pi\))-rules of \(\lambda\)HOL. Therefore we will be able to form the \(\Pi\)-types \(\Pi x : A, \Lambda x : A \vdash A\) to which the meta-variable declarations are mapped. Formally, one can prove by induction on the derivation that

\[
\text{if } \Gamma \vdash \lambda_\text{HOL} M : A \text{ then } |\Gamma| \vdash \lambda_\text{HOL} |M| : |A|
\]

This, combined with the fact that if \(M \rightarrow_\beta N\) then \(|M| \rightarrow_\beta |N|\), means that there are no infinite reduction paths of well-typed terms in \(\lambda_\text{HOL}\), because if there were such paths they would be translated by \(|\_|\) to infinite paths of well-typed terms in \(\lambda_\text{HOL}\). \(\Box\)

4.2.2. Examples. In this section we will show how \(\lambda_\text{HOL}\) is used to model the process of proof-term construction (compare with Section 3.3.1).

Suppose we have two propositions \(A\) and \(B\) and inhabitants \(h_1\) and \(h_2\) of the types \(\Pi p : A, \Pi q : B, C\) and \(\Pi p : A, B\) respectively. These assumptions are encoded in the context \(\Gamma_0\) in Figure 4.7. In this context we would like to construct a term of type \(\Pi p : A, C\). At step (1) we have the initial state where the unknown inhabitant

\[
\Gamma_0 \equiv A : \text{Prop}, B : \text{Prop}, h_1 : \Pi p : A, \Pi q : B, C, h_2 : \Pi p : A, B
\]

\[
\begin{align}
(1) & \quad \Gamma_0, m_1[\,] : \Pi p : A, C \vdash m_1[\,] : \Pi p : A, C \\
(2) & \quad \Gamma_0, m_2[p : A] : C \vdash \lambda i : A, m_2[i] : \Pi p : A, C \\
(3) & \quad \Gamma_0, m_3[p : A] : A, m_2[p] : B \vdash (\lambda i : A, h_1 m_3[i] m_2[i]) : \Pi p : A, C \\
(4) & \quad \Gamma_0, m_4[p : A] : A \vdash (\lambda i : A, h_1 (h_2 m_4[i])) : \Pi p : A, C \\
(5) & \quad \Gamma_0 \vdash \lambda i : A, (h_1 (h_2 i)) : \Pi p : A, C
\end{align}
\]

**Figure 4.7.** Stepwise proof-term construction in \(\lambda_\text{HOL}\).

is represented by the meta-variable \(m_1[\,]\) of type \(\Pi p : A, C\). As we are looking for an inhabitant of a product type, we decide to refine \(m_1\) by the abstraction \(\lambda p : A, m_2[p]\), where \(m_2[p : A]\) is a fresh meta-variable of type \(C\). By weakening we can introduce \(m_2\):

\[
\Gamma_0, m_2[p : A] : C, m_1[\,] : \Pi p : A, C \vdash m_1[\,] : \Pi p : A, C
\]

and because \(\Gamma_0, m_2[p : A] : C \vdash \lambda p : A, m_2[p] : \Pi p : A, C\), by the Cut Lemma we get that (2) is a derivable judgment.

Similarly, we can weaken (2) by \(m_3[p : A] : A\) and \(m_2[p] : B\). If we then instantiate \(m_2[p : A]\) by \(h_1 m_3[p] m_2[p]\), by the Cut Lemma we get that (3) is derivable.

To get (4) we weaken by \(m_4[p : A] : A\) and instantiate \(m_3[p : A]\) by \(p\) and \(m_2[p : A]\) by \(h_2 m_4[p]\). Finally (5) is obtained by instantiating \(m_4[p : A]\) by \(p\).

Let \(U\) be a type, let \(a, b\) and \(c\) be elements of \(U\) and let \(R\) be a binary predicate on \(U\). Assume that the type \(\Pi x y : U. \Pi p_1 : (R x y), \Pi p_2 : (R y z), (R x z)\) is inhabited by \(tr\) and that \(th\) is of type \(\Pi x : U. (R x b)\). In the context \(\Gamma_0\) (see Figure 4.8)
describing these assumptions we will present a partial development of a proof-term of type \( \text{Rac} \). We start with the judgment (1) in which the unknown term is represented by the meta-variable \( m[] \). We see that if in \( tr \) we instantiate \( x \) by \( a \) and \( z \) by \( c \) we get a product type with range \( \text{Rac} \). Since we do not need (nor know how) to instantiate the other arguments in any special way, we introduce meta-variables for them - \( y_1[] \) for \( y \), \( p_1[] \) for \( p_1 \) and \( p_2[] \) for \( p_2 \). Then the term \( \langle tr a y_1[] c p_1[] p_2[] \rangle \) is of type \( \text{Rac} \) and we can use it to instantiate \( m[] \). Therefore by weakening and the Cut Lemma we can show that (2) is a derivable judgment.

In the second step of this partial development we focus on the goal \( p_1[] \) which is of type \( \text{Rac} y_1 \). We would like to solve it by applying the term \( th \) of type \( \Pi x:U.(R x b) \). If we instantiate the product type by taking \( x := a \) we get \( \text{Rab} \) while our goal is \( \text{Rac} y_1 \). However \( y_1 \) is also a meta-variable and we can instantiate it by \( b \) and then the types match. Hence, in contrast to the previous step, starting from (2) we first have to instantiate \( y_1 \) by \( b \) and then instantiate the goal \( p_1[] \) by \( th \ a \). The Cut Lemma guarantees that the judgment (3) obtained in this way is derivable.

This development is only partial (we still have the open goal \( p_2[] \) but it illustrates the kinds of activities done at each step of the proof development and the ways to ensure that they are sound by preserving derivability at each step. □

We will use refinements as above in order to express the effect of the application of tactics on a proof state (see Chapter 6).

### 4.3. Typing for HOL with binding holes: \( \text{AxHOL} \)

In this section we will present the type theory \( \text{AxHOL} \) that corresponds to the extension of HOL with binding holes of Section 3.4.

Consider diagrams (a) and (b) in Figure 4.9. Diagram (a) represents a typical incomplete term constructed backwards (i.e. from the root to the leaves). We have an incomplete term consisting of known parts and several completely unknown subterms. The situation on Diagram (b) occurs when we also allow forward steps or when we have performed \( \beta \)-reductions. Then we have unknown (sub)terms with known parts. Both situations can be captured if we use meta-variables with parameters by giving the known part of the term as an argument to the meta-variable.

| \( \Gamma_0 \equiv \) | \( \begin{array}{l}
U: \text{Type}, a, b, c: U, R: U \rightarrow U \rightarrow \text{Prop}, th : \Pi x: U.(R x b), \\
tr : \Pi y z: U. \Pi p_1 : (R x y). \Pi p_2 : (R y z). (R x z)
\end{array} \) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ( \Gamma_0, m[] : (\text{Rac}) \vdash m[] : (\text{Rac}) )</td>
<td></td>
</tr>
<tr>
<td>(2) ( \Gamma_0, y_1[] : U, p_1[] : (\text{Rac} y_1), p_2[] : (\text{Rac} y_1) \vdash (tr a y_1[] c p_1[] p_2[]) : (\text{Rac}) )</td>
<td></td>
</tr>
<tr>
<td>(3) ( \Gamma_0, p_2[] : (\text{Rac}) \vdash (tr a b c (th \ a) p_2[]) : (\text{Rac}) )</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.8.** An example of partial proof-term development in \( \text{AxHOL} \).
However if we look closer there is a catch. Let us consider the possible scoping dependencies in an incomplete term. We can have a binder in the known part and an unknown in its scope (1) and we can have a variable in the known part of an unknown term bound by a binder outside the unknown (2), but how do we account for the situation when the binding for a variable in the known part is supposed to be provided by a binder in the unknown part (3)?

We have already seen similar situation in Section 3.4 where we introduced binding holes to HOL. We will take a similar approach here by introducing hereditarily parameterized meta-variables. These are meta-variables whose parameters are parameterized themselves. The instances of these meta-variables will act as binders and introduce fresh variables in the scope of the arguments of the meta-variable. In this way we handle the occurrences of variables that are supposed to be bound, but whose binder is not yet constructed as in (3) in Figure 4.9. The resulting system will be called $\lambda b$HOL.

Below we introduce $\lambda b$HOL and its properties. In Section 4.3.2 we will present two detailed examples of stepwise construction of terms in $\lambda b$HOL and show how they relate to the situation described above.

**4.3.1. The system.** Let $\mathcal{M}$ be a set of meta-variable names and $\mathcal{V}$ a set of variables. The set of pseudo-terms $T$ is defined inductively as shown in Figure 4.10. Some notions (e.g. instantiation, see p.76) are easier to define if we consider the variables as meta-variables with no parameters. However, the distinction between variables and meta-variables will be necessary, because only variables can be abstracted using $\lambda$ and $\Pi$.

As in the logic, a meta-variable declaration is an expression like $m[\Delta]:T$ with $\Delta$ being the list of the formal parameters of $m$ and $T$ its type. Again we assume that the name $m$ uniquely determines the number and the structure of the parameters.
of the meta-variable. The parameters of the meta-variable can be parameterized themselves, hence the term hereditary parameters. In the presentation of $\lambda$hOL we limit the level of nesting of the parameters to two, but in Section 4.4 we will present a typing system where we can have arbitrary level of nesting.

Examples of meta-variables are:

\[ p[\cdot]:U, \ m[x:U,y:V]:U, \ n[m[x:U_1,y:U_2]:V_1,z:V_2]:U \]

To make a term out of a meta-variable, one needs to provide the actual arguments for its formal parameters. A formal parameter without parameters expects a single term as argument while a formal parameter that itself has parameters $\vec{x}:\vec{A}$ needs an argument defined in an extended context. This extended context must be of the same size as the list of the parameters and is written between angle brackets $\langle \cdot \rangle$ in front of the argument. For example, if $M$ and $N$ are terms and $p$, $m$ and $n$ have parameters as above, then the following are well-formed terms:

\[ p[\cdot], \ m[M,N], \ n[(x:U_1,y:U_2)M,N], \ n[(z:U,y:V)N,N] \]

As meta-variables will stand for unknown terms, we can interpret $m[M,N]$ as unknown term with known subterms $M$ and $N$ and the term $n[(x:U_1,y:U_2)M,N]$ again as an unknown term with subterms $M$ and $N$, but the construction

\[ \langle x:U_1,y:U_2 \rangle M \]

says that $x$ and $y$ may occur free in $M$ and a solution for the unknown $n$ should bind them. Thus, similar to the object terms in logic, $\langle \vec{x}:\vec{A} \rangle M$ is not a term on its own, but can only be formed as part of a meta-variable instance.

**Definition 4.27 (Free Variable Occurrence).** The set $\text{FV}(M)$ of the variables with free occurrences in the term $M$ is defined by induction on the structure of $M$ as in $\text{HOL}$ (Definition 4.1) with the following extra clauses:

\[
\text{FV}(m[(\vec{x}_1:\vec{A}_1)M_1 \ldots (\vec{x}_n:\vec{A}_n)M_n]) = \bigcup_{i=1}^{n}(\text{FV}(M_i) \setminus \vec{x}_i \cup \text{FV}(\vec{x}_i:\vec{A}_i))
\]

\[
\text{FV}(\varepsilon) = \emptyset
\]

\[
\text{FV}(\vec{y};\vec{B},x:A) = \text{FV}(\vec{y};\vec{B}) \cup \text{FV}(A) \setminus \vec{y}
\]

An occurrence of a variable that is not free is bound. Note the binding in the arguments of the meta-variables. Please recall that with respect to the bound variables we apply Convention 4.2.
**Definition 4.28.** The result of substituting \( t \) for the free occurrences of \( x \) in \( M \) (denoted \( M[t/x] \)) is defined in \( \text{bHOL} \) (Definition 4.4) with an extra clause stating that \( (m[(\bar{x}_1:A_1)M_1 \ldots (\bar{x}_n:A_n)M_n])t/x) \) is defined as

\[
m[(\bar{x}_1:A_1)t/x]M_1[t/x] \ldots (\bar{x}_n:A_n)[t/x]M_n[t/x]
\]

Note that in contrast to \( \text{bHOL} \) (see Definition 3.27), here we need to propagate the substitution also in the local contexts of each argument. In the case of \( \text{bHOL} \) we did not need to do that because there only object variables could occur in the context and their domains are in general not affected by substitutions. In \( \lambda\text{bHOL} \) we can have also propositional parameters whose types may contain occurrences of the variable for which we substitute.

Substitution replaces free variable occurrences by a given term. As in \( \lambda\text{HOL} \) we have an analogous operation for meta-variables called instantiation:

**Definition 4.29 (Instantiation).** Let \( m[\Delta] : A \) be a (meta-)variable. The instantiation of \( m \) by an arbitrary term \( N \) in the term \( M \) (notation \( M \{ m[\Delta] := N \} \)) is defined by induction on \( M \) as in \( \lambda\text{HOL} \) (see Definition 4.18) with the following clauses for meta-variable instances:

\[
\begin{align*}
(m[(\Theta_1)u_1 \ldots (\Theta_n)u_n])^* &\ x_1(\Theta_1) := u_1^* \ldots x_n(\Theta_n) := u_n^* \\
n[(\Theta_1)u_1 \ldots (\Theta_k)u_k]^* &\ x_1(\Theta_1) := u_1^* \ldots x_k(\Theta_k) := u_k^*
\end{align*}
\]

where in the first clause \( x_1 \ldots x_n \) are the names of the arguments in \( \Delta \) and \( M^* \) abbreviates \( M \{ m[\Delta] := N \} \).

The difference in this definition in comparison to the instantiation in \( \lambda\text{HOL} \) is the need to recursively instantiate the parameters of the meta-variable. The difference with respect to \( \text{bHOL} \) is that the instantiation has to be propagated also in the contexts \( \Theta_i \).

The instantiation as given by Definition 4.29 is introduced also for variables by considering them as meta-variables with no parameters. Again (see Proposition 3.35) we can show that on variables instantiations and substitution behave in the same way.

**Proposition 4.30.** Let \( x \) be a variable and \( N \) be a term. Then for every \( M \)

\[
M[N/x] \equiv M \{ x[\cdot] := N \}
\]

The notions of \( \beta \)-reduction and \( \beta \)-equality are defined as the usual closures of \( (\lambda x:A. M)N \rightarrow^\beta M[N/x] \). They naturally extend to contexts (and thus also to variable and parameter lists) as shown in Figure 4.11 for \( =^\beta \).

![Figure 4.11. \( \beta \)-equality on contexts.](image)

**Lemma 4.31.** If \( \Delta =^\beta \Delta' \) then

\[
M \{ m[\Delta] := N \} \equiv M \{ m[\Delta'] := N \}
\]
4.3. Typing for HOL with Binding Holes: λbHOL

PROOF. Induction on $M$. □

For a context $\Gamma$ define the depth $|\Gamma|$ of $\Gamma$ as follows:

$$
|\varepsilon| = 0
$$
$$
|\Gamma, m[\Delta]: A| = \max\{1 + |\Delta|, |\Gamma|\}
$$
$$
|\Gamma, x:A| = \max\{1, |\Gamma|\}
$$

LEMMA 4.32 (Substitution Lemma).

1. Let $m[\Delta]$ be a meta-variable that does not occur in $Q$ and $\Sigma$. Then $M\{m[\Delta] := P\}{n[\Sigma] := Q} = M\{n[\Sigma] := Q\} = P\{n[\Sigma] := Q\}$

2. Let $x$ be a variable that does not occur in $Q$. Then

$$
M[P/x][Q/y] \equiv M[Q/y][M[Q/y]/x]
$$

PROOF. By Proposition 4.30 (2) follows from (1). To prove (1) we proceed by induction on the sum of the depths of $\Delta$ and $\Sigma$ and embedded induction on the structure of $M$. The interesting case are the meta-variable instances. The crucial observation is that if $x[\Theta]$ is a meta-variable, and $x[(\Theta_1)u_1 \ldots (\Theta_k)u_k]$ is an instance of $x[\Theta]$ then for each $i$ we have $|\Theta_i| + 1 \leq |\Theta|$. This allows us to apply the induction hypothesis to the instantiations for the parameters in the cases when $M$ is an instance of $m$ or $n$. □

LEMMA 4.33. Let $M \rightarrow_{\beta} M'$ and $N \rightarrow_{\beta} N'$. Then

$$
M\{m[\Delta] := N\} \rightarrow_{\beta} M'\{m[\Delta] := N'\}
$$

PROOF. Induction on the depth of $|\Delta|$ and a nested induction on $M$. Let us denote $\{m[\Delta] := N\}$ by $^*$ and $\{m[\Delta] := N'\}$ by $^\dagger$.

Consider the case when $M$ is $((\lambda x:A.R)Q)$ and $M'$ is $R[Q/x]$. Then $M^*$ is $(\lambda x:A^*.R^*)Q^*$ and by induction hypothesis it reduces to $(\lambda x:A^\dagger.R^\dagger)Q^{\dagger}$. After one $\beta$-reduction this gives us $R^\dagger[Q^{\dagger}/x]$ that by Substitution Lemma is $(R[Q/x])^\dagger$. The last is of course identical to $M'^\dagger$.

The other non-trivial case is when $M$ is $m[(\Theta_1)t_1 \ldots (\Theta_n)t_n]$ and $M'$ is the term $m[(\Theta'_1)t'_1 \ldots (\Theta'_n)t'_n]$ with $\Theta_i \rightarrow_{\beta} \Theta'_i$ and $t_i \rightarrow_{\beta} t'_i$. Then we have $M^* = N[x_i[\Theta'_i] := t'_i]$. By induction $t'_i \rightarrow_{\beta} t'^\dagger_i$ and by assumption $N \rightarrow_{\beta} N'$. By induction hypothesis and Lemma 4.31 we have $N[x_i[\Theta'_i] := t'^\dagger_i] \equiv N[x_i[\Theta'_i] := t'^\dagger_i]$. Now $|\Theta'_i| < |\Delta|$ and by induction $N[x_i[\Theta'_i] := t'^\dagger_i] \rightarrow_{\beta} N'[x_i[\Theta'_i] := t'^\dagger_i]$ that by Substitution Lemma is $(N'[x_i[\Theta'_i] := t'^\dagger_i])^\dagger$, i.e. $M'^\dagger$. □

The typing rules of λbHOL are the standard rules of λHOL (Figure 4.3) plus the two rules shown in Figure 4.12. The rule (weak)$_{\lambda\delta}$ is used to introduce meta-variables in the context. We are allowed to introduce meta-variables if their type is well-formed in the current context extended by the parameter list of the meta-variable. By putting restriction $s \in \{\text{Prop, Type}\}$ we limit the set of sorts over which meta-variables are allowed. There is no essential need for this restriction other than the fact that in bHOL we do not consider unknown domains (see Section 3.3.1.6) and thus we should not allow meta-variables of sort Kind. We could enforce certain dependencies between the sorts of the arguments and the
sort $s$ of the meta-variable (so that for example meta-variables for object terms depend only on object-level parameters), but here we will not do that to keep the setting as general as possible.

The rule $(\text{start})_M$ types instances of meta-variables that have been declared in the context. Each of the arguments $t_i$ has to be typable in the current context extended by the parameter list $\Theta_i$. As a parameter might occur in the type of the parameters declared after it, when giving an argument for a formal parameter we need to instantiate the parameter in the types of the following arguments. This is the reason for having the instantiations $\delta_k$ (compare with the rule (start)$_{\text{prop}}$ in Figure 4.6).

Examples of typing in $\lambda$HOL are given in Section 4.3.2.

**Example 4.34.** If $h|y:U, p|x:U|P x y, A[z:U]:\text{Prop}:A[y] \in \Gamma$ and one wants to derive $\Gamma \vdash h[M, (x:U)N, (z:U)\varphi]: \varphi[M/z]$ the premises of the $(\text{start})_M$ rule amount to showing that

\[
\begin{align*}
\Gamma, y:U, p|x:U|P x y, A[z:U]:\text{Prop} & \vdash A[y]:\text{Prop} \\
\Gamma & \vdash M : U \\
\Gamma, z:U & \vdash \varphi : \text{Prop}
\end{align*}
\]

Note that the meta-variables in the typing system are more ‘complicated’ than the ones in the logic as now we can have meta-variables like $m[x:U, p : P x]: Q x$ where the type of the meta-variable (or a parameter) depends on (another) parameter. We can also have meta-variables of type $\text{Prop}$ that represent unknown formulas and if we had allowed $s$ in the rule (\text{weak})$_M$ to be $\text{Kind}$, we could have had meta-variables representing unknown domains.

**Meta-theoretical Properties.** $\lambda$HOL enjoys the same meta-theoretical properties as $\lambda$HOL and $\lambda$HOL. They are stated and proven below.

**Property 4.35 (Confluence).** If $M \rightarrow_\beta P$ and $M \rightarrow_\beta Q$ then there is $N$ such that $P \rightarrow_\beta N$ and $Q \rightarrow_\beta N$.

**Proof.** We can translate the $\lambda$HOL-terms to $\lambda$HOL-terms as follows:

\[
\begin{align*}
\varphi(m[(\bar{x}_1; \bar{A}_1)M_1, \ldots, (\bar{x}_n; \bar{A}_n)M_n]) & = \overline{m} [\varphi(\bar{A}_1), \varphi(M_1), \ldots, \varphi(\bar{A}_n), \varphi(M_n)] \\
\varphi(s) & = s \\
\varphi(x) & = x \\
\varphi(MN) & = \varphi(M) \varphi(N) \\
\varphi(\lambda x: A.M) & = \lambda x: \varphi(A). \varphi(M) \\
\varphi(\Pi x: A.B) & = \lambda x: \varphi(A). \varphi(B)
\end{align*}
\]

**Figure 4.12.** Typing rules for $\lambda$HOL.
where for each meta-variable \( m \) we have introduced a unique fresh variable \( \overline{m} \). Recall that according to the variable convention the variables \( \overline{x}_i \) in the first clause above are different from all other bound and free variables. Therefore they become free in the translated term.

The map \( \phi \) obviously does not preserve the typability of terms, but it is still useful, because one can easily show that \( \phi(M) \rightarrow_\beta P \) if and only if there is a term \( N \) such that \( P \equiv \phi(N) \) and \( M \rightarrow_\beta N \). Hence, the confluence of \( \lambda \text{HOL} \) implies confluence of \( \lambda \text{HOL} \).

**Proposition 4.36** (Generation Lemma for \( \lambda \text{HOL} \)). For \( \lambda \text{HOL} \), the Generation Lemma for \( \lambda \text{HOL} \) (Proposition 4.7) holds with the following extra clauses:

- if \( \Gamma \vdash m[(\overline{x}_1; \overline{C}_1)t_1 \ldots (\overline{x}_n; \overline{C}_n)t_n]:C \) then there is a meta-variable declaration \( m[\Delta]:A \in \Gamma \) such that
  - \( \Delta \equiv p_1[\overline{x}_1; \overline{A}_1];B_1 \ldots p_n[\overline{x}_n; \overline{A}_n];B_n \)
  - \( \delta_0 := id \), \( \delta_{k+1} := \delta_k \circ \{ p_{k+1}[\overline{x}_{k+1}; \overline{C}_{k+1}] := t_{k+1} \} \)
  - \( \Gamma, \overline{x}_k; \overline{C}_k \vdash t_k : D_k \)
  - \( \delta_{k-1}(A_i) = \beta C_i \), \( \delta_{k-1}(B_i) = \beta D_i \) and \( \delta_n(A) = \beta C \)

- if \( \Gamma_1, m[\Delta]:A, \Gamma_2 \vdash M : C \) then there is a sort \( s \) such that \( \Gamma_1, \Delta \vdash A : s \).

**Proof.** Induction on the derivation. \( \square \)

**Lemma 4.37** (Weakening).

1. Let \( \Gamma, \Gamma' \vdash M : B \), \( \Gamma \vdash A : s \) and \( x \) be fresh. Then \( \Gamma, x:A, \Gamma' \vdash M : B \).
2. Let \( \Gamma, \Gamma' \vdash M : B \), \( \Gamma, \Delta \vdash A : s \) and \( m \) be fresh. Then \( \Gamma, m[\Delta]:A, \Gamma' \vdash M : B \).

**Proof.** Induction on the derivation. \( \square \)

**Lemma 4.38** (Cut Lemma).

1. If \( \Gamma_1, m[\Delta]:A, \Gamma_2 \vdash M : B \) and \( \Gamma_1, \Delta \vdash N : A \) then
   \( \Gamma_1, \Gamma_2 \vdash m[\Delta] := N \vdash M \{ m[\Delta] := N \} : B \{ m[\Delta] := N \} \)
2. If \( \Gamma_1, x:A, \Gamma_2 \vdash M : B \) and \( \Gamma_1 \vdash N : A \) then
   \( \Gamma_1, \Gamma_2 \vdash N[x] : M[N/x] : B[N/x] \)

Let us by \( \Gamma \vdash \text{Ok} \) denote the statement

'There are \( M \) and \( B \) such that \( \Gamma \vdash M : B \)

**Lemma 4.39.** If \( \Gamma, \Delta, \Theta \vdash M : B \), \( \Delta =_\beta \Delta' \) and \( \Gamma, \Delta' \vdash \text{Ok} \) then

\( \Gamma, \Delta', \Theta \vdash M : B \)

**Proof.** Induction on the derivation of \( \Gamma, \Delta, \Theta \vdash M : B \). Here we only consider the four rules of \( \lambda \text{HOL} \) that extend the context of the judgment. The cases for the other rules are straightforward applications of the induction hypothesis.

**Start:** We have several cases shown below depending on whether the variable being introduced is in \( \Gamma \), \( \Delta \) or \( \Theta \).

\[
\begin{align*}
(a) & \quad \Gamma \vdash A : s \\
& \quad \Gamma, x:A \vdash x:A \\
(b) & \quad \Gamma, \Delta \vdash A : s \\
& \quad \Gamma, \Delta, x:A \vdash x:A \\
(c) & \quad \Gamma, \Delta, \Theta \vdash A : s \\
& \quad \Gamma, \Delta, \Theta, x:A \vdash x:A
\end{align*}
\]
In case (a) $\Delta$ is empty and there is nothing to prove. In case (b) we have by assumption $\Delta, x:A \equiv_{\beta} \Delta', x:A'$ and $\Gamma, \Delta', x:A' \vdash \text{Ok}$. From the last by Generation we get $\Gamma, \Delta' \vdash A : s'$ for some $s'$. Since $A =_{\beta} A'$ and from the induction hypothesis we get by conversion $\Gamma, \Delta' \vdash x:A$.

In case (c) we straightforwardly apply the induction hypothesis.

**(weak)**: Again we consider three cases:

(a) $\Gamma \vdash M:B \quad \Gamma \vdash A:s \quad \Gamma, x:A \vdash M:B \quad \Gamma, \Delta, x:A \vdash M:B$

(b) $\Gamma, \Delta \vdash M:B \quad \Gamma, \Delta \vdash A:s \quad \Gamma, \Delta, x:A \vdash M:B$

(c) $\Gamma, \Delta, \Theta \vdash A:s \quad \Gamma, \Delta, \Theta \vdash M:B \quad \Gamma, \Delta, \Theta \vdash M:B$

Cases (a) and (c) are trivial. In case (b) we have by assumption $\Delta, x:A =_{\beta} \Delta', x:A'$ and $\Gamma, \Delta', x:A' \vdash \text{Ok}$. Therefore by Generation $\Gamma, \Delta' \vdash A' : s'$ for some $s'$. By induction $\Gamma, \Delta' \vdash M : B$ and hence applying the (weak) rule we get $\Gamma, \Delta', x:A' \vdash M : B$.

**Start** $\Lambda$: Straightforward.

**Weak** $\Lambda$: The three cases are:

(a) $\Gamma \vdash M:B \quad \Gamma, \Sigma \vdash A:s \quad \Gamma, m[\Sigma][A] \vdash M:B \quad \Gamma, \Delta, \Sigma \vdash A:s$

(b) $\Gamma, \Delta \vdash M:B \quad \Gamma, \Delta, \Sigma \vdash A:s \quad \Gamma, \Delta, \Theta, m[\Sigma][A] \vdash M:B$

(c) $\Gamma, \Delta, \Theta \vdash A:s \quad \Gamma, \Delta, \Theta \vdash M:B \quad \Gamma, \Delta, \Theta, m[\Sigma][A] \vdash M:B$

As above, in case (a) there is nothing to prove and (c) is a simple application of the induction hypothesis.

In case (b) we have by assumption $\Delta, m[\Sigma][A] =_{\beta} \Delta', m[\Sigma'][A']$ and $\Gamma, \Delta', m[\Sigma'][A'] \vdash \text{Ok}$. By Generation $\Gamma, \Delta', \Sigma' \vdash A' : s'$ and by (weak) we are done.

\[\square\]

**Proposition 4.40 (Correctness of types).** If $\Gamma \vdash M : A$ then either $A \equiv \text{Kind or there is a sort } s$ such that $\Gamma \vdash A : s$.

**Proof.** Induction on the derivation of $\Gamma \vdash M : A$. The cases for the rules (ax), (weak), (weak)$_\Lambda$, ($\lambda$) and (conv) are straightforward. Below we consider the rest:

**Start** $\Lambda$: Apply (weak) to the assumption $\Gamma \vdash A : s$ to get $\Gamma, x:A \vdash A : s$.

**Start** $\Lambda$: Use Lemma 4.39 on $\Gamma, \Theta_i \vdash t_i : \delta_i A_i$ to get $\Gamma, \delta_i A_i \vdash t_i : \delta_i A_i$ and then apply Cut Lemma $n$ times to $\Gamma, x_1[\Delta_1][A_1] \ldots x_n[\Delta_n][A_n] \vdash A : s$ to get $\Gamma \vdash \delta_n A : s$.

(II): Let $s_2$ be the type of $\Pi x : A. B$. Then from (ax) by weakening we get $\Gamma \vdash s_2 : s_3$ (because $s_2$ is either Prop or Type).

**App**: By induction hypothesis $\Gamma \vdash \Pi x : A. B : s$. By Generation $\Gamma, x:A \vdash B : s$. However $\Gamma \vdash N : A$, thus by Cut Lemma $\Gamma \vdash B[N/x] : s$.

\[\square\]

**Property 4.41 (Subject Reduction).** Let $\Gamma \vdash M:A$. Then

1. If $M \equiv_{\beta} N$ then $\Gamma \vdash N:A$
(2) If $\Gamma \rightarrow_\beta \Delta$ then $\Delta \vdash M : A$

**Proof.** We prove (1) and (2) simultaneously by induction on the derivation of $\Gamma \vdash M : A$ using Lemma 4.39. Here we will present only the interesting cases of the rules (start)$_M$, (app), (weak) and (weak)$_M$.

**(start)$_M$:**

$$
\Gamma, x_1[\Delta_1] : A_1, \ldots, x_n[\Delta_n] : A_n \vdash s; \quad \Gamma, \Theta_i \vdash t_i : \delta_i(A_i)
$$

$$
\Gamma \vdash m[(\Theta_1)t_1, \ldots, (\Theta_n)t_n] : \delta_n(A)
$$

where

$$m[x_1[\Delta_1] : A_1, \ldots, x_n[\Delta_n] : A_n] : A \in \Gamma$$

$$\delta_0 = \text{id}$$

$$\delta_{k+1} = \delta_k \circ \{ x_{k+1}[\Theta_{k+1}] := t_{k+1} \}$$

$$\Theta_i = \beta \delta_i - 1(\Delta_i)$$

(1) (1a) Suppose that the reduction was in $\Theta_k$: $\Theta_k \rightarrow_\beta \Theta'_k$. Define $\Theta'_i := \Theta_i$ for $i \neq k$. Then by induction hypothesis (2) we have $\Gamma, \Theta'_i \vdash t_i : \delta_i - 1 A_i$. By Lemma 4.31 we have $\delta_i - 1 A_i = \delta'_i - 1 A_i$ and $\delta_n A = \delta'_n A$. By applying the rule (start)$_M$ we are done.

(1b) Suppose that $t_k \rightarrow_\beta t'_k$. For $i \neq k$ define $t'_i := t_i$. The instantiations $\delta'_i$ are defined as above using $t'_i$ instead of $t_i$. By induction on $i$ we will show that

$$\bullet \quad \Gamma, x_i[\delta'_i - 1 \Delta_i] : \delta'_i - 1 A_i, x_n[\delta'_n - 1 \Delta_n] : \delta'_n - 1 A_n \vdash \delta'_i - 1 A : s$$

For $i = 1$ we have $\delta'_0 = \text{id}$ and the statement is true by assumption. By assumption (or induction hypothesis if $i = k$) we have $\Gamma, \Theta_i \vdash t'_i : \delta'_i - 1 A_i$. Since $\Theta_i = \beta \delta_i - 1 \Delta_i$ and $\delta_i - 1 \Delta_i = \delta'_i - 1 \Delta_i$ and because $\Gamma, \delta'_i - 1 \Delta_i \vdash \text{Ok}$ (by Generation on (1)), we can apply Lemma 4.39 to get $\Gamma, \delta'_i - 1 \Delta_i \vdash t'_i : \delta'_i - 1 A_i$. By Generation on (1) also follows that $\delta'_i - 1 A_i$ is a well-formed type, thus we can apply the conversion rule to get $\Gamma, \delta'_i - 1 \Delta_i \vdash t'_i : \delta'_i - 1 A_i$. To this judgment and (1) we can apply the Cut Lemma and hence get (1) for $i + 1$.

As a consequence of (1) we now know that $\Gamma, \delta'_i - 1 \Delta_i : \delta'_i - 1 A_i : s_i$ for every $i$. As $\delta'_i - 1 \Delta_i = \beta \Theta_i$ and $\Gamma, \Theta_i \vdash \text{Ok}$, by Lemma 4.39 we have that $\Gamma, \Theta_i \vdash \delta'_i - 1 A_i : s_i$. Hence from the induction hypothesis $\Gamma, \Theta_i \vdash t'_i : \delta_i - 1 A_i$, by the conversion rule we get $\Gamma, \Theta_i \vdash t'_i : \delta'_i - 1 A_i$. Now we can apply (weak)$_M$ and obtain

$$\Gamma \vdash m[(\Theta_1)t'_1, \ldots, (\Theta_n)t'_n] : \delta'_n A_n$$

As $\Gamma \vdash \delta_n A_n$ and $\delta_n A_n = \beta \delta'_n A_n$, with one last application of the conversion rule we have

$$\Gamma \vdash m[(\Theta_1)t'_1, \ldots, (\Theta_n)t'_n] : \delta_n A_n$$

(2) Trivial.

**(app):**

$$
\Gamma \vdash P : P x : B. C \quad \Gamma \vdash Q : B
\overline{\Gamma \vdash P Q : C[Q/x]}
$$
(1) Consider the case when $P$ is $\lambda x:B'.R$ and $N$ is $R[Q/x]$. By Generation we have $X$, $s_1$ and $s_2$ such that

\[
\begin{align*}
\Gamma, x:B' &\vdash R : X \\
\Gamma &\vdash B' : s_1 \\
\Gamma, x:B' &\vdash X : s_2 \\
\Pi x : B'.X &\equiv_{\beta} \Pi x : B.C
\end{align*}
\]

We know that $\Gamma \vdash Q : B$, but by the conversion rule also $\Gamma \vdash Q : B'$ holds as $B'$ is a well-formed type. Similarly by conversion we have $\Gamma, x:B' \vdash R : C$. Then by Cut Lemma we therefore get $\Gamma \vdash R[Q/x] : C[Q/x]$.

(2) Trivial.

(weak):

\[
\begin{align*}
\Gamma &\vdash M : B \\
\Gamma, x:A &\vdash M : B
\end{align*}
\]

(1) Trivial.

(2) Consider the case when $A \rightarrow_{\beta} A'$. Then by the induction hypothesis for (1) we have $\Gamma \vdash A' : s$. Hence by (weak) we get $\Gamma, x:A' \vdash M : B$.

(weak)$_M$:

\[
\begin{align*}
\Gamma &\vdash M : B \\
\Gamma, \Delta &\vdash A : s
\end{align*}
\]

(1) Trivial.

(2) The case when $A \rightarrow_{\beta} A'$ is similar to (weak) above. If $\Delta \rightarrow_{\beta} \Delta'$ then we need to apply the induction hypothesis for (2) to get $\Gamma, \Delta' \vdash A : s$ to which we apply the rule (weak)$_M$.

\[
\square
\]

**Proposition 4.42 (Uniqueness of Types).** If $\Gamma \vdash M : B$ and $\Gamma \vdash M : C$ then $B =_{\beta} C$.

**Proof.** Induction on the structure of $M$ using Generation Lemma. We treat only the case when $M$ is a meta-variable instance. Suppose

\[
\begin{align*}
\Gamma &\vdash m[\Theta_1 t_1 \ldots \Theta_n t_n] : B \\
\Gamma &\vdash m[\Theta_1 t_1 \ldots \Theta_n t_n] : C
\end{align*}
\]

By Generation Lemma we have a type $A$ such that $B =_{\delta_n} \delta_n A =_{\beta} C$.

\[
\square
\]

**Property 4.43 (Strong Normalization).** Let $\Gamma \vdash M : A$. Then every reduction sequence starting with $M$ is finite.

**Proof.** We define by induction a reduction- and typing- preserving map $\mid - \mid$ from $\lambda\beta$HOL into the Extended Calculus of Constructions ECC [Luo90] which is known to be strongly normalizing. Then an assumption that $\lambda\beta$HOL has an infinite reduction path induces infinite reduction path in ECC through the map.

\[
\begin{align*}
|x[(\bar{x}_1 : \bar{A}_1) M_1 \ldots (\bar{x}_n : \bar{A}_n) M_n]| &= x(\lambda \bar{x}_1 : [\bar{A}_1]_1 | M_1) \ldots (\lambda \bar{x}_n : [\bar{A}_n] | M_n) \\
|x : A . M| &= \lambda x : A . M | \text{Prop} | = \text{Prop} \\
|\Pi x : A . M| &= \Pi x : A . M | \text{Type} | = \text{Type}_0 \\
|P Q| &= |P| |Q| | \text{Kind} | = \text{Type}_1
\end{align*}
\]
4.3. Typing for HOL with Binding Holes: aHOL

For contexts, the map is defined inductively with \( |\varepsilon| = \varepsilon \) and

\[ |\Gamma, m[p_1[x_1:]A_1]; \ldots; p_n[x_n:]A_n]; A| \]

defined as \( |\Gamma|, x: (\Pi \exists !: A_1 \cdot \ldots \cdot \Pi \exists !: A_n \cdot |A|) \). Then we have

1. If \( \Gamma \vdash_{\text{aHOL}} M:A \) then \( |\Gamma| \vdash_{\text{ECC}} |M|:|A| \);
2. If \( M \xrightarrow{\beta} N \) then \( |M| \xrightarrow{\beta} |N| \).

Proof of (1): By induction on the derivation of \( \Gamma \vdash_{\text{aHOL}} M:A \). We consider the cases for the meta-variable rules:

- If the last rule is (weak)_{M}

\[ \Gamma \vdash M: B \quad \Gamma, p_1[x_1:]A_1]; \ldots; p_n[x_n:]A_n]; B \vdash A:s \]
\[ \Gamma, m[p_1[x_1:]A_1]; \ldots; p_n[x_n:]A_n]; A \vdash M:B \]

then by induction hypothesis we have

\[ |\Gamma| \vdash_{\text{ECC}} |M|: |B| \]
\[ |\Gamma|, p_1[\Pi \exists !: A_1 \cdot |B_1|]; \ldots; p_n[\Pi \exists !: A_n \cdot |B_n|] \vdash_{\text{ECC}} |A|: |s| \]

By Generation Lemma for ECC we have that \( \Pi \exists !: A_n \cdot |B_n| \) is a well-formed type of sort either Prop or Type_i for some \( i \). As \( s \) is either Prop or Type_0, we can form the type \( \Pi \exists !: A_n \cdot |B_n| \cdot A \) and it will be of sort Prop or Type_i. Iterating this process we can show that there is a sort \( s \) in ECC such that

\[ \Gamma \vdash_{\text{ECC}} (\Pi \exists !: (\Pi \exists !: A_1 \cdot |B_1|) \cdot \ldots \cdot \Pi \exists !: (\Pi \exists !: A_n \cdot |B_n|)) \cdot A : s \]

and therefore by weakening we get

\[ \Gamma, m:(\Pi \exists !: (\Pi \exists !: A_1 \cdot |B_1|) \cdot \ldots \cdot \Pi \exists !: (\Pi \exists !: A_n \cdot |B_n|)) \cdot A \vdash |M|: |B| \]

- If the last rule is (start)_{M}

\[ \Gamma, x_1[\Delta_1]: A_1 \ldots x_n[\Delta_n]: A_n \vdash A:s \quad \Gamma, \Theta_i \vdash t_i : \delta_i^{-1}(A_i) \]
\[ \Gamma \vdash m[(\Theta_1) t_1, \ldots, (\Theta_n) t_n] : \delta_n(A) \]

where

\[ m[x_1[\Delta_1]: A_1 \ldots x_n[\Delta_n]: A_n]: A \in \Gamma \]
\[ \delta_0 = \text{id} \]
\[ \delta_{k+1} = \delta_k \circ \{x_{k+1}[\Theta_{k+1}] := t_{k+1}\} \]
\[ \Theta_i = \beta \delta_i^{-1}(\Delta_i) \]

then by induction hypothesis \( |\Gamma|, |\Theta_i| \vdash_{\text{ECC}} |t_i| : |\delta_i^{-1}(A_i)| \). Therefore \( |\Gamma| \vdash_{\text{ECC}} \lambda(\Theta_1)[\Theta_1 \cdot |t_1|] : \Pi(\Theta_1)[\delta_1^{-1}(A_1)] \). On the other hand

\[ m : (\Pi x_1:(\Pi \Delta_1 \cdot |A_1|) \cdot \ldots \cdot \Pi x_n:(\Pi \Delta_n \cdot |A_n|) \cdot |A|) \in |\Gamma| \]

Therefore using the conversion and application rules we can get

\[ |\Gamma| \vdash_{\text{ECC}} m(\lambda(\Theta_1)[\Theta_1 \cdot |t_1|] \cdot \ldots \cdot (\lambda(\Theta_n)[\Theta_n])) : |\delta_n(A)| \]

Proof of (2): Induction on the structure of \( M \).
4.3.2. Examples. In this section we will give some examples of typing in \(\lambda\text{bHOL}\).

Consider the development in Figure 4.13. We have a three-step forward construction of a term of type \(A \rightarrow C\) given inhabitants of types \(A \rightarrow B \rightarrow C\) and \(A \rightarrow B\) (compare with Example 3.20). In the second and the third judgments we use hereditarily meta-variables in order to introduce a bound variable whose precise place of binding is not yet fixed. The binding effect is better seen in the tree representations of the terms in Figure 4.14. With black we have marked the

\[
\Gamma_0 \equiv A:\text{Prop}, B:\text{Prop}, C:\text{Prop}, p:A \rightarrow B, q:A \rightarrow B \rightarrow C
\]

\[
\Gamma_0, h_1[x:A \rightarrow B \rightarrow C, y:A \rightarrow B] : A \rightarrow C \vdash h_1[q, p] : A \rightarrow C
\]

\[
\Gamma_0, h_2[x[z:A] : A \rightarrow B \rightarrow C, y[z:A] : A \rightarrow B] : A \rightarrow C \vdash h_2[(u:A)q, (u:A)p] : A \rightarrow C
\]

\[
\Gamma_0, h_3[x[z:A] : B \rightarrow C, y[z:A] : B] : A \rightarrow C \vdash h_3[(u:A)(q, u), (u:A)(p, u)] : A \rightarrow C
\]

\[
\Gamma_0 \vdash \lambda u : A.((q, u)(p, u)) : A \rightarrow C
\]

**Figure 4.13.** A forward construction of a term of type \(A \rightarrow C\) assuming inhabitants of type \(A \rightarrow B \rightarrow C\) and \(A \rightarrow B\).
constructed part of the term and with gray the parts of the intended complete
term that are still missing. The top row shows the \(\lambda\)bHOL-terms and the bottom
row shows the parts of the intended complete term that are constructed at the
particular stage. Note that the second term on the bottom line only differs from
the first in the declaration \(u : A\). The binding meta-variable allows us to introduce
the binding without actually creating a \(\lambda\)-binder.

Starting with the first judgment in Figure 4.13 and checking that it is derivable,
we can obtain the other ones by consecutive applications of the Weakening Lemma
and/or the Cut Lemma. For example, to obtain the last judgment from the third,
we need to instantiate \(h_3\) in the following way:

\[
\frac{}{h_3[x;z:A] : B \rightarrow C, y[z:A] : B} \quad := \quad \lambda u : A. (x[u] y[u])
\]

One can easily check that \(\lambda u : A. (x[u] y[u])\) is of type \(A \rightarrow C\) and it satisfies the
premises of the Cut Lemma.

The ability to represent developments as sequences of derivable judgments
obtained from each other by weakening and instantiation forms a basis for a sound
model of the process of interactive proof-term development using \(\lambda\)bHOL.

Another example of stepwise development in \(\lambda\)bHOL is presented in Fig-
ure 4.15 (compare with Example 3.21). The tree representations of the proof-terms
are shown in Figure 4.16. We can see again how the binding meta-variable allows

\[
\begin{array}{c}
\Gamma \equiv \forall U: \text{Type, } A: \text{Prop, } P : \Pi x : U. \text{Prop, } a, b, \Pi z : U. \Pi s : A. P z, c : \Pi s : (\Pi y : U. P y). B
\\
\hline
\Gamma, h_1[m : A, n : (\Pi z : U. \Pi s : A. P z), p : (\Pi y : U. P y) \rightarrow B] : B \vdash h_1[a, b, c] : B
\\
\Gamma, h_2[p[x : U] : P x, q : (\Pi x : U. P x) \rightarrow B] : B \vdash h_2[(x : U) (b x a), c] : B
\\
\Gamma \vdash (c (\lambda x : U. (b x a))) : B
\end{array}
\]

\text{Figure 4.15. Another example of a term construction from the}
leaves to the root.

us to introduce a binding for a variable before the real binder is constructed. This
allows us to use the bound variable in the known subterms and maintain sound-
ness of the development process. As before, each judgment is obtained from the
previous one by weakening (to introduce any new meta-variables) and/or the Cut
Lemma (to instantiate a meta-variable).

### 4.4. Pure Type Systems with Hereditarily Parameterized Variables,
Constants and Definitions

In this section we will present an extension of the Pure Type Systems (PTSs)
with parameterized constants and definitions (C\(^\Omega\) PTS) presented in [BKN96,
BKLN02, KLN01]. Our extension introduces hereditary parametrization to the
constants and definitions as well as to the variables and the accompanying \(\lambda\) and
\(\Pi\)-abstractions.
Pure Type Systems (PTSs) were introduced by Berardi [Ber89] and Terlouw [Ter89] as a generalization of the systems of Barendregt’s \( \lambda \)-cube (see [Bar92]). We assume that the reader is acquainted with the background facts about PTSs (see for example [Bar92, Gen93]). Definitions were first introduced in the setting of PTSs by Poll and Severi [SP94]. Parameterized definitions were discussed later in work by Bloo, Kamareddine, Laan and Nederpelt [BKN96, BKLN02, KLN01]. As related to the work in this section we can mention also the systems of the \( \lambda \)\(|\) cube by Bognar [Bog02] (see also Section 2.2.5) where abstractions may be done over parameterized variables. We will present an extension of PTSs with parameterized constants and definitions as in [BKLN02], parameterized abstractions as in the \( \lambda \)\(|\)\)-cube and unlimited hereditary parametrization as in AbHOL (where we have only two levels of parametrization).

We extend the standard definition of a PTS by adding parametrization to the \( \lambda \)- and II- abstractions. A parameterized \( \lambda \)-abstraction \( \lambda m[\Delta] : A.M \) represents a term that has abstracted out the meta-variable \( m[\Delta] \) that potentially occurs in \( M \). Such a term would have a parameterized II-abstraction as a type: \( \Pi m[\Delta] : A.B \). As the use of \( \lambda \) suggests, we can apply parameterized \( \lambda \)-abstractions to arguments and that would act as an explicit notation for the instantiation operation. Meta-variables however have parameters that can be used in the term that instantiates them. Therefore, we need to introduce the parameters of a meta-variable into the argument of an application: \( (\lambda m[x:A] : A.m[z]) . (x:A)x \). This term represents explicitly the instantiation of the meta-variable \( m[x:A] \) by \( x \) in the term \( m[z] \) (indeed, we will see that it \( \beta \)-reduces to \( z \) as expected). Notice how the extended application \( M . (x:A)N \) introduces \( x \) in scope for the term \( N \).
4.4.1. Syntax. Every PTS is given by a tuple $\lambda S = \langle S, A, R \rangle$ where the elements of $S$ are called sorts, $A \subseteq S \times S$ is the set of axioms and $R \subseteq S \times S \times S$ is a set of triples that restrict the formation of $\Pi$-types (see e.g.\cite{Geu93, Bar92}).

The set of the pseudo-terms of the extended PTS is given by $T$ in the following grammar:

$$
T::=\ S\ |\ x[(\Delta)T]\ldots(\Delta)T\ |\ T\cdot(\Delta)T\ |\ T\cdot(\Delta)T\ |\ \Pi_x[(\Delta)T]\cdot T\cdot T\ |\ c[(\Delta)T]\ldots(\Delta)T\ |
$$

$$
\Delta::=\ \varepsilon\ |\ \Delta,\ x[(\Delta)T]
$$

$$
\Gamma::=\ \varepsilon\ |\ \Gamma,\ x[(\Delta)T]\ |\ \Gamma,\ c[(\Delta)T]=T\cdot T
$$

The syntactic category of terms is defined simultaneously with the category of parameter lists $\Delta$. The category $\Gamma$ will be used to denote the contexts in the typing judgments. The syntax is motivated by the intuitive meaning of the parameterized abstractions and application introduced above. $\lambda x[(\Delta)T] : A.M$, $\Pi x[(\Delta)T] : A.M$ and $\lambda x[(\Delta)T] : N : A.M$ introduce the parameterized variable/constant $x[(\Delta)T]$ in $M$ where it can be used provided it is given appropriate arguments. The variables declared in $\Delta$ can be used in $A$ and $N$, but their scope does not extend to $M$. The term $N$ is in the scope of the variables in $\Delta$ in an application $M \cdot (\Delta)N$. Similarly, in a variable/constant instance $x[(\Delta_1)N_1 \ldots (\Delta_n)N_n]$, each $N_i$ is in the scope of the variables in $\Delta_i$.

In order to ease the notation, we will drop the brackets after the variables or constants with empty parameter lists. When an empty parameter list is given as a subscript of an application we will drop both the - and the list and use the usual notation for application (e.g. we will write $x y$ for $x \langle \rangle y$).

On the level of parameter lists we define the notion of structural equivalence $\approx$ as follows:

$$
\begin{array}{c}
\varepsilon \approx \varepsilon \\
\Gamma_1 \approx \Gamma_2 \quad \Delta_1 \approx \Delta_2 \\
\Gamma_1, x_1[(\Delta_1)T] : A_1 \approx \Gamma_2, x_2[(\Delta_2)T] : A_2 \\
\end{array}
$$

The relation $\Gamma_1 \approx \Gamma_2$ should be read as “$\Gamma_1$ and $\Gamma_2$ have the same structure”. Note that the relation states properties of the structure of the contexts only. In particular, in the definition above $A_1$ and $A_2$ are not subject to any restrictions. We will assume that the names of the parameterized variables determine up to $\approx$-equivalence the context describing their parameters. Hence, if we talk about a variable $x[(\Delta)T]$ then its instances $x[(\Theta_1)t_1 \ldots (\Theta_n)t_n]$ must have the same number of actual parameters as there are elements in $\Delta$ and if $\Delta$ is the context $x_1[(\Delta_1)T] : A_1, \ldots, x_n[(\Delta_n)T] : A_n$ then $\Delta \approx \Theta$, $\Delta$.

Note also that the structural equivalence relation is a weaker notion than $\alpha$-equivalence as contexts that are not $\alpha$-convertible can have the same structure. The need to introduce this notion arises from the need to synchronize the context describing the parameters in the declaration of a variable and its instances as well as to define when an application of a $\lambda$-abstraction to a term is a redex (see the definition of $\beta$-reduction).

Example 4.44 (Well-formed pseudo-terms).
If $x[y[z[p:D]:E]:F]$ is a parameterized variable then the following term is well-formed:

$$\lambda y:([p:D]:A,B).x[y[z[p:D]:E](Y \cdot (p:D)z[p])]$$

$\lambda h[p[i:A]:B,q[j:A]:B \rightarrow C:A \rightarrow C,h[[i:A](a i),[j:A](b j)]]$ is a well-formed term.

**Definition 4.45 (Free and bound variables).** The set of the free variables $\text{FV}(\cdot)$ of a term or a context is defined as follows:

$\text{FV}(\varepsilon) = \emptyset$

$\text{FV}(\Delta, x[\Delta']:A) = \text{FV}(\Delta) \cup \text{FV}(A) \setminus \text{dom}(\Delta', \Delta) \cup \text{FV}(\Delta) \setminus \text{dom}(\Delta)$

$\text{FV}(s) = \emptyset$

$\text{FV}(x[\Delta_1]M_1, \ldots, \Delta_n]M_n]) = \{x\} \cup \bigcup_j (\text{FV}(M_j) \setminus \text{dom}(\Delta_i) \cup \text{FV}(\Delta_i))$

$\text{FV}(M : \Delta)N = \text{FV}(M) \cup \text{FV}(N) \setminus \text{dom}(\Delta) \cup \text{FV}(\Delta)$

$\text{FV}(\lambda x[\Delta]:A.M) = \text{FV}(M) \setminus \{x\} \cup \text{FV}(A) \setminus \text{dom}(\Delta) \cup \text{FV}(\Delta)$

$\text{FV}(\Pi x[\Delta]:A.M) = \text{FV}(M) \setminus \{x\} \cup \text{FV}(A) \setminus \text{dom}(\Delta) \cup \text{FV}(\Delta)$

$\text{FV}(M[\Delta] := N:A.M) = \text{FV}(M) \setminus \{x\} \cup (\text{FV}(N) \cup \text{FV}(A)) \setminus \text{dom}(\Delta) \cup \text{FV}(\Delta)$

An occurrence of a variable that is not free is bound. We assume that the names of the bound variables are always taken to be different from each other and from the names of the free variables.

This definition differs from the standard one in that it defines that the scope of the parameters $\Delta$ in $\Gamma$, $x[\Delta]:A$, $\lambda x[\Delta]:A.M$, $\Pi x[\Delta]:A.B$ and $!x[\Delta]:N:A.M$ is limited to $A$ and $N$ and that the actual parameters ($M_i$ in $x[\Delta_i]M_1 \ldots \Delta_n]M_n]$) and the arguments of applications ($N$ in $M : \Delta)N$) are in the scope of extra parameters ($\Delta_i$ and $\Delta$ resp.). This shows that the application and variable instances can behave as binders.

The process of filling in a value for a parameterized variable is called instantiation. As instances of variables provide actual arguments for formal parameters, we need to propagate the arguments in the term instantiating the variable. This leads to the following definition:

**Definition 4.46 (Instantiation).** Let $\Delta$ be the context $x_1[\Delta_1]:A_1, \ldots, x_n[\Delta_n]:A_n$ and $M[\Delta]:A$ be a meta-variable. The instantiation of $M[\Delta]$ by an arbitrary term $N$ in the term $M$ (notation $M[\Delta] := N$) is defined as follows:

$s[M[\Delta] := N] = s$

$(m[(\Theta_1)M_1 \ldots (\Theta_n)M_n])[m[\Delta] := N] = N[x_1[\Theta_1^*] := M_1^*] \ldots x_n[\Theta_n^*] := M_n^*])$

$(n[(\Theta_1)M_1, \ldots, (\Theta_k)M_k])[m[\Delta] := N] = n[(\Theta_1^* M_1^*, \ldots, (\Theta_k^* M_k^*)]$

$(M_1, (\Theta)M_2)[m[\Delta] := N] = M_1^* (\Theta^*) M_2^*$

$(\lambda y[\Theta]:U.M)[m[\Delta] := N] = \lambda y[\Theta^*]:U^*.M^*$

$(\Pi y[\Theta]:U.B)[m[\Delta] := N] = \Pi y[\Theta^*]:U^*.B^*$

$(\lambda y[\Theta]:U.M)[m[\Delta] := N] = \lambda y[\Theta^*]:U^*.M^*$

where $\text{for readability } M^*$ abbreviates $M[\Delta] := N$ and $\Theta^*$ is the obvious extension to parameter lists.
Note that by the $\approx$-convention on variables $\Delta_i \approx \Theta_i$ and this allows us to form the instantiations $\{ x_i[\Theta_i^*] := u_i^* \}$

The well-foundedness of instantiation is not completely self-evident, because in the second clause of the definition we apply recursively instantiations to a possibly 'larger' term $N$. Note however that in that case the contexts involved in the instantiations become strictly 'smaller' (w.r.t the depth of the context, see Definition 4.50) and that ensures the termination of the process.

**Example 4.47.** A few examples of instantiation:

<table>
<thead>
<tr>
<th>Term</th>
<th>Instantiation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h[]$</td>
<td>${ \hat{h}[] := t }$</td>
<td>$t$</td>
</tr>
<tr>
<td>$h[a]$</td>
<td>${ h[x:A] := x }$</td>
<td>$a$</td>
</tr>
<tr>
<td>$h[t, (x:A)p(x, t)]$</td>
<td>${ h[y:A, q[x:A], P(x, y)] := q[y] }$</td>
<td>$p(t, t)$</td>
</tr>
<tr>
<td>$h[g, h[\lambda y:A,y,s]]$</td>
<td>${ h[f: \Pi x:A.A, x:A] := f x }$</td>
<td>$g((\lambda y:A.y)s)$</td>
</tr>
</tbody>
</table>

**Remark 4.48** (Substitution is instantiation with no parameters). Note that if $\Delta$ is empty in an instantiation $\{ x[\Delta] := t \}$ then the instantiation of $x$ by $t$ in $M$ is exactly the result of the substitution of $t$ for the free occurrences of $x$ in $M$.

For example:

$$(\lambda y[z:Ax]:B.x)\{x := t\} = \lambda y[z:At]:B.t$$

**Example 4.49** (Variable Capture). Due to the parameters, some variables may get 'captured'. For example in the term

$$(\lambda x:A.h[x])\{ h[x:A] := x \} = \lambda x:A.x$$

the variable $x$ is captured by the binder which is in contrast to

$$(\lambda x:A.h[ [] ]\{ h[ ] := x \}) = \lambda y:A.x$$

where $x$ is still free after the instantiation (Note the renaming). In both cases we instantiate $h$ by $x$ but in the first example $x$ is bound and in the second it is free. We note that only variables that have been declared as parameters can get captured.

The notion of depth reflects the number of levels of parametrization in a context or a parameterized variable.

**Definition 4.50** (Depth). The parameter depth of $x[\Delta]$ is by definition $d(\Delta)$, where the depth $d(\Delta)$ of a context $\Delta$ is defined as:

$$d(\varepsilon) = 0$$
$$d(\Gamma, x[\Delta]:A) = \max(d(\Gamma), d(\Delta) + 1)$$

Example: $d(A:*), h[x:A]A \rightarrow A) = 2$ and $d(A:*), x:A, h: \Pi y:A.Bx) = 1$

**Proposition 4.51.**

1. If $\Delta \approx \Theta$ then $d(\Delta) = d(\Theta)$.
2. For all $\Theta$ we have $\Theta \approx \Theta \{ x[\Delta] := N \}$.

**Lemma 4.52.** If $\Delta \approx \Theta$ and $\text{dom}(\Delta) = \text{dom}(\Theta)$ then for all $M$ and $N$

$$M \{ x[\Delta] := N \} = M \{ x[\Theta] := N \}$$
4.4.2. Representing States and Tactic Instances as Well-Typed Terms.

In the previous section we introduced the parameterized meta-variables as a mechanism to model incomplete terms. The process of stepwise construction of a (proof) term can be modelled by a sequence of open terms representing the incomplete proof at different stages. Let us take as an example the following problem: Assume that $A$ is a type and $a$, $b$ and $c$ are terms of this type. Assume that $R$ is a binary relation on $A$ that is transitive and for each $x R(x, b)$ holds. As a part of a larger proof we would like to prove $R(a, c)$. We can reduce this goal to the goal of proving $R(b, c)$ using the assumptions we have. The initial state of the prover can be depicted as:

$$
\text{thm} : (x : A)(R \times b) \\
\text{tr} : (x, y, z : A) (R x y) \rightarrow (R y z) \rightarrow (R x z) \\
\text{thm} : … \text{thm} : … \text{thm} : … \text{thm} : … \\
\text{tr} : … \text{tr} : … \text{tr} : … \text{tr} : … \\
\text{thm} : (x : A) (p? : (R a y?)) \text{cp} : (R y? c) \\
\text{thm} : (x : A) \text{cp} : (R a y?) \text{cp} : (R y? c) \\
\Delta : (\text{tr} a y? (\text{thm}, \text{tr})) c p? (\text{thm}, \text{tr}) q? (\text{thm}, \text{tr}) : \text{Rac}
$$

The declarations above the line are the assumptions under which we have to prove the goal $R(a, c)$. Let us collect them in the context $\Delta$:

$$
\Delta = \text{thm} : \Pi x : A. R x b, \text{tr} : \Pi x, y, z : A. R x y \rightarrow R y z \rightarrow R x z
$$

We can represent the unknown proof of the goal by a meta-variable $m$ with parameters $\Delta$ and type $\text{Rac}$. Then the initial state of the prover can be encoded by the judgment

$$m[\Delta]; \text{Rac}, \Delta \vdash m[\text{thm}, \text{tr}] : \text{Rac}
$$

At this moment we would like to use the transitivity of $R$ by instantiating $x$ and $z$ by $a$ and $c$. This produces three new goals — to find an instantiation for $y$ in the transitivity, and to prove the premises corresponding to $R x y$ and $R y z$.

$$
\text{thm} : … \text{thm} : … \text{thm} : … \text{thm} : … \\
\text{tr} : … \text{tr} : … \text{tr} : … \text{tr} : … \\
y? : A \quad p? : (R a y?) \quad q? : (R y? c)
$$

How do we encode this new state and how is it related to the previous one? We introduce a new meta-variable for each new goal and in the place of $m[\text{thm}, \text{tr}]$ we have an application of $\text{tr}$:

$$y? [\Delta : A] \\
p? [\Delta] : (R a y?[\text{thm}, \text{tr}]), \\
q? [\text{thm}, \text{tr}] : (R y?[\text{thm}, \text{tr}] c), \\
\Delta \vdash (\text{tr} a y?[\text{thm}, \text{tr}] c p?[\text{thm}, \text{tr}] q?[\text{thm}, \text{tr}]) : \text{Rac}
$$

Now we would like to use our other assumption, $\text{thm}$, to solve the second goal. At this point a theorem prover would use unification to match $R a y?$ to $R x b$ and find out that in order to apply $\text{thm}$, $x$ has to be instantiated by $a$ and $y?$ has to be $b$. This results in the following state:
And it can be represented by the judgment

\[ r[\Delta] : (R b c), \Delta \vdash (tr \ a \ b \ c \ (tm \ a) \ r[\ell \ m, tr]) : \text{Rac} \]

This does not complete the proof, but if we have a look at the two transitions between the three states, we notice that there are several steps that we do on the meta-level that are not part of our representation. We introduce new meta-variables, we use them to give solutions to (some of) the pre-existing ones and we propagate these solutions through the representation of the state. All these are the meta-steps we make at each of the two transitions. The question arises: Can we represent each of the states and the meta-transformations on them explicitly by internalizing them as well-typed terms in the calculus? We will give an affirmative answer to this question by using abstractions over meta-variables as a means to internalize the dependency of the state on its meta-variables. The corresponding application operation would play the role of explicit representation of the instantiation of meta-variables. In this system a state can be encapsulated in a term by abstracting over all its assumptions and meta-variables. The transformation steps then become functions that expect terms of appropriate types matching the types of the state terms and can also be encoded by \( \lambda \)-terms.

As an illustration, the first state can be encoded by \( \lambda m[\Delta] : \text{Rac} \cdot \lambda \Delta. m[\ell \ m, tr] \) and its type is \( \Pi m[\Delta] : \text{Rac} \cdot \Pi \Delta. \text{Rac} \). The transformation step leading to the second state can be given by:

\[
\begin{align*}
\lambda S : (\Pi m[\Delta] : \text{Rac} \cdot \Pi \Delta. \text{Rac}). \\
\lambda \gamma?{\Delta} : A. \\
\lambda \rho?{\Delta} : (R \ a \ y?[tm, tr]). \\
\lambda \mu?{\Delta} : (R \ y?[tm, tr] \ c). \\
(S \cdot (\Delta) \ (tr \ a \ y?[tm, tr] \ c \ p?[tm, tr], q?[tm, tr]))
\end{align*}
\]

If we apply this transformation term to the state term and normalize, we get the term

\[
\begin{align*}
\lambda \gamma?{\Delta} & : A. \\
\lambda \rho?{\Delta} & : (R \ a \ y?[tm, tr]). \\
\lambda \mu?{\Delta} & : (R \ y?[tm, tr] \ c). \\
\lambda \Delta. (tr \ a \ y?[tm, tr] \ c \ p?[tm, tr], q?[tm, tr])
\end{align*}
\]

which is exactly the encoding of the second state.

### 4.4.3. \( \beta \)-reduction and confluence

Normally \( \beta \)-reduction is defined in terms of the capture-avoiding meta-substitution:

\[
(\lambda x : A. M) t \rightarrow _{\beta} M[t/x]
\]

We extend this definition to our pseudo-terms as follows:

\[
(\lambda x \Theta : A. M) \cdot (\Delta) \rightarrow _{\beta} M \{x[\Delta] := t\} \quad \text{if} \quad \Theta \approx \Delta
\]

Note that on unparameterized terms the two reduction relations coincide. The side condition \( \Theta \approx \Delta \) is needed, because we need to know that the two contexts have the same structure in order for the instantiation \( \{x[\Delta] := t\} \) to be well-defined.
To establish the confluence property we follow the modular confluence proof of Takahashi [Tak95]. Definition 4.54 introduces the notions of parallel reduction \( M \Rightarrow N \) and complete development \(#(M)\) and Lemma 4.55 states the relevant properties:

**Definition 4.54 (Parallel reduction \( M \Rightarrow N \) and complete development \(#(M)\)).**

\[
\begin{align*}
\Delta &\Rightarrow \Delta' \quad A \Rightarrow A' \quad M \Rightarrow M' & &\lambda x[\Delta]: A.M \Rightarrow \lambda x[\Delta']: A'.M' \\
\text{for} & &\Pi x[\Delta]: A.B \Rightarrow \Pi x[\Delta']: A'.B' \\
M &\Rightarrow M' \quad N \Rightarrow N' \quad \Delta \Rightarrow \Delta' & &\frac{M \Rightarrow M' \quad N \Rightarrow N' \quad \Delta \Rightarrow \Delta'}{M \langle \Delta \rangle N \Rightarrow M' \langle \Delta' \rangle N'}
\end{align*}
\]

\[
\begin{align*}
\#(s) &= s \\
\#(x[(\Theta_1)t_1 \ldots (\Theta_n)t_n]) &= x[(\#(\Theta_1)) \ldots (\#(\Theta_n)) \ldots (\#(t_n))]
\end{align*}
\]

Lemma 4.55 (Properties of \( \Rightarrow \) and \#).

1. If \( M_1 \Rightarrow M_2 \), \( N_1 \Rightarrow N_2 \) and \( \Delta_1 \Rightarrow \Delta_2 \) then \( M_1 \{ x[\Delta_1] := N_1 \} \Rightarrow M_2 \{ x[\Delta_2] := N_2 \} \).
2. \( \Rightarrow \Rightarrow \).
3. If \( M \Rightarrow N \) then \( N \Rightarrow \Rightarrow \).
4. If \( M \Rightarrow N \) then \( M \Rightarrow_\beta N \).
5. If \( M \Rightarrow_\beta N \) then \( M \Rightarrow N \).

**Proof.**

1. We proceed by induction on the depth of the context \( \Delta \). Let us fix \( \Delta \) and assume that the lemma holds for all terms and for all context with strictly smaller depth. For this \( \Delta \) we use induction on \( M_1 \). The non-trivial cases are the ones when \( M_1 \) is an instance of \( x \) or a \( \beta \)-redex. Let \( M_1 = x[\Theta] \) and \( M_2 = x[\Sigma] \) with \( \Theta \Rightarrow \Sigma \) and \( t \Rightarrow \tilde{s} \). Then we have

\[
\begin{align*}
M_1 \{ x[\Delta_1] := N_1 \} &= N_1 \{ x[t^*_i] := t'_i \}_{i=1}^n \\
M_2 \{ x[\Delta_2] := N_2 \} &= N_2 \{ x[s^*_i] := s'_i \}_{i=1}^n
\end{align*}
\]

where on the first (resp. second) line \( \ast \) denotes the application of \( \{ x[\Delta_1] := N_1 \} \) (resp. \( \{ x[\Delta_2] := N_2 \} \)). As the depth of \( \Delta \) is equal to one plus the maximum of the depths of \( \Theta \) and the depth of a context is preserved under substitution, we can apply the first induction hypothesis to \( N_1 \Rightarrow N_2 \),
4.4. Pure type systems with hereditary parametrization

\[ t \mapsto s \text{ and } \Theta^*_i \Rightarrow \Sigma^*_i \text{ obtaining} \]
\[ N_1 \{ x_i[\Theta^*_i] := t^*_i \}_{i=1}^n \Rightarrow N_2 \{ x_i[\Sigma^*_i] := s^*_i \}_{i=1}^n \]

but then we have \( N_2 \{ x_i[\Sigma^*_i] := s^*_i \}_{i=1}^n \equiv x[\Sigma^*] \{ x[\Delta_2] := N_2 \} \equiv M_2 \{ x[\Delta_2] := N_2 \}. \)

In the second case, when \( M_1 \) is a redex, we have
\[ M_1 = (\lambda y[\Theta]; A, P_1) \cdot (\Sigma_1) Q_1 \]
\[ M_2 = P_2 \{ y[\Sigma_2] := Q_2 \} \]

with \( P_1 \Rightarrow P_2, Q_1 \Rightarrow Q_2 \) and \( \Sigma_1 \Rightarrow \Sigma_2 \) and \( \Theta \approx \Sigma \). Applying the instantiations we get
\[ M_1 \{ x[\Delta_1] := N_1 \} = (\lambda y[\Theta^*]; A^*, P^*_1) \cdot (\Sigma^*_1) Q^*_1 \]
\[ M_2 \{ x[\Delta_2] := N_2 \} = (P_2 \{ y[\Sigma_2] := Q_2 \}^*) \]

where again for readability * abbreviates \( \{ x[\Delta_1] := N_1 \} \) on the first line and \( \{ x[\Delta_2] := N_2 \} \) on the second. From the definition of \( \Rightarrow \) we have \( (\lambda y[\Theta^*]; A^*, P^*_1) \cdot (\Sigma^*_1) Q^*_1 \Rightarrow P^*_1 \{ y[\Sigma_1^*] := Q_1^* \} \) which by the two induction hypotheses is equal to \( P^*_2 \{ y[\Sigma_2^*] := Q_2^* \} \). By Lemma 4.53 this is equal to \( (P_2 \{ y[\Delta_2] := Q_2 \}^*)^* \). \( \square \)

(2) Induction on \( M \) with non-trivial case when \( M \equiv (\lambda x[\Theta]; A, P) \cdot (\Delta) Q \). From the definition of \( \Rightarrow \) we have \( M \equiv (\lambda x[\Theta]; A, P) \cdot (\Delta) Q \Rightarrow P \{ x[\Delta] := Q \} \). By induction we have \( \Delta \Rightarrow \#(\Delta), P \Rightarrow \#(P) \) and \( Q \Rightarrow \#(Q) \) and thus from (1) we get \( P \{ x[\Delta] := Q \} \Rightarrow \#(P) \{ x[\#(\Delta)] := \#(Q) \} \) which is exactly \( \#((\lambda x[\Theta]; A, P) \cdot (\Delta) Q) \). \( \square \)

(3) The proof proceeds by induction on the generation of \( \Rightarrow \). Below we consider the two non-trivial cases:

Let the last rule in the derivation be \( M \Rightarrow M \). Then from (2) we have \( M \Rightarrow \#(M) \).

If the last rule is
\[ M \Rightarrow M' \quad N \Rightarrow N' \quad \Delta \Rightarrow \Delta' \]
\[ (\lambda x[\Theta]; A, M) \cdot (\Delta) N \Rightarrow M' \{ x[\Delta'] := N' \} \]

by induction we have \( M' \Rightarrow \#(M), N' \Rightarrow \#(N) \) and \( \Delta' \Rightarrow \#(\Delta) \). By (1) we have \( M' \{ x[\Delta'] := N' \} \Rightarrow \#(M) \{ x[\#(\Delta)] := \#(N) \} \) but the latter term is equal by definition to \( \#((\lambda x[\Theta]; A, M) \cdot (\Delta) N) \). \( \square \)

(4) Induction on the derivation of \( M \Rightarrow N \).

(5) Trivial.

\[ \square \]

**Corollary 4.56.** \( \Rightarrow_\beta \) is confluent.

**Proof.** From Lemma 4.55.3 it follows easily that \( \Rightarrow \) has the diamond property (i.e. if \( M \Rightarrow P \) and \( M \Rightarrow Q \) then there is a term \( N \) such that \( P \Rightarrow N \) and \( Q \Rightarrow N \)). Then, given \( M, P \) and \( Q \) such that \( M \Rightarrow_\beta P \) and \( M \Rightarrow_\beta Q \), we have \( M \Rightarrow^* P \) and \( M \Rightarrow^* Q \) using Lemma 4.55.5. But then iterating the diamond property for \( \Rightarrow \) we get a term \( N \) such that \( P \Rightarrow^* N \) and \( Q \Rightarrow^* N \). But then from Lemma 4.55.3 we have \( P \Rightarrow_\beta N \) and \( Q \Rightarrow_\beta N \) using the transitivity of \( \Rightarrow_\beta \). \( \square \)
4.4.4. $\delta$-reduction and confluence of $\beta\delta$-reduction.

**Definition 4.57** (Sound context). The set of sound contexts (notation $\text{Snd}(\Gamma)$) is defined inductively as follows:

- $\text{Snd}(\varepsilon)$
- If $\text{Snd}(\Gamma, \Delta)$ and $\text{FV}(A) \subseteq \text{dom}(\Gamma, \Delta)$ then $\text{Snd}(\Gamma, x[\Delta]:A)$;
- If $\text{Snd}(\Gamma, \Delta)$, $\text{FV}(N) \cup \text{FV}(A) \subseteq \text{dom}(\Gamma, \Delta)$ then $\text{Snd}(\Gamma, x[\Delta]:=N:A)$.

The fact that $\text{FV}(M) \subseteq \Gamma$ and $\text{Snd}(\Gamma)$ will be denoted by $M \in \text{Snd}(\Gamma)$. From now on we will consider only sound contexts.

Unfolding of a defined name is the replacement of the defined name by the value that it abbreviates. The notion of $\delta$-reduction refers to the reduction relation generated by the unfolding operation in a given context.

**Definition 4.58** ($\delta$-reduction).

\[(\delta_1) \quad \Gamma \vdash !x[\Delta]:=N:A. M \to_\delta M\]
\[(\delta_2) \quad \Gamma_1, x[\Delta]:=N:A, \Gamma_2 \vdash x[(\Theta_1)M_1 \ldots (\Theta_n)M_n] \to_\delta N\{x_i(\Theta_i) := M_i\}_{i=1}^{n}\]

where in $(\delta_1)$ $x \notin \text{FV}(M)$ and in $(\delta_2)$ we have $\Delta = x_1[\Delta_1]:A_1, \ldots, x_n[\Delta_n]:A_n$ and $\Theta_i \approx \Delta_i$.

To establish the confluence of $\beta\delta$ for our system we take a variant of the proof in [BKLN02]. We define the mapping $|-|$ and show that $|M|_\Gamma$ is the $\delta$-normal form of $M$ under $\Gamma$. Then we show that $\delta$ commutes with $\beta\delta$ and using the confluence of $\beta$ we will get the diagram in Figure 4.17.

**Figure 4.17.** Schematic representation of the confluence proof for $\beta\delta$. 
4.4. Pure Type Systems with Hereditary Parametrization

**Definition 4.59** ($|M|_\Gamma$). The mapping $| - |_\Gamma$ is defined by induction on the structure of the terms and parameter lists\(^1\) as follows:

$$
|s|_\Gamma = s \\
|x[(\Theta_1)_{t_1} \ldots (\Theta_n)_{t_n}]|_\Gamma = x[(\Theta_1)[t_1]_{\Gamma, \Theta_1} \ldots (\Theta_n)[t_n]_{\Gamma, \Theta_n}] \\
|e[(\Theta_1)_{t_1} \ldots (\Theta_n)_{t_n}]|_\Gamma = |N|_{\Gamma, \Delta} \{x_i[\Theta_i] := [t_i]_{\Gamma, \Theta_i}\} \\
|\lambda x[\Delta]::A.M|_\Gamma = \lambda x[\Delta][|A|_{\Gamma, \Delta}]_{\Gamma, \Delta, x} : |M|_{\Gamma, x}[\Delta] : A \\
|\Pi x[\Delta]::A.M|_\Gamma = \Pi x[|\Delta|][|A|_{\Gamma, \Delta}]_{\Gamma, \Delta, x} : |M|_{\Gamma, x}[\Delta] : A \\
|\langle x \rangle@\Delta:=[N::A.M]|_\Gamma = |M|_{\Gamma, x}[:N::A]_{\Gamma, x}[\Delta] : A \\
|\varepsilon|_\Gamma = \varepsilon \\
|\Delta, x[\Delta'][::A]|_\Gamma = |\Delta|, x[|\Delta'|_{\Gamma, \Delta'}] : |A|_{\Gamma, \Delta, x}[\Delta'] : A
$$

where in the third clause $\Gamma = \Gamma_1, e[\Delta] := N::A, \Gamma_2$ with $\Delta = x_1[\Delta_1] : A_1 \ldots x_n[\Delta_n] : A_n$ and $\Delta_i \approx \Theta_i$.

**Lemma 4.60.** For every $M$ and $\Gamma$ holds $\Gamma \vdash M \rightarrow_\delta |M|_\Gamma$.

**Proof.** Induction on $M$. We treat the cases when $M$ is a variable instance and $\lambda$-abstraction:

Suppose $M$ is $x[\tilde{\Theta}t]$. We have two cases - $x$ is a 'real' variable or $x$ is a defined constant in $\Gamma$. In the first case by induction we get $\Gamma, \Theta_i \vdash t_i \rightarrow_\delta |t_i|_{\Gamma, \Theta_i}$ and $\Gamma \vdash \Theta_i \rightarrow_\delta |\Theta_i|_\Gamma$. Therefore we have:

$$
\Gamma \vdash x[\tilde{\Theta}t] \rightarrow_\delta x[\tilde{\Theta}[t]_{\Gamma, \Theta}] \rightarrow_\delta x[\Theta[t]_{\Gamma, \Theta}]
$$

If $x[\Delta] := N::A \in \Gamma$ then by induction we still have $\Gamma, \Theta_i \vdash t_i \rightarrow_\delta |t_i|_{\Gamma, \Theta_i}$ and therefore $\Gamma \vdash x[\tilde{\Theta}t] \rightarrow_\delta x[\tilde{\Theta}[t]_{\Gamma, \Theta}]$ but $\Gamma \vdash x[\tilde{\Theta}[t]_{\Gamma, \Theta}] \rightarrow_\delta N\{x_i[\Theta_i] := [t_i]_{\Gamma, \Theta_i}\}_{i=1}^n$. Suppose $M$ is $\lambda x[\Delta]::A.P$. By induction we get:

$$
\Gamma \vdash \Delta \rightarrow_\delta |\Delta|_\Gamma \\
\Gamma, \Delta \vdash A \rightarrow_\delta |A|_{\Gamma, \Delta} \\
\Gamma, x[\Delta]::A \vdash P \rightarrow_\delta |P|_{\Gamma, x[\Delta]::A}
$$

Therefore

$$
\Gamma \vdash \lambda x[\Delta]::A.P \rightarrow_\delta \lambda x[\Delta]::A.|P|_{\Gamma, x[\Delta]::A} \\
\Gamma \vdash \lambda x[\Delta]::A.|P|_{\Gamma, x[\Delta]::A} \rightarrow_\delta \lambda x[\Delta]::A.|A|_{\Gamma, \Delta, x} : |P|_{\Gamma, x[\Delta]::A} \\
\Gamma \vdash \lambda x[\Delta]::A.|A|_{\Gamma, \Delta, x} : |P|_{\Gamma, x[\Delta]::A} \rightarrow_\delta \lambda x[\Delta][|A|_{\Gamma, \Delta, x} : |P|_{\Gamma, x[\Delta]::A}]
$$

and by transitivity we are done. \(\square\)

**Lemma 4.61.** If $\Gamma \vdash |M|_\Gamma \rightarrow_\delta N$ then $N \equiv |M|_\Gamma$.

**Proof.** By induction on $M$ we can see that $|M|_\Gamma$ contains no defined constants and no local definitions. \(\square\)

**Lemma 4.62.** If $\Gamma \vdash M \rightarrow_\beta N$ then $\Gamma \vdash |M|_\Gamma \rightarrow_\beta |N|_\Gamma$.

\(^1\)not contexts, note that there is no clause for $|\Delta, e[\Delta'][::N::A]|_\Gamma$. There is no problem to add such a clause, but we would never use it.
4. Typing for Incomplete Terms and Proofs

Proof. If $\Gamma \vdash M \rightarrow \beta N$ then by induction on $M$ we have $|M|_\Gamma \equiv |N|_\Gamma$. It is also not difficult to see that if $M \rightarrow \beta N$ then $|M|_\Gamma \rightarrow \beta |N|_\Gamma$. Since a redex may be an argument of a defined constant, after unfolding it may appear multiple times in the body and this is the reason to have $\rightarrow \beta$.

Corollary 4.63 (Confluence of $\beta\delta$). If $M \in \text{Snd}(\Gamma)$, $\Gamma \vdash M \rightarrow_{\beta\delta} P$ and $\Gamma \vdash M \rightarrow_{\beta\delta} Q$ then there is a term $N$ such that $\Gamma \vdash P \rightarrow_{\beta\delta} N$ and $\Gamma \vdash Q \rightarrow_{\beta\delta} N$.

Proof. Follows using the diagram in Figure 4.17 using the above lemmas. □

4.4.5. Typing System. The derivation rules of a typing system give an inductive definition of the typing relation that assigns types to terms in a given context that specifies the types of the free variables. The standard derivation rules for PTSs (see e.g. [Par92, Geu93]) are parameterized by three sets $\langle S, A, R \rangle$ and the different PTSs can be obtained by giving particular values to the three parameters. $S$ is the set of sorts, $A$ is a subset of $S \times S$ and its elements are called axioms. The set $R$ is a subset of $S \times S \times S$ and its elements are used to restrict the $\Pi$-formation rule.

For the purposes of typing terms with parameterized variables we introduce one extra set $\mathcal{P}$ that would be a subset of $S \times S$ and it will be used to denote the dependencies between the type of a parameter of a variable and the type of the variable itself. Hence a PTS with parametric variables will be given by a parametric specification that is a 4-tuple $\lambda s = \langle S, A, R, \mathcal{P} \rangle$.

Notation 4.64. 
- Let $\Gamma \vdash \Delta_1 =_{\beta\delta} \Delta_2$ denote the $\beta\delta$-convertibility of $\Delta_1$ and $\Delta_2$ in context $\Gamma$.
- We will write $x[\Theta\vec{t}]$ for $x[(\Theta_1)t_1\ldots(\Theta_n)t_n]$.
- Let $\Delta = x_1[\Delta_1]:A_1, \ldots, x_n[\Delta_n]:A_n$ and $\vec{t} = \langle t_1, \ldots, t_n \rangle$. We will write $\Gamma \vdash \Theta\vec{t} : \Delta$ for the conjunction of the judgments $\Gamma, \Theta_k \vdash t_k : \delta_{k-1}A_k$ with $k \in [1 \ldots n]$ where $\delta_0 = \text{id}$, $\delta_{k+1} = \delta_k \circ \{x_{k+1}[\Theta_{k+1}] := t_{k+1}\}$ and $\Gamma \vdash \Theta_k =_{\beta\delta} \delta_{k-1}\Delta_k$.
- By $\{\Delta := \Theta\vec{t}\}$ we will denote $\delta_n$ from above and $\Delta_{\vec{t}}$ will denote the context $x_1[\Delta_1]:A_1, \ldots, x_{i-1}[\Delta_{i-1}]:A_{i-1}, \Delta_i$.

Figure 4.18 contains the derivation rules for a PTS with hereditarily parameterized variables, constants and definitions $(V^hC^hD^h\text{PTS})$. As usual $s$ and $s_i$ denote sorts from $S$.

We briefly comment on the modifications to the rules in order to explain the intuition behind them. In the (start) rule we type occurrences of the parameterized variables and constants introduced in the context. An instance of a variable/constant is well-typed if it has a correct number and type of arguments. The premise $\Gamma, \Delta \vdash A:s$ is necessary in order to ensure that $\Gamma$ is a valid context in case there are no parameters. Each actual parameter $t_i$ can be given a context $\Theta_i$ that locally introduces variables that can be used in $t_i$. The context $\Theta_i$ is required to be $\beta\delta$-convertible, but not necessarily syntactically equal to $\delta_{i-1}\Delta_i$ because...
for the Subject Reduction property we should be able to type instances in which \( \beta \)-reductions have been executed in \( \Theta_i \).

Using the weakening rules (weak-var) and (weak-def) we can add variables, constants and definitions to a context. Note that the parameters of the variable/constant can be used in its type. Very much like in the \( C^nD^pPTSs \)[BKLN02], by a suitable choice of \( P \) the condition \( (s_i, s) \in P \) is used to restrict the possible parameters that a variable of a given sort can take.

The (let-form) rule is used to type local definitions in sorts. The rule is only interesting for the topports because for the definitions in other sorts the rule is derivable: we can use the conversion rule with \( \Gamma \vdash x[\Delta] := N : A, s =_\delta s \).

As usual, the II-formation rule is restricted by \( R \). The new aspect is that the bound variable may have parameters. Again, the parameters in \( \Delta \) can be used in \( A \) (but not in \( B \), see Definition 4.45).

\[
\begin{array}{ll}
\Gamma, \Delta \vdash A : s & \Gamma \vdash \Theta \varepsilon \Delta \\
\Gamma \vdash x[(\Theta_1)t_1 \ldots (\Theta_n)t_n] : A\{\Delta := \Theta\} & (s_1, s_2) \in A \quad (\text{axiom})
\end{array}
\]

\[
\begin{array}{ll}
\Gamma \vdash M : B & \Gamma, \Delta \vdash A : s \\
\Gamma, \Delta_1 \vdash A : s_i & (s_i, s) \in P \quad (\text{start})
\end{array}
\]

\[
\begin{array}{ll}
\Gamma \vdash M : B & \Gamma, \Delta \vdash N : A : s \\
\Gamma, \Delta \vdash A : s_i & (s_i, s) \in P \quad (\text{weak-def})
\end{array}
\]

\[
\begin{array}{ll}
\Gamma, \Delta \vdash A : s_1 & \Gamma, x[\Delta] : A \vdash B : s_2 \\
\Gamma \vdash \Pi x[\Delta] : A : B : s_3 & (s_1, s_2, s_3) \in R \quad (\Pi\text{-form})
\end{array}
\]

\[
\begin{array}{ll}
\Gamma, x[\Delta] : N : A \vdash M : B & \Gamma \vdash !x[\Delta] : N : A : B : s \\
\Gamma \vdash (\Pi x[\Delta] : N : A) & (let)
\end{array}
\]

\[
\begin{array}{ll}
\Gamma, x[\Delta] : A \vdash M : B & \Gamma \vdash \Pi x[\Delta] : A : B : s \\
\Gamma \vdash (\Pi x[\Delta] : A : B) & (\lambda)
\end{array}
\]

\[
\begin{array}{ll}
\Gamma \vdash M : \Pi x[\Delta] : A : B & \Gamma, \Delta \vdash N : A \\
\Gamma \vdash M : \Pi x[\Delta] N : B \{x[\Delta] := N\} & (app)
\end{array}
\]

\[
\begin{array}{ll}
\Gamma \vdash M : A & \Gamma \vdash B : s \\
\Gamma \vdash M : B & \Gamma \vdash A \Rightarrow B \quad (\text{conv})
\end{array}
\]

\textbf{Figure 4.18.} Derivation rules for V^hC^hD^hPTSs.
The rule (let) is used to type local definitions provided that their type is valid.

The (\lambda) rule abstracts parameterized variables. If we want to apply such an abstraction to an argument, the argument should be typed in a context that is extended with the parameters. This is done by the (app) rule. Note how the application \texttt{\lambda(\Delta)} introduces the parameters in the context of the argument.

We now proceed by establishing the important meta-properties of the system.

**Lemma 4.65 (Weakening).** Let $\Gamma_1, \Gamma_2 \vdash M : B$, $\Gamma_1, \Delta \vdash A : s$ and $\Gamma_1, \Delta_1 \vdash A_1 : s_1$ with $(s, s) \in \mathcal{P}$. Let also $x$ be a fresh variable/constant name. Then

1. $\Gamma_1, x[\Delta] : A, \Gamma_2 \vdash M : B$.
2. If $\Gamma_1, \Delta \vdash N : A$ then $\Gamma_1, x[\Delta] := N : A, \Gamma_2 \vdash M : B$.

**Proof.** Simultaneous induction on the derivation of $\Gamma_1, \Gamma_2 \vdash M : B$.  

**Lemma 4.66 (Generation Lemma).** Let $\lambda S = \langle S, A, R, \mathcal{P} \rangle$ be a parametric specification. Then

1. If $\Gamma \vdash s : D$ then there is $s' \in S$ such that $\Gamma \vdash D =_{\beta \delta} s'$ and $(s, s') \in A$;
2. If $\Gamma \vdash x[\Theta] : D$ then we have the following two cases:
   - $\Gamma = \Gamma_1, x[\Delta] : A, \Gamma_2$ and there is an $s$ such that $\Gamma_1, \Delta \vdash A : s$, $\Gamma \vdash \Theta : \Delta$ and $\Gamma \vdash D =_{\beta \delta} A\{\Delta := \Theta\}$;
   - $\Gamma = \Gamma_1, x[\Delta] := N : A, \Gamma_2$ and there is an $s$ such that $\Gamma_1, \Delta \vdash N : A : s$, $\Gamma \vdash \Theta : \Delta$ and $\Gamma \vdash D =_{\beta \delta} A\{\Delta := \Theta\}$;
3. If $\Gamma \vdash (\Pi x[\Delta] : A : B) : D$ then there are sorts $(s_1, s_2, s_3) \in R$ such that $\Gamma, \Delta \vdash A : s_1$ and $\Gamma, x[\Delta] : A \vdash B : s_2$ and $\Gamma \vdash D =_{\beta \delta} s_3$.
4. If $\Gamma \vdash (\lambda x[\Delta] : A : M) : D$ then there are $s$ and $B$ such that $\Gamma \vdash \Pi x[\Delta] : A : B : s$, $\Gamma, x[\Delta] : A \vdash M : B$ and $\Gamma \vdash \Pi : A : B =_{\beta \delta} D$.
5. If $\Gamma \vdash M \cdot (\Delta) : N : D$ then there are $A$ and $B$ such that $\Gamma \vdash M : \Pi x[\Delta] : A : B$, $\Gamma, \Delta \vdash N : B$ and $\Gamma \vdash D =_{\beta \delta} B\{\Delta := N\}$.
6. If $\Gamma \vdash x[\Delta] := N : A : M : D$ then there are $s$ and $B$ such that $\Gamma, x[\Delta] := N : A \vdash M : B$, $\Gamma \vdash x[\Delta] := N : A : B : s$ and $\Gamma \vdash D =_{\beta \delta} x[\Delta] := N : A : B$.
7. If $\Gamma, x[\Delta] : A, \Gamma' \vdash M : D$ then there are $s$ and $s_1$ such that $\Gamma, \Delta \vdash A : s$ and $\Gamma, x[\Delta] =_{\beta \delta} \Gamma' : A : s_1, (s, s) \in \mathcal{P}$.
8. If $\Gamma, x[\Delta] := N : A, \Gamma' \vdash M : D$ then there are $s$ and $s_1$ such that $\Gamma, \Delta \vdash N : A : s$ and $\Gamma, x[\Delta] : A : s_1, (s, s) \in \mathcal{P}$.

**Proof.** Induction on the generation of the typing relation.  

**Lemma 4.67 (Substitution Lemma).**

1. If $\Gamma, x[\Delta] : A, \Gamma' \vdash M : B$ and $\Gamma, \Delta \vdash N : A$ then $\Gamma, \Gamma' \vdash x[\Delta] := N \vdash M\{x[\Delta] := N\} : B\{x[\Delta] := N\}$.
2. If $\Gamma, x[\Delta] := N : A, \Gamma' \vdash M : B$ then $\Gamma, \Gamma' \vdash x[\Delta] := N \vdash M\{x[\Delta] := N\} : B\{x[\Delta] := N\}$.

**Proof.** By induction on the depth of $\Delta$ and a nested induction on the derivation.
Corollary 4.68 (Correctness of types). If $\Gamma \vdash M : A$ then either $\Gamma \vdash A =_{\beta \delta} s$ or $\Gamma \vdash A : s$ for some $s$.

Proof. Consider the possible cases for $M$ and apply the Generation Lemma and the Substitution Lemma.

Proposition 4.69. If $\Gamma, \Delta, \Theta \vdash A =_{\beta \delta} B$ and $\Gamma, \Delta =_{\beta \delta} \Gamma', \Delta'$ then

$$\Gamma, \Delta', \Theta \vdash A =_{\beta \delta} B$$

Definition 4.70. We will call a context $\Gamma$ legal (notation $\Gamma \vdash \Omega k$) if there are $M$ and $A$ such that $\Gamma \vdash M : A$.

Lemma 4.71. Let $\Gamma, \Delta, \Theta \vdash M : A$, $\Gamma', \Delta' \vdash \Omega k$ and $\Gamma, \Delta =_{\beta \delta} \Gamma'$. Then

$$\Gamma, \Delta', \Theta \vdash M : A$$

Proof. Induction on the derivation of $\Gamma, \Delta, \Theta \vdash M : A$.

(weak-var): There are three cases depending on whether the variable that we introduce by weakening is in the sections covered by $\Gamma$, $\Delta$ or $\Theta$.

- In the first case we have $\Delta$ and $\Theta$ empty and the statement of the lemma is trivial.
- In the second case $\Theta$ is empty and $\vdash \Gamma, \Delta, x[\Sigma] : A =_{\beta \delta} \Gamma, \Delta', x[\Sigma'] : A'$:

$$\begin{array}{c}
\Gamma, \Delta \vdash M : B \\
\Gamma, \Delta, \Sigma \vdash A : s \\
\Gamma, \Delta, \Sigma | x \vdash A_i : s_i \\
\hline
\Gamma, \Delta, x[\Sigma] : A \vdash M : B
\end{array}$$

By induction hypothesis we have $\Gamma, \Delta' \vdash M : B$. Furthermore, because $\Gamma, \Delta', x[\Sigma'] : A' \vdash \Omega k$ by Generation we have $s'$ and $s'_i$ such that $(s', s'_i) \in \mathcal{P}$ and

$$\begin{array}{c}
\Gamma, \Delta', x[\Sigma'] \vdash A' : s' \\
\Gamma, \Delta', x[\Sigma']_i \vdash A'_i : s'_i
\end{array}$$

Therefore we can apply (weak-var) and get

$$\Gamma, \Delta', x[\Sigma'] : A' \vdash M : B$$

- In this case the variable being introduced is in the last part of the context:

$$\begin{array}{c}
\Gamma, \Delta, x[\Sigma] : A \vdash M : B
\end{array}$$

We can apply directly the induction hypothesis to the premises:

$$\begin{array}{c}
\Gamma, \Delta', \Theta \vdash M : B \\
\Gamma, \Delta', \Theta, \Sigma \vdash A : s \\
\Gamma, \Delta', \Theta, \Sigma | x \vdash A_i : s_i
\end{array}$$

(weak-def): Similar to (weak-var).

(II-form):

$$\begin{array}{c}
\Gamma, \Delta, \Theta, \Sigma \vdash A : s_1 \\
\Gamma, \Delta, \Theta, x[\Sigma] : A \vdash B : s_2 \\
\hline
\Gamma, \Delta, \Theta \vdash \Pi x[\Sigma] : A. B : s_3
\end{array}$$

By the induction hypothesis we have $\Gamma, \Delta', \Theta, \Sigma \vdash A : s_1$ and $\Gamma, \Delta', \Theta, x[\Sigma] : A \vdash B : s_2$ and after applying the rule (II-form) we get $\Gamma, \Delta', \Theta \vdash \Pi x[\Sigma] : A. B : s_3$. 

4.4. Pure Type Systems with Hereditary Parametrization
\(\lambda\):

\[
\Gamma, \Delta, \Theta, x[\Sigma]: A \vdash M : B \quad \Gamma, \Delta, \Theta \vdash \Pi x[\Sigma]: A.B : s
\]

\[
\Gamma, \Delta, \Theta \vdash (\lambda x[\Sigma]: A.M) : (\Pi x[\Sigma]: A.B)
\]

By induction we get \(\Gamma, \Delta', \Theta, x[\Sigma]: A \vdash M : B\) and \(\Gamma, \Delta', \Theta \vdash \Pi x[\Sigma]: A.B : s\).

Now we can apply the rule (\(\lambda\)) to get \(\Gamma, \Delta', \Theta \vdash (\lambda x[\Sigma]: A.M) : (\Pi x[\Sigma]: A.B)\).

(app):

\[
\Gamma, \Delta, \Theta \vdash M : \Pi x[\Sigma]: A.B \quad \Gamma, \Delta, \Theta, \Sigma \vdash N : A
\]

\[
\Gamma, \Delta, \Theta \vdash M \cdot (\Sigma) N : B\{x[\Sigma] := N\}
\]

The induction gives us \(\Gamma, \Delta', \Theta \vdash M : \Pi x[\Sigma]: A.B\) and \(\Gamma, \Delta', \Theta, \Sigma \vdash N : A\).

We can apply the rule (app) to get \(\Gamma, \Delta', \Theta \vdash M \cdot (\Sigma) N : B\{x[\Sigma] := N\}\)

(conv):

\[
\Gamma, \Delta, \Theta \vdash M : A \quad \Gamma, \Delta, \Theta \vdash B : s \quad \Gamma, \Delta, \Theta \vdash A =_{\beta\delta} B
\]

\[
\Gamma, \Delta, \Theta \vdash M : B
\]

By induction we get \(\Gamma, \Delta', \Theta \vdash M : A\) and \(\Gamma, \Delta', \Theta \vdash B : s\). Because \(\vdash \Gamma, \Delta, \Delta' =_{\beta\delta} \Gamma, \Delta'\), by Proposition 4.69 we get \(\Gamma, \Delta', \Theta \vdash A =_{\beta\delta} B\) and we can apply the conversion rule to get \(\Gamma, \Delta', \Theta \vdash M : B\).

(start), (let-form), (let): Similar to (II-form), (\(\lambda\)) and (app) using the induction hypothesis.

\[\square\]

**Lemma 4.72 (Subject Reduction).**

1. If \(\Gamma \vdash M : A\) and \(\Gamma \vdash M \rightarrow_{\beta\delta} N\) then \(\Gamma \vdash N : A\).
2. If \(\Gamma \vdash M : A\) and \(\Gamma \rightarrow_{\beta\delta} \Delta\) then \(\Delta \vdash M : A\).

**Proof.** Simultaneous induction on the derivation of \(\Gamma \vdash M : A\). Consider the last rule of the derivation:

\(\text{(start)}\):

\[
\Gamma, \Delta \vdash A : s \quad \Gamma, \Theta_i \vdash t_i; \delta_{i-1}A_i
\]

\[
\Gamma \vdash x[(\Theta_1)t_1 \ldots (\Theta_n)t_n]: \Delta := \Theta_i
\]

1. Consider the possible reduction steps.
   - Let \(\Gamma, \Theta_k \vdash t_k \rightarrow_{\beta\delta} t'_k\) for some \(k \in [1..n]\). For \(i \neq k\) define \(t'_i = t_i\). Define also \(\delta'_0 = id\) and \(\delta'_{i+1} = \delta'_i \circ \{x_{i+1}[\Theta_{i+1}] := t'_{i+1}\}\).
   
   We will use the following scheme: prove that (a) \(\Gamma, \Theta_i \vdash t'_i : \delta'_{i-1}A_i\) and (b) \(\Gamma, \Theta_i =_{\beta\delta} \Gamma, \delta'_{i-1} A_i\). Then by the start rule we get \(\Gamma \vdash x[\Theta_i t'] : \delta'_n A\). Then since \(\Gamma \vdash \delta_n A =_{\beta\delta} \delta'_n A\) by the conversion rule we will get the needed \(\Gamma \vdash x[\Theta_i t'] : \delta'_n A\).

By the side condition of the (start) rule we have \(\Gamma, \Theta_i =_{\beta\delta} \Gamma, \delta_{i-1} A_i\). It is easy to see that then \(\Gamma, \delta_{i-1} A_i =_{\beta\delta} \Gamma, \delta'_{i-1} A_i\) and thus we have (b) by transitivity.

Getting (a) is more complicated. By induction on \(i\) we will show that \(\Gamma, \Theta_i \vdash \delta'_{i-1} A_i : s_i\). Using that fact and the fact
that $\Gamma, \Theta, 1 \vdash \delta_{i-1}^{i} A_i =_{\beta\delta} \delta_{i-1} A_i$ by conversion we will have (a).
For $i = 1$ taking into account that $\delta_0 = id$, from the generation lemma we have:

$$\Gamma, x_1[\delta_0^{i} \Delta_1]:\delta_0^{i} A_1 \ldots x_i[\delta_0^{i} \Delta_1]:\delta_0^{i} A_i, \delta_0^{i} \Delta_{i+1} \vdash \delta_0^{i} A_{i+1} : s_{i+1}$$

We know by induction that $\Gamma, \Theta_1 \vdash t'_1 : \delta_0^{i} A_1$. But $\Gamma, \Theta_1 =_{\beta\delta} \Gamma, \delta_0^{i} \Delta_1 =_{\beta\delta} \delta_0^{i} A_1$. By generation we have $\Gamma, \delta_0^{i} \Delta_1 \vdash \text{Ok}$ and therefore by Lemma 4.71 we have $\Gamma, \delta_0^{i} \Delta_1 \vdash t'_1 : \delta_0^{i} A_1$. Now we can apply Substitution Lemma and get

$$\Gamma, x_2[\delta_0^{i} \Delta_2]:\delta_0^{i} A_2 \ldots x_i[\delta_0^{i} \Delta_1]:\delta_0^{i} A_i, \delta_0^{i} \Delta_{i+1} \vdash \delta_0^{i} A_{i+1} : s_{i+1}$$

By the first induction hypothesis we have $\Gamma, \Theta_2 \vdash t'_2 : \delta_1 A_2$. By the second we have $\Gamma, \Theta_2 \vdash \delta_1 A_2 : s_2$ therefore by (conv) we have $\Gamma, \Theta_2 \vdash t'_2 : \delta_1 A_2$. By Generation $\Gamma, \delta_1^{i} \Delta_2 \vdash \text{Ok}$. Therefore by Lemma 4.71 we have $\Gamma, \delta_1^{i} \Delta_2 \vdash t'_2 : \delta_1 A_2$. By Substitution then

$$\Gamma, x_3[\delta_0^{i} \Delta_3]:\delta_0^{i} A_3 \ldots x_i[\delta_0^{i} \Delta_1]:\delta_0^{i} A_i, \delta_0^{i} \Delta_{i+1} \vdash \delta_0^{i} A_{i+1} : s_{i+1}$$

We continue in this manner until we get

$$\Gamma, \delta_i^{i} \Delta_{i+1} \vdash \delta_i^{i} A_{i+1} : s_{i+1}$$

but $\Gamma \Theta_{i+1} \vdash \text{Ok}$ and $\Gamma, \Theta_{i+1} =_{\beta\delta} \Gamma, \delta_i^{i} \Delta_{i+1}$. By Lemma 4.71 we have

$$\Gamma, \Theta_{i+1} \vdash \delta_i^{i} A_{i+1} : s_{i+1}$$

- If the reduction step was performed in $\Theta_k$, we have $\Gamma, \Theta_k \rightarrow_{\beta\delta} \Gamma, \Theta_k$ and we use the induction hypothesis for (2) to get

$$\Gamma, \Theta'_k \vdash t'_k : \delta_{k-1} A_k$$. It is easy to see that the side conditions for (start) are true and we can apply it to get

$$\Gamma \vdash x[\Theta'] : A[\Delta := \Theta']$$

(since $\vdash \Gamma, \Delta =_{\beta\delta} \Gamma$, implicates $M\{x[\Delta] := N\} = M\{x[\Theta] := N\}$).

- The last possibility is that $x$ is a defined constant in $\Gamma$ and the reduction step has unfolded it. Suppose $x[\Delta] := N : A \in \Gamma$ and the reduction was $\Gamma \vdash x[\Theta'] \rightarrow_{\delta} N[\Delta := \Theta']$. By Generation we have

$$\Gamma, x_1[\delta_0^{i} \Delta_1]:\delta_0^{i} A_1, \ldots, x_n[\delta_0^{i} \Delta_n]:\delta_0^{i} A_n \vdash \delta_0^{i} N : \delta_0^{i} A$$

On the other hand we have $\Gamma, \Theta_1 \vdash t_1 : \delta_0^{i} A_1$. Using Lemma 4.71 as in the previous cases we have $\Gamma, \delta_0^{i} \Delta_1 \vdash t_1 : \delta_0^{i} A_1$ and by Substitution Lemma we get

$$\Gamma, x_2[\delta_1^{i} \Delta_2]:\delta_1^{i} A_2, \ldots, x_n[\delta_1^{i} \Delta_n]:\delta_1^{i} A_n \vdash \delta_1^{i} N : \delta_1^{i} A$$

Iterating this process we get

$$\Gamma \vdash \delta_n^{i} N : \delta_n^{i} A$$
(recall that \( \{ \Delta = \emptyset \} \) is another name for \( \delta_n \)).

(2) Let \( \Gamma \vdash \beta_\Sigma. \) Then \( \Gamma, \Delta \vdash \beta_\Sigma, \Delta. \) By Induction then we have 
\( \Sigma, \Delta \vdash A : s. \) Similarly \( \Sigma, \Theta_i \vdash t_i : \delta_{i-1}A_i. \) From the side condition
\( \Gamma_i, \Theta_i = \beta_\delta, \Gamma, \delta_i \vdash \Sigma, \Theta_i = \beta_\delta, \Sigma, \delta_i - 1 \Delta_i. \) Therefore we can apply the (start) rule.

\[
(\text{let}): \quad \Gamma, x[\Sigma] := t; B \vdash P : B \quad \Gamma \vdash \lambda x[\Sigma] := t; B.C : s \\
\Gamma \vdash (\lambda x[\Sigma] := t; B.P) : (\lambda x[\Sigma] := t; B.C)
\]

(1) We distinguish several cases depending on the kind of reduction step
that was made:
- The reduction step was \( \Gamma \vdash !x[\Sigma] := t; B.P \rightarrow xP \) with \( x \notin \text{FV}(P). \) From the premise \( \Gamma, x[\Sigma] := t; B \vdash P : C \) by Substitution
Lemma we get \( \Gamma \vdash P : C \{ x[\Sigma] := t \}. \) But \( \Gamma \vdash !x[\Sigma] := t; B.C \rightarrow \beta_\delta C \{ x[\Sigma] := t \} \) and from the second premise of the rule we have \( \Gamma \vdash !x[\Sigma] := t; B.C : s. \) Therefore by the conversion rule we get \( \Gamma \vdash P : !x[\Sigma] := t; B.C. \)
- The reduction step was \( \Gamma \vdash !x[\Sigma] := t; B.P \rightarrow \beta_\delta !x[\Sigma]' := t'; B'.P \)
with exactly one of the pairs \((\Sigma, \Sigma'), (t, t')\) and \((B, B')\) contains different terms. Then \( \Gamma, x[\Sigma] := t; B \rightarrow \beta_\delta, \Gamma, x[\Sigma]' := t'; B' \)
and from the induction hypothesis for (2) we get
\[ (*) \quad \Gamma, x[\Sigma]' := t'; B' \vdash P : C \]
Similarly we have \( \Gamma \vdash !x[\Sigma] := t; B.C \rightarrow \beta_\delta x[\Sigma]' := t'; B'.C \) and by the induction hypothesis on the second premise we get
\[ (**) \quad \Gamma \vdash !x[\Sigma]' := t'; B'.C : s \]
Applying (let) to \((*)\) and \((**)\) we get \( \Gamma \vdash !x[\Sigma]' := t'; B'.P : !x[\Sigma]' := t'; B'.C. \) But by the conversion rule using the second
premise of the rule we have \( \Gamma \vdash !x[\Sigma]' := t'; B'.P : !x[\Sigma] := t; B.C. \)
- The reduction step was \( \Gamma \vdash !x[\Sigma] := t; B.P \rightarrow \beta_\delta x[\Sigma] := t; B.P'. \)
Then \( \Gamma, x[\Sigma] := t; B \vdash P \rightarrow \beta_\delta P' \) and by induction \( \Gamma, x[\Sigma] := t; B \vdash P' : C. \) Applying (let) we get \( \Gamma \vdash !x[\Sigma] := t; B.P' : !x[\Sigma] := t; B.C. \)

(2) Straightforward application of the induction hypothesis.

\[
(\text{app}): \quad \Gamma \vdash P : \Pi x[\Delta] : A.B \quad \Gamma, \Delta \vdash Q : A \\
\Gamma \vdash P : \langle \Delta \rangle Q : B \{ x[\Delta] := Q \}
\]

(1) If the redex being contracted is in \( P, \Delta \) or \( Q, \) then we can simply apply the induction hypothesis. If the redex is \( P : \langle \Delta \rangle Q \) itself then \( P = \lambda y[\Theta] : C.R, \) reduces to \( R \{ y[\Delta] := Q \}. \) Since \( \Gamma \vdash \lambda y[\Theta] : C.R : \Pi x[\Delta] : A.B \) is derivable, then we go up this derivation
until the node in which the \( \lambda \) was introduced:

\[
\Gamma', y[\Theta] : C \vdash R : D \quad \Gamma' \vdash \Pi y[\Theta] : C.D : s \\
\Gamma' \vdash (\lambda y[\Theta] : C.R) : \Pi y[\Theta] : C.D
\]

where \( \Gamma' \) is an initial segment of \( \Gamma \) and \( B \) is convertible to \( D. \) Using
weakening we get \( \Gamma', y[\Theta] : C \vdash R : D \) and by Substitution we get
\[ \Gamma \vdash R\{x[\Delta] := Q\}; D\{x[\Delta] := Q\} \quad \text{which (if necessary using the conversion rule) yields } \Gamma \vdash R\{x[\Delta] := Q\}; B\{x[\Delta] := Q\} \]

(\textbf{weak-var}): \[
\begin{array}{c}
\Gamma \vdash M : B \\
\Gamma, \Delta \vdash A : s \\
\Gamma, \Delta_i \vdash A_i : s_i \\
\hline
\Gamma, x[\Delta]: A \vdash M : B
\end{array}
\]

(1) Straightforward application of the induction hypothesis.
(2) Suppose \( \vdash \Gamma, x[\Delta]: A \rightarrow_{\beta} \Gamma', x[\Delta'] : A' \). We have several cases:
• If \( \Delta = \Delta' \) and \( A = A' \), we are done by the induction hypothesis.
• Let \( \Gamma = \Gamma' \) and \( \Delta = \Delta' \). From the induction hypothesis (1) we have \( \Gamma, \Delta \vdash A' : s \). Applying (\textbf{weak-var}) we get \( \Gamma, x[\Delta]: A\vdash M : B \).
• Let \( \Gamma = \Gamma' \) and \( A = A' \). Then we have \( \vdash \Gamma, \Delta \rightarrow_{\beta} \Gamma, \Delta' \) and
\( \vdash \Gamma, \Delta_i \rightarrow_{\beta} \Gamma, \Delta_i' \). By the induction hypothesis (2) we get
\( \Gamma, \Delta' \vdash A : s \) and \( \Gamma, \Delta_i' \vdash A_i : s_i \). Therefore by (\textbf{weak-var}) we have
\( \Gamma, x[\Delta'] : A \vdash M : B \).

(\textbf{weak-def}): \[
\begin{array}{c}
\Gamma \vdash M : B \\
\Gamma, \Delta \vdash N : A : s \\
\Gamma, \Delta_i \vdash A_i : s_i \\
\hline
\Gamma, x[\Delta] := N; A \vdash M : B
\end{array}
\]

(1) Straightforward application of the induction hypothesis.
(2) Suppose \( \vdash \Gamma, x[\Delta] := N; A \rightarrow_{\beta} \Gamma', x[\Delta'] := N'; A' \). We have several cases:
• If \( \Delta = \Delta' \) \( N = N' \) and \( A = A' \), we are done by the induction hypothesis.
• Let \( \Gamma = \Gamma' \) and \( \Delta = \Delta' \). From the induction hypothesis (1) we have \( \Gamma, \Delta \vdash A' : s \) and \( \Gamma, \Delta \vdash N' : A \). By (\textbf{conv}) this becomes \( \Gamma, \Delta \vdash N' : A' \). Applying (\textbf{weak-def}) we get
\( \Gamma, x[\Delta] := N'; A' \vdash M : B \).
• Let \( \Gamma = \Gamma' \) and \( A = A' \). Then we have \( \vdash \Gamma, \Delta \rightarrow_{\beta} \Gamma, \Delta' \) and
\( \vdash \Gamma, \Delta_i \rightarrow_{\beta} \Gamma, \Delta_i' \). By the induction hypothesis (2) we get
\( \Gamma, \Delta' \vdash A : s \) and \( \Gamma, \Delta_i' \vdash A_i : s_i \). Therefore by (weak-var) we have
\( \Gamma, x[\Delta'] : A \vdash M : B \).

(\textbf{axiom}), (\textbf{let-form}), (\textbf{If-form}), (\textbf{lambda}), (\textbf{conv}):
Straightforward, using the induction hypothesis.
\[ \square \]

\textbf{Definition 4.73 (Functional specification).} A specification \( \lambda S = \langle S, A, R, P \rangle \) is called functional if:
• for all sorts \( s_1, s_2, s' \) and \( s'' \) if \( (s_1, s_2, s') \in R \) and \( (s_1, s_2, s'') \in R \) then \( s' = s'' \);
• for all sorts \( s_1, s' \) and \( s'' \) if \( (s_1, s') \in A \) and \( (s_1, s'') \in A \) then \( s' = s'' \).

\textbf{Lemma 4.74 (Uniqueness of types).} If \( \lambda S \) is functional, \( \Gamma \vdash M : A \) and \( \Gamma \vdash M : B \) then \( A =_{\beta} B \).
PROOF. By induction on $M$ using Generation. The functionality condition is used when proving the uniqueness of the types of $\Pi$-terms and sorts.

Let $\lambda S^P$ be $\langle S, A, R, P \rangle$ and $\lambda S'$ be $\langle S', A', R' \rangle$. Define by induction the mapping $| - |$ from $\lambda S^P$ into $\lambda S'$:

$$|s|_\Gamma = s$$
$$|x(\Theta)|_\Gamma = x(\lambda(\Theta_1|_\Gamma|_\Gamma_1, \ldots, \lambda(\Theta_n|_\Gamma|_\Gamma_n)$$
$$|c(\Theta)|_\Gamma = (\lambda(\Delta|_\Gamma|_\Gamma)|_\Gamma, (\lambda(\Delta|_\Gamma|_\Gamma)|_\Gamma)|_\Gamma, \ldots, (\lambda(\Delta|_\Gamma|_\Gamma)|_\Gamma)|_\Gamma)$$

$$|\lambda x[\Delta]: A M|_\Gamma = \lambda x: (\Pi(\Delta|_\Gamma|_\Gamma) A) |_\Gamma, M |_\Gamma, \Delta|_\Gamma$$

$$|\Pi x[\Delta]: A M|_\Gamma = \Pi x: (\Pi(\Delta|_\Gamma|_\Gamma) A) |_\Gamma, M |_\Gamma, \Delta|_\Gamma$$

$$|P \cdot \Delta Q|_\Gamma = |P|_\Gamma, (\lambda(\Delta|_\Gamma|_\Gamma)|_\Gamma, Q|_\Gamma)$$

$$|c[\Delta] = t A M|_\Gamma = (\lambda c: (\Pi(\Delta|_\Gamma|_\Gamma) A) |_\Gamma, M |_\Gamma, c|_\Gamma, \Delta|_\Gamma)$$

where in the third clause we have $\Gamma \equiv \Gamma', c[\Delta] = t A, \Gamma''$.

Assuming that in $\lambda S'$ we can form all $\Pi$-types necessary to type the extra redexes created by $| - |$, we can show that this map preserves typing in the sense that if $\Gamma \vdash \lambda S^P M : A$ then $|\Gamma|_\Gamma \vdash \lambda S' M : A|_\Gamma$. Furthermore, $\beta$-redexes are mapped into $\beta$-redexes and $\delta$-redexes into $\beta$-redexes and $| - |$ commutes with $\beta \delta$, i.e. one can show that if $\Gamma \vdash M \rightarrow \beta \delta N$ then $|M|_\Gamma \rightarrow \beta \delta |N|_\Gamma$. Hence, we can conclude that if $\lambda S'$ contains enough $\Pi$-types and is strongly normalizing, then $\lambda S^P$ will also be strongly normalizing, because if we assume that $\lambda S^P$ has an infinite reduction path induces an infinite reduction path in $\lambda S'$ through the map.

One way to ensure that $\lambda S'$ can accommodate all necessary type would be to require that $S \subseteq S'$, $A \subseteq A'$ and $R \subseteq R'$ and for each $s_1, s_2 \in S'$ there is an $s_3 \in S'$ such that $(s_1, s_2, s_3) \in R'$. Weaker conditions are also possible.

Corollary 4.75. The systems of the $\lambda$-cube extended with hereditary parameters are strongly normalizing.

Proof. These systems can be embedded in the Extended Calculus of Constructions [Luo90].
The Formulas-as-Types Embedding

In this chapter we discuss the extension of the formulas-as-types embedding (also known as the Curry-Howard-de Bruijn embedding) to the logic with incomplete terms and proofs.

5.1. Introduction

Brouwer, Heyting and Kolmogorov’s interpretation of a proof of an implication \( \varphi \rightarrow \psi \) in intuitionistic logic (BHK-interpretation) [TvD88] is a construction that gives a proof of \( \varphi \) produces a proof of \( \psi \). Similarly, a proof of \( \forall x.\varphi(x) \) is a construction that gives any term \( t \) produces a proof of \( \varphi(t) \). This idea was formalized first by Kleene in his recursive realizability interpretation of intuitionistic number theory [Kle45]. Later the work of Curry [CF68] and Howard [How80] revealed the correspondence between provability in intuitionistic propositional logic and type inhabitation in Church’s simple type theory. This correspondence, known as the formulas-as-types isomorphism, identifies logical propositions with simple types and proofs with \( \lambda \)-terms in such a way that every proof of a proposition can systematically be mapped into a term inhabiting the type that the proposition is mapped to and vice versa. Furthermore, Howard showed that the reduction of proofs studied by Prawitz [Pra65] corresponds to \( \beta \)-reduction of the \( \lambda \)-term that encode the proof.

The practical use of the formulas-as-types interpretation dates back to 1970s and the AUTOMATH project of de Bruijn [dB80] who was the first to apply it on a large scale for representing mathematical structures as terms. For this reason the formulas-as-types interpretation is sometimes called the Curry-Howard-de Bruijn embedding. Since de Bruijn’s AUTOMATH a myriad of systems have emerged that in different degrees exploit the formulas-as-types interpretation. This has motivated the extension of the interpretation to stronger systems like second and higher-order predicate logics. Sometimes the correspondence is weaker as instead of isomorphism there is only an embedding from the logic into the typing system. As an example we can point out the logics corresponding to the systems of the Barendregt’s cube [Bar92] studied by Geuvers [Geu93].

Constable [Con83] proposes another view on the formulas-as-types interpretation that allows its wider application in computer science. He suggests that a proposition can be viewed as a specification and its proof can be seen as a program that implements this specification.

In this chapter we will be interested in extending the embedding of HOL into AHOL to an embedding of the logic with incomplete terms and proofs (bHOL)
to its corresponding typing system ($\lambda bHOL$). After showing that this embedding is sound and complete (Section 5.2) we will see that in its original formulation, the cut elimination does not directly correspond to $\beta$-reduction. We discuss the reasons and suggest how this correspondence can be regained in Section 5.3. We will conclude the chapter with a comparison of the representation of unknowns by functional variables and parameterized meta-variables from the viewpoint of the formulas-as-types embedding (Section 5.4).

### 5.2. Embedding $bHOL$ into $\lambda bHOL$

In this section we show how $bHOL$ can be embedded into $\lambda bHOL$, effectively lifting the embedding from HOL into $\lambda HOL$ to the logic of open terms. The exposition follows closely the approach taken in [Gen93].

For each domain $U$ and object term $t$ of $bHOL$ we define their $\lambda bHOL$ translations $\langle U \rangle$ and $\langle t \rangle$ respectively as shown in Figure 5.1. The translation is pretty straightforward except for the fact that $\rightarrow$ and $\forall$ are encoded by $\Pi$. Although we use $\Pi$ in three different roles (functional domain, implication and universal quantifier) we will always be able to decode it using the typing information. To see that $\langle \cdot \rangle$ is properly defined on terms and domains we need to show that it maps them to well-typed terms. To type these terms we need a context that assigns types to their free variables. The context is defined in Figure 5.2. In the definitions

<table>
<thead>
<tr>
<th>$U$</th>
<th>$\langle U \rangle$</th>
<th>$t$</th>
<th>$\langle t \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Prop}$</td>
<td>$\epsilon$</td>
<td>$x^U$</td>
<td>$x$</td>
</tr>
<tr>
<td>$U \in B$</td>
<td>$U$</td>
<td>$\lambda x^U.t$</td>
<td>$\lambda x:\langle U \rangle,\langle t \rangle$</td>
</tr>
<tr>
<td>$U \rightarrow V$</td>
<td>$\Pi x:\langle U \rangle,\langle V \rangle$</td>
<td>$(f \ t)$</td>
<td>$\langle f \rangle \langle t \rangle$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\forall x^U.\varphi$</td>
<td>$\Pi x:\langle U \rangle,\langle \varphi \rangle$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\varphi \rightarrow \psi$</td>
<td>$\Pi p:\langle \varphi \rangle,\langle \psi \rangle$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$m^U[\ldots(\vec{x}^U_i)t_i \ldots]</td>
<td>m[\ldots(\vec{x}^U)\langle t_i \rangle \ldots]$</td>
</tr>
</tbody>
</table>

**Figure 5.1.** Translating domains and object terms to $\lambda bHOL$-terms.

<table>
<thead>
<tr>
<th>$U$</th>
<th>$\Gamma_U$</th>
<th>$t$</th>
<th>$\Gamma_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Prop}$</td>
<td>$\epsilon$</td>
<td>$x^U$</td>
<td>$\Gamma_U, x:\langle U \rangle$</td>
</tr>
<tr>
<td>$U \in B$</td>
<td>$U: \text{Type}$</td>
<td>$\lambda x^U.t$</td>
<td>$\Gamma_U \cup \Gamma_t \setminus {x:\langle U \rangle}$</td>
</tr>
<tr>
<td>$U \rightarrow V$</td>
<td>$\Gamma_U \cup \Gamma_V$</td>
<td>$(f \ t)$</td>
<td>$\Gamma_f \cup \Gamma_t$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\forall x^U.t$</td>
<td>$\Gamma_U \cup \Gamma_t \setminus {x:\langle U \rangle}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\varphi \rightarrow \psi$</td>
<td>$\Gamma_{\varphi} \cup \Gamma_{\psi}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$m^U[\ldots(\vec{x}^U_i)t_i \ldots]</td>
<td>m[\ldots(\vec{x}^U)\langle t_i \rangle \ldots]$</td>
</tr>
</tbody>
</table>

**Figure 5.2.** Computing the $\lambda bHOL$ contexts for domains and object terms.

of the contexts $\Gamma_U$ and $\Gamma_t$ we use union of contexts. The union $\Gamma_1 \cup \Gamma_2$ is the
concatenation of $\Gamma_1$ and $\Gamma_2$ from which we have removed duplications and entries with the same name but different types or parameters. Formally it is defined as follows:

$$
\begin{align*}
\Gamma \cup \epsilon &= \Gamma \\
\Gamma \cup (\Gamma', x : A) &= (\Gamma \cup \Gamma'), x : A \quad x \notin \text{dom}(\Gamma) \\
\Gamma \cup (\Gamma', x : A) &= (\Gamma \setminus \{x : B\}) \cup \Gamma' \quad x : B \in \Gamma, A \neq B \\
\Gamma \cup (\Gamma', x : A) &= \Gamma \cup \Gamma' \quad x : A \in \Gamma \\
\Gamma \cup (\Gamma', m[\Delta] : A) &= (\Gamma \cup \Gamma'), m[\Delta] : A \quad m \notin \text{dom}(\Gamma) \\
\Gamma \cup (\Gamma', m[\Delta] : A) &= (\Gamma \setminus \{m[\Delta] : B\}) \cup \Gamma' \quad m[\Delta] : B \in \Gamma, (A \neq B \lor \Delta \neq \Delta') \\
\Gamma \cup (\Gamma', m[\Delta] : A) &= \Gamma \cup \Gamma' \quad m[\Delta] : A \in \Gamma
\end{align*}
$$

Often from the fact that $\Gamma_1$ and $\Gamma_2$ are valid contexts we will conclude that $\Gamma_1 \cup \Gamma_2$ is also valid. In general this is not true, but our contexts will not be arbitrary. Given a derivation in bHOL, a variable occurring in two different subtrees or terms has to have the same domain. Therefore, given a derivation that conforms to the variable conventions (see Chapter 3) it can not happen that a variable or meta-variable is defined in different ways in two different contexts that will be used to type subderivations or terms in it.

Here are some basic properties of $\langle \cdot \rangle$ that we will need later

**Property 5.1.** For each $M$ and $t$ we have

$$\langle M[t/x] \rangle \equiv \langle M \rangle\langle t/x \rangle$$

**Proof.** Straightforward induction on $M$. 

**Corollary 5.2.** If $M = _{\beta} N$ then $\langle M \rangle = _{\beta} \langle N \rangle$.

**Proof.** It suffices to show that if $M \rightarrow _{\beta} N$ then $\langle M \rangle \rightarrow _{\beta} \langle N \rangle$. To do that we proceed by induction on $M$. The interesting case is when $M$ is $(\lambda x U. P) Q$ and $N$ is $P[Q/x]$. Then $\langle (\lambda x U. P) Q \rangle = (\lambda x : \langle U \rangle. \langle P \rangle) \langle Q \rangle \rightarrow _{\beta} \langle P \rangle \langle Q/x \rangle$, but by Property 5.1 the last term is equal to $\langle P[Q/x] \rangle$ and that is of course $\langle N \rangle$. 

**Lemma 5.3** (Characterization of object contexts).

1. Let $U$ be a domain. Then $x : A \in \Gamma_U$ if and only if $A \equiv \text{Type}$ and there is a basic domain $V$ occurring in $U$ such that $\langle V \rangle \equiv x$.
2. Let $t$ be a term. Then $x : A \in \Gamma_t$ if and only if one of the following holds:
   - there is a variable $x U$ in $\text{FV}(t)$ such that $A \equiv \langle U \rangle$.
   - there is a variable $y U$ in $\text{FV}(t)$ such that $x : A \in \Gamma_U$.
   - $t$ contains a binder for $y U$ and $x : A \in \Gamma_U$.

**Proof.**

1. Induction on $U$. If $U$ is a basic domain then $\Gamma_U \equiv U : \text{Type}$. If $U$ is the domain $V_1 \rightarrow V_2$ then $\Gamma_U \equiv \Gamma_{V_1} \cup \Gamma_{V_2}$. It is clear that $\Gamma_U$ contains exactly the declarations of $\Gamma_{V_1}$ and $\Gamma_{V_2}$ because by inductions each of them only contains declarations like $x : \text{Type}$ where $x$ is a basic domain. If $x$ occurs in both contexts, it has the same type and is thus also in the union.
2. Induction on $t$. The cases when $t$ is application or abstraction illustrate the idea of the proof:
\[ t = (f \, s) \]; Then \( \Gamma_t \equiv \Gamma_f \cup \Gamma_s \). It suffices to show that \( x : A \in \Gamma_t \) if and only if \( x : A \in \Gamma_f \) or \( x : A \in \Gamma_s \). By the definition of \( \cup \) this is not the case only if it happens that \( x : A_1 \in \Gamma_f \), \( x : A_2 \in \Gamma_s \) and \( A_1 \neq A_2 \). Since \( x : A_1 \in \Gamma_f \), by induction we have three possible cases. If \( x^U \in \text{FV}(f) \) then \( A_1 \equiv \{ U \} \) and (because the set of basic domains is disjoint from the object variables) \( x^V \in \text{FV}(s) \) and hence \( A_2 \equiv \{ V \} \). But \( x \) may have only one domain, hence \( U \equiv V \) and therefore \( A_1 \equiv A_2 \).

In the second case we have that there is \( y^U \) such that \( y^U \in \text{FV}(f) \) and \( x : A_1 \in \Gamma_U \). But then \( x \) must be a basic domain and \( A_1 = \text{Type} \) by (1). If \( x \) is a basic domain and \( x : A_2 \in \Gamma_s \), then \( A_2 = \text{Type} \).

The last case when there is a binder for \( y^U \) in \( f \) and \( x : A_1 \in \Gamma_U \) is handled similarly.

The converse direction is straightforward.

\[ t = \lambda y^U.M : \text{Then } \Gamma_t \equiv \Gamma_U \cup \Gamma_M \setminus \{ y : \{ U \} \}. \]

Let \( x : A \in \Gamma_t \). We have two cases: If \( x : A \in \Gamma_U \) we have the third option because \( \lambda y^U.M \) contains a binder for a variable of domain \( U \). If \( x : A \in \Gamma_M \setminus \{ y : \{ U \} \} \) we can apply the induction hypothesis for \( M \).

For the opposite direction, the interesting case is the fresh binder \( \lambda y^U.M \). We need to show that every element of \( \Gamma_U \) is contained in \( \Gamma_t \), but this is the case because by definition \( \Gamma_t \equiv \Gamma_U \cup \Gamma_M \setminus \{ y : \{ U \} \} \).

\[ \square \]

The characterization of the typing contexts provided by Lemma 5.3 is useful for showing the following soundness result for the translation of object terms:

**Lemma 5.4.**

1. If \( U \) is a domain then \( \Gamma_U \vdash \{ U \} : \text{Type} \).
2. Let \( t \) be a term of domain \( U \). Then \( \Gamma_t \vdash \{ t \} : \{ U \} \)^{\text{type}}

**Proof.** Induction on the structure of \( U \) and \( t \) respectively. Below we consider some cases of the proof of (2).

\[ t = x^U : \text{From (1) we have } \Gamma_U \vdash \{ U \} : \text{Type}. \text{ Then by the start rule we get } \Gamma_U, x : \{ U \} \vdash x : \{ U \} \]

\[ t = \lambda x^U.M : \text{By (1) and the induction hypothesis we have } \Gamma_U \vdash \{ U \} : \text{Type} \text{ and } \Gamma_M \vdash \{ M \} : \{ V \} : \text{Type} \text{ where } V \text{ is the domain of } M. \text{ In this case we have } \Gamma_t \equiv \Gamma_U \cup \Gamma_M \setminus \{ x : \{ U \} \}. \text{ As } x \text{ does not occur in } \Gamma_U \text{ (Lemma 5.3) and in } \Gamma_M \setminus \{ x : \{ U \} \}, \text{ the context } \Gamma_t, x : \{ U \} \text{ is valid. It is also a weakening of } \Gamma_M. \text{ Therefore we have } \Gamma_t, x : \{ U \} \vdash \{ M \} : \{ V \}. \text{ As it is easy to see that } \Gamma_t \vdash \Pi x : \{ U \}. \{ V \} : \text{Type}, \text{ we can apply the rule } (\lambda) \text{ to get } \Gamma_t \vdash \lambda x : \{ U \}. \{ M \} : \Pi x : \{ U \}. \{ V \}. \text{ This of course means that } \Gamma_t \vdash \{ \lambda x^U.M \} : \{ U \rightarrow V \}. \]

\[ t = m[x \, t_1 : U_1, \ldots, x_n : U_n : t_n] : \text{Let } m^U[p_1^{x_1^{U_1}} \ldots p_n^{x_n^{U_n}}] \text{ be the declaration of } m \text{ and let } t_i \text{ be of domain } V_i. \text{ By induction we have that } \Gamma_t \vdash \{ t_i \} : \{ U_i \}. \text{ Because of Lemma 5.3 the context } \Gamma_t, x_i : \{ U_i \} \text{ is valid.} \]
It is a weakening of $\Gamma_i$ and therefore we have $\Gamma_i, \vec{x}_i ; (\| U_i \|) \vdash (\| t_i \|) : (\| U_i \|)$. Then using the typing rule for meta-variable instances we have

$$\Gamma_i \vdash m[\langle \vec{x}_i ; (\| U_i \|) \rangle (\| t_1 \|) \ldots (\| t_n \|)] : (\| U \|)$$

\[ \square \]

**Definition 5.5.** Let $\Gamma$ be a valid context. Define $B_\Gamma$ as the set

$$\{ x \mid x : \text{Type} \in \Gamma \}$$

**Lemma 5.6 (Surjectivity of $\langle \cdot \cdot \rangle$ on domains and terms).**

1. If $\Gamma \vdash A : \text{Type}$ then there is a domain $U$ in bHOL with basic domains $\mathcal{B}_U$ such that $\langle U \rangle \equiv A$.
2. If $\Gamma \vdash M : A : \text{Type}$ then there is a term $t$ and a domain $U$ in bHOL with basic domains $\mathcal{B}_U$ such that $\langle t \rangle \equiv M$ and $\langle U \rangle \equiv A$.

**Proof.** Induction on $A$ and $M$ respectively using Generation Lemma for $\lambda$HOL (Lemma 4.36). \[ \square \]

After establishing that $\langle \cdot \cdot \rangle$ is sound and complete on the level of domains and object terms, we can focus our attention to the proofs. Next we translate the bHOL derivations to proof terms in $\lambda$HOL. Given a derivation $\Sigma$ we define its translation $\langle \Sigma \rangle$ together with a context $\Gamma_\Sigma$ that will be used to type it (see Lemma 5.8 below). The definitions of $\langle \Sigma \rangle$ and $\Gamma_\Sigma$ for the standard rules of HOL are presented in Figure 5.3 and the case of the claim rule in Figure 5.4. Please note that the translation of the instances of the claim rule is context-sensitive.

The same instance will give rise to a different goal when occurring in different derivations, because variables that occur free in it may get bound later in the derivation. This shows that the inductive definitions in Figure 5.3 should be seen as inductions on the subderivations of a given proof rather than as induction on the derivation itself. On that figure we also define the notion of skeleton for a proof (i.e. $\langle \Sigma \rangle$). It will be used later (see Definition 5.10) to define a notion of equivalence on proofs.

As shown in Figure 5.3, we assume that all formulas at the leaves of a proof are labelled, not only the discharged ones. This is done for technical reasons and given a derivation with the standard labelling we can always choose suitable labels.

**Lemma 5.7 (Correctness of proof contexts).** Let $\Sigma$ be a derivation.

1. $\varphi^i$ is an undischarged assumption of $\Sigma$ if and only if there is $A$ such that $i : A \in \Gamma_\Sigma$, $\Gamma_\Sigma \vdash A : \text{Prop}$ and $A \equiv \langle \varphi \rangle$.
2. $x^U$ is a free variable of $\Sigma$ if and only if $x : \langle U \rangle \in \Gamma_\Sigma$.
3. The object meta-variable $m^U[\Delta]$ occurs in $\Sigma$ if and only if $m[\langle \Delta \rangle ; \langle U \rangle] \in \Gamma_\Sigma$.
4. $\vec{x}^U$, $\{ \vec{x}_1^U, \vec{p}_1^U : A_1 \} B_1 \ldots B_n$ is a goal of $\Sigma$ if and only if there is a proof meta-variable $m[\vec{x}^U, \vec{p}_1^U] ; p_1[\vec{x}_1^U, \vec{p}_1^U] ; \vec{q}_1[: A_1] ; B_1 \ldots : \vec{\psi} \in \Gamma_\Sigma$.

**Proof.** By induction on $\Sigma$ inspecting the definitions of $\Gamma_\Sigma$ and $\langle \| t \| \rangle$. In the case of the introduction rule for the universal quantifier we need to make sure that $(\Gamma_\Sigma \cup \Gamma_U) \setminus \{ i : \langle U \rangle \}$ is valid. The problem could potentially occur
if $x$ occurs in it. To see that this does not happen we consider 4 cases given by the induction hypothesis for $\Sigma_1$. In the case when $i:(\varphi \vdash) \in \Gamma_{\Sigma_1}$ we use Lemma 5.3 and the side condition for the introduction rule that states that $x$ does not occur in the undischarged assumptions of $\Sigma_1$. The other three cases are straightforward. □

**Lemma 5.8 (Soundness).** If $\Sigma$ is a bHOL derivation of $\varphi$ then $\Gamma_{\Sigma} \vdash (\varphi \vdash)$.

**Proof.** Induction on the subderivations. Consider the cases for the last rule of the subderivation $\Sigma$:

**axiom:** As $\varphi$ is a term of domain $\text{Prop}$, we have $\Gamma_{\varphi} \vdash (\varphi \vdash) : \text{Prop}$. Hence by the start rule we have $\Gamma_{\varphi}, i:(\varphi \vdash) \vdash i : (\varphi \vdash)$.

**→ elim:** By induction hypothesis we have

- $\Gamma_{\Sigma_1} \vdash (\Sigma_1) : (\varphi \rightarrow \psi)$
- $\Gamma_{\Sigma_2} \vdash (\Sigma_2) : (\varphi \vdash)$
Let $\vec{x}: U, \{x_1^{\vec{a}_1}, A_1\} \varphi_1 \cdots \{x_n^{\vec{a}_n}, A_n\} \varphi_n \vdash \varphi$ be the goal (see p. 52) corresponding to the instance of the (claim) rule.

Then we define $\Gamma_\Sigma$ as the context

$$\bigcup_{j=1}^n \Gamma_j \cup \Gamma_\varphi \cup \{m[\vec{x}: U], p_1[\Delta_1](\varphi_1), \ldots, p_n[\Delta_n](\varphi_n)\}$$

where $\Gamma_j \equiv (\Gamma_{\Sigma_j} \cup \Gamma_{\bar{\varphi}_j} \cup \Gamma_{\bar{A}_j}) \setminus \Delta_j$ and $\Delta_j \equiv \vec{x}_j : ((\bar{U}_j), \bar{\varphi}_j, \bar{A}_j)$. If $\Sigma_i$ are the subderivations in the premises of the rule, we define $\langle \Sigma \rangle$ as

$$\frac{\{\Sigma_1\} \varphi_1 \cdots \{\Sigma_n\} \varphi_n}{\varphi}$$

$\langle \Sigma \rangle$ is defined as the term $m[\vec{x}, (\Delta_1)(\varphi_1) \ldots (\Delta_n)(\varphi_n)]$

**Figure 5.4.** Defining $\Gamma_\Sigma$, $(\Sigma)$ and $(\Sigma)$ when the last rule of $\Sigma$ is (claim).

Because $\Sigma_1$ and $\Sigma_2$ are subderivations of $\Sigma$, the variables shared by $\Gamma_{\Sigma_1}$ and $\Gamma_{\Sigma_2}$ have the same type and therefore $\Gamma_{\Sigma_1} \cup \Gamma_{\Sigma_2}$ is a weakening of both $\Gamma_{\Sigma_1}$ and $\Gamma_{\Sigma_2}$. Thus $\Gamma_{\Sigma_1} \vdash (\varphi_1)$ and $\Gamma_{\Sigma_2} \vdash (\varphi_2)$. Since $(\varphi_1 \rightarrow \varphi_2)$ is true by weakening, if it is in $\Gamma_{\Sigma}$, no other element of the context contains free occurrences of $\varphi$.

By the inference rule $\lambda$ we have $\Gamma_{\Sigma} \vdash \lambda \bar{x} : (\varphi), (\Sigma_1 \vdash (\varphi_1))$.

$(\forall \text{ elim}), (\forall \text{ intro})$: Similar to the rules for $\rightarrow$ In the case of the introduction rule we use the fact that $x^U$ does not occur free in the undischarged hypotheses of $\Sigma$.

$(\text{conv})$: By induction $\Gamma_{\Sigma_1} \vdash (\varphi_1)$. As before we can show that $\Gamma_{\Sigma_1} \cup \Gamma_{\varphi}$ is a weakening of $\Gamma_{\varphi}$ and $\Gamma_{\Sigma_1}$. Hence we have $\Gamma_{\Sigma} \vdash (\varphi) : \text{Prop}$ and $\Gamma_{\Sigma} \vdash (\Sigma_1 \vdash (\varphi))$. As $\Gamma_{\Sigma}$ preserves $\equiv$, we have also $(\varphi) = (\psi)$.

Therefore by the conversion rule of $\text{AbHOL}$ we have $\Gamma_{\Sigma} \vdash (\Sigma) : (\psi)$.

$(\text{claim})$: Let $\vec{x}: U, \{x_1^{\vec{a}_1}, A_1\} \varphi_1 \cdots \{x_n^{\vec{a}_n}, A_n\} \varphi_n \vdash \varphi$ be the goal corresponding to the instance of the (claim) rule and let $\Sigma_i$ be the derivations in its premises. By induction we have $\Gamma_{\Sigma_i} \vdash (\varphi_i)$. Because of the side-conditions in the (claim) rule, $\vec{x}_j$ may not appear free in any undischarged assumptions of $\Sigma_j$ other than $A_j$ (these are discharged by the (claim) rule before abstracting $\vec{x}_j$). Therefore we may assume that $\Gamma_{\Sigma_j}$ is of the form $\Gamma_{\Sigma_j} \equiv (\Gamma, \vec{x}_j : (\bar{U}_j), \bar{\varphi}_j, \bar{A}_j)$. As before, we can weaken the context to get $\Gamma_{\Sigma} \vdash \vec{x}_j : ((\bar{U}_j), \bar{\varphi}_j, \bar{A}_j) \vdash (\Sigma_j) : (\varphi_j)$. Furthermore, the
free variables $\vec{x}^{\vec{M}}$ have declaration in $\Gamma_{\Sigma}$ and $\Gamma_{\Sigma} \vdash \vec{x} : \langle U \rangle$. Therefore by the rule (start),$_{\mathcal{M}}$ we have

$$\Gamma_{\Sigma} \vdash m[\vec{x}, (\vec{x}_1: \langle U_1 \rangle, \vec{x}_{i_1}: \langle A_{i_1} \rangle) \ldots (\vec{x}_n: \langle U_n \rangle, \vec{x}_{i_n}: \langle A_{i_n} \rangle) (\Sigma_n)] : \langle \varphi \rangle$$

□

Next, we need to show that every inhabitant of a propositional type in $\lambda bHOL$ corresponds to a proof in $bHOL$. We do that by defining a mapping $[-]_\Gamma$ from the $\lambda bHOL$ terms typable in a context $\Gamma$ to the different elements of $bHOL$. Figure 5.5 shows how to map the object types to HOL domains and the terms of object types to object terms in $bHOL$.

<table>
<thead>
<tr>
<th>$U$</th>
<th>From Gen. Lemma</th>
<th>$[U]_\Gamma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U$</td>
<td>$U : \text{Type} \in \Gamma$</td>
<td>$[U]<em>\Gamma \rightarrow [V]</em>{\Gamma, x : U}$</td>
</tr>
<tr>
<td>$\Pi x : U, V$</td>
<td>$\Gamma \vdash U : \text{Type}$</td>
<td>$x : U \in \Gamma$</td>
</tr>
<tr>
<td>$t$</td>
<td>$\Gamma, x : U \vdash V : \text{Type}$</td>
<td>$[t]_\Gamma$</td>
</tr>
<tr>
<td>$x$</td>
<td>$\lambda x : A.M$</td>
<td>$x : U \in \Gamma$</td>
</tr>
<tr>
<td>$M N$</td>
<td>$m[\Theta_1 M_1 \ldots]$</td>
<td>$[M]<em>\Gamma [N]</em>\Gamma$</td>
</tr>
<tr>
<td>$\Pi x : A.B$</td>
<td>$\Gamma \vdash A : \text{Prop}$</td>
<td>$\lambda x[A]<em>\Gamma \rightarrow [M]</em>{\Gamma, x : A}$</td>
</tr>
<tr>
<td>$\Pi x : A.B$</td>
<td>$\Gamma \vdash A : \text{Type}$</td>
<td>$\Gamma, x : A \vdash B : \text{Prop}$</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>$\forall x \in [A]<em>\Gamma, [B]</em>\Gamma$</td>
<td>$\Theta$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\Theta, x : A$</td>
<td>$\Theta, x : A : \text{Type}$</td>
</tr>
<tr>
<td>$\Theta, x : A$</td>
<td>$\Gamma, \Theta \vdash A : \text{Type}$</td>
<td>$[\Theta]<em>\Gamma, x : [A]</em>\Gamma, \Theta$</td>
</tr>
</tbody>
</table>

Figure 5.5. Translating the domains, the object terms and parameter lists from $\lambda bHOL$ to $bHOL$, assuming that $\Gamma \vdash U : \text{Type}$ and $\Gamma \vdash t : V : \text{Type}$ for some $V$.

The following lemma shows that this translation distributes over substitution and hence preserves $\beta$-reduction.

**Lemma 5.9.**

1. Let $\Gamma, x : U \vdash M : V : \text{Type}$ and $\Gamma \vdash N : U : \text{Type}$. Then $[M[N/x]]_\Gamma \equiv [M]_\Gamma [[N]_\Gamma/x]$.  
2. Let $\Gamma \vdash M : U : \text{Type}$ and $M \rightarrow_\beta N$. Then $[M]_\Gamma \rightarrow_\beta [N]_\Gamma$.

For each derivable judgment $\Gamma \vdash \varphi$ we define a derivation denoted $[\Gamma \vdash M : \varphi]_\Gamma$ by induction on $M$ using Generation Lemma as shown in Figure 5.6. Note that the derivation $[\Gamma \vdash M : A]_\Gamma$ is influenced by the choice of types provided by the Generation Lemma. Derivations that differ only by a permutation of the conversion rule are mapped to the same $\lambda$-term. This fact was used in [Geu93] to
define a notion of skeleton (Σ) and a relation on proofs ≈ identifying those with the same skeleton. We will extend this relation to bHOL and show that up to ≈ the mapping [·] is well-defined, ⟨·⟩ is injective and [·] is its inverse.

**Definition 5.10.** Let Σ₁ and Σ₂ be bHOL derivations. Define Σ₁ ≈ Σ₂ if and only if (Σ₁) ≡ (Σ₂) (see Figure 5.3 and Figure 5.4).

**Lemma 5.11.** Let Γ ⊢ M : A and Γ ⊢ M : B. Then [Γ ⊢ M : A] ≈ [Γ ⊢ M : B].

**Proof.** Induction on M using the fact that ⟨[Γ ⊢ M : A]⟩ is determined by M and Γ only. □

**Lemma 5.12.** Let Γ ⊢ M : A and let Γ’ be a valid weakening of Γ. Then

\[ [M]_{Γ'} = [M]_Γ \]

Given a valid context Γ, define [Γ] as the set of formulas

\[ \{ [φ]_Γ : \exists x : φ ∈ Γ, Γ ⊢ φ : \text{Prop} \} \]

**Lemma 5.13 (Completeness).** If Γ ⊢ M : A : Prop then [Γ ⊢ M : A] is a well-formed bHOL derivation of \([A]_Γ\) with assumptions among [Γ].
5. THE FORMULAS-AS-TYPES EMBEDDING

Proof. The proof is rather straightforward induction on the derivation of \( \Gamma \vdash M : A \). In the cases when the last rule is (\textit{app}) or (\textit{\lambda}) we need to consider two cases depending on whether the \( \Pi \)-types involved encode implication or universal quantifiers. One can easily see that all side conditions in the logical rules are met because the context \( \Gamma \) is valid.

What remains to be shown is that the embedding of bHOL into \( \lambda \text{bHOL} \) given by

\[
CH : \Sigma \longrightarrow \Gamma_{\Sigma} \vdash \emptyset_{\Sigma} : (\emptyset_{\Sigma})
\]

(where \( \emptyset_{\Sigma} \) is the conclusion of \( \Sigma \)) is injective. This is a consequence of the following lemma:

Lemma 5.14.

1. If \( U \) is a domain then \( \|U\|_{\Gamma_{U}} \equiv U \).
2. If \( t \) is a term of domain \( U \) then \( \|t\|_{\Gamma_{U}} \equiv t \).
3. For each bHOL derivation \( \Sigma \) with conclusion \( \varphi \) we have \( \|\Gamma_{\Sigma} \vdash \emptyset_{\Sigma} : (\emptyset_{\Sigma}) \| \sim \Sigma \)

Proof.

1. If \( U \) is a basic domain then \( \Gamma_{U} \equiv U : \text{Type} \) and \( \|U\| \equiv U \) and \( \|U\|_{\Gamma_{U}} = U \). For a functional domain \( U \rightarrow V \) we have \( \|U \rightarrow V\| \equiv \Pi x : \|U\|, \|V\| \) and \( \Gamma_{U \rightarrow U} \equiv \Gamma_{U} \cup \Gamma_{V} \). Then

\[
\Pi x : \|U\|, \|V\| \Gamma_{U \rightarrow U} \equiv \Pi x \|U\|_{\Gamma_{U} \cup \Gamma_{V}} \rightarrow \Pi x \|V\|_{\Gamma_{U} \cup \Gamma_{V}}
\]

But \( \Gamma_{U \rightarrow U} \) is a weakening of \( \Gamma_{U} \) and \( \Gamma_{U} \vdash \emptyset_{\Sigma} : \text{Type} \). Hence

\[
\Pi x : \|U\|, \|V\| \Gamma_{U \rightarrow U} \equiv \Pi x \|U\|_{\Gamma_{U}} \rightarrow \Pi x \|V\|_{\Gamma_{V}} \equiv U \rightarrow V
\]

2. Induction on the structure of the terms. To give the reader a flavour of the proof we treat three cases:
   \( t \equiv x^U \): By definition \( \Gamma_{t} \equiv \Gamma_{U}, x : \|U\| \). Then

\[
\|x^U\|_{\Gamma_{t}} \equiv x \|U\|_{\Gamma_{U}, x : \|U\|} \equiv x \|U\|_{\Gamma_{U}} \equiv x^U
\]

\( t \equiv MN \): Then

\[
\|MN\|_{\Gamma_{MN}} \equiv \Pi x \|M\|_{\Gamma_{M}}, \|N\|_{\Gamma_{N}} \equiv \Pi x \|M\|_{\Gamma_{M}} \Pi x \|N\|_{\Gamma_{N}}
\]

\( t \equiv m(\Theta_1 t_1 \ldots) \): Note that none of the variables in \( \|\Theta_1\| \) is propositional, therefore using the induction hypothesis we have

\[
\Pi x \|\Theta_1\|_{\Gamma_{\Theta_1}} \equiv m(\Pi x \|\Theta_1\|_{\Gamma_{\Theta_1}}, \Pi x \|\Theta_1\|_{\Gamma_{\Theta_1}}, \ldots) \equiv m(\|\Theta_1\|_{\Gamma_{\Theta_1}}, \ldots)
\]

3. Induction on \( \Sigma \).

Remark 5.15. In contrast to the case of terms (Lemma 5.6), \( \| - \| \) is not surjective on proofs, i.e. it is not the case that \( \| - \| \) is the inverse of \( \| - \| \) as shown by the following example. Let \( \Gamma \) be the context \( U : \text{Type}, f : U \rightarrow x : U, P : U \rightarrow \text{Prop}, t : (\Pi x : U.(P x)) \). Then if \( \Sigma \) is the derivation

\[
\Gamma \vdash \emptyset_{\Sigma} \equiv \Pi x \|U\|_{\Gamma_{U}}, \|f\|_{\Gamma_{U}}, \|t\|_{\Gamma_{U}} : (P(f)x)
\]

then \( \Gamma_{\Sigma} \vdash \emptyset_{\Sigma} : (P(f)x) \) is the judgment

\[
\Gamma, n_1 : P(fx) \vdash n_2 : (P(fx)) \]

\[
\Gamma, n_1 : P(fx) \vdash n_2 : (P(fx)) \]

\[
\Gamma : P(fx) \vdash \emptyset_{\Sigma} : (P(fx))
\]

\[
\Gamma, n_1 : P(fx) \vdash n_2 : (P(fx)) \]

\[
\Gamma : P(fx) \vdash \emptyset_{\Sigma} : (P(fx))
\]
Note the change in the parameters: the object-term parameter $z$ has disappeared, the type of the proof-parameter has changed and each occurrence of the meta-variable has now a separate declaration.

5.3. Proof Reduction and $\beta$-Reduction

Besides a mapping of proofs to $\lambda$-terms, the Curry-Howard embedding is also associated with the relation between proof reduction (cut elimination) and $\beta$-reduction. In this section we will see that the formulation of oHOL and bHOL from Chapter 3 does not allow us to encode enough information to get this correspondence. After discussing the reasons, we give some ideas how it could be regained.

The combination of an introduction rule and immediately following elimination rule is known as a cut. Using the rules for proof reduction (see Figure 5.7) we can remove cuts from proofs. Consider the derivations in Figure 5.8. Note that according to the proof reduction rules proof (1) reduces to proof (2). We see however that the lambda term $(\lambda x:U.\lambda p:Ax.m[x,p])t$ that represents (1) does not $\beta$-reduce to the term $\lambda p:At.m[p]$ that encodes (2). This happens because in the claim rule we are not explicit about the dependency of the unknown part on object terms.

The mismatch between the logic and the typing system comes from the fact that in a logical derivation a claim rule is only implicitly dependent on the bound variables that may occur free in it while in the typing system this dependency is made explicit by the need to declare a meta-variable before we use it. For example, if we drop the parameter $x$ in the declaration $m[x:U,p:A(x)]:A(x)$, we would get an occurrence of $x$ in $A(x)$ that is not declared. We cannot declare $x$ outside the declaration of $m$ because then we would make it a global variable and by the variable convention it will be different from the bound variable $x$ that actually occurs in the claim rule.

Hence the dependency on object terms is essential for the representation of unknown proofs in the typing system and we cannot drop it. Let us look again closer at what happens to the two derivations in Figure 5.8. If the claim rule at (1) were instantiated by a proof $\Sigma$, then the one at (2) after reduction was going...
to be $\Sigma[t/x]$. The problem is that in the logic we cannot denote this explicitly and therefore the rules in Figure 5.8 do not give us a proper notion of proof reduction.

To regain the connection between proof reduction and $\beta$-reduction we would need to modify the logic in two respects: the (claim) rule needs to be able to record object term dependencies and the notion of goal needs to be made explicit. A goal would be a reduction-independent description of the unknown at the moment of its introduction, we can think of it as of the declaration of a meta-variable in a typing context. Then every instance of the claim rule will have an associated goal of which it is an instance, very much like a meta-variable instance in a $\lambda$HOL or $\lambda$bHOL term.

Let us illustrate this by showing how the example in Figure 5.8 would be affected. In our presentation in Chapter 3 we inferred the goals from the derivation and in that sense they were implicit. The idea now is that the goals would be given first and then the claim rule would be an instance of a goal. The goal that corresponds to the claim rule instance in (1) is

$\Gamma \equiv U : \text{Type}, A : U \rightarrow \text{Prop}, t : U$

\begin{figure}
\centering
\begin{align}
\frac{[A(x)]^i}{\vdash \lambda x : U. \lambda p : Ax. m[x, p] : A(t) \rightarrow A(t)}
\end{align}
\end{figure}

The new claim rule would have to explicitly record all components of the goal, including the term $x$ as shown below:

\begin{align}
\Gamma, m[x : U, p : Ax] : Ax \\
\frac{\exists x^U. A(x) \rightarrow A(x)}{\vdash (\lambda x : U. \lambda p : Ax. m[x, p]) t : A(t) \rightarrow A(t)}
\end{align}

\begin{align}
\frac{[A(t)]^i}{\vdash \lambda p : At. m[p] : A(t) \rightarrow A(t)}
\end{align}

\begin{align}
\frac{\exists x^U. A(x) \rightarrow A(x)}{\vdash \lambda x : U. \lambda p : Ax. m[x, p] : A(t) \rightarrow A(t)}
\end{align}

\begin{align}
\frac{\lambda x : U. \lambda p : Ax. m[x, p]}{t}
\end{align}
This will allow us to record the substitution executed over a subderivation when performing a proof reduction step. For example, if we reduce this proof we get

\[
\begin{array}{c}
\frac{t}{A(t)} \quad [A(t)]^i \\
\frac{t \vdash A \tau}{A \tau \vdash A(t)} \\
\frac{t \vdash A(t)}{A \tau \vdash A(t)}^i
\end{array}
\]

where the \( x \) in the premise of the claim rule also was substituted. If we now do the translation to \( \lambda \)-terms, we get the following typing judgments:

\[
\begin{align*}
\Gamma, m \vdash \lambda x : \mathcal{X} \cdot A x & \tau \vdash (\lambda \mathcal{X} : \mathcal{U} \cdot \lambda \mathcal{X} \cdot \lambda \mathcal{X} m[x, p]) t : A(t) \rightarrow A(t) \\
\Gamma, m \vdash \lambda x : \mathcal{X} \cdot A x & \tau \vdash \lambda \mathcal{X} : \mathcal{U} \cdot \lambda \mathcal{X} m[t, p] : A(t) \rightarrow A(t)
\end{align*}
\]

Note that as we now know that the claim rule is an instance of a particular goal that does not change with proof reduction, we translate the second derivation using the meta-variable that represents the original goal and as expected we get that

\[
(\lambda \mathcal{X} : \mathcal{U} \cdot \lambda \mathcal{X} \cdot \lambda \mathcal{X} m[x, p]) t \rightarrow _\beta \lambda \mathcal{X} : \mathcal{U} \cdot \lambda \mathcal{X} m[t, p]
\]

One may ask the question why didn’t we introduce this form of the claim rule from the beginning and work with it all the way. The first reason is that we wanted that the logic of incomplete proofs and terms remains as intuitive as possible. The occurrence of terms in the premises of the claim rule is certainly not very intuitive for a logician. The second one is related: the main application of the logic of incomplete terms would be the interactive proof construction and not so much the study of the proof-theoretical properties of the underlying system. Although it is not inconceivable that proof reduction on incomplete proofs turns out to have interesting applications, we believe that its relevance to the process of interactive proof construction would probably be limited.

### 5.4. Are Terms with Free Variables Open Terms?

The idea that an unknown can be seen as a function of the variables that may occur free in it is not a new one. It has been used in the setting of unification problems like Huet’s unification algorithm [Hue72]; Miller’s [Mil92] higher-order unification problems under a mixed prefix or the proof synthesis method of Dowek [Dow93]. Therefore one may justifiably ask why do we need to introduce meta-variables for representing unknowns. In this section we answer this question by comparing our approach of representing unknowns by parameterized variables in type theory (i.e. meta-variables as introduced in Chapter 4) to the alternative approach where as representation of the unknowns we take functional variables. The comparison that we make is biased in the sense that we only look at the adequacy of the two approaches to represent open terms and proofs and the possibility to extend the Curry-Howard embedding. We do not make claims with respect to other applications.

A meta-variable $m[x:A]:B$ is a map taking $x$ as arguments and producing a result of type $B$. In that sense it can also be viewed as a function $f$ of type $\Pi x:A.B$. Its instances $m(t_1 \ldots t_n)$ of $m$ can be seen as full applications $(f t_1 \ldots t_n)$.

A binding meta-variable $m[p_1[x_1:A_1]:B_1 \ldots p_n[x_n:A_n]:B_n]:C$ can be seen as a function $f$ of type $\Pi p_i:(\Pi x_i:A_i.B_i).C$ and its instances $m[(x_1:A_1)t_1 \ldots (x_n:A_n)t_n]$ can again be viewed as full applications:

$$(f(\lambda x_1:A_1.t_1) \ldots (\lambda x_n:A_n.t_n))$$

It can be shown that this translation of the (hereditarily parameterized) meta-variables to (higher-order) functional variables commutes with the instantiation of meta-variables and the reduction behaviour of the open terms.

Therefore, meta-variables are function variables with respect to the object-level information that they carry. However, meta-variables carry also meta-level information that makes a difference in our setting.

5.4.2. Differences. The two approaches differ in a number of aspects that we discuss below:

With respect to the typing system needed in both cases we immediately see several differences. First, the functional approach requires that we are able to form the higher-order functional types in the first place. This may require the need to extend the object calculus with higher-order types with all its consequences. In particular this would be a serious problem for first-order systems or systems without product types (Conor Mcbride gives the example of a system with $\Sigma$-types only). Furthermore, if we allow the formation of more product types we may make the system inconsistent and this is not a desirable property for a calculus that is supposed to be used as a basis for a theorem prover.

The experience of this thesis shows that if we do not quantify over meta-variables (as is the case in the extensions $\lambda$-HOL and $\lambda$-HOL from Chapter 4) it seems to be relatively easy to verify that the calculus with open terms satisfies the desired meta-properties.

In applications to unification, the functional approach has as disadvantage the fact that it does not allow the lifting of first-order techniques such as narrowing to the higher-order case.

A third and maybe most relevant difference between the two approaches is found in the (im)possibility to lift the Curry-Howard-de Bruijn embedding of logical proofs to $\lambda$-terms from the object calculus to the calculus of open terms. Let us elaborate on this point:

In the logic with incomplete terms and proofs the unknown terms and proofs are 'first class citizens'. In particular this means that we have chosen to make a distinction between the object-level functional dependencies expressed by functions and the meta-level functional dependencies expressed by meta-variables and the claim rule. When we extend the Curry-Howard embedding to the logic with open
5.4. Are terms with free variables open terms?

Terms we would expect that on the level of the type theory this distinction is preserved (because by definition any embedding is an injective function).

Let us look at several examples. Consider the deductions shown in Figure 5.9.

\[
\begin{array}{c|c|c|c}
\text{?} & [A]^i & \text{?} \\
A \rightarrow B & \overline{B} & A \rightarrow B \\
(a) & (b) & (c)
\end{array}
\]

\[
\begin{align*}
h_a &: A \rightarrow B \vdash h_a &: A \rightarrow B \\
h_b &: A \rightarrow B \vdash \lambda x : A. h_b[x] &: A \rightarrow B \\
h_c &: A \rightarrow B \vdash \lambda x : A. (h_c[x]) & : A \rightarrow B
\end{align*}
\]

Their translations in parameterized and functional form are given at the bottom of the figure. There are two points of interest here. Consider the derivations \((a)\) and \((b)\) first. Derivation \((b)\) can be seen as a refinement of derivation \((a)\) because the claim rule in \((a)\) is replaced by an introduction rule and a claim rule in \((b)\). In \((a)\) our goal is \(A \rightarrow B\) and in \((b)\) it is \(B\) under the assumption \(A\). Therefore by refinement we have transformed the object-level functional dependency \(A \rightarrow B\) into a meta-level dependency.

This transformation can be seen in the type theory where the unknowns are represented by meta-variables. In the purely functional case on the right, both dependencies are identified and are thus indistinguishable. Therefore, looking at the type-theory encoding of the incomplete proof we will not be able to tell whether our goal is to prove \(A \rightarrow B\) or \(B\) under assumption \(A\) because both of them are represented by a function \(f\) of type \(A \rightarrow B\). This impossibility to 'simulate the Intro tactic' was already noticed by Muñoz [Mn97], but as we see in the paragraph below this is not limited only to this tactic. Therefore, if we do not distinguish between the object- and meta-level abstractions we cannot track the progress being made towards solving the goals. We cannot even say which functions represent goals and which are 'real functions'.

Moving on to the derivations \((b)\) and \((c)\) we see another example where the progress being done in the proof construction is not reflected in the encodings. Since the two goals in \((b)\) and \((c)\) could potentially admit different completions to HOL proofs, we definitely should consider the two derivations as distinct. However both of them have exactly the same translation to type theory. This means that the lifting of the Curry-Howard embedding to the logic with open terms would not be injective.
The second example shown in Figure 5.10 demonstrates the necessity to use hereditarily parameterized instead of higher-order arguments to encode binding holes. In the first derivation the variable $x$ is bound, but its binder has not been constructed yet while in the second this has already been done. We would like to distinguish between these two derivations because the two unknowns have different sets of solutions. These two derivations however are identified under the functional representation.

The examples above show that if we commit to first-class representation of unknowns in logic, we need to do the same on the type theory level if we want to have an extension of the Curry-Howard embedding. In that sense higher-order parametricity is necessary in a type theory with open terms.

One of the readers of the manuscript noted an interesting historical parallel between the use of functions and meta-variables. First came meta-variables standing for unknown objects, i.e. these are first-order meta-variables. Then, in systems with binders, we needed second-order meta-variables to express the functional dependency of an unknown on bound variables that may occur in it. The examples above show the need for third-order meta-variables in some situations.

This repeats the historical development of the notion of function, starting with parametric number (Newton and Leibnitz), functions on numbers (Euler), functions on functions, quantification on functions, etc. The parallel can be seen also by comparing the notion of instantiation and the second order substitution by Goldfarb [Gold81] or [Far88].

![Figure 5.10: Comparing hereditarily parameterized meta-variables and representation with higher-order functions.](image-url)
CHAPTER 6

Applications to Tactics

In this chapter we use the calculus of open terms presented in Chapter 4 as a basis on which we build a model of a hypothetical interactive theorem prover for higher-order logic.

We present a calculus that allows us to encode proof-search procedures. This is illustrated by giving terms that correspond to some common primitive tactics found in modern tactics-based theorem provers. We define a typed semantics for this calculus and show that this semantics is deterministic and that it is sound and complete with respect to the calculus of open terms.

As an application, we show how to define a decision procedure for intuitionistic minimal propositional logic (based on LJT [Dyc92]). The soundness of this decision procedure comes for free from the soundness theorem of the semantics. However using the semantics, we can give a proof of its completeness too.

6.1. Introduction

The tactic-based approach to theorem proving was introduced first in the LCF system [GMW79]. There the proofs are created through a controlled interface providing a set of basic primitives that are supposed to be sound and are the only way to construct proofs. These primitives can then be used by external procedures called tactics that are written in a general-purpose programming language like ML or OCAML [Cril]. The advantage of using a Turing complete language is that we can program tactics to implement very complex (semi-)decision procedures. The price to be paid lays in several areas. First, such tactics may not terminate. Pollack [Pol94] suggests using a powerful dependently typed calculus instead of ML as a possible solution to the termination problem. He also proposes that tactics can be seen as admissible rules and can be used to give indirect proofs without the need for normalization. On the other hand, letting users write tactics in the implementation language of the prover is not the best solution for a system that seeks to address a wide audience. Apart from the expert knowledge needed to write such tactics, one depends on implementation details that may make the tactic non-portable to the next version. A dedicated tactic language (like $L_{tac}$ [Del00] for Coq) which is independent of the implementation of the system has also the theoretical advantage that we can give semantics to the tactics written in it without having to refer to the semantics of the general-purpose implementation language. In $L_{tac}$ the semantics is given in terms of sequences of calls to the atomic tactics provided by Coq.
In the chapters so far we have developed theory that allows us to model the static aspects of the processes in an interactive theorem prover for higher-order logic like the representation of incomplete objects and proofs and modelling of states. In this chapter we want to address the dynamic aspects represented by the tactics. We will start with \( \lambda \text{HOL} \) and extend it with primitives for proof search and construction to obtain a calculus capable to encode some common tactics as terms. The definition of operational semantics will give us a way to execute the tactics and the soundness theorem of the semantics will guarantee us that successful tactic applications behave 'properly'. The general soundness result means that every tactic is sound by construction. Additional properties like termination (i.e. completeness) can be proven using the definition and the properties of the semantics. We demonstrate this by presenting a tactic for deciding minimal propositional intuitionistic logic and proving its completeness.

### 6.2. An Integrated Model of an Interactive Theorem Prover

Figure 6.1 presents the general relation between the main components of the model that we have adopted. We have two categories – tactics and pre-tactics. The general idea is that tactics are terms encoding proof-state transformations and pre-tactics are terms that encode procedures to create (proof)terms. A tactic execution is always associated to a goal from the state that it is supposed to solve, while pre-tactics may be evaluated without knowing the current goal. There are operations to use pre-tactics in tactics and conversely. Tactics can also be built up using tacticals in LCF-style.
6.2. AN INTEGRATED MODEL OF AN INTERACTIVE THEOREM PROVER

Each of the components of the model is to a certain degree related to work previously done in other systems or in other contexts, but as far the author is aware this is the first integrated model that covers all aspects together (see Section 6.8 for a discussion on related work). Starting from the meaning of open terms as incomplete derivations in logic, through the encoding of states as (contexts with) open terms, to the definition of typed operational semantics based on that encoding, we have a single framework that describes all main processes in an tactic-based interactive theorem prover in type theory.

6.2.1. Object Layer. The open terms that we will work with are essentially the terms of \(\lambda\text{HOL}\) described in Chapter 3. This means that they are built up from the sorts Prop, Type and Kind, variable and meta-variable instances using application and \(\lambda\) and \(\Pi\)-abstraction. A meta-variable stands for an unknown term for which we seek instantiation. The tactics manipulate incomplete terms by introducing and/or solving meta-variables. As we will see later in the chapter, it is practically very convenient to have a way to denote explicitly the fact that a meta-variable has been solved. The fact that the meta-variable \(m[\Delta]:A\) is solved by the term \(N\) can naturally be expressed by a definition \(m[\Delta] = N:A\). Since meta-variables in \(\lambda\text{HOL}\) are always global (i.e. declared in the context of the typing judgment) it is enough to allow definitions of meta-variables in the contexts.

The extension with global parameterized definitions [SP94, BKL02] (see also Section 4.4) of \(\lambda\text{HOL}\) is rather standard and straightforward and we will not go into its details. We will bring to the attention of the reader that the only change to the typing system is the addition of a weakening rule to introduce definitions in the context and a modification of the conversion rule in which the \(\beta\)-equality is replaced by \(\beta\delta\)-equality. All usual properties of \(\lambda\text{HOL}\) continue to hold after the introduction of global definitions. For precise formulation of the syntax and the typing rules, see Appendix A.

For our further discussions we will need the following modified versions of the weakening and cut lemmata given below:

\[ \text{Lemma 6.1 (Cut Lemma). If } \Gamma, m[\Delta]:A, \Gamma' \vdash M : B \text{ and } \Gamma, \Delta \vdash N : A \text{ then} \]
\[ \begin{align*}
(1) & \quad \Gamma, \Gamma'\{m[\Delta] = N\} \vdash M\{m[\Delta] = N\} : B\{m[\Delta] = N\} \\
(2) & \quad \Gamma, m[\Delta] = N:A, \Gamma' \vdash M : B
\end{align*} \]

\[ \text{Lemma 6.2 (Weakening). Let } \Gamma, \Delta \vdash M : B. \]
\[ \begin{align*}
(1) & \quad \text{If } \Gamma, \Delta \vdash A : s \text{ then } \Gamma, m[\Delta]:A, \Gamma' \vdash M : B. \\
(2) & \quad \text{If } \Gamma, \Delta \vdash N : A : s \text{ then } \Gamma, m[\Delta] = N:A, \Gamma' \vdash M : B.
\end{align*} \]

6.2.2. Proof-states and Proof-state Transformations. The state of the prover at a given moment can be represented by a typing judgment in the calculus of open terms. A judgment \(\Gamma \vdash M : A\) asserts that the incomplete term \(M\) is of type \(A\) in context \(\Gamma\). The term \(M\) is the incomplete proof we are developing and \(\Gamma\) contains all declarations of the language that we work in, the global assumptions and the meta-variables representing the unknown parts of the proof. The fact that the judgment is derivable in the calculus of open terms guarantees the ‘soundness’ of the state in the sense that if we find instantiations of the meta-variables in \(\Gamma\)
and propagate them over \( M \) we will get a typable term of the type we want to inhabit.

\[
\begin{array}{|l|l|l|l|}
\hline
\text{State}_1 & \text{State}_2 & \text{State}_3 & \text{State}_4 \\
\hline
\Gradient{\text{Intro.}} & x : U & \Gradient{\text{Intro.}} & x : U \\
\Gradient{\text{Intro.}} & \vdash \forall x^U, P \vdash P \rightarrow P \ & \Gradient{\text{Assumption.}} & \vdash P \\
\hline
\end{array}
\]

**Figure 6.2.** A sample interactive session. The commands above the arrows are the tactics invoked by the user. Above the line in each state are the assumptions and below is the goal to be proven.

Consider the interactive session depicted in Figure 6.2. There we have a sample tactic-based development of a proof of \( \forall x^U, P \rightarrow P \). The user invokes tactics (Intro and Assumption) that solve the goal in the current state in terms of subgoals that appear in the next state. We are interested in how each state can be encoded as a judgment and what is the relation between the states that is induced by the tactics.

Each of the four states can be encoded in a \( \lambda \)HOL judgement as shown in Figure 6.3. For example, \( \text{State}_3 \) is given by

\[
\Gamma, m_3[x^U, p^P], P \vdash \lambda x^U. \lambda p^P. m_3[x, p] : \Pi x^U. P x \rightarrow P x
\]

where \( \Gamma \) is a context that declares \( U \) and \( P \) with their respective types. The goal is encoded as the meta-variable \( m_3 \). Its assumptions are declared as parameters and its type is the proposition that we want to prove. The term \( \lambda x^U. \lambda p^P. P x, m_3[x, p] \) after the \( \vdash \) sign represents the (incomplete) proof of the original goal that we started with in \( \text{State}_1 \). The tactics that the user invokes induce the transitions between the states. This implies that there is a relation between the judgments that represents the states. Let us look at the transition from \( \text{State}_2 \) to \( \text{State}_3 \). How can we ‘obtain’ the judgment corresponding to \( \text{State}_3 \) from the one that corresponds to \( \text{State}_2 \)? First, we can apply clause (1) from the weakening lemma (Lemma 6.2) to the second judgment on Figure 6.3 to get

\[
\begin{align*}
\Gamma, m_1[\cdot] & : \Pi x^U. P x \rightarrow P x \vdash m_1[\cdot] : \Pi x^U. P x \rightarrow P x \\
\Gamma, m_2[x^U] & : P x \rightarrow P x \vdash \lambda x^U. m_2[x] : \Pi x^U. P x \rightarrow P x \\
\Gamma, m_3[x^U, p^P] & : P x \vdash \lambda x^U. \lambda p^P. m_3[x, p] : \Pi x^U. P x \rightarrow P x \\
\Gamma & \vdash \lambda x^U. \lambda p^P. P x, p : \Pi x^U. P x \rightarrow P x
\end{align*}
\]

**Figure 6.3.** Encoding the proof states as \( \lambda \)HOL judgments.
which is exactly the judgment encoding State₃.

In general, the effect of applying a tactic would be the introduction of a number of meta-variables (possibly zero) and finding instantiations for pre-existing or new meta-variables. In the encoding in Figure 6.3 every next judgment can be obtained from the previous one by a number of applications of clause (1) of the Weakening Lemma and (1) of the Cut Lemma. Since proof construction is potentially a branching process (e.g. the proof term is an application) and meta-variables are global, instantiations of meta-variables found in one branch need to be remembered for use in the other branches. For that reason we will prefer not to propagate the instantiations of meta-variables immediately, but instead record them as definitions in the context.

This means that in the places where we used clause (1) from the Cut Lemma, we will use clause (2), or in other words, to solve a meta-variable declared as $m[\Delta]:A$ by $N$ we convert its declaration to a definition $m[\Delta] = N:A$. In the case of our example this means that the judgments representing the states become as shown in Figure 6.4. The new representation is very verbose as it contains not only the state, but also an encoding of its 'history'. Another interesting side-effect of

\[
\begin{array}{ll}
\Gamma, & \Gamma, \\
m_3[\cdot] : \Pi x : U. P x \rightarrow P x & m_4[\cdot] : \Pi x : U. P x \rightarrow P x \\
\vdash & \vdash \\
m_1[\cdot] : \Pi x : U. P x \rightarrow P x & m_4[\cdot] : \Pi x : U. P x \rightarrow P x \\
\text{State}_1 & \text{State}_2
\end{array}
\]

\[
\begin{array}{ll}
\Gamma, & \Gamma, \\
m_3[\cdot] : \Pi x : U, p : P x \rightarrow P x & m_4[\cdot] : \Pi x : U, p : P x \rightarrow P x \\
m_2[\cdot] : \lambda p : P x. m_3[\cdot, x, p] & m_2[\cdot] : \lambda p : P x. m_3[\cdot, x, p] \\
m_1[\cdot] = \lambda x : U. m_2[\cdot] & m_4[\cdot] = \lambda x : U. m_2[\cdot] \\
\vdash & \vdash \\
m_1[\cdot] : \Pi x : U. P x \rightarrow P x & m_4[\cdot] : \Pi x : U. P x \rightarrow P x \\
\text{State}_3 & \text{State}_3
\end{array}
\]

**Figure 6.4.** Encoding the proof states as judgments in $\lambda\text{hoHOL}$ with definitions. Note that the context contains virtually all essential information. For readability, in the definitions $m[\Delta] = N:A$, we have omitted the type $A$ (since it can be inferred).

this approach is that virtually all essential information for the state is now encoded in the context. Assuming that we start from a judgment $\Gamma \vdash m[\cdot] : A$ the tactic applications will only change $\Gamma$, but never $m[\cdot]$ of $A$ as we do not propagate the instantiations of the meta-variables. Therefore, for most purposes we will consider states as encoded by (valid) contexts.
6.3. Encoding Proof-Search Procedures: Pre-tactics and Tactics

As we already saw in the previous section, part of the effect of the execution of a tactic is expressed by an instantiation of a meta-variable. To actually find this instantiation, a tactic must encode a proof-search procedure. To do that we will extend the calculus of open terms with proof-search primitives and we will call the terms of this calculus pre-tactics. We will define operational semantics for the calculus of pre-tactics and the idea is that a successful evaluation of a pre-tactic results in a well-typed term that we consequently can use to solve our goal. A pre-tactic is therefore a codified proof-search procedure that evaluates to an instantiating term (if successful). To do this proof-search we will add the possibility to construct terms by using primitives like unification constraints and search in the context, introducing and solving of fresh meta-variables, failing of the search and handling of failures as well as (unrestricted) recursion.

The term obtained from a pre-tactic can be used to instantiate a goal in a proof-state. This is done by the tactics that actually work on proof-states as we saw in the previous section. On the level of tactics we need to handle things like focus (the goal that we want to work with at a given moment), the use of term found by a pre-tactic to solve the current goal, the composition of tactics (i.e. the so-called tacticals) and different issues related to the correct handling of (fresh) names in tactic terms. In other words, the ‘big picture’ is as presented in

![Figure 6.5](image)

Figure 6.5. The calculus of pre-tactics extends the calculus of open terms with proof-search primitives. Tactics are defined using pre-tactics and tacticals.

Figure 6.5 we have two syntactic categories - tactics and pre-tactics. They are defined by a mutually inductive definition as shown in Figure 6.6.

Before explaining more thoroughly the details of the calculus, let us illustrate it by considering the example in the following section.

6.3.1. Example: The pre-tactic Apply. The pre-tactic Apply tries to solve a goal \( \varphi \) by specializing a given theorem \( \psi \) with a given proof \( M \). For example, if \( \varphi \) is \( P(t) \) and \( \psi \) is \( \forall x. P(x) \) then Apply produces the term \( (M t) \) which is a proof of \( P(t) \).

In more complicated cases the pre-tactic may introduce new meta-variables for unknown terms needed to instantiate \( \psi \). For example, let \( a, b \) and \( c \) be terms of type \( U \) and \( R \) be a binary relation on \( U \). Let \( M \) be the proof that \( R \) is transitive.
6.3. Encoding Proof-search Procedures: Pre-tactics and Tactics

<table>
<thead>
<tr>
<th>sorts</th>
<th>$S ::= \text{Prop} \mid \text{Type} \mid \text{Kind}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>open term</td>
<td>$B ::= x \mid m[B \ldots B] \mid BB \mid \lambda x:B.B \mid \Pi x:B.B \mid S$</td>
</tr>
<tr>
<td>parameter list</td>
<td>$\Delta ::= \varepsilon \mid \Delta, x:B$</td>
</tr>
<tr>
<td>meta-variable list</td>
<td>$\Sigma ::= \varepsilon \mid \Sigma, ?m[\Delta]:B$</td>
</tr>
<tr>
<td>pre-tactic term</td>
<td>$\mathcal{P} ::= x \mid m[B \ldots B] \mid t[B \ldots B] \mid \mathcal{P}\mathcal{P} \mid \lambda x:B.\mathcal{P} \mid \Pi x:B.\mathcal{P} \mid S \star \mid ?m[\Delta]:B.\mathcal{P} \mid ?m[\Delta] = \langle T \rangle.B.\mathcal{P} \mid [B \sim_{\Sigma} B].\mathcal{P} \mid [\sim_{\Sigma} x:B].\mathcal{P} \mid (\mathcal{P} \ \text{else} \ \mathcal{P}) \mid \text{Fail}_{a}$</td>
</tr>
<tr>
<td>tactic</td>
<td>$T ::= (\text{use} \ \mathcal{P}) \mid \text{Bind} (x, n).T \mid \text{Clear} \ x \mid T; T \mid T; [T \ldots [T] \mid T[T, \ldots, T]$</td>
</tr>
<tr>
<td>pre-tactic defn.</td>
<td>$t[x:B \ldots x:B] := \mathcal{P}$</td>
</tr>
<tr>
<td>tactic defn.</td>
<td>$T[x_1, \ldots, x_n] := T$</td>
</tr>
<tr>
<td>context</td>
<td>$C ::= \varepsilon \mid C, x:B \mid C, ?m[\Delta]:B \mid C, ?m[\Delta] := B:B$</td>
</tr>
</tbody>
</table>

Figure 6.6. Syntax for object, pre-tactics and tactics terms

Then an application of \texttt{Apply} with argument $M$ to the goal $R(a, c)$ would produce new meta-variables $y[i]:U$, $p[i]:R(a, y)$ and $q[i]:R(y, c)$ and instantiation term $(M \ a \ y[i] \ c \ p[i] \ q[i])$ that can be used to solve the original goal.

To further illustrate the complex behaviour required from \texttt{Apply}, let us assume that there is also a term $N$ representing the proof of $\forall x.R(x, b)$. When we use \texttt{Apply} with $N$ on the goal $p$ from above we observe two things - first, we see that \texttt{Apply} needs unification (as opposed to matching) to solve the goal because it needs to unify $R(a, y)$ and $R(x, b)$ where both $x$ and $y$ are unknowns. Second, the instantiation found by unification affects not only the variable $x$, but also the goal $y$. Therefore as a side effect a (pre-)tactic may force other goals to be solved.

The proof-search done by \texttt{Apply} can be given in the calculus of pre-tactics with the following recursive definition:

\[
\text{Apply} [\varphi : \text{Prop}, \psi : \text{Prop}, \ M : \psi] :=
\begin{align*}
\ & [\varphi \sim \psi].M \\
\text{else} & [\psi \sim ?A.?B: \text{Prop} \ \Pi x:A.B]. ?m:A.\text{Apply}[\varphi, B, (Mm)] \\
\text{else} & [\psi \sim ?A.?B: \text{Prop} \ \Pi x:A.B[x]]. ?m:A.\text{Apply}[\varphi, B[m], (Mm)]
\end{align*}
\]

We have dropped the brackets around else by defining it as left-associative.

Let us quickly go through the definition and explain how \texttt{Apply} works. We start by postulating that \texttt{Apply} is a pre-tactic with three parameters with their respective types. We have used suggestive names for the parameters that correspond to the informal description that we started with (i.e. $\varphi$ is the goal and $M$ the proof of the theorem $\psi$ that we apply to solve it). The term constituting the body of the definition of \texttt{Apply} consists of three terms separated by else. The idea is that we first try to compute the term on the left-hand side of else and if we fail we continue with the right-hand side.

Therefore, the first of the three subterms to be evaluated will be $[\varphi \sim \psi].M$. It should be read as "if $\varphi$ unifies with $\psi$ with instantiation $\sigma$ then the resulting
term is the value of \( \sigma M' \). Hence this term represents the base case in the recursive
definition of Apply.

If the unification in \([\varphi \sim \psi], M \) fails (this may happen because there is no
unifier or because the unification procedure did not find it\(^1\)) we have to look at
the second subterm, namely

\[
[\psi \sim \_A, \_B: \text{Prop}, \Pi x: A. B]. ?m:A. \text{Apply}[\varphi, B, (M m)]
\]

As before, we start with a unification constraint. We try to unify our goal \( \psi \) with
\( \Pi x: A.B \) in order to check whether \( \psi \) is an implication. However, the question
arises "Where do \( A \) and \( B \) come from and are they variables or meta-variables?"
Well, our idea is certainly that we want \( A \) and \( B \) to be solved by the unification and
therefore they should be meta-variables. Furthermore, \( A \) and \( B \) should be 'fresh'
in order to correctly capture the idea of the check whether \( \psi \) is an implication.
To address this situation we declare \( A \) and \( B \) locally as new meta-variables in the
unification constraint. The idea is that they are bound by the constraint and get
their values when the constraint is evaluated.

Hence, \([\psi \sim \_A, \_B: \text{Prop}, \Pi x: A. B] \ldots \) should be read as "if there are \( A \) and \( B \)
such that \( \psi = A \to B \) then \ldots " The rest of the term \( ?m:A. \text{Apply}[\varphi, B, (M m)] \)
means that we introduce a fresh meta-variable \( m \) (with the current local context
as parameter list) of type \( A \) and we do a recursive call to Apply. Note that the
second argument is \( B \) and that the type of \( (M m) \) will also be \( B \).

In case \( \psi \) is not an implication, we do a similar last attempt checking whether
\( \psi \) is a universally quantified formula.

To summarize, while a proof of the goal is not found and the provided theorem
is implication or universally quantified, we introduce meta-variables to eliminate
the antecedent or the universal quantifier of the given theorem. Some or all of
these meta-variables are solved once we reach a state in which the goal unifies
with the statement of theorem (after elimination).

To illustrate this, let us come back to the transitive relation \( R \). We would
expect that when evaluating the term

\[
\text{Apply}[R(a, c), \Pi x, y, z: U. R(x, y) \to R(y, z) \to R(x, z), M]
\]

we would have to do roughly the steps shown in Figure 6.7. At the last step we
see that the constraint \([R(a, c) \sim R(x, z)](M x y z p q) \) produces an instantiation
that affects some of the meta-variables that we introduced at the previous steps.
Therefore, when we evaluate terms of the form \([M \sim N], P \) and get an instantiation \( \sigma \), we need to propagate \( \sigma \) not only in the body \( P \), but also in the whole state.
In that respect the evaluation of terms with unification constraints is non-local.

In general, the evaluation of one subterm of a pre-tactic term may affect
whether and to what another subterm evaluates. For example, in the context
\( A: \text{Prop}, g:A \to A, ?m[]=\text{Prop} \) term \( (\lambda x:A. m[])(\lambda x:A.gx) \) cannot be
evaluated to a well-typed term while the term \( (\lambda x:A.gx)(\lambda x:A.gx) \) evaluates
to \( (\lambda f:A \to A, f)(\lambda x:A.gx) \) (after unfolding of definitions).

Hence we should make sure that the semantics correctly captures the side-
effects of the evaluation of pre-tactic terms.

\(^1\)In general here we deal with higher-order unification problems which are not decidable.
state context | pre-tactic term to evaluate
--- | ---
?x:U | Apply\([\text{R}(a, c), \text{I}x, y, z:U. \text{R}(x, y) \rightarrow \text{R}(y, z) \rightarrow \text{R}(x, z), M]\)
?y:U | Apply\([\text{R}(a, c), \text{I}y, z:U. \text{R}(x, y) \rightarrow \text{R}(y, z) \rightarrow \text{R}(x, z), (M x y)]\)
?x, y:U | Apply\([\text{R}(a, c), \text{I}x:U. \text{R}(x, y) \rightarrow \text{R}(y, z) \rightarrow \text{R}(x, z), (M x y)]\)
?x, y, z:U | Apply\([\text{R}(a, c), \text{R}(x, y) \rightarrow \text{R}(y, z) \rightarrow \text{R}(x, z), (M x y z)]\)
?x, y, z:U | Apply\([\text{R}(a, c), \text{R}(x, y) \rightarrow \text{R}(y, z) \rightarrow \text{R}(x, z), (M x y z)]\)
?x, y, z:U | Apply\([\text{R}(a, c), \text{R}(x, y) \rightarrow \text{R}(y, z) \rightarrow \text{R}(x, z), (M x y z)]\)

Figure 6.7. Part of the steps one needs to go through when evaluating a call to Apply. In the last step \(x\) and \(z\) are solved by the unification constraint. We have omitted the prefix of the state context that declares \(U, R, a, c\) and \(M\).

6.3.2. A Calculus of Tactics and Pre-tactics. In Figure 6.6 we presented the syntax of the calculus and in the previous section we saw an example of a pre-tactic defined in it. Now we will go through the different constructions and explain more precisely their intended meaning.

As we can see in Figure 6.6 the syntactic categories of pre-tactics \(P\) and of tactics \(T\) are mutually recursive and \(P\) contains the calculus of open terms. We will start with a presentation of the constructors of the tactic terms.

\(\text{(use } M\text{)}\) represents a tactic that solves the current goal by the term to which the pre-tactic \(M\) evaluates to. If the current goal is \(m[\Delta]:A\) then \(M\) is evaluated in the context \(\Delta\). In case the evaluation is successful and we have obtained a term \(N\) of type \(A\), the meta-variable declaration \(m[\Delta]:A\) is changed to the definition \(m[\Delta] = N:A\). Of course, the situation is not that simple, because the evaluation of \(M\) may have side effects on the state that also need to be taken into account. We discuss the handling of these side effects in more detail in Section 6.5.

\(\text{(T}_1;\text{T}_2\text{)}\) This is the tactical for tactic composition. The idea is that \(T_1\) is applied to the current goal and \(T_2\) to all subgoals generated by \(T_1\). It fails if any of these tactic applications fails.

\(\text{T} : [\text{T}_1 | \ldots | \text{T}_n]\) This is the tactical for general composition. First \(T\) is applied to the current goal and if it produces exactly \(n\) subgoals, then \(T_i\) is applied to the \(i\)th subgoal. If any of the tactic applications fails or if there is a mismatch between the number of goals and tactics, the composition also fails.

\(\text{(Clear } x\text{)}\) This tactic removes \(x\) from the list of hypotheses of the current goal. It fails if the removal would invalidate the state. We need such a primitive tactic because all other constructions that introduce a meta-variable are monotonic in the sense that they only extend the context of the current goal. For an example of the use of Clear, see Section 6.7.
Bind\((x, i) \cdot T\) This binding tactical allows at 'runtime' to associate the bound variable \(x\) with the \(i\)th (from the back) hypothesis of the current goal. This is necessary in order to be able to refer to hypotheses that have been introduced by previous tactics (when we compose them). For a discussion on Bind and an example for its use, see Section 6.4.1.

\(T[T_1 \ldots T_n]\) Defined tactic call. We allow global recursive definitions of tactics with untyped parameters.

After presenting the tacticals that are used to define the tactics from pre-tactics, it is time to look at the pre-tactics themselves. Please recall that a pre-tactic describes a proof-search procedure that if successful must yield an open term together with the impact of the search procedure on the current state.

\((M \text{ else } N)\) The pre-tactic first evaluates \(M\) and if successful, the value of \(M\) is the value of the pre-tactic. Otherwise the value of \(N\) is taken. An exception is the case when \(M\) evaluates to \(\text{Fail}_{n+1}\) in which case \(N\) is not evaluated and the pre-tactic gets value \(\text{Fail}_n\).

\(\text{Fail}_n\) This is the always failing pre-tactic. The index \(n\) denotes at which level of nested else's should the failure be handled.

Open terms. The terms of the open terms calculus can be seen as constant pre-tactics without side effects. In addition, the application, \(\lambda\)-abstraction and \(\Pi\)-abstraction can also be used on pre-tactics, for example \(\lambda x:A.\text{Fail}_0\).

\(*\) This is a dedicated variable that will be substituted by the current goal at the moment the pre-tactic is called. This is done, because we do not want to give explicitly the current goal each time as a parameter. The use of \(*\) allows us to make the tactic terms applicable to a range of goals. For example, consider the pre-tactic Apply from Section 6.3.1. Its first parameter is the goal that we want to solve. If we write \(\text{use}\ \text{Apply}[^* , \varphi, M]\), this would be a tactic that will fail for any goal different from \(\varphi\). Using \(*\) we can express the much more general tactic \(\text{use}\ \text{Apply}[^* , \varphi, M]\) that would succeed for any goal that unifies with a specialization of \(\varphi\).

The use of \(*\) in recursive definitions should be avoided because the substitution of the current goal for \(*\) is only done once at the toplevel call in \(\text{use}\ M\) and unfolding of recursively defined pre-tactics may bring up irreducible \(*\) subterms. In these cases one should use a parameter describing the goal and then give \(*\) as an argument to the topmost call to the recursive pre-tactic as with Apply above.

\([M \sim_\Sigma N]\cdot P\) This term represents a unification constraint. The open terms \(M\) and \(N\) are unified in the local context, extended with the meta-variables in \(\Sigma\). If the unification fails, the whole pre-tactic fails. If it succeeds with an instantiation \(\sigma\), then the value of the pre-tactic is computed by evaluating \(\sigma(P)\) in a state obtained from the current one by executing \(\sigma\).

\([\sim_\Sigma h : A] \cdot P\) This pre-tactic allows us to search the current local context. The value of \([\sim_\Sigma h : A] \cdot P\) is obtained by computing the value of \(P[x_i / h]\) where \(x_i : B_i\)
is a declaration in the local context and $A$ and $B_i$ are unifiable. If there are multiple such $i$, the first one for which $P[x_i/h]$ succeeds is taken.

To be more precise, if the local context contains the declarations $x_i : B_i$ ($1 \leq i \leq n$) then $[\sim h : A].P$ is equivalent to

$$([A \sim B_1].P[x_1/h]) \text{ else } \ldots \text{ else } ([A \sim B_n].P[x_n/h]) \text{ else } \text{Fail}_0$$

where the last $\text{Fail}_0$ is only needed if $n = 0$.

$?m[\vec{x} : A] : B.M$ This is a construction that prescribes that a new meta-variable $m$ of type $B$ should be added to the current state. Since this construction may occur under binders, we need to take the local context into account. If $\Delta$ is this local context, then the actual meta-variable that is added is $m[\Delta, \vec{x} : A] : B$. The evaluation of the pre-tactic then continues with the body $M$ in which $m$ has been augmented with extra arguments for $\Delta$.

$?m[\vec{x} : A] = (T) : B.M$ This construction is similar to $?m[\vec{x} : A] : B.M$, but after introducing $m$ to the state, we solve it by applying the tactic $T$ to it. If $T$ succeeds, we continue with $M$. This construction allows us to use tactics to define pre-tactics and it is necessary when we need to solve a goal with a tactic that is not definable by a pre-tactic term. An example of such a case can be seen in Section 6.7.

$t[M_1 \ldots M_n]$ Call to a defined pre-tactic. We allow global definitions of pre-tactic terms with unrestricted recursion. The parameters of $t$ are typechecked before substituting them into its body.

### 6.4. Definability of Some Basic Tactics and Tactics

Below we give (recursive) definitions in our language that represent some common tactics in tactic-based interactive theorem provers. As a matter of convention, the first parameter of the tactics will always be the goal that the tactic solves.

| Cut $[\varphi : \text{Prop}, \psi : \text{Prop}] := ?p : \varphi \rightarrow ?q : \psi. (p q)$ |
| Assert$_1 [\varphi : \text{Prop}, \psi : \text{Prop}] := ?q : \psi. ?p : x : \varphi. p[q]$ |
| Assert$_2 [\varphi : \text{Prop}, \psi : \text{Prop}] := ?q : \psi. (\lambda x : \psi. ?p : \varphi. p) q$ |

**Figure 6.8.** Encoding of Cut and two different encodings of Assert.

The pre-tactics Cut and Assert presented in Figure 6.8 solve the goal $\varphi$ by introducing a statement $\psi$ and proving $\psi \rightarrow \varphi$ ($\varphi$ under assumption $\psi$ in the case of Assert) and $\psi$. The difference between the two is the order in which the new goals are generated. Note also that the two implementations of Assert will produce identical goals but the proof-terms will be different. Assert$_2$ will create a $\beta$-redex, while Assert$_1$ will result in its reduct.

Figure 6.9 depicts two pre-tactics that encode the tactics Intro and Intros. Intro applies to any goal which is of $\Pi$-type and solves it by a $\lambda$-abstraction.
with a new meta-variable as body. Intro does the same, but iterates the process as much as possible. For example, \(\text{Intro}[\forall x U. P(x) \to Q(x)]\) results in the term \(\lambda x : U. \lambda p : P(x). m[x, p]\) where \(m\) is a new meta-variable with declaration 
\[m[x : U, p : P(x)] : Q(x)\]

\[
\begin{align*}
\text{Intro}[\varphi : \text{Prop}] := \\
&[\varphi \sim_{A : \text{Prop}} \Pi x : A. B], \lambda x : A. \, ?m : \varphi. m \\
&\text{else} \\
&[\varphi \sim_{A : \text{Type}, B[x : A] : \text{Prop}} \Pi x : A. B[x]], \lambda x : A. \, ?m : \varphi. m
\end{align*}
\]

**Figure 6.9.** Encoding Intro and Intros.

\[
\begin{align*}
\text{Apply}[\varphi : \text{Prop}, \psi : \text{Prop}, M : \psi] := \\
&[\varphi \sim \psi], M \\
&\text{else} \\
&[\psi \sim_{A : \text{Prop}} \Pi x : A. B], \, ?m : A. \text{Apply}[\varphi, B, (Mm)] \\
&\text{else} \\
&[\psi \sim_{A : \text{Type}, B[x : A] : \text{Prop}} \Pi x : A. B[x]], \, ?m : A. \text{Apply}[\varphi, B[m], (Mm)]
\end{align*}
\]

**Figure 6.10.** Encoding Apply.

The pre-tactic Apply shown in Figure 6.10 was already introduced in Section 6.3.1. It tries to produce a term of type \(\varphi\) by specializing the theorem \(\psi\) with proof \(M\).

The tactic Generalize shown in Figure 6.11 is in a sense the opposite of Apply. Given the goal \(\varphi\) and a term \(t\) it produces a new goal \(\forall x. B[x]\) such that \(\varphi\) is its specialization by \(t\) (i.e. such that \(B[t] \equiv \varphi\)). This tactic is used in cases it is easier to prove a statement for all values of a variable rather than for a specific one. Its effectiveness is heavily influenced by the power of the unification mechanism one uses.

The same holds for the tactic Pattern which performs \(\beta\)-expansion on the current goal. If the goal is \(B[t]\) then it is replaced by \((\lambda x : U. B[x]) t\). Figure 6.12 shows the straightforward definition of Assumption. Figure 6.13 shows how we can en-
6.4. Definability of Some Basic Tactics and Tacticals

\begin{figure}[h]
\centering
\begin{align*}
\text{Generalize}[\varphi : \text{Prop}, U : \text{Type}, t : U] & := \\
\end{align*}
\begin{align*}
\text{Pattern}[\varphi : \text{Prop}, U : \text{Type}, t : U] & := \\
[B[t] \sim ?B[x : U] : \text{Prop } \varphi]. ?m : (\lambda x : U. B[x]) t. m
\end{align*}
\caption{Encoding Generalize and Pattern.}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{Assumption}[\varphi : \text{Prop}] & := [\sim A : \text{Prop } h : A]. h
\end{align*}
\caption{Encoding Assumption.}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{OrElse}[T_1, T_2] & := (\text{use } (?m = (T_1); \ast . m) \text{ else } (?m = (T_2); \ast . m)) \\
\text{Id} & := (\text{use } ?m; \ast . m) \\
\text{Fail}_n & := (\text{use } \text{Fail}_n) \\
\text{Repeat}[T] & := (T; \text{Repeat}[T]) \text{ OrElse Id} \\
\text{Try}[T] & := T \text{ OrElse Id} \\
\text{Solve}[T_1 \ldots T_n] & := (T_1; \text{Fail}_0) \text{ OrElse } \ldots \text{ OrElse } (T_n; \text{Fail})
\end{align*}
\caption{Encoding some tactics.}
\end{figure}

code some common tactics. The tactical OrElse first applies $T_1$ and if it succeeds, the tactical succeeds. If $T_1$ fails, then $T_2$ is called. Id is the identity tactical that solves the current goal by a copy of the current goal. It always succeeds by making a copy of the current goal solving it using that copy (practically, this looks to the user as if the goal was left unchanged). Fail is the tactical that always fails. Repeat$[T]$ keeps applying $T$ to the current goal and any generated subgoals as long as possible. The tactical Try$[T]$ allows safe calls to tactics that may fail. It tries to execute the tactic $T$ and if it fails, Try just returns the original goal.

The last tactical is Solve$[T_1 \ldots T_n]$. It applies successively the tactics given as parameters to the current goal until one of them solves it without producing subgoals. Note that $T_i; \text{Fail}$ will succeed if and only if $T_i$ succeeds and produces no goals.

Some rather trivial extensions of the calculus could allow us to define tactics dealing with computations on well-typed terms (see Section 6.8.2.2 for suggestions on how to implement a normalization tactic).

6.4.1. Fresh Names in Tactic Terms. Consider the goal of proving the trivial goal $A \rightarrow A$. By applying the tactic Intro to the goal $m[\cdot] : A \rightarrow A$ we get a new goal $n[x : A] : A$ and then a call to Apply$[A, A, x]$ closes the proof. Given the
fact that we can compose tactics, one could expect that the tactic
\( \text{(use Intros[*]); (use Apply[*], A, x)} \)
would solve the goal \( m [] : A \rightarrow A \). However this tactic will fail because the variable \( x \) is neither a hypothesis of the goal neither is it declared in the state context. What we want to do in the call to apply is to use the hypothesis \( x \) that will be introduced by the call to Intros. This however is impossible, because \( x \) is not in scope.

To address this issue we introduce the tactical Bind. It allows 'runtime' binding of a variable \( x \) to a variable from the local context. The term \( \text{Bind}(x, 1). \text{Apply[*], A, x} \)

is executed by substituting the last variable of the local context in \( \text{Apply[*], A, x} \)

and then evaluating the result. Therefore the tactic term
\( \text{(use Intros[*]); Bind(x, 1). (use Apply[*], A, x)} \)

will solve the goal \( m [] : A \rightarrow A \).

The problem with giving names to hypotheses is present in existing theorem provers. For example, in Coq one writes

\[ \text{Intros; Apply H.} \]

to solve a goal \( A \rightarrow A \), but the name \( H \) is automatically generated by the tactic \( \text{Intros} \) and is context sensitive. This means that the same script applied to the goal \( A \rightarrow A \) may not work if executed in a state where \( H \) is already bound. Coq gives us the possibility to write

\[ \text{Intros x; Apply x.} \]

therefore allowing us to give an explicit name to the hypothesis so that we can refer to it later. This does not really solve the problem because if \( x \) is already declared, the tactic will fail. Note however that the meaning of the tactic would remain the same if we rename \( x \) to \( y \):

\[ \text{Intros y; Apply y.} \]

This clearly suggests that \( x \) can be seen as a bound variable whose scope starts \textit{after} the execution of \( \text{Intros} \). Note that also the end of the scope of this binding may not be the rest of the tactic script. For example the scope of \( y \) in

\[ \text{Apply thm; [Intros y; Apply y | M]} \]

does not extend to \( M \).

This intuition stands behind the introduction of the binding tactical Bind in our calculus. It provides a clean way to handle the names of the variables by binding them and it clearly defines their scope.

6.5. Semantics

In this section we define what it means to 'execute' a tactic, or in other words how to compute the result of applying a tactic to a state. We will do that by presenting the evaluation relation \( \models \) defined inductively using the typing relation on the level of the open terms.
6.5. SEMANTICS

Both for tactics and pre-tactics we have semantic judgments that represent successful evaluations and judgments that represent failed evaluations. The successful application of a tactic $T$ to the goal $m[\Delta] : A$ in state context $\Gamma$ will be denoted by a judgment of the form

$$\vdash \Gamma, m[\Delta] \langle T \rangle : A \Rightarrow \Gamma', m[\Delta] = N : A$$

It should be read as follows: “The application of the tactic $T$ to the goal $m[\Delta] : A$ in a state $\Gamma$, $m[\Delta] : A$ is successful and yields the state $\Gamma'$, $m[\Delta] = N : A'$.

The judgment representing successful evaluation of a pre-tactic is

$$\vdash (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright N : A$$

and should be read as “The evaluation of the pre-tactic $M$ in state context $\Gamma$ and local context $\Delta$ is successful and yields the open term $N$ of type $A$ in the (new) state context $\Gamma'$ and local context $\Delta'$. Note that the evaluation of $M$ changes $\Gamma$, but the local context $\Delta$ is preserved.

The judgments representing failed evaluations of tactics and pre-tactics look respectively like this:

$$\vdash \Gamma, m[\Delta] \langle T \rangle : A \Rightarrow \text{Fail}_n$$

$$\vdash (\Gamma; \Delta) \triangleright M \Rightarrow \text{Fail}_n$$

Before we define $\vdash$, let us first see how it is intended to be used. As we said, we model proof-states by typing judgments with global definitions and most of the information about the state is contained in the context. Actually, with respect to the semantics, the context contains all we need to work with.

So let $\Gamma_1, m[\Delta] : A, \Gamma_2$ be a state. Suppose the user has chosen the goal $m$ as his/her current goal and wants to apply the tactic $T$ to it. To compute the resulting state, we first compute $\Gamma'_1$ and $N$ such that

$$\vdash \Gamma_1, m[\Delta] \langle T \rangle : A \Rightarrow \Gamma'_1, m[\Delta] = N : A$$

and then the new state is $\Gamma'_1, m[\Delta] = N : A, \Gamma_2$.

By Consistency (Theorem 6.7) if we manage to evaluate $T$, the result is unique and therefore $\Gamma'_1$ and $N$ are properly defined. Next, from the Soundness theorem (Theorem 6.12) we know that $N$ will be of type $A$ in context $\Gamma'_1$, $\Delta$ and that $\Gamma'_1$ is a weakening (with meta-variables and meta-variable definitions) of $\Gamma_1$. Therefore using the Weakening Lemma and Cut Lemma (see Section 6.2.1) we have that the new state $\Gamma'_1, m[\Delta] = N : A, \Gamma_2$ is valid.

As in some of the proof-search primitives we need to solve unification problems, we will assume that an external oracle function $\text{uni}()$ is provided. In general the unification problems that we need to solve will be higher-order unification problems. Since higher-order unification is undecidable [Hue73], we sacrifice completeness by assuming that $\text{uni}()$ is a total recursive function. It either returns the value Fail indicating that no unifiers were found (not necessarily that there are no unifiers), or it returns a unifier $\sigma$. We will call $\text{uni}()$ with four parameters: if $\text{uni}(\Gamma, \Sigma, M, N) = \sigma$ where $\Gamma$ is a context, $\Sigma$ is a list of meta-variables and $M$ and $N$ are open terms, then $\sigma(\Gamma) + \sigma(M) =_{\Delta} \sigma(N)$ and all meta-variables of $\Sigma$ have solutions in $\sigma$. $\sigma(\Gamma)$ denotes the context $\Gamma$ after all declarations of meta-variables.
in the domain of $\sigma$ are converted into definitions according to $\sigma$. Of course, we also assume that $\text{uni}(\cdot)$ only produces well-typed unifiers and in particular it does not introduce fresh meta-variables. The oracle function $\text{uni}(\cdot)$ will be used in defining the semantics of pre-tactic terms like $[M \sim_{\Sigma} N].P$ and $[\sim_{\Sigma} h:A].P$.

Now we are ready to define the semantics of tactics and pre-tactics. For each term constructor there are typically two groups of rules — one that handles the successful evaluations and one that handles the failed evaluations. For most of the cases the rules for handling of failed evaluations are just passing the information about the failure to the outer level and are therefore not very interesting. For that reason we have left them out below, but one can find the full list of rules in Appendix B.

The rules presented in Figure 6.14, Figure 6.15 and Figure 6.16 represent the core of the semantics for pre-tactics. Let us briefly go through those rules and see how they work.

Starting with rule (Fail) in Figure 6.14, we see that in every valid context $\text{Fail}_0$ produces a failure. The next four rules are designed to handle failed evaluations. The rule (else$^+$) states that if $M$ evaluates successfully to a term $P$ of type $A$ then so does $(M$ else $N)$. The next two rules handle the case when the evaluation of $M$ has failed. Then obviously we need to evaluate $N$ and take its value as a value for $(M$ else $N)$. An exception to this is given by (else$^-\Sigma$) which handles the case when the value of $M$ is $\text{Fail}_k$ with $k > 0$. This is a sign that the evaluation of else should be aborted and the result is $\text{Fail}_{k-1}$.

Next come two rules for handling unification constraints $[M \sim_{\Sigma} N].P$. After making sure that the context we work in is valid, we call the oracle function $\text{uni}((\Gamma, \Delta), \Sigma, M, N)$. It either succeeds, producing a unifier $\sigma$, or fails by producing $\text{Fail}$ in which case the evaluation of $[M \sim_{\Sigma} N].P$ also fails. If $\sigma$ is produced, we apply it to $\Gamma$ (this causes some meta-variable declarations to be converted to definitions). In the resulting state $\sigma(\Gamma)$ and unchanged local context $\Delta$ we evaluate the body of the constraint after applying $\sigma$ to it. The result becomes the value for the term $[M \sim_{\Sigma} N].P$.

The next rule assumes that the pre-tactic $T$ is (potentially recursively) defined as $T[e:A] := M$. After checking the number and the types of the arguments, we substitute the arguments in the body $M$ and evaluate the result which after that becomes the value of the whole term $T[e]$.

The last rule in Figure 6.14 allows evaluation of context-search pre-tactics. If the local context at the moment of evaluation is $\Delta \equiv x_1:A_1 \ldots x_n:A_n$, then the value of the term $[\sim_{\Sigma} h:A].P$ is defined as the value of

$$([A \sim_{\Sigma} A_1].M[x_1/h]) \text{ else } \ldots \text{ else } ([A \sim_{\Sigma} A_n].M[x_n/h]) \text{ else } \text{Fail}_0$$

The idea is of course to find the first element of the context $x_i:A_i$ for which $A_i$ unifies with $A_i$, and $M[x_i/h]$ can successfully be evaluated. This effectively represents a backtracking search algorithm.

Figure 6.15 presents the rules for introducing new meta-variables to a state. The rule $(M^{\text{intra}})$ prescribes that in order to evaluate a term $?m[\theta]:A.M$ under a local context $\Delta$, we need to add a new meta-variable $n[\Delta, \theta]:A$ to the state (i.e. to $\Gamma$), then replace $m[\theta]$ by $n[\Delta, \theta]$ in $M$ and evaluate the result.
\[
\frac{\Gamma, \Delta \vdash \text{Prop} : \text{Type}}{\models (\Gamma; \Delta) \triangleright \text{Fail}_n \Rightarrow \text{Fail}_n} \quad \text{(Fail)}
\]

\[
\frac{\models (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright P : A}{\models (\Gamma'; \Delta) \triangleright (M \text{ else } N) \Rightarrow (\Gamma'; \Delta) \triangleright P : A} \quad \text{(else\textsuperscript{+})}
\]

\[
\frac{\models (\Gamma; \Delta) \triangleright M \Rightarrow \text{Fail}_0}{\models (\Gamma; \Delta) \triangleright N \Rightarrow (\Gamma'; \Delta) \triangleright Q : A} \quad \text{(else\textsuperscript{1})}
\]

\[
\frac{\models (\Gamma; \Delta) \triangleright (M \text{ else } N) \Rightarrow \text{Fail}_n}{\models (\Gamma; \Delta) \triangleright M \Rightarrow \text{Fail}_{n+1}} \quad \text{(else\textsuperscript{2})}
\]

\[
\frac{\Gamma, \Delta \vdash \text{Prop} : \text{Type}}{\models (\Gamma; \Delta) \triangleright [M \sim_{\Sigma} N].P \Rightarrow (\Gamma'; \Delta) \triangleright Q : A} \quad \text{(uni\textsuperscript{+})}
\]

\[
\frac{\Gamma, \Delta \vdash \text{Prop} : \text{Type}}{\models (\Gamma; \Delta) \triangleright \text{Fail}} \quad \text{(uni\textsuperscript{-})}
\]

\[
\frac{\Gamma, \Delta \vdash t_i : A_i;[t_j/x_j]_{j<i}}{\models (\Gamma; \Delta) \triangleright M[#/\bar{x}] \Rightarrow (\Gamma'; \Delta) \triangleright N : B} \quad T[\bar{x};\bar{A}] := M \quad \text{defined tactic}
\]

\[
\frac{\models (\Gamma; \Delta) \triangleright \text{else}_{i=1}^{n}([A \sim_{\Sigma} A_i].M[x_i/h]) \Rightarrow (\Gamma'; \Delta) \triangleright N : B}{\models (\Gamma; \Delta) \triangleright [\sim_{\Sigma} h:A].M \Rightarrow (\Gamma'; \Delta) \triangleright N : B}
\]

\[
\Delta \equiv x_1:A_1 \ldots x_n:A_n \\
\text{else}_{i=1}^{n}M_i \equiv M_1 \text{ else } \ldots \text{ else } M_n \text{ else } \text{Fail}_0
\]

Figure 6.14. Rules for the pre-tactic terms (1).

The rule \((M\textsuperscript{tac})\) is similar, but instead of adding a declaration of a new meta-variable to the state, we add a definition. This effectively means that we introduce a new goal and then solve it with the tactic term \(T\).
The last group of rules shown in Figure 6.16 deals with the term constructors for open terms. These rules have a double role. They explain how to evaluate pre-tactic terms whose main constructor is a sort, variable, meta-variable, application, \(\lambda\)- or \(\Pi\)-abstraction. At the same time they have a controlling function to make sure that the contexts and the terms produced by pre-tactics are well-typed. In fact these rules perform typechecking for the calculus of open terms.

The first four rules are axioms of the semantics since their premises contain only typechecking conditions. They allow us to compute well-typed values given by sorts, variables and meta-variables. In the rules (II) and (\(\lambda\)) we need to evaluate the body of the respective abstractions. We see here that the local context \(\Delta\) is extended when we go under a binder. This is necessary not only for the typechecking of the body, but also to correctly handle the context of any new goals that may be introduced in the body (see rules (\(\text{MV}^{\text{intro}}\)) and (\(\text{MV}^{\text{tac}}\)).

Next comes the rule for typing applications. Here both subterms can be pre-tactics and they need to be evaluated sequentially. In an application \(MN\) we have chosen (for no other particular reason, but that a choice has to be made) to evaluate first \(M\) and then \(N\). This order can be seen if one looks at the state contexts involved in the two premises. When we evaluate \(M\), we start with \(\Gamma\), the context in which we want to evaluate \(MN\), and the resulting new state context \(\Gamma'\) is used as a start for the evaluation of \(N\). As we already noticed before, this order means that some side effects of the evaluation of \(M\) may affect the evaluation of \(N\).

The last rule is the conversion rule that we need to do the typechecking. Note that conversion is only done in the types of the result terms which are open terms and not pre-tactics.

Next, in Figure 6.17 and Figure 6.18 we see the rules for computing the values of tactic terms. The first rule is for the binding tactical \texttt{Bind}. The value of \texttt{Bind}(\(x, i\), \texttt{T}) is computed by first substituting the \(i\)th (from the back) variable of the local context \(\Delta\) for \(x\) in \(\texttt{T}\) and then computing the result.

The following two rules compute the tactic \texttt{Clear} \(x\) where \(x\) is a variable in the local context. After making sure that the context we compute in is valid, we check whether the context after removing \(x\) from \(\Delta\) is still valid. If this is the case, we
\[
\begin{array}{c}
\Gamma, \Delta \vdash \text{Prop} : \text{Type} \\
\models (\Gamma; \Delta) \triangleright \text{Prop} \Rightarrow (\Gamma; \Delta) \triangleright \text{Prop} : \text{Type} \quad (ax)
\end{array}
\]

\[
\begin{array}{c}
\Gamma, \Delta \vdash \text{Type} : \text{Kind} \\
\models (\Gamma; \Delta) \triangleright \text{Type} \Rightarrow (\Gamma; \Delta) \triangleright \text{Type} : \text{Kind} \quad (ax)
\end{array}
\]

\[
\begin{array}{c}
\Gamma, \Delta \vdash x : A \\
\models (\Gamma; \Delta) \triangleright x \Rightarrow (\Gamma; \Delta) \triangleright x : A \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma, \Delta \vdash A : s \\
\Gamma, \Delta \vdash N_i : A_i[N_j/x_j]_{i < j} \\
\models (\Gamma; \Delta) \triangleright m[\vec{N}] \Rightarrow (\Gamma; \Delta) \triangleright m[\vec{N}] : A[\vec{N}/\vec{x}] \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma, \Delta \vdash A : s_1 \\
\models (\Gamma; \Delta, x:A) \triangleright B \Rightarrow (\Gamma'; \Delta, x:A) \triangleright B' : s_2 \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash \Pi x:A.B : s \\
\models (\Gamma; \Delta, x:A) \triangleright M \Rightarrow (\Gamma'; \Delta, x:A) \triangleright N : B \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash \Pi x:A.B : s \\
\Pi x:A.B \triangleright \lambda x:A.M \Rightarrow (\Gamma'; \Delta) \triangleright \lambda x:A.N : \Pi x:A.B \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash \Pi x:A.B : s \\
\models (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright P : \Pi x:A.B \\
\models (\Gamma'; \Delta) \triangleright N \Rightarrow (\Gamma''; \Delta) \triangleright Q : A \\
\models (\Gamma; \Delta) \triangleright MN \Rightarrow (\Gamma''; \Delta) \triangleright PQ : B[Q/x] \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash A =_{\beta \delta} B \\
\Gamma', \Delta \vdash B : s \\
\models (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright N : A \\
\models (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright N : B \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash \text{Prop} : \text{Type} \\
\models (\Gamma; \Delta) \triangleright \text{Prop} \Rightarrow (\Gamma; \Delta) \triangleright \text{Prop} : \text{Type} \quad (ax)
\end{array}
\]

\[
\begin{array}{c}
\Gamma, \Delta \vdash \text{Type} : \text{Kind} \\
\models (\Gamma; \Delta) \triangleright \text{Type} \Rightarrow (\Gamma; \Delta) \triangleright \text{Type} : \text{Kind} \quad (ax)
\end{array}
\]

\[
\begin{array}{c}
\Gamma, \Delta \vdash x : A \\
\models (\Gamma; \Delta) \triangleright x \Rightarrow (\Gamma; \Delta) \triangleright x : A \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma, \Delta \vdash A : s \\
\Gamma, \Delta \vdash N_i : A_i[N_j/x_j]_{i < j} \\
\models (\Gamma; \Delta) \triangleright m[\vec{N}] \Rightarrow (\Gamma; \Delta) \triangleright m[\vec{N}] : A[\vec{N}/\vec{x}] \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma, \Delta \vdash A : s_1 \\
\models (\Gamma; \Delta, x:A) \triangleright B \Rightarrow (\Gamma'; \Delta, x:A) \triangleright B' : s_2 \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash \Pi x:A.B : s \\
\models (\Gamma; \Delta, x:A) \triangleright M \Rightarrow (\Gamma'; \Delta, x:A) \triangleright N : B \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash \Pi x:A.B : s \\
\Pi x:A.B \triangleright \lambda x:A.M \Rightarrow (\Gamma'; \Delta) \triangleright \lambda x:A.N : \Pi x:A.B \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash \Pi x:A.B : s \\
\models (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright P : \Pi x:A.B \\
\models (\Gamma'; \Delta) \triangleright N \Rightarrow (\Gamma''; \Delta) \triangleright Q : A \\
\models (\Gamma; \Delta) \triangleright MN \Rightarrow (\Gamma''; \Delta) \triangleright PQ : B[Q/x] \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma', \Delta \vdash A =_{\beta \delta} B \\
\Gamma', \Delta \vdash B : s \\
\models (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright N : A \\
\models (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright N : B \\
\end{array}
\]

Figure 6.16. Rules for the pre-tactic terms (3).

solve the current goal by a new goal of the same type which has \( x \) removed from its hypotheses. If the new context would be invalided by the removal of \( x \), the tactic fails.

The next rule handles tactic definitions by expanding them and evaluating their body. We would like to remind the reader that \( M \) may contain recursive calls to \( T \) which potentially is a source of non-termination.

The last rule in Figure 6.17 deals with the tactic terms of the form \( \text{(use } M \text{)} \) where \( M \) is a pre-tactic term. This tactic represents a top-level call to a pre-tactic that will compute an instantiation for the current goal. That’s why before evaluating \( M \) in the context of the current goal we substitute the type of the current goal for the special variable \( \ast \). Please note that the hypotheses \( \Delta \) of the goal become the local context for evaluating the pre-tactic term \( M \).
\[
\begin{align*}
| = \Gamma, m[\Delta](T[x_{n-i+1}/x]):A \Rightarrow \Gamma' \\
| = \Gamma, m[\Delta](\text{Bind}(x,i):T):A \Rightarrow \Gamma'
\end{align*}
\]

where \(\Delta \equiv x_1:A_1, \ldots, x_n:A_n\)

\[
\begin{align*}
\Gamma, m[\Delta]:A \vdash \text{Prop} : \text{Type} \\
\Gamma, n[\Delta'] : A \vdash \text{Prop} : \text{Type} \\
| = \Gamma, m[\Delta](\text{Clear } x):A \Rightarrow \Gamma, n[\Delta'] : A, m[\Delta] := n[\Delta']:A
\end{align*}
\]

\[
\begin{align*}
\Gamma, m[\Delta]:A \vdash \text{Prop} : \text{Type} \\
\Gamma, n[\Delta'] : A \not\vdash \text{Prop} : \text{Type} \\
| = \Gamma, m[\Delta](\text{Clear } x):A \Rightarrow \text{Fail}_0
\end{align*}
\]

where \((x:C) \in \Delta, \Delta' = \Delta \setminus (x:C)\)

\[
\begin{align*}
| = \Gamma, m[\Delta](M[t/x]):A \Rightarrow \Gamma' \\
| = \Gamma, m[\Delta](T[t_1 \ldots t_n]):A \Rightarrow \Gamma'
\end{align*}
\]

where \(T[x] := M\) is a defined tactic

\[
\begin{align*}
\Gamma, \Delta \vdash A : s \\
| = (\Gamma; \Delta) \triangleright t[A/s] \Rightarrow (\Gamma'; \Delta) \triangleright N : A \\
| = \Gamma, m[\Delta](\text{use } t):A \Rightarrow \Gamma', m[\Delta] = N:A
\end{align*}
\]

Figure 6.17. Rules for the tactic terms (1).

The two rules in Figure 6.18 deal with the two composition tacticals. The first rule of each group represents the case when all tactic calls have succeeded. In both cases we execute the first tactic and analyze the resulting state context. It is of the form \(\Gamma_0, \Theta\) where \(\Gamma_0\) is an initial segment with length equal to the length of the initial state context \(\Gamma\). It will become clear later (see Theorem 6.12) that \(\Gamma_0\) and \(\Gamma\) contain exactly the same meta-variables, but some declarations in \(\Gamma\) may have become definitions in \(\Gamma_0\). This means that the rest of the state context was created by the call to the first tactic. Therefore the meta-variable declarations in \(\Theta\) represent the new subgoals introduced by the tactic.

In the case of sequential composition we have to apply \(T_2\) to all subgoals generated by \(T_1\) sequentially. In the case of general composition we need to check that the number of subgoals matches the number of tactics and then we proceed as in the case of sequential composition but applying \(T_i\) to the \(i\)th goal. The rules to handle the cases when one of the tactic calls has failed are given in Appendix B.
6.6. Properties

**Proposition 6.3.**

1. If $\vdash (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta') \triangleright N : A$ then $\Delta \equiv \Delta'$.
2. If $\vdash \Gamma, m[\Delta](T):A \Rightarrow \Gamma_1$ then there is a term $N$ and a context $\Gamma'$ such that $\Gamma_1 \equiv (\Gamma', m[\Delta] := N:A)$.

**Theorem 6.4 (Completeness).** If $\Gamma, \Delta \vdash M : A$ then

$$\vdash (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma; \Delta) \triangleright M : A$$

**Proof.** By induction on $M$ using Generation lemma for the typing relation $\vdash$ in the calculus with open terms. \hfill \Box

**Definition 6.5 (Semantic values, equality on values).** A semantic value is an expression that occurs on the right-hand side of $\Rightarrow$ in a semantic judgment. Equality on values is defined as follows: $X_1 \equiv X_2$ if and only if one of the following holds:

- $X_1 \equiv (\Gamma; \Delta) \triangleright M : A$, $X_2 \equiv (\Gamma; \Delta) \triangleright M : B$ and $\Gamma, \Delta \vdash A =_{\beta k} B$.
- $X_1 \equiv \text{Fail}_n$, $X_2 \equiv \text{Fail}_k$ and $n = k$.
- $X_1 \equiv \Gamma \equiv X_2$.

**Definition 6.6.** Let $\Sigma$ be a derivation of $\vdash (\Gamma; \Delta) \triangleright M \Rightarrow X$. We define the main rule of $\Sigma$ as follows. If the last rule of $\Sigma$ is not the conversion rule, then that rule is the main rule of $\Sigma$. If the last rule of $\Sigma$ is the conversion rule, then the main rule of $\Sigma$ is the main rule of the derivation in the premise of the rule.

If $\Sigma$ is a derivation of $\vdash \Gamma, m[\Delta](T):A \Rightarrow X$ then its last rule is its main rule.

**Theorem 6.7 (Consistency).**

1. If $\vdash (\Gamma; \Delta) \triangleright M \Rightarrow X_1$ and $\vdash (\Gamma; \Delta) \triangleright M \Rightarrow X_2$ then $X_1 = X_2$.
2. If $\vdash \Gamma, m[\Delta](T):A \Rightarrow X_1$ and $\vdash \Gamma, m[\Delta](T):A \Rightarrow X_2$ then $X_1 = X_2$.
6. Applications to Tactics

Proof. We proceed by simultaneous induction on the derivation of \( \vdash (\Gamma; \Delta) \Rightarrow M \Rightarrow X_1 \) (resp. \( \vdash \Gamma, m[\Delta]((T):A) \Rightarrow X_1 \)).

Case 1: Let \( M \equiv (M_0 \text{ else } N_0) \). Consider the last rule of the derivation of \( \vdash (\Gamma; \Delta) \Rightarrow M_0 \text{ else } N_0 \Rightarrow X_1 \).

Case 1a: If it is

\[
\vdash (\Gamma; \Delta) \Rightarrow M_0 \Rightarrow (\Gamma'; \Delta) \Rightarrow P : A
\]

then the main rule of the derivation of \( \vdash (\Gamma; \Delta) \Rightarrow M_0 \text{ else } N_0 \Rightarrow X_2 \) must be

\[
\vdash (\Gamma; \Delta) \Rightarrow M_0 \Rightarrow (\Gamma'; \Delta) \Rightarrow P_1 : A_1
\]

because the premises of the other rules would lead to contradiction with the induction hypothesis. However by induction, for \( M_0 \) we have

\[
(\Gamma; \Delta) \Rightarrow P : A = ((\Gamma'; \Delta) \Rightarrow P_1 : A_1)
\]

and since the conversion rule preserves the semantic values, we have \( X_2 = ((\Gamma'; \Delta) \Rightarrow P_1 : A_1) \).

Case 1b: If the last rule is

\[
\vdash (\Gamma; \Delta) \Rightarrow M_0 \Rightarrow \text{Fail}_0
\]

\[
\vdash (\Gamma; \Delta) \Rightarrow N_0 \Rightarrow (\Gamma''; \Delta) \Rightarrow Q : A
\]

Then the induction hypothesis for \( M_0 \) limits the choice for the main rule of the derivation of \( X_2 \) to \((\text{else}_1)\) and \((\text{else}_2)\). The induction hypothesis for \( N \) however leaves us no choice but \((\text{else}_1)\) and therefore \( X_1 = X_2 \).

Case 1c: If the last rule is

\[
\vdash (\Gamma; \Delta) \Rightarrow M_0 \Rightarrow \text{Fail}_0
\]

\[
\vdash (\Gamma; \Delta) \Rightarrow \text{Fail}_n
\]

Again, then the induction hypothesis for \( M_0 \) limits the choice for the main rule of the derivation of \( X_2 \) to \((\text{else}_1)\) and \((\text{else}_2)\), but the induction hypothesis for \( N \) leaves us only with \((\text{else}_2)\) and therefore \( X_1 = X_2 \).

Case 1d: If the last rule is

\[
\vdash (\Gamma; \Delta) \Rightarrow M_0 \Rightarrow \text{Fail}_{n+1}
\]

then from the induction hypothesis for \( M_0 \) we can deduce that the only possibility for the main rule of the derivation of \( X_2 \) is \((\text{else}_3)\).

Case \( n+1 \): By inspecting the rules as in Case 1, we can check that for each main rule of the derivation of \( \vdash (\Gamma; \Delta) \Rightarrow M \Rightarrow X_1 \) (resp. \( \vdash \Gamma, m[\Delta]((T):A) \Rightarrow X_1 \)) there is exactly one possibility for the main rule of \( \vdash (\Gamma; \Delta) \Rightarrow M \Rightarrow X_2 \) (resp. \( \vdash \Gamma, m[\Delta]((T):A) \Rightarrow X_2 \)) whose premises are consistent with the induction hypothesis. Therefore we can conclude that \( X_1 = X_2 \).

\( \square \)
Definition 6.8 (Γ-morphism). Let Γ be a context. The pair \( \delta \equiv \langle \delta_i, \delta_\Sigma \rangle \) is called a \( \Gamma \)-morphism if the following hold:

- \( \delta_i \) is a map from a subset of the meta-variables declared in Γ to the set of B-terms (i.e. the set of open terms).
- \( \delta_\Sigma \) is a context containing no variable declarations (i.e. containing only meta-variable declarations and definitions).

The result of the application of a \( \Gamma \)-morphism \( \delta \) to Γ is defined as the context \( \delta_i(\Gamma), \delta_\Sigma \), where \( \delta_i(\Gamma) \) is defined inductively on the initial segments of Γ:

\[
\begin{align*}
\delta_i(\varepsilon) &= \varepsilon \\
\delta_i(\Gamma', x:A) &= \delta_i(\Gamma'), x:A \\
\delta_i(\Gamma', ?m[\Delta]:A) &= \delta_i(\Gamma'), ?m[\Delta]:A \\
\delta_i(\Gamma', !m[\Delta] := \delta_i(m):A) &= \delta_i(\Gamma'), !m[\Delta] := \delta_i(m):A \\
\delta_i(\Gamma', m[\Delta] := t:A) &= \delta_i(\Gamma'), m[\Delta] := t:A \\
\end{align*}
\]

Definition 6.9. Let Γ be a valid context. The \( \Gamma \)-morphism \( \delta \) is called well-typed if \( \delta(\Gamma) \) is a valid context.

Lemma 6.10. Let \( \Gamma, \Delta \vdash M : A \) and let \( \delta \) be a well-typed \( \Gamma \)-morphism. Then

\( \delta(\Gamma), \Delta \vdash M : A \)

Proof. Follows by Weakening and Cut Lemma for the calculus of open terms. □

Lemma 6.11. Let Γ be a valid context. If \( \delta_1 \) is a well-typed \( \Gamma \)-morphism and \( \delta_2 \) is a well-typed \( \delta_i(\Gamma) \)-morphism, then the composition \( \delta_2 \circ \delta_1 \) is a well-typed \( \Gamma \)-morphism.

Theorem 6.12 (Soundness).

1. Let \( \vdash (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright N : A \). Then
   a. \( \Gamma, \Delta \) is a valid context.
   b. There is a well-typed \( \Gamma \)-morphism \( \delta \) such that \( \delta(\Gamma) \equiv \Gamma' \).
   c. \( \Gamma', \Delta \vdash N : A \).
2. If \( \vdash \Gamma, m[\Delta](\langle T \rangle):A \Rightarrow \Gamma', m[\Delta] := N:A \) then
   a. \( \Gamma, m[\Delta] : A \) is a valid context.
   b. There is a well-typed \( \Gamma \)-morphism \( \delta \) such that \( \delta(\Gamma) \equiv \Gamma' \).
   c. \( \Gamma', \Delta \vdash N : A \).
3. Let \( \vdash (\Gamma; \Delta) \triangleright M \Rightarrow \text{Fail}_n \). Then \( \Gamma, \Delta \) is a valid context.
4. Let \( \vdash \Gamma, m[\Delta](\langle T \rangle):A \Rightarrow \text{Fail}_n \). Then \( \Gamma, m[\Delta] : A \) is a valid context.

Proof. Induction on the definition of \( \vdash \). We consider some of the non-trivial cases below.

Case 1: If the last rule is

\[
\Gamma, \Delta \vdash \text{Prop} : \text{Type} \\
\text{univ}(\langle \sigma(\Gamma); \Delta \rangle, \Sigma, M, N) = \sigma \\
\vdash \langle \sigma(\Gamma); \Delta \rangle \triangleright \sigma(P) \Rightarrow (\Gamma'; \Delta) \triangleright Q : A \\
\vdash (\Gamma; \Delta) \triangleright [M \sim_{\Sigma} N], P \Rightarrow (\Gamma'; \Delta) \triangleright Q : A
\]


then \((1a)\) holds by assumption. For \((1b)\), let \(\delta_0\) be the \(\sigma(\Gamma)\)-morphism from the induction hypothesis. Since \(\sigma\) itself is a well-typed morphism, we can take its composition with \(\delta_0\) to form a \(\Gamma\)-morphism that by Lemma 6.11 is well-typed. The fact that \(\Gamma', \Delta \vdash Q : A\) follows from the induction hypothesis.

**Case 2:** If the last rule is

\[
\frac{
\vdash \Gamma, ?n[\Delta, \Theta][A; \Delta] \triangleright M[n[\Delta, \Theta]/m[\Theta]] \Rightarrow \Gamma'; \Delta \triangleright N : B
}{
\vdash \Gamma; \Delta \triangleright ?m[\Theta][A.M] \Rightarrow \Gamma'; \Delta \triangleright N : B
}
\]

By induction hypothesis we have that \(\Gamma, ?m[\Delta, \Theta][A, \Delta]\) is valid which implies that also \(\Gamma, \Delta\) is. We also have a well-typed morphism \(\delta_0\) such that \(\delta_0(\Gamma, ?n[\Delta, \Theta][A]) \equiv \Gamma'\). Using \(\delta_0\) it is easy to construct a well-typed \(\delta\) such that \(\delta(\Gamma) \equiv \Gamma'\).

**Case 3:** If the last rule is

\[
\frac{
\vdash \Gamma, n[\Delta, \Theta][\langle T \rangle : A \Rightarrow \Gamma', n[\Delta, \Theta] := N : A]
}{
\vdash \Gamma'; n[\Delta, \Theta] := N : A; \Delta \triangleright M[n[\Delta, \Theta]/m[\Theta]] \Rightarrow \Gamma''; \Delta \triangleright P : B
}
\]

we need to use \((2)\) from the induction hypothesis. It gives us that \(\Gamma, n[\Delta, \Theta][A]\) is valid and therefore \(\Gamma, \Delta\) is also valid. Furthermore, there is a well-typed \(\delta_1\) such that \(\delta_1(\Gamma) \equiv \Gamma'\) and a well-typed \(\delta_2\) such that \(\delta_2(\Gamma', n[\Delta, \Theta] := N : A) \equiv \Gamma''\). From \(\delta_1\) and \(\delta_2\) and using the fact that \(\Gamma', \Delta, \Theta \vdash N : A\) (from the induction hypothesis) we can make a well-typed \(\delta\) such that \(\delta(\Gamma) \equiv \Gamma''\).

**Case 4:** If the last rule is

\[
\frac{
\begin{align*}
\Gamma, \Delta & \vdash A : s_1 \\
\vdash (\Gamma; \Delta, x : A) \triangleright B \Rightarrow (\Gamma'; \Delta, x : A) \triangleright B' : s_2
\end{align*}
}{
\vdash (\Gamma; \Delta) \triangleright \Pi x : A.B \Rightarrow (\Gamma'; \Delta) \triangleright \Pi x : A.B' : s_2
}
\]

We need to show that \(\Gamma', \Delta \vdash \Pi x : A.B' : s_2\). From the assumption \(\Gamma, \Delta \vdash A : s_1\) using the morphism \(\delta\) from the induction hypothesis by Lemma 6.10 we have \(\delta(\Gamma), \Delta \vdash A : s_1\). But \(\delta(\Gamma) \equiv \Gamma'\). By induction we also have \(\Gamma', \Delta, x : A \vdash B' : s_2\). Therefore by the rule for \(\Pi\)-formation we have \(\Gamma', \Delta \vdash \Pi x : A.B' : s_2\).

**Case 5:** If the last rule is

\[
\frac{
\begin{align*}
\vdash (\Gamma; \Delta) \triangleright M & \Rightarrow (\Gamma'; \Delta) \triangleright P : \Pi x : A.B \\
\vdash (\Gamma'; \Delta) & \triangleright N \Rightarrow (\Gamma''; \Delta) \triangleright Q : A
\end{align*}
}{
\vdash (\Gamma; \Delta) \triangleright MN \Rightarrow (\Gamma''; \Delta) \triangleright PQ : B(Q/x)
}
\]

To obtain a \(\Gamma\)-morphism we just compose the morphisms \(\delta_1\) and \(\delta_2\) provided by the induction hypotheses. We also know that \(\Gamma', \Delta \vdash P : \Pi x : A.B\) and \(\Gamma'', \Delta \vdash Q : A\). By Lemma 6.10 however we have \(\delta_2(\Gamma'), \Delta \vdash P : \Pi x : A.B\) and since \(\delta_2(\Gamma') \equiv \Gamma''\) by the application rule we can conclude \(\Gamma'', \Delta \vdash PQ : B(Q/x)\).

**Case 6:** If the last rule is

\[
\frac{
\begin{align*}
\vdash \Gamma, m[\Delta][A] & \vdash \text{Prop} : \text{Type} \\
\vdash \Gamma, n[\Delta][\text{Id}][A] & \Rightarrow \Gamma, n[\Delta][A], m[\Delta] := n[\Delta][A]
\end{align*}
}{
\vdash \Gamma, m[\Delta][A] \vdash \text{Prop} : \text{Type}
}
\]

By the assumption in the premise the context \(\Gamma, m[\Delta][A]\) is valid. Since \(n\) is a fresh meta-variable, the morphism \(\delta\) defined by the empty instantiation and the context \(n[\Delta][A]\) is well-typed and \(\delta(\Gamma) \equiv \Gamma, n[\Delta][A]\). Furthermore \(\Gamma, n[\Delta][A] : \Delta \vdash n[\Delta][A] : A\)
(here we use a sloppy notation \( n[\Delta] \) means \( n[\bar{x}] \) where \( \bar{x} \) are the variables declared in \( \Delta \)).

**Case 7:** If the last rule is

\[
\Gamma, m[\Delta]: A \vdash \text{Prop : Type} \\
\Gamma, n[\Delta']: A \vdash \text{Prop : Type}
\]

\[
\models \Gamma, m[\Delta]/(\text{Clear } x): A \Rightarrow \Gamma, n[\Delta']: A, m[\Delta] := n[\Delta']: A
\]

Take the morphism that maps \( \Gamma \) to \( \Gamma, n[\Delta']: A \). It is well-typed because of the second premise of the rule.

**Case 8:** If the last rule is

\[
\Gamma, \Delta \vdash A : s \\
\models (\Gamma; \Delta) \triangleright t[A/s] \Rightarrow (\Gamma'; \Delta) \triangleright N : A \\
\models \Gamma, m[\Delta]/(\text{use } t): A \Rightarrow \Gamma', m[\Delta] = N:A
\]

To prove (2) we use the premise \( \Gamma, \Delta \vdash A : s \) and the induction hypothesis (1).

**Case 9:** If the last rule is

\[
\models \Gamma, m[\Delta]/(T_1): A \Rightarrow \Gamma_0, \Theta \\
\models \Gamma_i-1, \Theta_i-1, n_i[\Delta]/(T_2): A_i \Rightarrow \Gamma_i \\
\models \Gamma, m[\Delta]/(T_1; T_2): A \Rightarrow \Gamma_k, \Theta_k
\]

use Lemma 6.11 to compose the morphisms obtained from the induction hypotheses for the premises of the rule. □

### 6.7. Example: A Tactic for Deciding Minimal IPC

In this section we will show how a simple decision procedure can be encoded in the calculi that we presented so far. We illustrate the use of the semantics to prove properties of this encoding.

As an illustration of the possibilities of \( \mathcal{L}_{tac} \), Delahaye [Del00] presents a decision procedure for intuitionistic propositional logic based on the contraction-free LJT of Dyckhoff [Dyc92]. We will use the minimal version of LJT to construct a tactic Decide that can decide tautologies in minimal propositional intuitionistic logic. The rules of LJT are presented in Figure 6.19 and the definition of Decide

\[
\begin{array}{c|c|c}
(ax) & \Gamma, A \vdash A & \Gamma, A \vdash B \\
& \Gamma \vdash A \rightarrow B & (\rightarrow R) \\
\leftarrow 1 & \Gamma, p, B \vdash A & \Gamma, D \rightarrow B \vdash C \rightarrow D \\
& \Gamma, p, p \rightarrow B \vdash A & \Gamma, B \vdash A \\
& \Gamma, (C \rightarrow D) \rightarrow B \vdash A & (\rightarrow 2) \\
\end{array}
\]

**Figure 6.19.** Rules of LJT for the implicational fragment of IPC. The proposition \( p \) is atomic.
in Figure 6.20. Since LJT considered backwards as goal-reduction system is terminating, we base the tactic on brute-force search trying consecutively to construct a proof that ends with one of the rules in Figure 6.19.

**Lemma 6.13** ([Dyc92]). $\Gamma \vdash_{\text{L JT}} \varphi$ if and only if $\Gamma \vdash_{\text{IPC}} \varphi$

![](figure.png)

**Figure 6.20.** Implementing a decision procedure for minimal IPC.

Let $\Gamma, m[\Delta]:A$ be a valid context. Define $|\Delta|_\Gamma$ as the multiset

$$\{A : (x:A) \in \Delta \text{ and } \Gamma, \Delta \vdash A : \text{Prop}\}$$

**Theorem 6.14** (Correctness and completeness of Decide).

Let $\Gamma, m[\Delta]:A \vdash A:\text{Prop}$.

1. $|\Delta|_\Gamma \vdash_{\text{LJ}} A$ if and only if there exists a context $\Gamma'$ and a term $N$ such that every unsolved meta-variable of $\Gamma'$ occurs in $\Gamma$ and

$$\models \Gamma, m[\Delta]|(\text{Decide} : A \rightarrow \Gamma', m[\Delta] := N : A)$$

2. $\models \Gamma, m[\Delta]|(\text{Decide} : A \rightarrow \text{Fail}_0$ if and only if $|\Delta|_\Gamma \not\vdash_{\text{LJ}} A$.

**Proof.** For the correctness we need to observe that Decide does not introduce unsolved meta-variables. Therefore by the soundness theorem of the semantics we have that $\Gamma'$ is obtained from $\Gamma$ by weakening with meta-variable definitions. Furthermore, the only free propositional variables of $N$ are from $\Delta$. Therefore by the Curry-Howard isomorphism $N$ represents encoding of a proof of $A$ from assumptions $|\Delta|_\Gamma$.

Define a measure function $\varphi$ on implicational propositional formulas as follows:

$$\varphi(p) = 1 \quad \text{if } p \text{ is atomic}$$

$$\varphi(A \rightarrow B) = \varphi(A) + \varphi(B) + 1 \quad \text{else}$$

Dyckhoff [Dyc92] uses the measure to define well-founded order on formulas ($A < B$ iff $\varphi(A) < \varphi(B)$). This order induces a well-founded order on finite multisets of formulas [DM79]. Following the idea of [Dyc92], for a given sequent $|\Delta|_\Gamma \vdash A$
we will prove completeness of Decide by induction on the well-founded order on the multiset \(|\Delta|_\Gamma, A\) using the definition of the semantics.

Suppose that \(\Gamma, m[\Delta]; E \vdash E : \text{Prop} \) and \(|\Delta|_\Gamma \vdash E\). We need to show that there are \(\Gamma'\) and \(N\) such that
\[
\models \Gamma, m[\Delta](\text{ Decide }): E \rightarrow \Gamma', m[\Delta] := N : E
\]
and \(\Gamma'\) is a weakening of \(\Gamma\) with meta-variable definitions only.

Following the structure of Decide we consider four cases.

Case 1: There is \(x\) such that \((x; E) \in \Delta\). According to the rules of the semantics we need to evaluate the term \([\sim h; E]; h\) in context \(\Gamma; \Delta\). Since \(x; E \in \Delta\), the unification will find the first such \(x\) in \(\Delta\) and we have
\[
\models (\Gamma, \Delta) \triangleright [\sim h; E], h \Rightarrow (\Gamma; \Delta) \triangleright x : E
\]
Therefore we get that
\[
\models (\Gamma, \Delta) \triangleright [\sim h; E], h \Rightarrow (\Gamma; \Delta) \triangleright \text{Fail}_0
\]

Therefore we need to evaluate \([E \sim_{A,B: \text{Prop}} A \rightarrow B], \lambda x: A.?n = (\text{ Decide }): B, n\) in context \(\Gamma; \Delta\). Since \(E\) is an implication and there is no \(x\) such that \((x; E) \in \Delta\). Since \(E\) is not among the assumptions in \(\Delta\), we have
\[
\models (\Gamma, \Delta) \triangleright [\sim h; E], h \Rightarrow (\Gamma; \Delta) \triangleright \text{Fail}_0
\]

This requires computing of
\[
\Gamma, n[\Delta, x; A_0](\text{ Decide }): B_0
\]
Now we notice that the sequent \(|\Gamma, x; A_0|_\Gamma \vdash B_0\) is strictly smaller than the sequent \(|\Delta|_\Gamma \vdash A_0 \rightarrow B_0\) and therefore we can apply the induction hypothesis. If \(|\Delta, x; A_0|_\Gamma \vdash B_0\) then
\[
\models \Gamma, m[\Delta](\text{ Decide }): (A_0 \rightarrow B_0) \rightarrow \Gamma', m[\Delta] := N : B_0
\]
and therefore
\[
\models \Gamma, m[\Delta](\text{ Decide }): (A_0 \rightarrow B_0) \rightarrow \Gamma', m[\Delta] := N : B_0, m[\Delta] := \lambda x: A_0.n[\Delta, x] : A_0 \rightarrow B_0
\]

On the other hand, if \(|\Delta, x; A_0|_\Gamma \vdash B_0\) is not provable, by induction we get
\[
\models \Gamma, n[\Delta, x; A_0](\text{ Decide }): B_0 \rightarrow \text{Fail}_0
\]
and hence
\[
\models \Gamma, m[\Delta](\text{ Decide }): (A_0 \rightarrow B_0) \rightarrow \text{Fail}_0
\]

Case 3: Assume that both previous cases have failed and that \(\Delta\) contains two assumptions \(h_1 : A \rightarrow B\) and \(h_2 : A\) for some propositions \(A\) and \(B\). First we need to evaluate \([A \sim_{C,D: \text{Prop}} C \rightarrow D], \text{Fail}_1\) in context \(\Gamma; \Delta\). There are two cases: If \(A\) is an implication (say \(C \rightarrow D\), we have
\[
\models (\Gamma; \Delta) \triangleright [A \sim_{C,D: \text{Prop}} C \rightarrow D], \text{Fail}_1 \Rightarrow (\Gamma; \Delta) \triangleright \text{Fail}_1
\]
which means that
\[ \models (\Gamma; \Delta) \triangleright ([A \sim_{C,D;\text{Prop}} C \rightarrow D], \text{Fail}_1) \text{else...} \Rightarrow (\Gamma; \Delta) \triangleright \text{Fail}_0 \]
and this will force a new choice of \( h_1 \) and \( h_2 \). Therefore if for all \( h_1 \) and \( h_2 \) the corresponding \( A \) is an implication, the term \([\sim_{A,B;\text{Prop}} h_1: A \rightarrow B], [\sim h_2: A] \ldots \) will evaluate to \( \text{Fail}_0 \). Let us consider the case when for a given choice of \( h_1 \) and \( h_2 \) the corresponding \( A \) is not an implication. Then following the rules of the semantics
\[ \models (\Gamma; \Delta) \triangleright ([A \sim_{C,D;\text{Prop}} C \rightarrow D], \text{Fail}_1) \Rightarrow (\Gamma; \Delta) \triangleright \text{Fail}_0 \]
and taking the else-branch we need to evaluate
\[ \exists n[x:B] = \langle \text{Clear } h_1; \text{Decide} \rangle \cdot E \cdot n[(h_1 h_2)] \]
in context \( \Gamma; \Delta \). This means computing \( \Gamma, n[\Delta, x:B] \langle \text{Clear } h_1; \text{Decide} \rangle \cdot E \) which of course is reduced to computing \( \Gamma, n[\Delta', x:B] \langle \text{Decide} \rangle \cdot E \) where \( \Delta' \) is \( \Delta \) without \( h_1 \). But now \( |\Delta', x:B|_{\Gamma}, E \) as a multiset is strictly smaller than \( |\Delta|_{\Gamma}, E \). Therefore by the induction hypothesis \( \Gamma, n[\Delta', x:B] \langle \text{Decide} \rangle \cdot E \) succeeds if and only if \( |\Delta', x:B|_{\Gamma} \vdash E \).

Suppose that there is proof of \( |\Delta', x:B|_{\Gamma} \vdash E \). Then by \( (\rightarrow \frac{1}{E}) \) there is a proof of \( |\Delta|_{\Gamma} \vdash E \). On the other hand, by induction we get
\[ \models \Gamma, n[\Delta', x:B] \langle \text{Decide} \rangle \cdot E \rightarrow \Gamma', n[\Delta'] := N : E \]
and hence
\[ \models \Gamma, m[\Delta] \langle \text{Decide} \rangle \cdot E \rightarrow \Gamma', n[\Delta'] := N : E, m[\Delta] := n[\Delta'] : E \]
If there is no proof of \( |\Delta', x:B|_{\Gamma} \vdash E \) for any \( h_1 \) and \( h_2 \), then
\[ \models \Gamma, m[\Delta] \langle \text{Decide} \rangle \cdot E \rightarrow \text{Fail}_0 \]
Case 4: Suppose that the previous cases have failed and there is an assumption \( h \) of type \( (C \rightarrow D) \rightarrow B \) for some \( B, C \) and \( D \). Then we need to compute
\[ \Gamma, k[\Delta, x:B] = \langle \text{Clear } h; \text{Decide} \rangle \cdot E \]
which reduces to computing
\[ \Gamma', k_1[\Delta', x:B] = \langle \text{Decide} \rangle \cdot E \]
where \( \Delta' \) is \( \Delta \) without the assumption \( h : (C \rightarrow D) \rightarrow B \). Again, \( |\Delta', x:B|_{\Gamma}, E \) as a multiset is smaller than \( |\Delta|_{\Gamma}, E \) and the induction hypothesis applies. If \( |\Delta|_{\Gamma} \vdash E \) is not provable we continue with a new search for \( h \). If it is provable, there is a term \( N_1 \) and a context \( \Gamma' \) such that
\[ \models \Gamma, k_1[\Delta', x:B] = \langle \text{Decide} \rangle \cdot E \rightarrow \Gamma', k_1[\Delta', x:B] = N_1 : E \]
Next, starting from the context \( \Gamma', k_1[\Delta', x:B] = N_1 : E, k[\Delta, x:B] := k_1[\Delta', x] : E \) (call it \( \Gamma_1 \)) and \( \Delta \) we need to evaluate \( ?n[x:D \rightarrow B] = \langle \text{Clear } h; \text{Decide} \rangle : C \rightarrow D \ldots \)

Of course, this means computing \( \Gamma_1, n[\Delta, x:D \rightarrow B] = \langle \text{Clear } h; \text{Decide} \rangle : C \rightarrow D \) and in turn it is reduced to computing \( \Gamma_1, n_1[\Delta', x:D \rightarrow B] = \langle \text{Decide} \rangle : C \rightarrow D \), where as before \( \Delta' \) is \( \Delta \) without \( h \). By induction we get that this evaluation is successful if and only if \( |\Delta', x:D \rightarrow B|_{\Gamma} \vdash E \).
In case it succeeds, we get \( \Gamma'' \) and \( N_2 \) such that
\[
| = \Gamma_1, n_1[\Delta, x:D \to B] = (\text{Clear } h; \text{Decide}) : (C \to D) \rightarrow \Gamma_1, n_1[\ldots] := N_2 : E, n[\ldots] := n_1[\Delta', x] : E
\]
In case it fails we need to look for another \( h \) and repeat the above procedure for it. If no more assumptions can be found, the procedure \text{Decide} fails.

Towards contradiction, assume that there is a proof of \( \Delta_1 \vdash E \) and
\[
\Gamma, m[\Delta] = (\text{Decide}) \rightarrow \text{Fail}_0
\]
Consider the last rule of that proof. If it was (\( ax \)), then we would have Case 1 and \text{Decide} would succeed. Similarly, if the rule was (\( \rightarrow_2 \)), \( E \) would be an implication \( A \to B \) and the sequent \( [\Delta, x:A]_1 \vdash B \) would be provable and we would have Case 2. If the last rule was (\( \rightarrow_1^{\Delta} \)) there would be two assumptions \( h_1 : A \to B \) and \( h_2 : A \) such that \( A \) is not implication and \( [\Delta', x:B]_1 \vdash E \) is provable and therefore Case 3 would succeed. Similarly, if the last rule is (\( \rightarrow_3^{\Delta} \)), Case 4 succeeds. However, since all four cases have failed, none of the four LJT rules is the last rule of the proof and therefore there is no proof. \( \square \)

6.8. Related and Future Work

6.8.1. Related Work. The approach to tactics that we take has been influenced by the Pure Pattern Type Systems (PPTs) of Barthe et al. [BCKL03] which is an extension on previous work on the rewriting calculus (see e.g. [HCL01]). Our approach relates to this work in the following way: In a PPT one abstracts a pattern and then this pattern is matched to the actual argument given to the function. As in our case where we have unification constraints, this results in an instantiation. The fact that matching is performed (as opposed to unification) means that the pattern reduction is localized only to the redex that we contract, while in our case there may be side effects that influence the whole state and this makes the reduction non-local.

One may choose for a declarative approach to tactics. For example, in [DB87] De Bruijn presents the Mathematical Vernacular (MV), where one writes formalized texts line by line, each of them being a statement or a definition. In Mizar [Rud92] the user writes the proofs in a dedicated proof languages that is quite readable for a human.

The notion of tactic was introduced in LCF [GMW79] and was taken up in many systems offering backward proof in LCF-style. Among those systems we can mention Coq [CDT03], Nuprl [CAB86], Isabelle [Pau93], Lego [Pol], etc. In [Gro92] Groote presents a formal mathematical vernacular based on a set of primitive tactics and sequential and alternative composition. The semantics is given by meta-functions on objects that encode the open obligations. The semantics is proven to be sound and complete with respect to a class of PTS called PTSs for logic.

The need for a dedicated language for tactics was recognized in the system Coq for which Delahaye presents the tactic language \( \mathcal{L}_{tac} \) [Del10]. The idea is to take the basic tactics of Coq and using LCF-style tacticals, (unrestricted) recursion and matching on proof-contexts to allow the user to define new tactics. He presents
informal semantics of $\mathcal{L}_{tac}$ that explains the execution of complex tactics in terms of sequences of calls to the basic tactics. The language is not Turing complete, but is powerful enough to encode some interesting tactics like a decision procedure for the full LJT of [Dyc92] (compare with Section 6.7), for type isomorphisms and for permutations on closed sets (see the Coq Manual [CDT03], Chapter 11).

Recently the author became aware of further work by Delahaye [Del02] (and [Del01] in French) that is closely related to the exposition in this chapter. The idea to provide a single calculus in which terms and tactics are more or less freely mixed is already present in the proof language $\mathcal{L}_{pdt}$ proposed there. It even goes further by offering support for the declarative style of proof. The discovery that the idea for a ‘tactics-as-terms’ calculus was already invented came after the main part of this chapter was written and was a bit of a disappointment for the author. We feel, however that it will be beneficial to compare the two approaches to this idea and see how they relate. A brief comparison follows here.

Leaving aside the declarative features of $\mathcal{L}_{pdt}$ which we do not address in this chapter, the language contains primitives to construct proof-terms either directly using the term constructors of the calculus of inductive constructions (CIC) or by calls to a fixed set of Coq tactics like Apply, Assumption, Intros, etc. $\mathcal{L}_{pdt}$ is the first language of tactics to be given a formal semantics. Its big-step semantics allows evaluation of terms into values that are generated from the terms of CIC by adding ?-terms. In [Del02] it is also claimed that the semantics is correct with respect to CIC without universes, but it is not specified in which respect and a proof is not provided.

In the work presented here we take an approach that is less system dependent because we base it on an extension of a calculus with open terms. The benefits of using a generic extension with open terms as a basis for the semantics are visible in details like the ability to handle goals with meta-variables for example. The study of the properties of the calculus of open terms allows us to formulate and prove the soundness criteria for the semantics.

Another difference lies in the choice of primitive non-term constructors for the tactics. While in $\mathcal{L}_{pdt}$ a fixed number of basic Coq tactics is provided, we have primitives based on unification, exceptions, explicit meta-variable introduction, and recursive definitions. As we have shown in Section 6.4, this allows us to define many commonly used tactics. The added value of this approach is that when we define a new tactic we do not have to reprove the soundness of the whole semantics – it comes for free.

In our calculus we have a special binding tactical Bind that allows us to address the issue of fresh name generation in tactic scripts. As far as we know the use of a binding tactical is a new approach to the problem.

One may ask how does the division that we make between tactics and pre-tactics relate to $\mathcal{L}_{pdt}$. It seems that this division is also present there in the form of the two evaluation modes of the semantics. When the type of the term being evaluated is known (from the goal), the evaluation goes into verification mode. Sometimes however the type has to be inferred (e.g. in application terms) and then the evaluation is in inference mode. Since many terms can be evaluated in both modes, this shows that there is some potential for flexibility in our division of
tactics and pre-tactics. We have chosen to put all features for which it is possible in the category of pre-tactics, but they can always be lifted to the tactic level with the help of use.

In this chapter we used contexts with definitions to model the state of the theorem prover. In [JNS03] such contexts are used to capture the information about the steps one has made in an informal proof when it is translated to type theory.

6.8.2. Future Work and Open Problems. There are still some interesting questions that one may try to answer.

6.8.2.1. Small-Step Semantics. We have presented a big-step semantics that relates a tactic term in a context to an open term in another context that represent its value. We believe that it is possible to define small-step semantics by giving rules that rewrite one tactic term (in a context) to another tactic term (in another context). For example, the rules for \([M \leadsto_{eaten} N].P\), \([\sim_{eaten} h; A].P\) and \(?m[\Theta]:A.M\) could be something like:

\[
\begin{align*}
(\Gamma; \Delta) \triangleright [M \leadsto_{eaten} N].P & \rightarrow (\Gamma; \Delta) \triangleright \text{Fail} \quad \text{uni}(\ldots) = \text{Fail}_0 \\
(\Gamma; \Delta) \triangleright [M \leadsto_{eaten} N].P & \rightarrow (\sigma\Gamma; \Delta) \triangleright \sigma P \quad \text{uni}(\ldots) = \sigma \\
(\Gamma; \Delta) \triangleright [\sim_{eaten} h; A].P & \rightarrow (\Gamma; \Delta) \triangleright \text{else}^n_{i=1}([A \sim_{eaten} A_i].P[x_i/h]) \\
(\Gamma; \Delta) \triangleright ?m[\Theta]:A.M & \rightarrow (\Gamma, n[\Delta, \Theta];A; \Delta) \triangleright M[n/m]
\end{align*}
\]

There are also some problems. A naive semantics of the \texttt{else} construct like

\begin{align*}
\text{else } N & \rightarrow M \quad M \text{ is pure} \\
\text{Fail}_0 \text{ else } N & \rightarrow N \\
\text{Fail}_{n+1} \text{ else } N & \rightarrow \text{Fail}_n
\end{align*}

does not take the context into consideration. When the \(M\) branch of \texttt{M else N} fails we need to undo all changes to the context inflicted by the evaluation of \(M\) before the failure. Hence, we need to record the state before the evaluation of \(M\) was started. This can be done with a two-phase rule:

\[
\begin{align*}
(\Gamma; \Delta) \triangleright M \text{ else } N & \rightarrow (\Gamma; \Delta) \triangleright M \text{ else}_\Gamma N \\
(\Gamma; \Delta) \triangleright M \text{ else}_\Gamma N & \rightarrow (\Gamma; \Delta) \triangleright M \quad M \text{ is pure} \\
(\Gamma; \Delta) \triangleright \text{Fail}_0 \text{ else}_\Gamma N & \rightarrow (\Gamma_0; \Delta) \triangleright N \\
(\Gamma; \Delta) \triangleright \text{Fail}_{n+1} \text{ else}_\Gamma N & \rightarrow (\Gamma_0; \Delta) \triangleright \text{Fail}_n
\end{align*}
\]

Another technical problem is the detection of the newly introduced goals when we evaluate tacticals. When we want to compute \(T_1; T_2\), the tactic \(T_2\) needs to be applied to all goals introduced by \(T_1\). To recognize which are those goals we need to put extra information in the state that allows us to detect the new goals. One option is to put a kind of stack structure on the context that is transparent to the other rules:

\[
\begin{align*}
\Gamma, m[\Delta](T_1; T_2):A & \rightarrow \Gamma, (T_2 m[\Delta](T_1):A) \\
\Gamma, (T\Gamma', m[\Delta] = N:A) & \rightarrow \Gamma, \Gamma'(T), m[\Delta] = N:A
\end{align*}
\]

Such a small-step semantics needs to be proven compatible with the big-step semantics by showing that a (pre-)tactic term in a context has a value in the big-step semantics if and only if there is a reduction path leading to this term in the small-step semantics. To ensure that, we need to show confluence, which can be
achieved by defining the rewrite rules in such a way that only one rule is applicable at every step.

6.8.2.2. Definability of computational tactics. Consider the term
\[ [M \sim_\Sigma (\lambda x:A.P[x])Q] \Rightarrow P \]
where \( \Sigma \) is the context \( A:\text{Type}, B[x:A]\text{Type}, P[x:A];B[x], Q:A \). If we restrict the unification procedure to \( \alpha\beta \)-convertibility, we could internalize the notion of \( \beta \)-reduction and possibly define a tactic that performs computations on well-typed terms. One can imagine a tactic \( \text{Norm} \) defined roughly like
\[
\text{Norm}[M] :=
\begin{cases}
M \sim_\Sigma (\lambda x:A.P[x])Q, \text{Norm}[P]\text{Norm}[Q] \\
\text{else}
M \sim_\Sigma \lambda x:A.P[x], \text{Norm}[A] \cdot \text{Norm}[P[x]]
\end{cases}
\]
that normalizes a term. To achieve that we need to make some changes like providing a means to specify what kind of unification algorithm is to be used in a particular unification constraint and allowing general pre-tactic terms in the types of abstractions. For example, the rule for \( \lambda \)-abstractions could become:
\[
\begin{array}{l}
\vdash (\Gamma; \Delta) \triangleright A_0 \Rightarrow (\Gamma'; \Delta) \triangleright A : s \\
\vdash (\Gamma'; \Delta, x:A) \triangleright M \Rightarrow (\Gamma'', \Delta, x:A) \triangleright N : B \\
\Gamma'', \Delta \vdash \Pi x:A.B : s
\end{array}
\]
\[
\vdash (\Gamma; \Delta) \triangleright \lambda x:A_0.M \Rightarrow (\Gamma''; \Delta) \triangleright \lambda x:A.N : \Pi x:A.B
\]

6.8.2.3. Typing System. There remains an open problem with respect to the possibility to provide a typing system that assigns types to tactic terms. This will most likely require restrictions on the possible recursive definitions, but that is not the only problem. For example, in the definition of \( \text{Apply}[\varphi, \psi; \text{Prop}, M; \psi] \) we had a subterm of the shape \( [\psi \sim_{A,B[x:A]} \Pi x:A.B[x]].?m:AAppl[y, B[m], (M m)] \) where \( M : \psi \) and \( \varphi \) is the current goal. To type the term we need to conclude that \( (M m) \) is of type \( B[m] \) under the assumptions imposed by the unification constraint.
APPENDIX A

\(\lambda\)oHOL with Global Definitions

\[\vdash \text{Prop} : \text{Type} \quad (ax_1)\]
\[\vdash \text{Type} : \text{Kind} \quad (ax_2)\]
\[\frac{\Gamma, x : A \vdash x : A}{\Gamma \vdash A : s} \quad (\text{start})\]
\[\frac{\Gamma \vdash M : B}{\Gamma, x : A \vdash M : B} \quad (\text{weak})\]
\[\frac{\Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A. B : s_2} \quad (\Pi)\]
\[\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A. M : \Pi x : A. B} \quad (\lambda)\]
\[\frac{\Gamma \vdash M : \Pi x : A. B \quad \Gamma \vdash N : A}{\Gamma \vdash \Pi N : B[N/x]} \quad (\text{app})\]
\[\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s \quad \Gamma \vdash A = \beta B}{\Gamma \vdash M : B} \quad (\text{conv})\]
\[\frac{\Gamma, x : A \vdash A_i: A_i[N_j/x_j][j<i]}{\Gamma \vdash m[N]: A[N/\vec{x}] \quad (MV_{\text{start}})}\]
\[\frac{\Gamma, \Theta \vdash A : s \quad \Gamma \vdash M : B}{\Gamma, m(\Theta): A \vdash M : B} \quad (MV_{\text{weak}})\]
\[\frac{\Gamma, \Theta \vdash N : A : s \quad \Gamma \vdash M : B}{\Gamma, m(\Theta) := N: A \vdash M : B} \quad (MD_{\text{weak}})\]

\[\Gamma_1, m[\vec{x}: A] := N : B, \Gamma_2 \vdash m[\vec{t}] \rightarrow^4 N[\vec{t}/\vec{x}]\]
\[\Gamma \vdash (\lambda x : A. M)N \rightarrow^\beta M[N/x]\]
APPENDIX B

Rules for Evaluation of Tactics and Pre-tactics

\[
\begin{array}{l}
\Gamma, \Delta \vdash \text{Prop : Type} \\
\Rightarrow (\Gamma; \Delta) \triangleright \text{Fail}_n \Rightarrow \text{Fail}_n \quad \text{(Fail)}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright P : A \\
\Rightarrow (\Gamma; \Delta) \triangleright (M \text{ else } N) \Rightarrow (\Gamma'; \Delta) \triangleright P : A \quad \text{(else$^+$)}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow (\Gamma; \Delta) \triangleright M \Rightarrow \text{Fail}_n \\
\Rightarrow (\Gamma; \Delta) \triangleright N \Rightarrow \text{Fail}_n \quad \text{(else$^-$)}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow (\Gamma; \Delta) \triangleright (M \text{ else } N) \Rightarrow \text{Fail}_n \\
\Rightarrow (\Gamma; \Delta) \triangleright M \Rightarrow \text{Fail}_{n+1} \quad \text{(else$^-_2$)}
\end{array}
\]

\[
\begin{array}{l}
\Gamma, \Delta \vdash \text{Prop : Type} \\
\Rightarrow \text{uni}((\Gamma, \Delta), \Sigma, M, N) = \sigma \\
\Rightarrow (\sigma(\Gamma); \Delta) \triangleright \sigma(P) \Rightarrow (\Gamma'; \Delta) \triangleright Q : A \quad \text{(uni$^+$)}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow (\Gamma; \Delta) \triangleright [M \sim_{\Sigma} N].P \Rightarrow (\Gamma'; \Delta) \triangleright Q : A
\end{array}
\]

\[
\begin{array}{l}
\Gamma, \Delta \vdash \text{Prop : Type} \\
\Rightarrow \text{uni}((\Gamma, \Delta), \Sigma, M, N) = \sigma \\
\Rightarrow (\sigma(\Gamma); \Delta) \triangleright \sigma(P) \Rightarrow \text{Fail}_n \quad \text{(uni$^{f\text{ail}}$)}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow (\Gamma; \Delta) \triangleright [M \sim_{\Sigma} N].P \Rightarrow \text{Fail}_n
\end{array}
\]

\[
\begin{array}{l}
\Gamma, \Delta \vdash \text{Prop : Type} \\
\Rightarrow \text{uni}((\Gamma, \Delta), \Sigma, M, N) = \text{Fail} \quad \text{(uni$^-$)}
\end{array}
\]

\[
\begin{array}{l}
\Rightarrow (\Gamma; \Delta) \triangleright [M \sim_{\Sigma} N].P \Rightarrow \text{Fail}_0
\end{array}
\]
\[
\begin{align*}
\Gamma, \Delta \vdash t_i : A_i[t_j/x_j]_{j<i} &; T[t/x] \Rightarrow (\Gamma'; \Delta) \triangleright N : B & T[x,\hat{A}] := M \\
\Rightarrow (\Gamma; \Delta) \triangleright T[t/x] \Rightarrow (\Gamma'; \Delta) \triangleright N : B & \text{ defined tactic} \\
\Gamma, \Delta \vdash t_i : A_i[t_j/x_j]_{j<i} &; M[t/x] \Rightarrow \text{Fail}_n \\
\Rightarrow (\Gamma; \Delta) \triangleright M[t/x] \Rightarrow \text{Fail}_n & \text{ defined tactic} \\
\Rightarrow (\Gamma; \Delta) \triangleright \text{else}^n_{i=1}([A \sim \Sigma A_i].M[x_i/h]) \Rightarrow (\Gamma'; \Delta) \triangleright N : B & \\
\Rightarrow (\Gamma; \Delta) \triangleright [\sim \Sigma h:A].M \Rightarrow (\Gamma'; \Delta) \triangleright N : B & \\
\Rightarrow (\Gamma; \Delta) \triangleright \text{else}^n_{i=1}([A \sim \Sigma A_i].M[x_i/h]) \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma; \Delta) \triangleright [\sim \Sigma h:A].M \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma, \Delta \vdash [\sim \Sigma h:A].M) \Rightarrow (\Gamma'; \Delta) \triangleright N : B & \\
\Rightarrow (\Gamma, \Delta) \triangleright ?m[\Theta] : A.M \Rightarrow (\Gamma'; \Delta) \triangleright N : B & \\
\Rightarrow (\Gamma, n[\Delta, \Theta] : A; \Delta) \triangleright M[n[\Delta, \Theta]/m[\Theta]] \Rightarrow (\Gamma'; \Delta) \triangleright N : B & \\
\Rightarrow (\Gamma, n[\Delta, \Theta] : A; \Delta) \triangleright M[n[\Delta, \Theta]/m[\Theta]] \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma', n[\Delta, \Theta] : A) \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma', n[\Delta, \Theta] := N : A) \triangleright A.M \Rightarrow (\Gamma''; \Delta) \triangleright P : B & \\
\Rightarrow (\Gamma, n[\Delta, \Theta] \triangleright [\sim \Sigma h:A].M) \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma, n[\Delta, \Theta] \triangleright ?m[\Theta] = \langle T \rangle : A.M) \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma, n[\Delta, \Theta] \triangleright ?m[\Theta] = \langle T \rangle : A.M) \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma, n[\Delta, \Theta] \triangleright ?m[\Theta] = \langle T \rangle : A.M) \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma', n[\Delta, \Theta] := N : A) \triangleright A.M \Rightarrow M[n[\Delta, \Theta]/m[\Theta]] \Rightarrow \text{Fail}_n & \\
\Rightarrow (\Gamma, \Delta) \triangleright ?m[\Theta] = \langle T \rangle : A.M \Rightarrow \text{Fail}_n & \\
\end{align*}
\]

\( n \) is fresh; \( T \) is a tactical.
\[
\begin{array}{l}
\Gamma, \Delta \vdash \text{Prop} : \text{Type} \\
\implies (\Gamma; \Delta) \triangleright \text{Prop} \Rightarrow (\Gamma; \Delta) \triangleright \text{Prop} : \text{Type} \\
\hline
\Gamma, \Delta \vdash \text{Type} : \text{Kind} \\
\implies (\Gamma; \Delta) \triangleright \text{Type} \Rightarrow (\Gamma; \Delta) \triangleright \text{Type} : \text{Kind} \\
\hline
\Gamma, \Delta \vdash x : A \\
\hline
\implies (\Gamma; \Delta) \triangleright x \Rightarrow (\Gamma; \Delta) \triangleright x : A \\
\hline
\Gamma, \Delta \vdash A : s \\
\Gamma, \Delta \vdash N_i : A_{[N_j/x_j]}_{j<i} \\
\hline
\implies (\Gamma; \Delta) \triangleright m[\vec{N}] \Rightarrow (\Gamma; \Delta) \triangleright m[\vec{N}] : A[\vec{N}/\vec{x}] \\
\hline
\Gamma, \Delta \vdash A : s_1 \\
\implies (\Gamma; \Delta, x:A) \triangleright B \Rightarrow (\Gamma'; \Delta, x:A) \triangleright B' : s_2 \\
\hline
\implies (\Gamma; \Delta) \triangleright \Pi x : A.B \Rightarrow (\Gamma'; \Delta) \triangleright \Pi x : A.B' : s_2 \\
\hline
\hline
\Gamma', \Delta \vdash \Pi x : A.B : s \\
\implies (\Gamma', \Delta, x:A) \triangleright M \Rightarrow (\Gamma', \Delta, x:A) \triangleright N : B \\
\hline
\implies (\Gamma; \Delta) \triangleright \lambda x : A.M \Rightarrow (\Gamma'; \Delta) \triangleright \lambda x : A.N : \Pi x : A.B \\
\hline
\hline
\Gamma', \Delta \vdash \lambda x : A.M = \beta \delta B \\
\Gamma', \Delta \vdash B : s \\
\implies (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright N : A \\
\hline
\implies (\Gamma; \Delta) \triangleright M \Rightarrow (\Gamma'; \Delta) \triangleright N : B \\
\end{array}
\]
\[\Gamma, m[\Delta] A \vdash \text{Prop} : \text{Type}\]

\[\Gamma, n[\Delta'] A \vdash \text{Prop} : \text{Type}\]

\[\Gamma, m[\Delta](\text{Clear } x) A \Rightarrow \Gamma, n[\Delta']: A, m[\Delta] := n[\Delta']: A\]

\[\Gamma, m[\Delta] A \vdash \text{Prop} : \text{Type}\]

\[\Gamma, n[\Delta'] A \not\vdash \text{Prop} : \text{Type}\]

\[\Gamma, m[\Delta](\text{Clear } x) A \Rightarrow \text{Fail}_0\]

where \((x:C) \in \Delta, \Delta' = \Delta \setminus (x:C)\)

\[\Gamma, m[\Delta] (M[\vec{t}/\vec{x}]) A \Rightarrow \Gamma'\]

\[\Gamma, m[\Delta](T[t_1 \ldots t_n]) A \Rightarrow \Gamma'\]

\[\Gamma, m[\Delta] (M[\vec{t}/\vec{x}]) A \Rightarrow \text{Fail}_n\]

\[\Gamma, m[\Delta] (T[t_1 \ldots t_n]) A \Rightarrow \text{Fail}_n\]

where \(T[\vec{x}] := M\) is a defined tactical

\[\Gamma, \Delta A : s\]

\[\Rightarrow (\Gamma; \Delta) \triangleright t[A/s] \Rightarrow (\Gamma'; \Delta) \triangleright N : A\]

\[\Rightarrow \Gamma, m[\Delta](\text{use } t) : A \Rightarrow \Gamma', m[\Delta] = N : A\]

\[\Gamma, \Delta A : s\]

\[\Rightarrow (\Gamma; \Delta) \triangleright t \Rightarrow (\Gamma'; \Delta) \triangleright \text{Fail}_n\]

\[\Rightarrow \Gamma, m[\Delta](\text{use } t) : A \Rightarrow \text{Fail}_n\]
\[ \begin{array}{c}
\vdash \Gamma, m[\Delta](\langle T \rangle_1): A \Rightarrow \Gamma_0, \Theta \\
\vdash \Gamma_{i-1}, \Theta_{i-1}, n_i[\Delta_i](\langle T_2 \rangle_2): A_i \Rightarrow \Gamma_i \\
\vdash \Gamma, m[\Delta](\langle T_1; T_2 \rangle): A \Rightarrow \Gamma_k, \Theta_k \\
\vdash \Gamma, m[\Delta](\langle T \rangle_1): A \Rightarrow \text{Fail}_n \\
\vdash \Gamma, m[\Delta](\langle T \rangle_2): A \Rightarrow \text{Fail}_n \\
\vdash \Gamma, m[\Delta](\langle T \rangle): A \Rightarrow \Gamma_0, \Theta \\
\vdash \Gamma_{i-1}, \Theta_{i-1}, m_i[\Delta_i](\langle T_1 \rangle_2): A_i \Rightarrow \Gamma_i \\
\vdash \Gamma, m[\Delta](\langle T; T_1; \ldots; T_k \rangle): A \Rightarrow \Gamma_k, \Theta_k \\
\vdash \Gamma, m[\Delta](\langle T \rangle): A \Rightarrow \text{Fail}_n \\
\vdash \Gamma, m[\Delta](\langle T; T_1; \ldots; T_k \rangle): A \Rightarrow \text{Fail}_n \\
\vdash \Gamma, m[\Delta](\langle T; T_1; \ldots; T_k \rangle): A \Rightarrow \text{Fail}_n \\
\vdash \Gamma, m[\Delta](\langle T; T_1; \ldots; T_k \rangle): A \Rightarrow \text{Fail}_n
\end{array} \]

\[ \begin{align*}
|\Gamma| &= |\Gamma_0| \\
\Theta &\equiv \Theta_0, n_1[\Delta_1]: A_1 \ldots n_k[\Delta_k]: A_k, \Theta_k \\
\Theta_i &\text{ contains only MV definitions.}
\end{align*} \]
Bibliography


Samenvatting

Een interactieve stellingbewijzer (ook wel bewijssistent genoemd) is een computerprogramma waarmee men theorieën en bewijzen ontwikkelt. Het maken van zulke bewijzen gaat meestal stap voor stap vanwege de hoge complexiteit van de theorieën waarmee men werkt. Daarom moet een bewijssistent in staat zijn om met incomplete objecten om te gaan. Voor bewijssistenten die gebaseerd zijn op type-theorie betekent dat dat ze incomplete (open) termen op een consistent manier moeten introduceren en gebruiken.

In dit proefschrift worden de volgende aspecten van het proces van formalisatie van incomplete termen in type-theorie en logica bestudeerd:

- de uitlegking van hogere-orde logica (HOL) met ‘first-class’ incomplete termen en bewijzen. Daarmee kunnen we de logische inhoud van de tussentoestanden van een bewijssistent tijdens een bewijssessie beschrijven.
- de uitlegking van het typesysteem λHOL met meta-variablen om incomplete termen te kunnen representeren.
- de introductie van ‘bindende gaten’. Een bindend gat is een onbekend deel van een term of een bewijs dat een of meerdere binders zou moeten bevatten. Daarmee kan men een gebonden variabele gebruiken nog voordat de binder geconstrueerd is.

We laten zien dat de bekende formules-as-types afbeelding uitgebreid kunnen worden tot incomplete bewijzen en termen. Verder gebruiken we de bovengenoemde theorie om een formele model te maken van een bewijssistent.

Tot slot definiëren we een calculus van tactieken en tactics en de getypeerde semantiek daarvan. Tactieken zijn de commando’s waarmee bewijzen gemaakt worden. Een tactiek kan beschouwd worden als een procedure voor het construeren van een (deel van) een bewijs. Tactics zijn meta-operaties op tactieken zoals composeitie, applicatie, enz. Wij definiëren de syntax en getypeerde evaluatieregels van de taal voor tactieken en tactics en bestuderen de belangrijkste eigenschappen daarvan.
Curriculum Vitae

Born on 29 April 1974 in Karlovo, Bulgaria


1998-1999 Master class in Mathematical Logic, MRI, the Netherlands.

1999-2000 Military draft service.

Titles in the IPA Dissertation Series


A.M. Geerling. Transformational Development of Data-Parallel Algorithms. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. Interactive Functional Programs: Models, Methods, and Implementation. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. Parallel Local Search. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. The Implementation of Functional Languages on Parallel Machines with Distributed Memory. Faculty of Mathematics and Computer Science, KUN. 1996-05


H. Doornbos. Reductivity Arguments and Program Construction. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. Functorial Operational Semantics and its Denotational Dual. Faculty of Mathematics and Computer Science, VUA. 1996-09


N.W.A. Arends. A Systems Engineering Specification Formalism. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. Normalisation in Lambda Calculus and its Relation to Type Inference. Faculty of Mathematics and Computing Science, TUE. 1996-12


M.M. Bonsangue. Topological Dualities in Semantics. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. Algorithms for Graphs of Small Treewidth. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. Process-algebraic Transformations in Context. Faculty of Computer Science, UT. 1997-02


C.J. Boo. Preservation of Termination for Explicit Substitution. Faculty of Mathematics and Computing Science, TUE. 1997-05


F.A.M. van den Beucken. A Functional Approach to Syntax and Typing. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. Ins and Outs in Refusal Testing. Faculty of Computer Science, UT. 1998-01

G. Naumeski and W. Albers. A Discrete-Event Simulator for Systems Engineering. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. Scheduling with Communication for Multiprocessor Computation. Faculty of Mathematics and Computer Science, UU. 1998-03


E. Voermans. Inductive Datatypes with Laws and Subtyping – A Relational Model. Faculty of Mathematics and Computing Science, TUE. 1999-01
H. ter Doest. Towards Probabilistic Unification-based Parsing. Faculty of Computer Science, UT. 1999-02


C.H.M. van Kemnade. Recombinative Evolutionary Search. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. Learning Reliability: a Study on Indecisiveness in Sample Selection. Faculty of Mathematics and Natural Sciences, RUG. 1999-05


J.P. Warners. Nonlinear approaches to satisfiability problems. Faculty of Mathematics and Computing Science, TUE. 1999-08


P.R. D’Argenio. Algebras and Automata for Timed and Stochastic Systems. Faculty of Computer Science, UT. 1999-10

G. Fábián. A Language and Simulator for Hybrid Systems. Faculty of Mechanical Engineering, TUE. 1999-11


R. Schiefer. Viper, A Visualization Tool for Parallel Program Construction. Faculty of Mathematics and Computing Science, TUE. 1999-15


T.E.J. Vos. UNITY in Diversity. A stratified approach to the verification of distributed algorithms. Faculty of Mathematics and Computer Science, UU. 2000-02


P.H.F.M. Verlooveen. The Design of the MathSpad Editor. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. Design of a Fruit Juice Blending and Packaging Plant. Faculty of Mechanical Engineering, TUE. 2000-06


P.A. Olivier. A Framework for Debugging Heterogeneous Applications. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. Another Formal Specification Language. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelassi. The Shape of Evolutionary Search Discovering and Representing Search Space Structure. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. Agents, Objects and Events a computational approach to knowledge, observation and communication. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. Reasoning about Java programs in higher order logic using PVS and Isabelle. Faculty of Science, KUN. 2001-03


S.C.C. Blom. Term Graph Rewriting: syntax and semantics. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. Studies in Interactive Visualization. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

J. Hage. Structural Aspects of Switching Classes. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. Towards Effective Model Checking. Faculty of Computer Science, UT. 2001-10

D. Chikliæev. Mechanical verification of concurrency control and recovery protocols. Faculty of Mathematics and Computing Science, TU/e. 2001-11


D. Bošnacki. Enhancing state space reduction techniques for model checking. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. Formal Specification and Analysis of Industrial Systems. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. Techniques for Understanding Legacy Software Systems. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03


R.J. Willemsen. School Timetable Construction: Algorithms and Complexity. Faculty of Mathematics and Computer Science, TU/e. 2002-05


N. van Vught. Models of Molecular Computing. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Felner. Citius, Viïus, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems. Faculty of Science, Mathematics and Computer Science, UvA. 2002-08

R. van Stee. On-line Scheduling and Bin Packing. Faculty of Mathematics and Natural Sciences, UL. 2002-09


M.B. van der Zwang. Models and Logics for Process Algebra. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.J. den Hartog. Probabilistic Extensions of Semantical Models. Faculty of Sciences, Division of Mathematics and Computer Science, VU. 2002-12


J.J. van Hemert. Applying Evolutionary Computation to Constraint Satisfaction and Data Mining. Faculty of Mathematics and Natural Sciences, UL. 2002-14


Y.S. Usenko. Linearization in μCRL. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. Random Redundant Storage for Video on Demand. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. To Reuse or To Be Reused: Techniques for component composition and construction. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Vasser. Generic Traversal over Typed Source Code Representations. Faculty
of Natural Sciences, Mathematics, and Computer Science, UvA, 2003-03

S.M. Bohte. Spiking Neural Networks. Faculty of Mathematics and Natural Sciences, UL, 2003-04


S.V. Nedea. Analysis and Simulations of Catalytic Reactions. Faculty of Mathematics and Computer Science, TU/e, 2003-06


H.P. Benz. Casual Multimedia Process Annotation – CoMPAs. Faculty of Electrical Engineering, Mathematics & Computer Science, UT, 2003-08


M.H. ter Beek. Teams Automata – A Formal Approach to the Modeling of Collaboration Between System Components. Faculty of Mathematics and Natural Sciences, UL, 2003-10

D.J.P. Leijen. The lambda Abroad – A Functional Approach to Software Components. Faculty of Mathematics and Computer Science, UU, 2003-11


G.I. Jojgov. Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving. Faculty of Mathematics and Computer Science, TU/e, 2001-02
STELLINGEN

behorende bij het proefschrift

INCOMPLETE PROOFS AND TERMS AND THEIR USE IN INTERACTIVE THEOREM PROVING

van G.I. Jojgov
(1) Only Moore’s law [4] (and hence faster computers) will not make proof-assistants applicable in a wider industrial and research setting. Next to studying the fundamentals, we need to build more application- and user-oriented proof-assistants. One step in that direction would be the development of a means of communication with the system in which the user can express what he has in mind, even if it is an incorrect or incomplete statement. This requires that the proof-assistant supports a number of notions of correctness of different strength.

(2) The need to internalize meta-information is pervasive. It has also been recognized in some modern CPU architectures (see [1] for an example) where meta-information about forthcoming branches, blocks of memory about to be accessed, instructions that are safe to execute in parallel, etc. can explicitly be passed by the compiler to the CPU. This leads to more efficient designs since the CPU does not have to re-discover the meta-information it needs in order to keep its multiple execution units busy.

(3) The recursive definition

\[
\text{Apply}([\varphi : \text{Prop}, \psi : \text{Prop}, M : \psi]) := \\
[\varphi \sim \psi].M \\
\text{else} \\
[\psi \sim ?A, ?B : \text{Prop} \Pi x : A.B].?m : A.\text{Apply}([\varphi, B, (Mm)]) \\
\text{else} \\
[\psi \sim ?A : \text{Type}, ?B[x : A] : \text{Prop} \Pi x : A.B[x]].?m : A.\text{Apply}([\varphi, B[m], (Mm)])
\]

represents an encoding of the \text{Apply} tactic in the calculus of tactics.

(4) “In the middle of difficulty lies opportunity.”

A. Einstein

With the theory developed in [2] one cannot do forward reasoning. The search for a solution to this problem led to the idea of binding holes that make the modeling of forward reasoning possible.
(5) The trend in the last years by IT companies from English-speaking countries to “off-shore” some of their operations to India is partly motivated by language and cultural ties. This is confirmed by the fact that it is not followed by IT companies in some European countries. They seem to prefer “near-shore” solutions offered by East-European countries (see [5]).

(6) Part of the value of fundamental research is that its output is the input for practical research.

(7) Contrary to the usual perception, type theory is capable of faithfully reflecting the structure of informal mathematical proofs. One way to do that is to use global definitions (see [3]).

(8) The desire to be able to define and manipulate tactics has its origins much before the first tactics-based theorem prover was invented:

“He who can modify his tactics in relation to his opponent and thereby succeed in winning, may be called heaven-born captain.”

Sun Tzu, ~500BC

(9) The impression of the wax sculptures in Madam Tussaud’s would be even greater if the eyes of the models could move to focus on and track the visitors that pass by.

REFERENCES
