Tabled Higher-Order Logic Programming

Brigitte Pientka
December 2003
CMU-CS-03-185

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This research was sponsored by the US Air Force (USAF) under grant F1962895C0050, the National Science Foundation (NSF) under grant CCR-9619584 and grant CCR-9988281, and by a generous donation from the Siebel Scholars Program. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the USAF, NSF, the U.S. government or any other entity.

Keywords: Logic programming, Type theory, Automated theorem proving, Logical frameworks

Abstract

A logical framework is a general meta-language for specifying and implementing deductive systems, given by axioms and inference rules. Based on a higher-order logic programming interpretation, it supports executing logical systems and reasoning with and about them, thereby reducing the effort required for each particular logical system.

In this thesis, we describe different techniques to improve the overall performance and the expressive power of higher-order logic programming. First, we introduce tabled higher-order logic programming, a novel execution model where some redundant information is eliminated using selective memoization. This extends tabled computation to the higher-order setting and forms the basis of the tabled higher-order logic programming interpreter. Second, we present efficient data-structures and algorithms for higher-order proof search. In particular, we describe a higher-order assignment algorithm which eliminates many unnecessary occurs checks and develop higher-order term indexing. These optimizations are crucial to make tabled higher-order logic programming successful in practice. Finally, we use tabled proof search in the meta-theorem prover to reason efficiently with and about deductive systems. It takes full advantage of higher-order assignment and higher-order term indexing.

As experimental results demonstrate, these optimizations taken together constitute a significant step toward exploring the full potential of logical frameworks in practice.

Contents

Acknowledgments

First of all I would like to thank my advisor Frank Pfenning for all his support and patience even before I started as a graduate student at CMU. His ideas shaped this thesis in many ways. His persistent questions and insistence on clarity and precision have been instrumental in understanding and writing about some of the issues presented in this thesis. In particular I appreciate his openness to always discuss technical problems, trying to make sense of half-baked ideas, and patiently explaining problems.

Second, I would like to thank the members of my thesis committee, Bob Harper, Dana Scott and David Warren for serving on my committee, giving valuable criticism and support. In particular, I would like to thank David Warren for his insights into XSB and his eagerness to understand some of the higher-order logic programming issues.

Third, my thanks go to Christoph Kreitz who has been a mentor during my year at Cornell and has been instrumental in my decision to come the US and join a graduate program. I also want to thank Alan Bundy and Bob Constable who gave me valuable research opportunities before I started graduate school.

Special thanks go to Carsten Schürmann for his support during my time at CMU, his friendship and all the technical discussions about Twelf. For many discussions and valuable feedback I would like to thank the members of the POP-group, in particular Andrew Bernard, Karl Crary, Aleksander Nanevski, Susmit Sarkar, and Kevin Watkins. Many thanks also go to Chad Brown for endless discussions about type-theory.

I could not have completed this journey without some friends in Pittsburgh and elsewhere. Women@scs has been a welcoming and supporting community and I have met many extraordinary women in computer science through them. It certainly has made my time at CMU more fun and rewarding. In particular, I would like to thank

Laurie Hiyakumoto, Jorjeta Jetcheva, Orna Raz, and Bianca Schröder for all their support, understanding, time and care of each other. They were always willing to spare some time, share the ups and downs in graduate school, and provide much needed distractions and breaks from work.

For providing me with an escape from Wean Hall and a different view to life and work, my thanks go to Uljana Feest who seemed to always enjoy life to the fullest while earning a philosophy PhD incidentally. For their support from abroad and for always welcoming me in Berlin and Jena, I would like to thank Vicky Temperton and Katrin Teske. I am also indebted to my parents who allowed me to go my way, although it was sometimes hard for them to understand and agree.

Last but not least, I want to thank Dirk Schlimm – without him I would not have made it to Pittsburgh. He always believed in me, patiently endured all my stubbornness and shared all the frustration and joy.

And, I almost forgot..., thanks also to Bruno, the cat, who has been teaching me that life isn't so hard after all ...

Chapter 1

Introduction

A logical framework is a general meta-language for specifying and implementing deductive systems, given by axioms and inference rules. Examples of deductive systems are plentiful in computer science. In computer security, we find authentication and security logics to describe access and security criteria. In programming languages, we use deductive systems to specify the operational semantics, type-systems or other aspects of the run-time behavior of programs. Recently, one major application of logical frameworks has been in the area of "certified code". To provide guarantees about the behavior of mobile code, safety properties are expressed as deductive systems. The code producer then verifies the program according to some predetermined safety policy, and supplies a binary executable together with its safety proof (certificate). Before executing the program, the host machine then quickly checks the code's safety proof against the binary. The safety policy and the safety proofs can be expressed in the logical framework thereby providing a general safety infrastructure.

There are two main variants of logical frameworks which are specifically designed to support the implementation of deductive systems. λProlog and Isabelle are based on hereditary Harrop formulas, while the Twelf system [53] is an implementation of the logical framework LF, a dependently typed lambda calculus. In this thesis, we will mainly focus on the latter. By assigning a logic programming interpretation to types [47], we obtain a higher-order logic programming language. Higher-order logic programming in Twelf extends traditional first-order logic programming in three ways: First, we have a rich type system based on dependent types, which allows the user to define her own higher-order data-types and supports higher-order abstract syntax [52]. Variables in the object language can be directly represented as variables in the meta-language thereby directly inheriting capture-avoiding substitution and bound variable renaming. Second, we not only have a static set of program clauses, but clauses may be introduced dynamically and used within a certain scope during proof search. Third, we have an explicit notion of proof, i.e., the logic programming interpreter does not only return an answer substitution for the free variables in the query, but also the actual proof of the query as a term in the dependently typed lambda-calculus. This stands in sharp contrast to higher-order features supported in many traditional logic programming languages (see for example [13]) where we can encapsulate predicate expressions within terms to later retrieve and invoke such stored predicates. Twelf's higher-order logic programming interpreter is complemented by a meta-theorem prover, which combines generic proof search based on higher-order logic programming with inductive reasoning [53, 63].

The Twelf system has been successfully used to implement, execute and reason about a wide variety of deductive systems. However, experience with real-world applications in different projects on certified code [4, 15, 3] have increasingly demonstrated the limitations of Twelf's higher-order logic programming proof search. To illustrate, let us briefly consider the foundational proof-carrying code project at Princeton. As part of this project, the researchers at Princeton have implemented between 70,000 and 100,000 lines of Twelf code, which includes data-type definitions and proofs. The higher-order logic program, which is used to execute safety policies, consists of over 5,000 lines of code, and over 600 – 700 clauses. Such large specifications have put to test implementations of logical frameworks and exposed several problems. First, performance of the higher-order logic programming interpreter may be severely hampered by redundant computation, leading to long response times and slow development of formal specifications. Second, many straightforward specifications of formal systems, for example recognizers and parsers for grammars, rewriting systems, type systems with subtyping or polymorphism, are not executable, thus requiring more complex and sometimes less efficient implementations. Thirdly, redundancy severely hampers the reasoning with and about deductive systems in general, limiting the use of the meta-theorem prover.

In applications to certified code, efficient proof search techniques not only play an important role to execute safety polices and generate a certificate that a given program fulfills a specified safety policy, but it also can be used to check the correctness of a certificate [42]. Necula and Rahul [42] propose as a certificate a bit-string of the non-deterministic choices in the proof. Hence, a proof can be checked by guiding the higher-order logic programming interpreter with the bit-string and reconstructing the actual proof. As pointed out by Necula and Lee, typical safety proof in the context of certified code commonly have repeated sub-proofs that should be hoisted out and proved only once. The replication of common sub-proofs leads to redundancy in the bit-strings representing the safety proof and it may take longer to reconstruct the safety proof using a guided higher-order logic programming interpreter.

In this thesis, we develop different techniques which improve the overall performance and the expressive power of the higher-order logic programming interpreter. We also apply these ideas in the meta-theorem prover to overcome existing limitations when reasoning about deductive systems. These optimizations taken together constitute a significant step toward exploring the full potential of logical frameworks in real-world applications. Some of the work in this thesis has been previously published in different forms [54, 55, 56, 57, 41]

Contributions

The contributions in this thesis are in three main areas: First, we introduce tabled higher-order logic programming, a novel execution model where some redundant information is eliminated using selective memoization. This forms the basis of the tabled higher-order logic programming interpreter. Second, we develop efficient data-structures and algorithms for higher-order proof search. These optimizations are crucial to make tabled higher-order logic programming successful in practice. Although we develop these techniques in the context of tabled logic programming, they are also independently useful and important to other areas such as higher-order rewriting, higher-order theorem proving and higher-order proof checking. Third, we use memoization-based proof search in the meta-theorem prover, to reason efficiently with and about deductive systems. This demonstrates the importance of memoization in general. Next, we will discuss briefly each of these contributions.

Tabled higher-order logic programming

Tabled first-order logic programming has been successfully applied to solve complex problems such as implementing recognizers and parsers for grammars [68], representing transition systems CCS and writing model checkers [16]. The idea behind it is to eliminate redundant computation by memoizing sub-computation and re-using its results later. The resulting search procedure is complete and terminates for programs with the bounded-term size property. The XSB system [62], a tabled logic programming system, demonstrates impressively that tabled together with non-tabled programs can be executed efficiently in the first-order setting.

The success of memoization in first-order logic programming strongly suggests that memoization may also be valuable in higher-order logic programming. In fact, Necula and Lee point out in [44] that typical safety proofs in the context of certified code commonly have repeated sub-proofs that should be hoisted out and proved only once. Memoization has potentially three advantages. First, proof search is faster thereby substantially reducing the response time to the programmer. Second, the proofs themselves are more compact and smaller. This is especially important in applications to secure mobile code where a proof is attached to a program, as smaller proofs take up less time to check and transmit to another host. Third, substantially more specifications, for example recognizers and parser for grammars, evaluators based on rewriting or type systems with subtyping, are executable under the new paradigm thereby extending the power of the existing system.

Using memoization in higher-order logic programming poses several challenges, since we have to handle type dependencies and may have dynamic assumptions which are introduced during proof search. This is unlike tabling in XSB, where we have no types and it suffices to memoize atomic goals. Moreover, most descriptions of tabling in the first-order setting are closely oriented on the WAM (Warren Abstract Machine) making it hard to transfer tabling techniques and design extensions to other logic programming interpreters.

In this thesis, we introduce a novel execution model for logical frameworks based on selective memoization.

Proof-theoretic characterization of uniform proofs and memoization We give a proof-theoretic characterization of tabling based on uniform proofs, and show soundness of the resulting interpreter. This provides a high-level description of a tabled logic programming interpreter and separates logical issues from procedural ones leaving maximum freedom to choose particular control mechanisms.

Implementation of a tabled higher-order logic programming interpreter We give a high-level description of a semi-functional implementation for adding tabling to a higher-order logic programming interpreter. We give an operational interpretation of the uniform proof system and discuss some of the implementation issues such as suspending and resuming computation, retrieving answers, and trailing. Unlike other description, it does not require an understanding or modifications, and extensions to the WAM (Warren abstract machine). It is intended as a high-level explanation and guide for adding tabling to an existing logic programming interpreter. This is essential for rapidly prototyping tabled logic programming interpreters, even for linear logic programming and other higher-order logic programming systems.

Case studies We discuss two case studies to illustrate the use of memoization in the higher-order setting. We consider a parser and recognizer for first-order formulas into higher-order abstract syntax. To model left and right associativity of the different connectives, we mix left and right recursion in the specification of the parser. Although this closely models the grammar, it leads to an implementation which is not executable with traditional logic programming interpreters which are based on depth-first search.

The second case study is an implementation of a bi-directional type-checker by Davies and Pfenning [17]. The type-checker is executable with the original logic programming interpreter, which performs a depth-first search. However, redundant computation may severely hamper its performance as there are several derivations for proving that a program has a specified type.

Efficient data-structures and algorithms

Efficient data-structures and implementation techniques play a crucial role in utilizing the full potential of a reasoning environment in large scale applications. Although this need has been widely recognized for first-order languages, efficient algorithms for higher-order languages are still a central open problem.

Proof-theoretic foundation for existential variables based on modal logic We give a dependent modal lambda calculus, which extends the theory of the logical framework LF [29] conservatively with modal variables. Modal variables can be interpreted as existential variables, thereby clearly distinguishing them from ordinary bound variables. This is critical to achieve a simplified account of higher-order unification and allows us to justify different optimizations such as as lowering, raising, and linearization [57, 41]. It also serves as a foundation for designing higher-order term indexing strategies.

Optimizing unification Unification lies at the heart of logic programming, theorem proving, and rewriting systems. Thus, its performance affects in a crucial way the global efficiency of each of these applications. Higher-order unification is in general undecidable, but decidable fragments, such as higher-order patterns unification, exist. Unfortunately, the complexity of this algorithm is still at best linear, which is impractical for any useful programming language or practical framework. In this thesis, we present an assignment algorithm for linear higher-order patterns which factors out unnecessary occurs checks. Experiments show that we get a speed-up by up to a factor 2 – 5 making the execution of some examples feasible. This is a significant step toward efficient implementation of higher-order reasoning systems in general [57].

Higher-order term indexing Proof search strategies, such as memoization, can only be practical if we can access the memo-table efficiently. Otherwise, the rate of drawing new conclusions may degrade sharply both with time and with an increase of the size of the memo-table. Term indexing aims at overcoming program degradation by sharing common structure and factoring common operations. Higher-order term indexing has been a central open problem, limiting the application and the potential impact of higher-order reasoning systems. In this thesis, we develop and implemented higher-order term indexing techniques. They improve performance by up to a factor of 9, illustrating the importance of indexing [56].

Meta-theorem proving based on memoization

The traditional approach for supporting theorem proving in logical frameworks is to guide proof search using tactics and tacticals. Tactics transform a proof structure with some unproven leaves into another. Tacticals combine tactics to perform more complex steps in the proof. Tactics and tacticals are written in ML or some other strategy language. To reason efficiently about some specification, the user implements specific tactics to guide the search. This means that tactics have to be rewritten for different specifications. Moreover, the user has to understand how to guide the prover to find the proof, which often requires expert knowledge about the systems. Proving the correctness of the tactic is itself a complex theorem proving problem.

The approach taken in the Twelf system is to endow the framework with the operational semantics of logic programming and design general proof search strategies for it. Twelf's meta-theorem prover combines general proof search based on higher-order logic programming with inductive reasoning. Using the proof-theoretic characterization of tabling, we develop a general memoization-based proof search strategy which is incorporated in Twelf's meta-theorem prover. As experiments demonstrate, eliminating redundancy in meta-theorem proving is critical to prove properties about larger and more complex specifications. We discuss several examples including type preservation proofs for type-system with subtyping, several inversion lemmas about refinement types, and reasoning in classical natural deduction. These examples include several lemmas and theorems which were not previously provable. Moreover, we show that in many cases no bound is needed on memoization-based search. As a consequence, if a sub-case is not provable, the user knows, that in fact no proof exists. This in turn helps the user to revise the formulation of the theorem or the specification. Overall the benefits of memoization are an important step towards a more robust and more efficient meta-theorem prover.

Chapter 2

Dependently typed lambda calculus based on modal type theory

In this chapter, we introduce a dependently typed lambda calculus based on modal type theory. The underlying motivation for this work is to provide a logical foundation for the implementation of logical frameworks and the design choices one has to make in practice. One such choice is for example the treatment of existential variables. Existential variables are usually implemented via mutation. However, previous formulations of logical frameworks [28, 29] do not capture or explain this technique. The framework presented in this chapter serves as a foundation for the subsequent chapters on higher-order unification, higher-order proof search, and higher-order term indexing.

We present here an abstract view of existential variables based on modal type theory. Existential variables u (or v) are treated as modal variables. This allows us to reason about existential variables as first-class objects and directly explain many optimizations done in implementations. In this chapter, we will conservatively extend the theory of the logical framework LF given by Harper and Pfenning in [29] with modal variables. Following Harper and Pfenning, we will introduce this language and show that definitional equality remains decidable. In addition, normalization and type-checking are decidable.

2.1 Motivation

Before presenting the foundation for dependently typed existential variables, we briefly motivate our approach based on modal logic. Following the methodology of Pfenning and Davies [51], we can assign constructive explanations to modal operators. A key characteristic of this view is to distinguish between propositions that are true and propositions that are valid. A proposition is valid if its truth does not depend on the truth of any other propositions. This leads to the basic hypothetical judgment

$$A_1\ \mathit{valid}, \ldots A_n\ \mathit{valid}; B_1\ \mathit{true}, \ldots, B_m\ \mathit{true} \vdash C\ \mathit{true}.$$

Under the multiple-world interpretation of modal logic, C *valid* corresponds to C *true* in *all* reachable worlds. This means C *true* without any assumptions, except those that are assumed to be true in all worlds. We can generalize this idea to also capture truth relative to a set of specified assumptions by writing C *valid* Ψ, where Ψ is the abbreviation for $C_1\ \mathit{true}, \ldots, C_n\ \mathit{true}$. In terms of the multiple world semantics, this means that C is true in any world where C_1 through C_n are all true and we say C is valid relative to the assumptions in Ψ. Hypotheses about relative validity are more complex now, so our general judgment form is

$$A_1\ \mathit{valid}\ \Psi_1, \ldots, A_n\ \mathit{valid}\ \Psi_n; B_1\ \mathit{true}, \ldots, B_m\ \mathit{true} \vdash C\ \mathit{true}$$

While it is interesting to investigate this modal logic above in its own right, it does not come alive until we introduce proof terms. In this chapter, we investigate the use of a modal proof term calculus as a foundation for existential variables. We will view existential variables u as modal variables of type A in a context Ψ while bound variables are treated as ordinary variables. This allows us to distinguish between existential variables $u{::}(\Psi{\vdash}A)$ for relative validity assumptions A *valid* Ψ declared in a modal context, and $x{:}A$ for ordinary truth assumptions A *true* declared in an (ordinary) context. If we have an assumption A *valid* Ψ we can only conclude A *true* if we can verify all assumptions in Ψ.

$$\frac{\Delta, A\ \mathit{valid}\ \Psi, \Delta'; \Gamma \vdash \Psi}{\Delta, A\ \mathit{valid}\ \Psi, \Delta'; \Gamma \vdash A\ \mathit{true}}\ (*)$$

In other words, if we know A *true* in Ψ, and all elements in Ψ can be verified from the assumptions in Γ, then we can conclude A *true* in Γ. As we will see in the next section, this transition from one context Ψ to another context Γ, can be achieved via a substitutions from Ψ to Γ.

2.2 Syntax

We conservatively extend LF [28] with modal variables. Existential variables u (or v) are treated as modal variables and x denotes ordinary variables. Existential variables are declared in a modal context Δ, while bound variables $x{:}A$ are declared in a context Γ or Ψ. Note that the modal variables u declared in Δ carry their own context of bound variables Ψ and type A. The substitution σ is part of the syntax of existential variables. c and a are constants, which are declared in a signature.

$$\begin{array}{rrcl}
\text{Substitutions} & \sigma, \tau & ::= & \cdot \mid \sigma, M/x \\
\text{Objects} & M, N & ::= & c \mid x \mid u[\sigma] \mid \lambda x{:}A.\ M \mid M_1\, M_2 \\
\text{Families} & A, B, C & ::= & a \mid A\, M \mid \Pi x{:}A_1.\ A_2 \\
\text{Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}A.\ K \\
\\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\
\text{Contexts} & \Gamma, \Psi & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Modal Contexts} & \Delta & ::= & \cdot \mid \Delta, u{::}(\Psi{\vdash}A)
\end{array}$$

We use K for kinds, and A, B, C for type families, M, N for object. Signatures, contexts and modal contexts may declare each constant and variable at most once. For example, when we write $\Gamma, x{:}A$ we assume that x is not already declared in Γ. If necessary, we tacitly rename x before adding it to the context Γ. Similarly, when we write $\Delta, u{::}(\Psi \vdash A)$, we assume that u is not already declared in Δ. Terms that differ only in the names of their bound and modal variables are considered identical.

2.3 Substitutions

We write the substitution operation as a defined operations in prefix notation $[\sigma]P$, for an object, family, or kind P. These operations are capture-avoiding as usual. Moreover, we always assume that all free variables in P are declared in σ. Substitutions that are part of the syntax are written in postfix notation, $u[\sigma]$. Note that such explicit substitutions occur only for variables u labeling relative validity assumptions. Substitutions are defined in a standard manner.

Substitutions

$$\begin{array}{rcll} [\sigma](\cdot) & = & (\cdot) & \\ [\sigma](\tau, N/y) & = & ([\sigma]\tau, [\sigma]N/y) & \text{provided } y \text{ not declared or free in } \sigma \end{array}$$

Objects

$$\begin{array}{rcll} [\sigma]c & = & c & \\ [\sigma_1, M/x, \sigma_2]x & = & M & \\ [\sigma](u[\tau]) & = & u[[\sigma]\tau] & \\ [\sigma](N_1\, N_2) & = & ([\sigma]N_1)\,([\sigma]N_2) & \\ [\sigma](\lambda y{:}A.\ N) & = & \lambda y{:}[\sigma]A.\ [\sigma, y/y]N & \text{provided } y \text{ not declared or free in } \sigma \end{array}$$

Families

$$\begin{array}{rcll} [\sigma]a & = & a & \\ [\sigma](A\, M) & = & ([\sigma]A)\,([\sigma]M) & \\ [\sigma](\Pi y{:}A_1.\ A_2) & = & \Pi y{:}[\sigma]A_1.\ [\sigma, y/y]A_2 & \text{provided } y \text{ not declared or free in } \sigma \end{array}$$

Kinds

$$\begin{array}{rcll} [\sigma]\mathsf{type} & = & \mathsf{type} & \\ [\sigma](\Pi y{:}A.\ K) & = & \Pi y{:}[\sigma]A.\ [\sigma, y/y]K & \text{provided } y \text{ not declared or free in } \sigma \end{array}$$

The side conditions can always be verified by (tacitly) renaming bound variables. We do not need an operation of applying a substitution σ to a context. The last principle makes it clear that $[\sigma]\tau$ corresponds to composition of substitutions, which is sometimes written as $\tau \circ \sigma$.

Lemma 1 (Composition of substitution)

1. $[\sigma]([\tau]\tau') = [[\sigma]\tau]\tau'$

2. $[\sigma]([\tau]M) = [[\sigma]\tau]M$

3. $[\sigma]([\tau]A) = [[\sigma]\tau]A$

4. $[\sigma]([\tau]K) = [[\sigma]\tau]K$

Proof: By simultaneous induction on the definition of substitutions. □

Note substitutions σ are defined only on ordinary variables x and not modal variables u. We write id_Γ for the identity substitution $(x_1/x_1, \dots, x_n/x_n)$ for a context $\Gamma = (\cdot, x_1{:}A_1, \dots, x_n{:}A_n)$. We will use π for a substitution which may permute the variables, i.e $\pi = (x_{\Phi(1)}/x_1, \dots, x_{\Phi(n)}/x_n)$ where Φ is a total permutation and π is defined on the elements from a context $\Gamma = (\cdot, x_1{:}A_1, \dots, x_n{:}A_n)$. We only consider well-typed substitutions, so π must respect possible dependencies in its domain. We also streamline the calculus slightly by always substituting simultaneously for all ordinary variables. This is not essential, but saves some tedium in relating simultaneous and iterated substitution. Moreover, it is also closer to the actual implementation where we use de Bruijn indices and postpone explicit substitutions.

A new and interesting operation arises from the substitution principles for modal variables. Modal substitutions are defined as follows.

$$\text{Modal Substitutions} \quad \theta \quad ::= \quad \cdot \mid \theta, M/u$$

We write θ for a simultanous substitution $[\![M_1/u_1, \dots, M_n/u_n]\!]$ where $u_1, \dots, u_n$ are distinct modal variables. The new operation of substitution is compositional, but two interesting situations arise: when a variable u is encountered, and when we substitute into a λ-abstraction (or a dependent type Π respectively).

Substitutions

$$\begin{array}{rcl} [\![\theta]\!](\cdot) & = & \cdot \\ [\![\theta]\!](\sigma, N/y) & = & ([\![\theta]\!]\sigma, [\![\theta]\!]N/y) \end{array}$$

Objects

$$\begin{array}{rcl} [\![\theta]\!]c & = & c \\ [\![\theta]\!]x & = & x \\ [\![\theta_1, M/u, \theta_2]\!](u[\sigma]) & = & [[\![\theta_1, M/u, \theta_2]\!]\sigma]M \\ [\![\theta]\!](N_1\, N_2) & = & ([\![\theta]\!]N_1)\,([\![\theta]\!]N_2) \\ [\![\theta]\!](\lambda y{:}A.\ N) & = & \lambda y{:}[\![\theta]\!]A.\ [\![\theta]\!]N \end{array}$$

Families

$$\begin{array}{rcl} [\![\theta]\!]a & = & a \\ [\![\theta]\!]x & = & x \\ [\![\theta]\!](A\,M) & = & ([\![\theta]\!]A)\,([\![\theta]\!]M) \\ [\![\theta]\!](\Pi y{:}A_1.\ A_2) & = & \Pi y{:}[\![\theta]\!]A_1.\ [\![\theta]\!]A_2 \end{array}$$

Kind

$$\begin{array}{rcl} [\![\theta]\!]\mathsf{type} & = & \mathsf{type} \\ [\![\theta]\!](\Pi y{:}A.\ K) & = & \Pi y{:}[\![\theta]\!]A.\ [\![\theta]\!]K \end{array}$$

Contexts

$$\begin{array}{rcl} [\![\theta]\!](\cdot) & = & \cdot \\ [\![\theta]\!](\Gamma, y{:}A) & = & ([\![\theta]\!]\Gamma, y{:}[\![\theta]\!]A) \end{array}$$

We remark that the rule for substitution into a λ-abstraction and similarly the rule for Π-abstraction does not require a side condition. This is because the object M is defined in a different context, which is accounted for by the explicit substitution stored at occurrences of u. This ultimately justifies implementing substitution for existential variables by mutation.

Finally, consider the case of substituting into a closure, which is the critical case of this definition.

$$[\![\theta_1, M/u, \theta_2]\!](u[\sigma]) \quad = \quad [[\![\theta_1, M/u, \theta_2]\!]\sigma]M$$

This is clearly well-founded, because σ is a subexpression (so $[\![M/u]\!]\sigma$ will terminate) and application of an ordinary substitution has been defined previously without reference to the new form of substitution. We will continue the discussion of modal variables in chapter 3 and focus here on type-checking and definitional equality of the conservative extension of LF.

2.4 Judgments

We now give the principal judgments for typing and definitional equality. We presuppose a valid signature Σ, which will be omitted for sake of brevity. If J is a typing or equality judgment, then we write $[\sigma]J$ for the obvious substitution of J by σ. For example. if J is $M : A$, then $[\sigma]J$ stands for the judgment $[\sigma]M : [\sigma]A$.

	$\vdash$	$\Delta\ \mathsf{mctx}$	Δ is a valid modal context
Δ	$\vdash$	$\Psi\ \mathsf{ctx}$	Ψ is a valid context
$\Delta;\Gamma$	$\vdash$	$\sigma : \Psi$	Substitution σ matches context Ψ
$\Delta;\Gamma$	$\vdash$	$M : A$	Object M has type A
$\Delta;\Gamma$	$\vdash$	$A : K$	Family A has kind K
$\Delta;\Gamma$	$\vdash$	$K : \mathsf{kind}$	K is a valid kind
$\Delta;\Gamma$	$\vdash$	$M \equiv N : A$	M is definitional equal to N
$\Delta;\Gamma$	$\vdash$	$A \equiv B : K$	A is definitional equal to B
$\Delta;\Gamma$	$\vdash$	$K \equiv L : \mathsf{kind}$	K is definitional equal to L

We start by defining valid modal context and valid context.

Modal Context

$$\frac{}{\vdash (\cdot)\ \mathsf{mctx}} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta \vdash \Psi\ \ \mathsf{ctx} \quad \Delta;\Psi \vdash A : \mathsf{type}}{\vdash (\Delta, u{::}(\Psi{\vdash}A))\ \mathsf{mctx}}$$

Context

$$\frac{}{\Delta \vdash (\cdot)\ \mathsf{ctx}} \qquad \frac{\Delta \vdash \Psi\ \mathsf{ctx} \quad \Delta;\Psi \vdash A : \mathsf{type}}{\Delta \vdash (\Psi, x{:}A)\ \mathsf{ctx}}$$

Note that there may be dependencies among the modal variables defined in Δ.

2.5 Typing rules

In the following, we will concentrate on the typing rules and definitional equality rules for substitutions, objects, types families, and kinds. We will follow the formulation of the typing rules given in [29]. We presuppose that all modal contexts Δ and bound variable context Γ in judgments are valid.

Substitutions

$$\frac{}{\Delta;\Gamma \vdash (\cdot) : (\cdot)} \qquad \frac{\Delta;\Gamma \vdash \sigma : \Psi \quad \Delta;\Gamma \vdash M : [\sigma]A}{\Delta;\Gamma \vdash (\sigma, M/x) : (\Psi, x{:}A)}$$

Object

$$\frac{}{\Delta;\Gamma, x{:}A, \Gamma' \vdash x : A}\,var \qquad \frac{\Delta, u{::}(\Psi {\vdash} A), \Delta';\Gamma \vdash \sigma : \Psi}{\Delta; u{::}(\Psi {\vdash} A), \Delta';\Gamma \vdash u[\sigma] : [\sigma]A}\,mvar$$

$$\frac{\Delta;\Gamma, x{:}A_1 \vdash M : A_2 \quad \Delta;\Gamma \vdash A_1 : \mathsf{type}}{\Delta;\Gamma \vdash \lambda x{:}A_1.\ M : \Pi x{:}A_1.\ A_2} \qquad \frac{\Delta;\Gamma \vdash M_1 : \Pi x{:}A_2.\ A_1 \quad \Delta;\Gamma \vdash M_2 : A_2}{\Delta;\Gamma \vdash M_1\,M_2 : [\mathsf{id}_\Gamma, M_2/x]A_1}$$

$$\frac{\Delta;\Gamma \vdash M : B \quad \Delta;\Gamma \vdash A \equiv B : \mathsf{type}}{\Delta;\Gamma \vdash M : A}\,conv$$

Families

$$\frac{a{:}K \text{ in } \Sigma}{\Delta;\ \Gamma \vdash a : K} \qquad \frac{\Delta;\ \Gamma \vdash A : \Pi x{:}B.K \quad \Delta;\Gamma \vdash M : B}{\Delta;\ \Gamma \vdash A\ M : [\mathsf{id}_\Gamma, M/x]K}$$

$$\frac{\Delta;\ \Gamma \vdash A_1 : \mathsf{type} \quad \Delta;\ \Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}}{\Delta;\ \Gamma \vdash \Pi x{:}A_1.A_2 : \mathsf{type}} \qquad \frac{\Delta;\ \Gamma \vdash A : K \quad \Delta;\ \Gamma \vdash K \equiv L : \mathsf{kind}}{\Delta;\ \Gamma \vdash A : L}$$

Kind

$$\frac{}{\Delta;\Gamma \vdash \mathsf{type} : \mathsf{kind}} \qquad \frac{\Delta;\Gamma \vdash A : \mathsf{type} \quad \Delta;\Gamma, x{:}A \vdash K : \mathsf{kind}}{\Delta;\Gamma \vdash \Pi x{:}A.K : \mathsf{kind}}$$

Note that the rule for modal variables is the rule (*) presented earlier, annotated with proof terms and slightly generalized, because of the dependent type theory we are working in. This rule also justifies our implementation choice of using existential variables only in the form $u[\sigma]$.

2.6 Definitional equality

Next, we give some rules for definitional equality for objects, families and kinds in Δ. Some of the typing premises marked as $\{\ldots\}$ are redundant, but we cannot prove this until validity has been established. We do not include reflexivity, since it is admissible. The interesting case is the one for modal variables. Two modal (or existential) variables are considered definitional equal, if they actually are the same variable and the associated substitutions are definitionally equal. This means that if we implement existential variables via references, then two uninstantiated existential variables are only definitional equal if they point to the same reference under the same substitution.

Substitutions

$$\frac{}{\Delta;\Gamma\vdash\cdot\equiv\cdot:\cdot}\qquad\frac{\Delta;\Gamma\vdash M\equiv N:[\sigma]A\quad \Delta;\Gamma\vdash\sigma\equiv\sigma':\Psi}{\Delta;\Gamma\vdash(\sigma,\ M/x)\equiv(\sigma',N/x):(\Psi,\ x{:}A)}$$

Simultaneous Congruence

$$\frac{\Delta;\Gamma\vdash A\equiv A':\mathsf{type}\quad \Delta;\Gamma\vdash A''\equiv A':\mathsf{type}\quad \Delta;\Gamma,x{:}A\vdash M\equiv N:B}{\Delta;\Gamma\vdash\lambda x{:}A'.M\equiv\lambda x{:}A''.N:\Pi x{:}A.B}$$

$$\frac{\Delta;\Gamma\vdash M_1\equiv N_1:\Pi x{:}A_2.\,A_1\quad \Delta;\Gamma\vdash M_2\equiv N_2:A_2}{\Delta\Gamma\vdash M_1\,M_2\equiv N_1\,N_2:[\mathsf{id}_\Gamma,M_2/x]A_1}$$

$$\frac{\Delta,u{::}(\Psi{\vdash}A),\Delta';\Gamma\vdash\sigma\equiv\sigma':\Psi}{\Delta,u{::}(\Psi{\vdash}A),\Delta';\Gamma\vdash u[\sigma]\equiv u[\sigma']:[\sigma]A}$$

$$\frac{}{\Delta;\Gamma,x{:}A,\Gamma'\vdash x\equiv x:A}\qquad\frac{c{:}A\text{ in signature }\Sigma}{\Delta;\Gamma\vdash c\equiv c:A}$$

Parallel conversion

$$\frac{\{\Delta;\Gamma\vdash A:\mathsf{type}\}\quad \Delta;\Gamma,x:A\vdash M_2\equiv N_2:B\quad \Delta;\Gamma\vdash M_1\equiv N_1:A}{\Delta;\Gamma\vdash(\lambda x{:}A.M_2)\,M_1\equiv[\mathsf{id}_\Gamma,N_1/x]N_2:[\mathsf{id}_\Gamma,M_1/x]B}$$

Extensionality

$$\frac{\begin{array}{l}\{\Delta;\Gamma\vdash N:\Pi x{:}A_1.A_2\}\\ \{\Delta;\Gamma\vdash M:\Pi x{:}A_1.A_2\}\end{array}\qquad \Delta;\Gamma\vdash A_1:\mathsf{type}\qquad \Delta;\Gamma,x{:}A_1\vdash M\,x\equiv N\,x:A_2}{\Delta;\Gamma\vdash M\equiv N:\Pi x{:}A_1.A_2}$$

Equivalence

$$\frac{\Delta; \Gamma \vdash N \equiv M : A}{\Delta; \Gamma \vdash M \equiv N : A} \qquad \frac{\Delta; \Gamma \vdash M \equiv N : A \quad \Delta; \Gamma \vdash N \equiv O : A}{\Delta; \Gamma \vdash M \equiv O : A}$$

Type conversion

$$\frac{\Delta; \Gamma \vdash M \equiv N : A \quad \Delta; \Gamma \vdash A \equiv B : \mathsf{type}}{\Delta; \Gamma \vdash M \equiv N : B}$$

Family congruence

$$\frac{a{:}K \text{ in } \Sigma}{\Delta; \Gamma \vdash a \equiv a : K} \qquad \frac{\Delta; \Gamma \vdash A \equiv B : \Pi x{:}C.K \quad \Delta; \Gamma \vdash M \equiv N : C}{\Delta; \Gamma \vdash A\,M \equiv B\,N : [\mathsf{id}_\Gamma, M/x]K}$$

$$\frac{\Delta; \Gamma \vdash A_1 \equiv B_1 : \mathsf{type} \quad \{\Delta; \Gamma \vdash A_1 : \mathsf{type}\} \quad \Delta; \Gamma, x{:}A_1 \vdash A_2 \equiv B_2 : \mathsf{type}}{\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 \equiv \Pi x{:}B_1.B_2 : \mathsf{type}}$$

Family equivalence

$$\frac{\Delta; \Gamma \vdash A \equiv B : K}{\Delta; \Gamma \vdash B \equiv A : K} \qquad \frac{\Delta; \Gamma \vdash A \equiv B : K \quad \Delta; \Gamma \vdash B \equiv C : K}{\Delta; \Gamma \vdash A \equiv C : K}$$

Kind conversion

$$\frac{\Delta; \Gamma \vdash A \equiv B : K \quad \Delta; \Gamma \vdash K \equiv L : \mathsf{kind}}{\Delta; \Gamma \vdash A \equiv B : L}$$

Kind congruence

$$\frac{}{\Delta; \Gamma \vdash \mathsf{type} \equiv \mathsf{type} : \mathsf{kind}}$$

$$\frac{\Delta; \Gamma \vdash A \equiv B : \mathsf{type} \quad \{\Delta; \Gamma \vdash A : \mathsf{type}\} \quad \Delta; \Gamma, x{:}A \vdash K \equiv L : \mathsf{kind}}{\Delta; \Gamma \vdash \Pi x{:}A.K \equiv \Pi x{:}B.L : \mathsf{kind}}$$

Kind equivalence

$$\frac{\Delta; \Gamma \vdash K \equiv L : \mathsf{kind}}{\Delta; \Gamma \vdash L \equiv K : \mathsf{kind}} \qquad \frac{\Delta; \Gamma \vdash K \equiv L' : \mathsf{kind} \quad \Delta; \Gamma \vdash L' \equiv L : \mathsf{kind}}{\Delta; \Gamma \vdash K \equiv L : \mathsf{kind}}$$

2.7 Elementary properties

We will start by showing some elementary properties.

Lemma 2 (Weakening)

1. *If* $\Delta; \Gamma, \Gamma' \vdash J$ *then* $\Delta; \Gamma, x{:}A, \Gamma' \vdash J$
2. *If* $\Delta,\ \Delta'; \Gamma \vdash J$ *then* $\Delta, u{::}(\Psi{\vdash}A), \Delta'; \Gamma \vdash J$

Proof: Proof by induction over the structure of the given derivation. □

Next, we show that reflexivity is admissible.

Lemma 3 (Reflexivity)

1. *If* $\Delta;\ \Gamma \vdash \sigma : \Psi$ *then* $\Delta;\ \Gamma \vdash \sigma \equiv \sigma : \Psi$
2. *If* $\Delta;\ \Gamma \vdash M : A$ *then* $\Delta;\ \Gamma \vdash M \equiv M : A$
3. *If* $\Delta;\ \Gamma \vdash A : K$ *then* $\Delta;\ \Gamma \vdash A \equiv A : K$
4. *If* $\Delta;\ \Gamma \vdash K : \mathsf{kind}$ *then* $\Delta;\ \Gamma \vdash K \equiv K : \mathsf{kind}$

Proof: Proof by induction over the structure of the given derivation. □

First, we show some simple properties about substitutions, which will simplify some of the following proofs. We always assume that the contexts for bound variables Γ and Ψ are valid. Similarly, the modal context Δ is valid.

Lemma 4 *Let* $\Delta; \Gamma \vdash \sigma_1 : \Psi_1$ *and* $\Delta; \Gamma \vdash (\sigma_1, \sigma_2) : (\Psi_1, \Psi_2)$.

1. *If* $\Delta; \Psi_1 \vdash \tau : \Psi'$ *then* $[\sigma_1, \sigma_2](\tau) = [\sigma_1](\tau)$.
2. *If* $\Delta; \Psi_1 \vdash M : A$ *then* $[\sigma_1, \sigma_2]M = [\sigma_1](M)$ *and* $[\sigma_1, \sigma_2]A = [\sigma_1](A)$.
3. *If* $\Delta; \Psi_1 \vdash A : K$ *then* $[\sigma_1, \sigma_2]A = [\sigma_1](A)$ *and* $[\sigma_1, \sigma_2]K = [\sigma_1](K)$.
4. *If* $\Delta; \Psi_1 \vdash K : \mathsf{kind}$ *then* $[\sigma_1, \sigma_2]K = [\sigma_1](K)$.

5. *If* $\Delta;\Gamma \vdash \mathsf{id}_\Gamma : \Gamma$ *and* $\Delta;\Gamma \vdash \sigma : \Psi$ *then* $[\mathsf{id}_\Gamma](\sigma) = \sigma$

6. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash \mathsf{id}_\Psi : \Psi$ *then* $[\sigma](\mathsf{id}_\Psi) = \sigma$

Proof: The statement (1) to (4) are proven by induction on the first derivation. The statement (5) by induction on the derivation $\Delta;\Gamma \vdash \sigma : \Psi$ and the last one by induction on the size of id_Ψ using statement (1). □

Lemma 5 (Inversion on substitutions)
If $\Delta;\Gamma \vdash (\sigma_1, N/x, \sigma_2) : (\Psi_1, x:A, \Psi_2)$ *then* $\Delta;\Gamma \vdash N : [\sigma_1]A$ *and* $\Delta;\Gamma \vdash \sigma_1 : \Psi_1$

Proof: Structural induction on σ_2. □

Lemma 6 (General substitution properties)

1. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash \tau : \Psi'$ *then* $\Delta;\Gamma \vdash [\sigma]\tau : \Psi'$.

2. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash \tau_1 \equiv \tau_2 : \Psi'$ *then* $\Delta;\Gamma \vdash [\sigma]\tau_1 \equiv [\sigma]\tau_2 : \Psi'$.

3. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash N : C$ *then* $\Delta;\Gamma \vdash [\sigma]N : [\sigma]C$.

4. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash N \equiv M : A$ *then* $\Delta;\Gamma \vdash [\sigma]N \equiv [\sigma]M : [\sigma]A$.

5. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash A : K$ *then* $\Delta;\Gamma \vdash [\sigma]A : [\sigma]K$.

6. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash A \equiv B : K$ *then* $\Delta;\Gamma \vdash [\sigma]A \equiv [\sigma]B : [\sigma]K$.

7. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash K : \mathsf{kind}$ *then* $\Delta;\Gamma \vdash [\sigma]K : \mathsf{kind}$.

8. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Psi \vdash K \equiv L : \mathsf{kind}$ *then* $\Delta;\Gamma \vdash [\sigma]K \equiv [\sigma]L : \mathsf{kind}$.

Proof: By simultaneous induction over the structure of the second derivation. We give some cases for (3).

Case $\mathcal{D} = \dfrac{}{\Delta; (\Psi_1, x{:}A, \Psi_2) \vdash x : A}$

$\Delta;\Psi_1 \vdash A : \mathsf{type}$	by validity of ctx Ψ
$\Delta;\Gamma \vdash (\sigma_1, N/x, \sigma_2) : \Psi_1, x{:}A, \Psi_2$	by assumption
$\Delta;\Gamma \vdash N : [\sigma_1]A$ and $\Delta;\Gamma \vdash \sigma_1 : \Psi_1$	by lemma 5
$\Delta;\Gamma \vdash N : [\sigma_1, N/x, \sigma_2]A$	by lemma 4

Case $\mathcal{D} = \dfrac{\Delta; \Psi, x{:}A_1 \vdash M : A_2 \qquad \Delta; \Psi \vdash A_1 : \mathsf{type}}{\Delta; \Psi \vdash \lambda x{:}A_1.M : \Pi x{:}A_1.A_2}$

$\Delta; \Gamma \vdash \sigma : \Psi$	by assumption
$\Delta; \Gamma \vdash [\sigma]A_1 : \mathsf{type}$	by i.h.
$\Delta \vdash \Gamma\ \mathsf{ctx}$	by asumption
$\Delta \vdash \Gamma, x{:}[\sigma]A_1\ \mathsf{ctx}$	by rule
$\Delta; \Gamma, x{:}[\sigma]A_1 \vdash \sigma : \Psi$	by weakening
$\Delta; \Gamma, x{:}[\sigma]A_1 \vdash x : [\sigma]A_1$	by rule
$\Delta; \Gamma, x{:}[\sigma]A_1 \vdash (\sigma, x/x) : (\Psi, x{:}A_1)$	by rule
$\Delta; \Gamma, x{:}[\sigma]A_1 \vdash [\sigma, x/x]M : [\sigma, x/x]A_2$	by i.h.
$\Delta; \Gamma \vdash \lambda x{:}[\sigma]A_1.[\sigma, x/x]M : \Pi x{:}[\sigma]A_1.[\sigma, x/x]A_2$	by rule
$\Delta; \Gamma \vdash [\sigma](\lambda x{:}A_1.M) : [\sigma](\Pi x{:}A_1.A_2)$	by subst. definition

Case $\mathcal{D} = \dfrac{\Delta; \Psi \vdash M_1 : \Pi x{:}A_2.A_1 \qquad \Delta; \Psi \vdash M_2 : A_2}{\Delta; \Psi \vdash (M_1\ M_2) : [\mathsf{id}_\Psi, M_2/x]A_1}$

$\Delta; \Gamma \vdash [\sigma]M_1 : [\sigma]\Pi x{:}A_2.A_1$	by i.h.
$\Delta; \Gamma \vdash [\sigma]M_1 : \Pi x{:}[\sigma]A_2.[\sigma, x/x]A_1$	by subst. definition
$\Delta; \Gamma \vdash [\sigma]M_2 : [\sigma]A_2$	by i.h.
$\Delta; \Gamma \vdash ([\sigma]M_1)\ ([\sigma]M_2) : [\mathsf{id}_\Gamma, [\sigma]M_2/x]([\sigma, x/x]A_1)$	by rule
$[\mathsf{id}_\Gamma, [\sigma]M_2/x](\sigma, x/x) = ([\mathsf{id}_\Gamma, [\sigma]M_2/x]\sigma, [\sigma]M_2/x)$	by subst. definition
$= ([\sigma], [\sigma]M_2/x)$	by lemma 4
$= ([\sigma](\mathsf{id}_\Psi), [\sigma]M_2/x)$	by lemma 4
$= [\sigma](\mathsf{id}_\Psi, M_2/x)$	by subst. definition
$\Delta; \Gamma \vdash [\sigma](M_1\ M_2) : [\sigma]([\mathsf{id}_\Psi, M_2/x]A_1)$	by subst. definition

Case $\mathcal{D} = \dfrac{\Delta, u{::}\Psi_1{\vdash}A_1, \Delta';\ \Psi \vdash \tau : \Psi_1}{\Delta, u{::}\Psi_1{\vdash}A_1, \Delta';\ \Psi \vdash u[\tau] : [\tau]A_1}$

$\Delta, u{::}\Psi_1{\vdash}A_1, \Delta';\ \Gamma \vdash ([\sigma]\tau) : \Psi_1$	by i.h.
$\Delta, u{::}\Psi_1{\vdash}A_1, \Delta';\ \Gamma \vdash u[[\sigma]\tau] : [[\sigma]\tau]A_1$	by rule

$\Delta, u{::}\Psi_1{\vdash}A_1, \Delta'; \Gamma \vdash [\sigma](u[\tau]) : [\sigma]([\tau]A_1)$ by lemma 1 and subst. definition

Case $\mathcal{D} = \dfrac{\Delta;\Psi \vdash M : B \qquad \Delta;\Psi \vdash B \equiv A : \mathsf{type}}{\Delta;\Psi \vdash M : A}$

$\Delta;\Gamma \vdash [\sigma]M : [\sigma]B$ by i.h.
$\Delta;\Gamma \vdash [\sigma]B \equiv [\sigma]A : \mathsf{type}$ by i.h.
$\Delta;\Gamma \vdash [\sigma]M : [\sigma]A$ by rule

□

Lemma 7 (Renaming substitution) $\Delta;\Gamma, y{:}A \vdash (\mathsf{id}_\Gamma, y/x) : (\Gamma, x{:}A)$.

Proof:
$\Delta;\Gamma \vdash \mathsf{id}_\Gamma : \Gamma$ by definition
$\Delta;\Gamma, y{:}A \vdash \mathsf{id}_\Gamma : \Gamma$ by weakening
$\Delta;\Gamma, y{:}A \vdash y : A$ by rule
$\Delta;\Gamma, y{:}A \vdash y : [\mathsf{id}_\Gamma]A$ by definition
$\Delta;\Gamma, y{:}A \vdash (\mathsf{id}_\Gamma, y/x) : (\Gamma, x{:}A)$ by rule

□

Lemma 8 (Context Conversion)
Assume $\Gamma, x{:}A$ *is a valid context and* $\Gamma \vdash B : \mathsf{type}$.
If $\Delta;\Gamma, x{:}A \vdash J$ *and* $\Delta;\Gamma \vdash A \equiv B : \mathsf{type}$ *then* $\Delta;\Gamma x{:}B \vdash J$.

Proof: direct using weakening and substitution property (lemma 6).
$\Delta;\Gamma, x{:}B \vdash x : B$ by rule
$\Delta;\Gamma \vdash B \equiv A : \mathsf{type}$ by symmetry
$\Delta;\Gamma, y{:}A \vdash (\mathsf{id}_\Gamma, y/x) : (\Gamma, x{:}A)$ renaming substitution
$\Delta;\Gamma, y{:}A \vdash [\mathsf{id}_\Gamma, y/x]J$ renaming of assumption
$\Delta;\Gamma, x{:}B, y{:}A \vdash [\mathsf{id}_\Gamma, y/x]J$ weakening
$\Delta;\Gamma, x{:}B \vdash (\mathsf{id}_\Gamma, x/x) : (\Gamma, x{:}B)$ by definition
$\Delta;\Gamma, x{:}B \vdash B \equiv A : \mathsf{type}$ by weakening

$\Delta; \Gamma, x{:}B \vdash x : A$ by rule (conv)
$\Delta; \Gamma, x{:}B \vdash x : [\mathsf{id}_\Gamma, x/x]A$ by subst. definition
$\Delta; \Gamma, x{:}B \vdash (\mathsf{id}_\Gamma, x/x, x/y) : (\Gamma, x{:}B, y{:}A)$ by rule
$\Delta; \Gamma, x{:}B \vdash [\mathsf{id}_\Gamma, x/x, x/y]([\mathsf{id}_\Gamma, y/x]J)$ by substitution property
$\Delta; \Gamma, x{:}B \vdash [\mathsf{id}_\Gamma, x/x]J$ by subst. definition and lemma 4
$\Delta; \Gamma, x{:}B \vdash J$ by subst. definition

□

Next, we prove a general functionality lemma which is suggested by the modal interpretation.

Lemma 9 (Functionality of typing under substitution) *Assume* $\Delta; \Gamma \vdash \sigma : \Psi$, $\Delta; \Gamma \vdash \sigma' : \Psi$ *and* $\Delta; \Gamma \vdash \sigma \equiv \sigma' : \Psi$.

1. *If* $\Delta; \Psi \vdash \tau : \Psi'$ *then* $\Delta; \Gamma \vdash ([\sigma]\tau) \equiv ([\sigma']\tau) : \Psi'$
2. *If* $\Delta; \Psi \vdash M : A$ *then* $\Delta; \Gamma \vdash [\sigma]M \equiv [\sigma']M : [\sigma]A$.
3. *If* $\Delta; \Psi \vdash A : K$ *then* $\Delta; \Gamma \vdash [\sigma]A \equiv [\sigma']A : [\sigma]K$.
4. *If* $\Delta; \Psi \vdash K : \mathsf{kind}$ *then* $\Delta; \Gamma \vdash [\sigma]K \equiv [\sigma']K : \mathsf{kind}$.

Proof: Simultaneous induction on the given derivation. First, the proof for (1).

Case $\mathcal{D} = \dfrac{}{\Delta; \Gamma \vdash \cdot : \cdot}$

$\Delta; \Gamma \vdash \cdot \equiv \cdot : \cdot$ by rule
$\Delta; \Gamma \vdash [\sigma](\cdot) \equiv [\sigma'](\cdot) : \cdot$ by subst. definition

Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta; \Psi \vdash N : [\tau]A} \qquad \overset{\mathcal{D}_2}{\Delta; \Psi \vdash \tau : \Psi'}}{\Delta; \Psi \vdash (\tau,\ N/x) : (\Psi', x{:}A)}$

$\Delta; \Gamma \vdash ([\sigma]\tau) \equiv ([\sigma']\tau) : \Psi'$ by i.h. on $\mathcal{D}_2$

$\Delta; \Gamma \vdash [\sigma]N \equiv [\sigma']N : [\sigma]([\tau]A)$	by i.h. on $\mathcal{D}_1$
$\Delta; \Gamma \vdash [\sigma]N \equiv [\sigma']N : [[\sigma]\tau]A$	by lemma 1
$\Delta; \Gamma \vdash ([\sigma]\tau,\ [\sigma]N/x) \equiv ([\sigma']\tau,\ [\sigma']N/x) : (\Psi', x{:}A)$	by rule
$\Delta; \Gamma \vdash [\sigma](\tau, N/x) \equiv [\sigma'](\tau, N/x) : (\Psi', x{:}A)$	by subst. definition

Next, the proof for (2). We only show the case for modal variable u.

Case $\mathcal{D} = \dfrac{\Delta_1, u{::}\Psi'{\vdash}A,\ \Delta_2; \Psi \vdash \tau : \Psi'}{\Delta_1, u{::}\Psi'{\vdash}A,\ \Delta_2; \Psi \vdash u[\tau] : [\tau]A}$

$\Delta_1, u{::}\Psi'{\vdash}A,\ \Delta_2; \Gamma \vdash ([\sigma]\tau) \equiv ([\sigma']\tau) : \Psi'$	by i.h.
$\Delta_1, u{::}\Psi'{\vdash}A,\ \Delta_2; \Gamma \vdash u[[\sigma]\tau] \equiv u[[\sigma']\tau] : [[\sigma]\tau]A$	by rule
$\Delta_1, u{::}\Psi'{\vdash}A,\ \Delta_2; \Gamma \vdash [\sigma](u[\tau]) \equiv [\sigma'](u[\tau]) : [\sigma]([\tau]A)$	by lemma 1 and subst. definition

□

Lemma 10 (Inversion on products)

1. *If* $\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 : K$ *then* $\Delta; \Gamma \vdash A_1 : \mathsf{type}$ *and* $\Delta; \Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$

2. *If* $\Delta; \Gamma \vdash \Pi x{:}A.K : \mathsf{kind}$ *then* $\Delta; \Gamma \vdash A : \mathsf{type}$ *and* $\Delta; \Gamma, x{:}A_1 \vdash K : \mathsf{kind}$

Proof: Part (1) follows by induction on the derivation. Part (2) is immediate by inversion. □

Lemma 11 (Validity)

1. *If* $\Delta; \Gamma \vdash M : A$ *then* $\Delta; \Gamma \vdash A : \mathsf{type}$*.*

2. *If* $\Delta; \Gamma \vdash A : K$ *then* $\Delta; \Gamma \vdash K : \mathsf{kind}$*.*

3. *If* $\Delta;\ \Gamma \vdash \sigma \equiv \sigma' : \Psi$ *then* $\Delta; \Gamma \vdash \sigma : \Psi$ *and* $\Delta; \Gamma \vdash \sigma' : \Psi$

4. *If* $\Delta; \Gamma \vdash M \equiv N : A$*, then* $\Delta; \Gamma \vdash M : A$*,* $\Delta; \Gamma \vdash N : A$ *and* $\Delta; \Gamma \vdash A : \mathsf{type}$*.*

5. *If* $\Delta; \Gamma \vdash A \equiv B : K$*, then*$\Delta; \Gamma \vdash A : K$*,* $\Delta; \Gamma \vdash B : K$ *and* $\Delta; \Gamma \vdash K : \mathsf{kind}$*.*

6. If $\Delta; \Gamma \vdash K \equiv L : \mathsf{kind}$*, then* $\Delta; \Gamma \vdash K : \mathsf{kind}$*,* $\Delta; \Gamma \vdash L : \mathsf{kind}$*.*

Proof: Simultaneous induction on derivations. The functionality lemma is needed for application and modal variables. In addition, the cases for modal variables use the validity of the modal context Δ. The typing premises on the rule of extensionality ensure that strengthening is not required. First, we show the case for modal variables in the proof for (1).

Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta, u{::}\Psi{\vdash}A, \Delta'; \Gamma \vdash \sigma : \Psi}}{\Delta, u{::}\Psi{\vdash}A, \Delta'; \Gamma \vdash u[\sigma] : [\sigma]A}$

$\Delta;\ \Psi \vdash A : \mathsf{type}$	by validity of mctx
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Psi \vdash A : \mathsf{type}$	by weakening
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Gamma \vdash [\sigma]A : \mathsf{type}$	by lemma 6

Consider the proof for (4).

Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta, u{::}\Psi{\vdash}A, \Delta'; \Gamma \vdash \sigma \equiv \sigma' : \Psi}}{\Delta, u{::}\Psi{\vdash}A, \Delta'; \Gamma \vdash u[\sigma] \equiv u[\sigma'] : [\sigma]A}$

$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Gamma \vdash \sigma : \Psi$	by i.h.
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Gamma \vdash u[\sigma] : [\sigma]A$	by rule
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Gamma \vdash \sigma' : \Psi$	by i.h.
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Gamma \vdash u[\sigma'] : [\sigma']A$	by rule
$\Delta;\ \Psi \vdash A : \mathsf{type}$	by validity of mctx
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Psi \vdash A : \mathsf{type}$	by weakening
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Psi \vdash A \equiv A : \mathsf{type}$	by reflexivity (lemma 3)
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Gamma \vdash \sigma' \equiv \sigma : \Psi$	by symmetry
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Gamma \vdash [\sigma']A \equiv [\sigma]A : \mathsf{type}$	by functionality (lemma 9)
$\Delta, u{::}\Psi{\vdash}A, \Delta';\ \Gamma \vdash u[\sigma'] : [\sigma]A$	by type conversion

Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta; \Gamma \vdash A_1 : \mathsf{type}} \quad \overset{\mathcal{D}_2}{\Delta; \Gamma, x{:}A_1 \vdash M_2 \equiv N_2 : A_2} \quad \overset{\mathcal{D}_3}{\Delta; \Gamma \vdash M_1 \equiv N_1 : A_1}}{\Delta; \Gamma \vdash (\lambda x{:}A_1.M_2)\ M_1 \equiv [\mathsf{id}_\Gamma, N_1/x]N_2 : [\mathsf{id}_\Gamma, M_1/x]A_2}$

$\Delta; \Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$ by i.h.
$\Delta; \Gamma, x{:}A_1 \vdash M_2 : A_2$ by i.h.
$\Delta; \Gamma \vdash \lambda x{:}A_1.M_2 : \Pi x{:}A_1.A_2$
by rule
$\Delta; \Gamma \vdash M_1 : A_1$ by i.h.
$\Delta; \Gamma \vdash (\lambda x{:}A_1.M_2)\ M_1 : [\mathsf{id}_\Gamma, M_1/x]A_2$ by rule
$\Delta; \Gamma, x{:}A_1 \vdash N_2 : A_2$ by i.h.
$\Delta; \Gamma \vdash N_1 : A_1$ by i.h.
$\Delta; \Gamma \vdash \mathsf{id}_\Gamma : \Gamma$ by definition
$\Delta; \Gamma \vdash N_1 : [\mathsf{id}_\Gamma]A_1$ by definition
$\Delta; \Gamma \vdash (\mathsf{id}_\Gamma, N_1/x) : (\Gamma, x{:}A_1)$ by rule
$\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, N_1/x]N_2 : [\mathsf{id}_\Gamma, N_1/x]A_2$ by substitution (lemma 6)
$\Delta; \Gamma, x{:}A_1 \vdash A_2 \equiv A_2 : \mathsf{type}$ by reflexivity (lemma 3)
$\Delta; \Gamma \vdash \mathsf{id}_\Gamma \equiv \mathsf{id}_\Gamma : \Gamma$ by defintion
$\Delta; \Gamma \vdash M_1 \equiv N_1 : A_1$ by assumption $\mathcal{D}_3$
$\Delta; \Gamma \vdash M_1 \equiv N_1 : [\mathsf{id}_\Gamma]A_1$ by definition
$\Delta; \Gamma \vdash (\mathsf{id}_\Gamma, M_1/x) \equiv (\mathsf{id}_\Gamma, N_1/x) :)\Gamma, x{:}A_1$ by defintion
$\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, M_1/x]A_2 \equiv [\mathsf{id}_\Gamma, N_1/x]A_2 : \mathsf{type}$ by functionality lemma 9
$\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, N_1/x]A_2 \equiv [\mathsf{id}_\Gamma, M_1/x]A_2 : \mathsf{type}$ by symmetry
$\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, N_1/x]N_2 : [\mathsf{id}_\Gamma, M_1/x]A_2$ by type conversion
$\Delta, \Gamma \vdash \mathsf{id}_\Gamma : \Gamma$ by definition
$\Delta; \Gamma \vdash M_1 : A_1$ by i.h.
$\Delta; \Gamma \vdash M_1 : [\mathsf{id}_\Gamma]A_1$ by definition
$\Delta; \Gamma \vdash (\mathsf{id}_\Gamma, M_1/x) : \Gamma, x{:}A_1$ by rule
$\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, M_1/x]A_2 : \mathsf{type}$ by substitution lemma 6

□

Lemma 12 (Typing Inversion)

1. *If* $\Delta; \Gamma \vdash x : A$ *then* $x{:}B$ *in* Γ *and* $\Delta; \Gamma \vdash A \equiv B : \mathsf{type}$ *for some* B.

2. *If* $\Delta; \Gamma \vdash c : A$ *then* $c{:}B$ *in* Σ *and* $\Delta; \Gamma \vdash A \equiv B : \mathsf{type}$ *for some* B.

3. *If* $\Delta; \Gamma \vdash M_1\, M_2 : A$ *then* $\Delta; \Gamma \vdash M_1 : \Pi x{:}A_2.A_1$, $\Delta; \Gamma \vdash M_2 : A_2$ *and* $\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, M_2/x]A_1 \equiv A : \mathsf{type}$ *for some* A_1 *and* A_2.

4. *If* $\Delta; \Gamma \vdash \lambda x{:}A.M : B$ *then* $\Delta; \Gamma \vdash B \equiv \Pi x{:}A.A' : \mathsf{type}$ *and* $\Delta; \Gamma, x{:}A \vdash M : A'$ *and* $\Delta; \Gamma \vdash A : \mathsf{type}$ *for some* A *and* A'.

5. *If* $\Delta; \Gamma \vdash u[\sigma] : B$ *then* $(\Delta_1, u{::}\Psi{\vdash}A, \Delta_2) = \Delta$ *and* $\Delta_1, u{::}\Psi{\vdash}A, \Delta_2; \Gamma \vdash \sigma : \Psi$ *and* $\Delta_1, u{::}\Psi{\vdash}A, \Delta_2; \Gamma \vdash B \equiv [\sigma]A : \mathsf{type}$ *for some* Ψ, A, Δ_1 *and* Δ_2.

6. *If* $\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 : K$ *then* $\Delta; \Gamma \vdash K \equiv \mathsf{type} : \mathsf{kind}$, $\Delta; \Gamma \vdash A_1 : \mathsf{type}$ *and* $\Delta; \Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$.

7. *If* $\Delta; \Gamma \vdash a : K$ *then* $a{:}L$ *in* Σ *and* $\Delta; \Gamma \vdash K \equiv L : \mathsf{kind}$ *for some* L.

8. *If* $\Delta; \Gamma \vdash A\, M : K$, *then* $\Delta; \Gamma \vdash A : \Pi x{:}A_1.K_2$, $\Delta; \Gamma \vdash M : A_1$ *and* $\Delta; \Gamma \vdash K \equiv [\mathsf{id}_\Gamma, M/x]K_2 : \mathsf{kind}$ *for some* A_1 *and* K_2.

Proof: By straightforward induction on typing derivations. Validity is needed in most cases in order to apply reflexivity. □

Lemma 13 (Redundancy of typing premises) *The indicated typing premises in the rules of parallel conversion and instantiation of modal variables are redundant.*

Proof: By inspecting the rules and validity. □

Lemma 14 (Equality inversion)

1. *If* $\Delta; \Gamma \vdash K \equiv \mathsf{type} : \mathsf{kind}$ *or* $\Delta; \Gamma \vdash \mathsf{type} \equiv K : \mathsf{kind}$ *then* $K = \mathsf{type}$.

2. *If* $\Delta; \Gamma \vdash K \equiv \Pi x{:}B_1.L_2 : \mathsf{kind}$ *or* $\Delta; \Gamma \vdash \Pi x{:}B_1.L_2 \equiv K : \mathsf{kind}$ *then* $K = \Pi x{:}A_1.K_2$ *such that* $\Delta; \Gamma \vdash A_1 \equiv B_1 : \mathsf{kind}$ *and* $\Delta; \Gamma, x{:}A_1 \vdash K_2 \equiv L_2 : \mathsf{kind}$.

3. *If* $\Delta; \Gamma \vdash A \equiv \Pi x{:}B_1.B_2 : \mathsf{type}$ *or* $\Delta; \Gamma \vdash \Pi x{:}B_1.B_2 \equiv A : \mathsf{type}$ *then* $A = \Pi x{:}A_1.A_2$ *for some* A_1 *and* A_2 *such that* $\Delta; \Gamma \vdash A_1 \equiv B_1 : \mathsf{type}$ *and* $\Delta; \Gamma, x{:}A_1 \vdash A_2 \equiv B : \mathsf{type}$.

Proof: By induction on the given equality derivations. □

Lemma 15 (Injectivity of Products)

1. *If* $\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 \equiv \Pi x{:}B_1.B_2 : \mathsf{type}$ *then* $\Delta; \Gamma \vdash A_1 \equiv B_1 : \mathsf{type}$ *and* $\Delta; \Gamma, x{:}A_1 \vdash A_2 \equiv B_2 : \mathsf{type}$.

2. *If* $\Delta; \Gamma \vdash \Pi x{:}A_1.K_2 \equiv \Pi x{:}B_1.L_2 : \mathsf{kind}$ *then* $\Delta; \Gamma \vdash A_1 \equiv B_1 : \mathsf{type}$ *and* $\Delta; \Gamma, x{:}A_1 \vdash K_2 \equiv L_2 : \mathsf{kind}$.

Proof: Immediate by equality inversion (lemma 14). □

2.8 Type-directed algorithmic equivalence

One important question in practice is whether it is still possible to effectively decide whether two terms are definitionally equal. In [29], Harper and Pfenning present a type-directed equivalence algorithm. We will extend this algorithm to allow modal variables. Crucial in the correctness proof for the algorithmic equality in [29] is the observation that we can erase all dependencies among types to obtain a simply typed calculus and then show that algorithm for equality is correct in this simply typed calculus. This idea carries over to the modal extension straightforwardly. Following [29], we write α for simple base types and have a special type constants, type^-.

$$\begin{array}{rcll}
\text{Simple Kinds} & \kappa & ::= & \mathsf{type}^- \mid \tau \rightarrow \kappa \\
\text{Simple Types} & \tau & ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \\
\text{Simple contexts} & \Omega, \Phi & ::= & \cdot \mid \Omega,\ x{:}\tau \\
\text{Simple modal contexts} & \Lambda & ::= & \cdot \mid \Lambda,\ x{:}(\Phi \vdash \tau)
\end{array}$$

We write A^- for the simple type that results from erasing dependencies in A, and similarly K^-.

$$\begin{array}{rcl}
(a)^- & = & a^- \\
(A\ M)^- & = & A^- \\
(\Pi x{:}A_1.A_2)^- & = & A_1^- \to A_2^- \\
\\
\mathsf{type}^- & = & \mathsf{type}^- \\
(\Pi x{:}A.K)^- & = & A^- \to K^- \\
\\
(\cdot)^- & = & \cdot \\
(\Gamma,\ x{:}A)^- & = & \Gamma^-,\ x{:}A^- \\
\\
(\cdot)^- & = & \cdot \\
(\Delta,\ x{:}(\Psi \vdash A)^- & = & \Delta^-,\ x{:}(\Psi^- \vdash A^-) \\
(\mathsf{kind})^- & = & \mathsf{kind}^-
\end{array}$$

Lemma 16 (Erasure preservation)

1. *If* $\Delta; \Gamma \vdash A \equiv B : K$ *then* $A^- = B^-$.

2. *If* $\Delta; \Gamma \vdash K \equiv L : \mathsf{kind}$ *then* $K^- = L^-$.

3. *If* $\Delta; \Gamma \vdash B : K$ *and* $\Delta; \Psi \vdash \sigma : \Gamma$ *then* $B^- = [\sigma]B^-$.

4. *If* $\Delta; \Gamma \vdash K : \mathsf{kind}$ *and* $\Delta; \Psi \vdash \sigma : \Gamma$ *then* $K^- = [\sigma]K^-$.

5. *If* $\Delta; \Gamma \vdash \mathsf{id}_\Gamma : \Gamma$ *then* $\mathsf{id}_\Gamma = \mathsf{id}_{\Gamma^-}$.

Proof: By induction over the structure of the given derivation. □

The following four judgments describe algorithmic equality:

$$\begin{array}{ll}
\Lambda; \Omega \vdash M \xrightarrow{whr} M' & M \text{ weak head reduces to } M' \\
\Lambda; \Omega \vdash M \Longleftrightarrow N : \tau & M \text{ is equal to } N \\
\Lambda; \Omega \vdash M \longleftrightarrow N : \tau & M \text{ is structurally equal to } N \\
\Lambda; \Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi & \sigma \text{ is structurally equal to } \sigma'
\end{array}$$

For the weak head reduction, it is not strictly necessary to carry around Λ and Ω explicitely, but it will make the weak head reduction rules more precise. Next, we give the type-directed equality rules.

Substitution equality

$$\frac{}{\Lambda;\Omega \vdash \cdot \longleftrightarrow \cdot : \cdot} \qquad \frac{\Lambda;\Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi \quad \Lambda;\Omega \vdash M \Longleftrightarrow N : \tau}{\Lambda;\Omega \vdash (\sigma, M/x) \longleftrightarrow (\sigma', N/x) : (\Phi, x{:}\tau)}$$

Weak head reduction

$$\frac{}{\Lambda;\Omega \vdash (\lambda x{:}A_1.M_2)\, M_1 \xrightarrow{whr} [\mathsf{id}_\Omega, M_1/x]M_2} \qquad \frac{\Lambda;\Omega \vdash M_1 \xrightarrow{whr} M_1'}{\Lambda;\Omega \vdash M_1\, M_2 \xrightarrow{whr} M_1'\, M_2}$$

Type-directed object equality

$$\frac{\Lambda;\Omega \vdash M \xrightarrow{whr} M' \quad \Lambda;\Omega \vdash M' \Longleftrightarrow N : \alpha}{\Lambda;\Omega \vdash M \Longleftrightarrow N : \alpha} \qquad \frac{\Lambda;\Omega \vdash N \xrightarrow{whr} N' \quad \Lambda;\Omega \vdash M \Longleftrightarrow N' : \alpha}{\Lambda;\Omega \vdash M \Longleftrightarrow N : \alpha}$$

$$\frac{\Lambda;\Omega \vdash M \longleftrightarrow N : \alpha}{\Lambda;\Omega \vdash M \Longleftrightarrow N : \alpha} \qquad \frac{\Lambda;\Omega, x{:}\tau_1 \vdash M\, x \Longleftrightarrow N\, x : \tau_2}{\Lambda;\Omega \vdash M \Longleftrightarrow N : \tau_1 \to \tau_2}$$

Structural object equality

$$\frac{x{:}\tau \text{ in } \Omega}{\Lambda;\Omega \vdash x \longleftrightarrow x : \tau} \qquad \frac{c{:}A \text{ in } \Sigma}{\Lambda;\Omega \vdash c \longleftrightarrow c : A^-}$$

$$\frac{u{::}(\Phi \vdash \tau) \text{ in } \Lambda \quad \Lambda;\Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi}{\Lambda;\Omega \vdash u[\sigma] \longleftrightarrow u[\sigma'] : \tau} *$$

$$\frac{\Lambda;\Omega \vdash M_1 \longleftrightarrow N_1 : \tau_2 \to \tau_1 \quad \Lambda;\Omega \vdash M_2 \Longleftrightarrow N_2 : \tau_2}{\Lambda;\Omega \vdash M_1\, M_2 \longleftrightarrow N_1\, N_2 : \tau_1}$$

Note that in the rule *, we do not apply σ to the type τ. Since all dependencies have been erased, τ cannot depend on any variables in Ω. The algorithm is essentially deterministic in the sense that when comparing terms at base type, we first weakly head normalize both sides and then compare the results structurally. Two modal variables are only structurally equal if they actually are the same modal variable. This means that if we implement existential variables via references, then two uninstantiated existential variables are only structurally equal if they point to the same reference. This algorithm closely describes the actual implementation and the treatment of existential variables in it. We mirror these equality judgements on the family and kind level.

Kind-directed object equality

$$\frac{\Lambda;\Omega \vdash A \longleftrightarrow B : \mathsf{type}^-}{\Lambda;\Omega \vdash A \Longleftrightarrow B : \mathsf{type}^-} \qquad \frac{\Lambda;\Omega, x{:}\tau \vdash A\,x \Longleftrightarrow B\,x : \kappa}{\Lambda;\Omega \vdash A \Longleftrightarrow B : \tau \rightarrow \kappa}$$

Structural family equality

$$\frac{a{:}K \text{ in } \Sigma}{\Lambda;\Omega \vdash a \longleftrightarrow a : K^-} \qquad \frac{\Lambda;\Omega \vdash A \longleftrightarrow B : \tau \rightarrow \kappa \quad \Lambda;\Omega \vdash M \Longleftrightarrow N : \tau}{\Lambda;\Omega \vdash A\,M \longleftrightarrow B\,N : \kappa}$$

Algorithmic kind equality

$$\frac{}{\Lambda;\Omega \vdash \mathsf{type} \Longleftrightarrow \mathsf{type} : \mathsf{kind}^-} \qquad \frac{\Lambda;\Omega \vdash A \Longleftrightarrow B : \mathsf{type}^- \quad \Lambda;\Omega \vdash K \Longleftrightarrow L : \mathsf{kind}^-}{\Lambda;\Omega \vdash \Pi x{:}A.K \Longleftrightarrow \Pi x{:}B.L : \mathsf{kind}^-}$$

The algorithmic equality satisfies some straightforward structural properties, such as exchange, contraction, strengthening, and weakening. Only weakening is required in the proof of its correctness.

Lemma 17 (Weakening)

1. *If* $\Lambda;\ \Omega, \Omega' \vdash \sigma \longleftrightarrow \sigma' : \Omega''$ *then* $\Lambda;\ \Omega, x{:}\tau, \Omega' \vdash \sigma \longleftrightarrow \sigma' : \Omega''$.
2. *If* $\Lambda;\ \Omega, \Omega' \vdash M \Longleftrightarrow N : \kappa$ *then* $\Lambda;\ \Omega, x{:}\tau, \Omega' \vdash M \Longleftrightarrow N : \kappa$.
3. *If* $\Lambda;\ \Omega, \Omega' \vdash M \longleftrightarrow N : \kappa$ *then* $\Lambda;\ \Omega, x{:}\tau, \Omega' \vdash M \longleftrightarrow N : \kappa$.
4. *If* $\Lambda;\ \Omega, \Omega' \vdash A \Longleftrightarrow B : \kappa$ *then* $\Lambda;\ \Omega, x{:}\tau, \Omega' \vdash A \Longleftrightarrow B : \kappa$
5. *If* $\Lambda;\ \Omega, \Omega' \vdash A \longleftrightarrow B : \kappa$ *then* $\Lambda;\ \Omega, x{:}\tau, \Omega' \vdash A \longleftrightarrow B : \kappa$
6. *If* $\Lambda;\ \Omega, \Omega' \vdash K \Longleftrightarrow L : \mathsf{kind}^-$ *then* $\Lambda;\ \Omega, x{:}\tau, \Omega' \vdash K \Longleftrightarrow L : \mathsf{kind}^-$

To show the above extension to the equality algorithm is correct, we can follow the development in [29]. We first consider determinacy of algorithmic equality.

Theorem 18 (Determinacy of algorithmic equality)

1. *If* $\Lambda;\Omega \vdash M \xrightarrow{whr} M_1$ *and* $\Lambda;\Omega \vdash M \xrightarrow{whr} M_2$ *then* $M_1 = M_2$.
2. *If* $\Lambda;\Omega \vdash M \longleftrightarrow N : \tau$ *then there is no* M' *such that* $\Lambda;\Omega \vdash M \xrightarrow{whr} M'$.
3. *If* $\Lambda;\Omega \vdash M \longleftrightarrow N : \tau$ *then there is no* N' *such that* $\Lambda;\Omega \vdash N \xrightarrow{whr} N'$.
4. *If* $\Lambda;\Omega \vdash M \longleftrightarrow N : \tau$ *and* $\Lambda;\Omega \vdash M \longleftrightarrow N : \tau'$ *then* $\tau = \tau'$.
5. *If* $\Lambda;\Omega \vdash A \longleftrightarrow B : \kappa$ *and* $\Lambda;\Omega \vdash A \longleftrightarrow B : \kappa'$ *then* $\kappa = \kappa'$.

Proof: The proofs follow [29]. Proof of (1) is a straightforward induction on the derivation. For (2), we assume

$$\Lambda;\Omega \vdash M \overset{\mathcal{S}}{\longleftrightarrow} N : \tau \quad \text{and} \quad \overset{\mathcal{W}}{\Lambda;\Omega \vdash M \xrightarrow{whr} M'}$$

for some M'. We show by induction over $\mathcal{S}$ and $\mathcal{W}$ that these assumptions are contradictory. Whenever, we have constructed a judgment such that there is no rule that could conclude this judgment we say we obtain a contradiction by inversion.

Case $\mathcal{S} = \dfrac{x{:}\tau \text{ in } \Omega}{\Lambda;\, \Omega \vdash x \longleftrightarrow x : \tau}$

$\Lambda;\Omega \vdash x \xrightarrow{whr} M'$ by assumption
Contradiction by inversion

Case $\mathcal{S} = \dfrac{}{\Lambda, u{::}\Phi{\vdash}\tau, \Lambda';\, \Omega \vdash u[\sigma] \longleftrightarrow u[\sigma] : \tau}$

$\Lambda, u{::}\Phi{\vdash}\tau, \Lambda';\Omega \vdash u[\sigma] \xrightarrow{whr} M'$ by assumption
Contradiction by inversion

Case $\mathcal{S} = \dfrac{\Lambda;\, \Omega \vdash M_1 \overset{\mathcal{S}_1}{\longleftrightarrow} N_1 : \tau_2 \to \tau_1 \qquad \Lambda;\, \Omega \vdash M_2 \Longleftrightarrow N_2 : \tau_2}{\Lambda;\, \Omega \vdash M_1\, M_2 \longleftrightarrow N_1\, N_2 : \tau_1}$

$\Lambda;\Omega \vdash M_1\, M_2 \xrightarrow{whr} M'$ by assumption

Sub-case $\mathcal{W} = \dfrac{}{\Lambda;\Omega \vdash (\lambda x{:}A_1.M'_1)\, M_2 \xrightarrow{whr} [\mathsf{id}_\Omega, M_2/x]M'_1}$

$M_1 = \lambda x{:}A_1.M_1'$

$\Lambda;\,\Omega \vdash (\lambda x{:}A_1.M_1') \longleftrightarrow N_2 : \tau_2 \to \tau_1$ by $\mathcal{S}_1$

contradiction by inversion

Sub-case $\mathcal{W} = \dfrac{\overset{\mathcal{W}_1}{\Lambda;\Omega \vdash M_1 \xrightarrow{whr} M_1'}}{\Lambda;\Omega \vdash M_1\, M_2 \xrightarrow{whr} M_1'\, M_2}$

contradiction by i.h. on $\mathcal{S}_1$ and $\mathcal{W}_1$

□

Using determinacy, we can then prove symmetry and transitivity of algorithmic equality.

Theorem 19 (Symmetry of algorithmic equality)

1. *If* $\Lambda;\Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi$ *then* $\Lambda;\Omega \vdash \sigma' \longleftrightarrow \sigma : \Phi$.

2. *If* $\Lambda;\Omega \vdash M \longleftrightarrow N : \tau$ *then* $\Lambda;\Omega \vdash N \longleftrightarrow M : \tau$.

3. *If* $\Lambda;\Omega \vdash M \Longleftrightarrow N : \tau$ *then* $\Lambda;\Omega \vdash N \Longleftrightarrow M : \tau$.

4. *If* $\Lambda;\Omega \vdash A \longleftrightarrow B : \kappa$ *then* $\Lambda;\Omega \vdash B \longleftrightarrow A : \kappa$.

5. *If* $\Lambda;\Omega \vdash A \Longleftrightarrow B : \kappa$ *then* $\Lambda;\Omega \vdash B \Longleftrightarrow A : \kappa$.

6. *If* $\Lambda;\Omega \vdash K \longleftrightarrow L : \mathsf{kind}^-$ *then* $\Lambda;\Omega \vdash L \longleftrightarrow K : \mathsf{kind}^-$.

Proof: By simultaneous induction on the given derivation. □

Theorem 20 (Transitivity of algorithmic equality)

1. *If* $\Lambda;\Omega \vdash \sigma_1 \longleftrightarrow \sigma_2 : \Phi$ *and* $\Lambda;\Omega \vdash \sigma_2 \longleftrightarrow \sigma_3 : \Phi$ *then* $\Lambda;\Omega \vdash \sigma_1 \longleftrightarrow \sigma_3 : \Phi$.

2. *If* $\Lambda; \Omega \vdash M \Longleftrightarrow N : \tau$ *and* $\Lambda; \Omega \vdash N \Longleftrightarrow O : \tau$ *then* $\Lambda; \Omega \vdash M \Longleftrightarrow O : \tau$.

3. *If* $\Lambda; \Omega \vdash M \longleftrightarrow N : \tau$ *and* $\Lambda; \Omega \vdash N \longleftrightarrow O : \tau$ *then* $\Lambda; \Omega \vdash M \longleftrightarrow O : \tau$.

4. *If* $\Lambda; \Omega \vdash A \Longleftrightarrow B : \kappa$ *and* $\Lambda; \Omega \vdash B \Longleftrightarrow C : \kappa$ *then* $\Lambda; \Omega \vdash A \Longleftrightarrow C : \kappa$.

5. *If* $\Lambda; \Omega \vdash A \longleftrightarrow B : \kappa$ *and* $\Lambda; \Omega \vdash B \longleftrightarrow C : \kappa$ *then* $\Lambda; \Omega \vdash A \longleftrightarrow C : \kappa$.

6. *If* $\Lambda; \Omega \vdash K \Longleftrightarrow L : \mathsf{kind}^-$ *and* $\Lambda; \Omega \vdash L \Longleftrightarrow L' : \mathsf{kind}^-$
   then $\Lambda; \Omega \vdash K \Longleftrightarrow L' : \mathsf{kind}^-$.

Proof: By simultaneous inductions on the structure of the given derivations. In each case, one of the two derivations is strictly smaller, while the other derivation is either smaller or the same. The proof requires determinacy and follows the proof in [29]. □

2.9 Completeness of algorithmic equality

We now develop the completeness theorem for the type-directed equality algorithm by an argument via logical relations. The logical relations are defined inductively on the approximate type of an object.

The completeness theorem can be stated as follows for type-directed equality:

$$\text{If } \Delta;\, \Gamma \vdash M \equiv N : A \text{ then } \Delta^-;\, \Gamma^- \vdash M \Longleftrightarrow N : A^-.$$

Here we define a logical relation $\Lambda;\ \Omega \vdash M = N \in [\![\tau]\!]$ that provides a stronger induction hypothesis s.t.

1. if $\Delta;\ \Gamma \vdash M \equiv N : A$ then $\Delta^-;\ \Gamma^- \vdash M = N \in [\![A^-]\!]$

2. if $\Delta;\ \Gamma \vdash M = N \in [\![A^-]\!]$ then $\Delta^-;\ \Gamma^- \vdash M \Longleftrightarrow N : A^-$

Following Harper and Pfenning, we define a Kripke logical relation inductively on simple types. At base type we require the property we eventually want to prove. At higher types we reduce the property to those for simpler types. We extend here the logical relation given in [29] to allow modal contexts. Since we do not introduce any new modal variables in the algorithm for type-directed equivalence, the modal context does not change and is essentially just carried along. In contrast, the context Ω for bound variables may be extended. This is accounted for in the case for the function type. We say Ω' extends Ω (written $\Omega' \geq \Omega$) if Ω' contains all declarations in Ω and possibly more.

1. $\Lambda; \Omega \vdash \sigma = \theta \in [\![\,\cdot\,]\!]$ iff $\sigma = \cdot$ and $\theta = \cdot$.
2. $\Lambda; \Omega \vdash \sigma = \theta \in [\![\Phi, x{:}\tau]\!]$ iff $\sigma = (\sigma', M/x)$ and $\theta = (\theta', N/x)$ where $\Lambda; \Omega \vdash \sigma' = \theta' \in [\![\Phi]\!]$ and $\Lambda; \Omega \vdash M = N \in [\![\tau]\!]$.
3. $\Lambda; \Omega \vdash M = N \in [\![\alpha]\!]$ iff $\Lambda; \Omega \vdash M \Longleftrightarrow N : \alpha$
4. $\Lambda; \Omega \vdash M = N \in [\![\tau_1 \to \tau_2]\!]$ iff for every Ω' extending Ω and for all M_1 and N_1 s. t. $\Lambda; \Omega' \vdash M_1 = N_1 \in [\![\tau_1]\!]$ we have $\Lambda; \Omega' \vdash M\ M_1 = N\ N_1 \in [\![\tau_2]\!]$.
5. $\Lambda; \Omega \vdash A = B \in [\![\mathsf{type}^-]\!]$ iff $\Lambda; \Omega \vdash A \Longleftrightarrow B : \mathsf{type}^-$
6. $\Lambda; \Omega \vdash A = B \in [\![\tau \to \kappa]\!]$ iff for every Ω' extending Ω and for all M and N s. t. $\Lambda; \Omega' \vdash M = N \in [\![\tau]\!]$ we have $\Lambda; \Omega' \vdash A\ M = B\ N \in [\![\kappa]\!]$.

The general structural properties of logical relations that we can show directly by induction are exchange, weakening, contraction and strengthening. We only prove and use weakening.

Lemma 21 (Weakening)
For all logical relations R, if $\Lambda; (\Omega, \Omega') \vdash R$ then $\Lambda; (\Omega, x{:}\tau, \Omega') \vdash R$.

Proof: By induction on the structure of the definition of R. □

It is straightforward to show that logically related terms are considered identical by the algorithm.

Theorem 22 (Logically related terms are algorithmically equal)

1. *If $\Lambda;\, \Omega \vdash \sigma = \sigma' \in [\![\Phi]\!]$ then $\Lambda;\, \Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi$.*
2. *If $\Lambda;\, \Omega \vdash M = N \in [\![\tau]\!]$ then $\Lambda; \Omega \vdash M \Longleftrightarrow N : \tau$.*

3. *If* $\Lambda; \Omega \vdash M \longleftrightarrow N : \tau$ *then* $\Lambda; \Omega \vdash M = N \in [\![\tau]\!]$.

4. *If* $\Lambda; \Omega \vdash A = B \in [\![\kappa]\!]$ *then* $\Lambda; \Omega \vdash A \Longleftrightarrow B : \kappa$.

5. *If* $\Lambda; \Omega \vdash A \longleftrightarrow B : \kappa$ *then* $\Lambda; \Omega \vdash A = B \in [\![\kappa]\!]$.

Proof: The statements are proven by simultaneous induction on the structure of τ or Φ respectively.
Proof of (1): Induction on Φ.

Case $\Phi = \cdot$

$\Lambda; \Omega \vdash \cdot = \cdot \in [\![\,\cdot\,]\!]$ by assumption
$\Lambda; \Omega \vdash \cdot = \cdot : \cdot$ by rule

Case $\Phi = \Phi', x{:}\tau$

$\Lambda; \Omega \vdash \sigma_1 = \sigma_2 \in [\![\Phi', x{:}\tau]\!]$ by assumption
$\sigma_1 = (\sigma, M/x)$ and $\sigma_2 = (\sigma', N/x)$ by definition
$\Lambda; \Omega \vdash \sigma = \sigma' \in [\![\Phi']\!]$
$\Lambda; \Omega \vdash M = N \in [\![\tau]\!]$
$\Lambda; \Omega \vdash M \Longleftrightarrow N : \tau$ by i.h. (2)
$\Lambda; \Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi'$ by i.h. (1)
$\Lambda; \Omega \vdash (\sigma, M/x) \longleftrightarrow (\sigma', N/x) : (\Phi', x{:}\tau)$ by rule

Proof of (2) by induction on the structure of τ:

Case $\tau = \alpha$

$\Lambda; \Omega \vdash M = N \in [\![\alpha]\!]$ by assumption
$\Lambda; \Omega \vdash M \Longleftrightarrow N : \alpha$ by definition

Case $\tau = \tau_1 \to \tau_2$

$\Lambda; \Omega \vdash M = N \in [\![\tau_1 \to \tau_2]\!]$ by assumption
$\Lambda; \Omega, x{:}\tau_1 \vdash x \longleftrightarrow x : \tau_1$ by rule
$\Lambda; \Omega, x{:}\tau_1 \vdash x = x \in [\![\tau_1]\!]$ by i.h. (3)
$\Lambda; \Omega, x{:}\tau_1 \vdash M\,x = N\,x \in [\![\tau_2]\!]$ by definition
$\Lambda; \Omega, x{:}\tau_1 \vdash M\,x \Longleftrightarrow N\,x : \tau_2$ by i.h. (2)
$\Lambda; \Omega \vdash M \Longleftrightarrow N : \tau_1 \to \tau_2$ by rule

Proof of (3):

Case $\tau = \alpha$

$\Lambda; \Omega \vdash M \longleftrightarrow N : \alpha$	by assumption
$\Lambda; \Omega \vdash M \Longleftrightarrow N : \alpha$	by rule
$\Lambda; \Omega \vdash M = N \in \llbracket\alpha\rrbracket$	by def.

Case $\tau = \tau_1 \rightarrow \tau_2$

$\Lambda; \Omega \vdash M \longleftrightarrow N : \tau_1 \rightarrow \tau_2$	by assumption
$\Lambda; \Omega^+ \vdash M_1 = N_1 \in \llbracket\tau_1\rrbracket$ for any arbitrary $\Omega^+ \geq \Omega$	by new assumption
$\Lambda; \Omega^+ \vdash M_1 \Longleftrightarrow N_1 : \tau_1$	by i.h. (2)
$\Lambda; \Omega^+ \vdash M \longleftrightarrow N : \tau_1 \rightarrow \tau_2$	by weakening
$\Lambda; \Omega^+ \vdash M\, M_1 \longleftrightarrow N\, N_1 : \tau_2$	by rule
$\Lambda; \Omega^+ \vdash M\, M_1 = N\, N_1 \in \llbracket\tau_2\rrbracket$	by i.h. (3)
$\Lambda; \Omega \vdash M = N \in \llbracket\tau_1 \rightarrow \tau_2\rrbracket$	by definition

□

Lemma 23 (Closure under head expansion)

1. *If* $\Lambda; \Omega \vdash M \xrightarrow{whr} M'$ *and* $\Lambda; \Omega \vdash M' = N \in \llbracket\tau\rrbracket$ *then* $M = N \in \llbracket\tau\rrbracket$.

2. *If* $\Lambda; \Omega \vdash N \xrightarrow{whr} N'$ *and* $\Lambda; \Omega \vdash M = N' \in \llbracket\tau\rrbracket$ *then* $M = N \in \llbracket\tau\rrbracket$.

Proof: By induction on the structure of τ.

Case $\tau = \alpha$

$\Lambda; \Omega \vdash M \xrightarrow{whr} M'$	by assumption
$\Lambda; \Omega \vdash M' \Longleftrightarrow N : \alpha$	by definition of $\llbracket\alpha\rrbracket$
$\Lambda; \Omega \vdash M \Longleftrightarrow N : \alpha$	by rule
$\Lambda; \Omega \vdash M = N \in \llbracket\alpha\rrbracket$	by definition

Case $\tau = \tau_1 \rightarrow \tau_2$

$\Lambda; \Omega \vdash M \xrightarrow{whr} M'$	by assumption
$\Lambda; \Omega \vdash M' = N \in \llbracket\tau_1 \rightarrow \tau_2\rrbracket$	by assumption
$\Lambda; \Omega^+ \vdash M_1 = N_1 \in \llbracket\tau_1\rrbracket$ for $\Omega^+ \geq \Omega$	by new assumption
$\Lambda; \Omega^+ \vdash M'\, M_1 = N\, N_1 \in \llbracket\tau_2\rrbracket$	definition

$\Lambda; \Omega \vdash M\, M_1 \xrightarrow{whr} M'\, M_1$ by whr rule
$\Lambda;\, \Omega^+ \vdash M\, M_1 = N\, N_1 \in [\![\tau_2]\!]$ by i.h.
$\Lambda;\, \Omega \vdash M = N \in [\![\tau_1 \rightarrow \tau_2]\!]$ by definition $[\![\tau_1 \rightarrow \tau_2]\!]$

□

Note that the proof of this lemma only depends on the structure of τ and not on the derivations of weak head reduction. This lemma is critical in order to prove that definitional equal terms are logically related under substitutions.

Lemma 24 (Symmetry of logical relations)

1. *If* $\Lambda; \Omega \vdash \sigma = \theta \in [\![\Phi]\!]$ *then* $\Lambda; \Omega \vdash \theta = \sigma \in [\![\Phi]\!]$
2. *If* $\Lambda; \Omega \vdash M = N \in [\![\tau]\!]$ *then* $\Lambda; \Omega \vdash N = M \in [\![\tau]\!]$
3. *If* $\Lambda; \Omega \vdash A = B \in [\![\kappa]\!]$ *then* $\Lambda; \Omega \vdash B = A \in [\![\kappa]\!]$

Proof: By induction on the structure of τ and Φ, using lemma on symmetry of algorithmic equality. □

Lemma 25 (Transitivity of logical relations)

1. *If* $\Lambda; \Omega \vdash \sigma = \theta \in [\![\Phi]\!]$ *and* $\Lambda; \Omega \vdash \theta = \rho \in [\![\Phi]\!]$ *then* $\Lambda; \Omega \vdash \sigma = \rho \in [\![\Phi]\!]$
2. *If* $\Lambda; \Omega \vdash M = N \in [\![\tau]\!]$ *and* $\Lambda; \Omega \vdash N = O \in [\![\tau]\!]$ *then* $\Lambda; \Omega \vdash M = O \in [\![\tau]\!]$
3. *If* $\Lambda; \Omega \vdash A = B \in [\![\kappa]\!]$ *and* $\Lambda; \Omega \vdash B = C \in [\![\kappa]\!]$ *then* $\Lambda; \Omega \vdash A = C \in [\![\kappa]\!]$

Proof: By induction on the structure of τ and Φ using lemma on transitivity of algorithmic equality. □

Lemma 26 (Definitionally equal terms are logically related under substitutions)

1. *If* $\Delta; \Gamma \vdash \rho \equiv \rho' : \Psi$ *and* $\Delta^-; \Omega \vdash \sigma = \theta \in [\![\Gamma^-]\!]$
   then $\Delta^-; \Omega \vdash ([\sigma]\rho) = ([\theta]\rho') \in [\![\Psi^-]\!]$.
2. *If* $\Delta; \Gamma \vdash M \equiv N : A$ *and* $\Delta^-; \Omega \vdash \sigma = \theta \in [\![\Gamma^-]\!]$
   then $\Delta^-; \Omega \vdash [\sigma]M = [\theta]N \in [\![A^-]\!]$.

3. *If* $\Delta;\Gamma \vdash A \equiv B : K$ *and* $\Delta^-;\Omega \vdash \sigma = \theta \in [\![\Gamma^-]\!]$
then $\Delta^-;\Omega \vdash [\sigma]A = [\theta]B \in [\![K^-]\!]$.

Proof: By induction on the derivation of definitional equality. The proof follows the development in [29]. We only give the case for modal variables.

Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Delta_1, u{::}\Psi{\vdash}A, \Delta_2;\Gamma \vdash \rho \equiv \rho' : \Psi}}{\Delta_1, u{::}\Psi{\vdash}A, \Delta_2;\Gamma \vdash u[\rho] \equiv u[\rho'] : [\rho]A}$

$\Delta_1^-, u{::}\Psi^-{\vdash}A^-, \Delta_2^-;\Omega \vdash \sigma = \theta \in [\![\Gamma^-]\!]$	by assumption
$\Delta_1^-, u{::}\Psi^-{\vdash}A^-, \Delta_2^-;\Omega \vdash ([\sigma]\rho) = ([\theta]\rho') \in [\![\Psi^-]\!]$	by i.h.
$\Delta_1^-, u{::}\Psi^-{\vdash}A^-, \Delta_2^-;\Omega \vdash ([\sigma]\rho) \longleftrightarrow ([\theta]\rho') : \Psi^-$	by lemma 22
$\Delta_1^-, u{::}\Psi^-{\vdash}A^-, \Delta_2^-;\Omega \vdash u[[\sigma]\rho] \longleftrightarrow u[[\theta]\rho'] : A^-$	by rule
$\Delta_1^-, u{::}\Psi^-{\vdash}A^-, \Delta_2^-;\Omega \vdash u[[\sigma]\rho] = u[[\theta]\rho'] \in [\![A^-]\!]$	by lemma 22
$\Delta_1^-, u{::}\Psi^-{\vdash}A^-, \Delta_2^-;\Omega \vdash [\sigma](u[\rho]) = [\theta](u[\rho']) \in [\![A^-]\!]$	by subst. definition
$\Delta_1^-, u{::}\Psi^-{\vdash}A^-, \Delta_2^-;\Omega \vdash [\sigma](u[\rho]) = [\theta](u[\rho']) \in [\![[\rho]A^-]\!]$	by erasure lemma 16

□

Lemma 27 (Identity substitutions are logically related)
$\Delta^-;\Gamma^- \vdash \mathsf{id}_\Gamma = \mathsf{id}_\Gamma \in [\![\Gamma^-]\!]$

Proof: By structural induction on $[\![\Gamma^-]\!]$ and theorem 22(3). □

Theorem 28 (Definitionally equal terms are logically related)

1. *If* $\Delta;\Gamma \vdash M \equiv N : A$ *then* $\Delta^-;\Gamma^- \vdash M = N \in [\![A^-]\!]$.

2. *If* $\Delta;\Gamma \vdash A \equiv B : K$ *then* $\Delta^-;\Gamma^- \vdash A = B \in [\![K^-]\!]$.

Proof: Direct by lemma 27 and lemma 26. □

Theorem 29 (Completeness of algorithmic equality)

1. *If* $\Delta;\Gamma \vdash M \equiv N : A$ *then* $\Delta^-;\Gamma^- \vdash M \Longleftrightarrow N : A^-$.

2. *If* $\Delta;\Gamma \vdash A \equiv B : K$ *then* $\Delta^-;\Gamma^- \vdash A \Longleftrightarrow B : K^-$.

Proof: Direct by theorem 28 and theorem 22 □

2.10 Soundness of algorithmic equality

We prove soundness of algorithmic equality via subject reduction. The algorithm for type-directed equality is not sound in general. However, when applied to valid objects of the same type, in a valid modal context Δ it is sound and relates only equal terms.

Lemma 30 (Subject reduction)
If $\Delta^-;\Gamma^- \vdash M \xrightarrow{whr} M'$ *and* $\Delta;\Gamma \vdash M : A$ *then* $\Delta;\Gamma \vdash M' : A$ *and* $\Delta;\Gamma \vdash M \equiv M' : A$.

Proof: By induction on the definition of weak head reduction. The two cases follow the proof in [29].

Case $\mathcal{W} = \dfrac{}{\Delta^-;\Gamma^- \vdash (\lambda x{:}A_1.M_2)\ M_1 \xrightarrow{whr} [\mathsf{id}_{\Gamma^-}, M_1/x]M_2}$

$\Delta;\Gamma \vdash (\lambda x{:}A_1.M_2)\ M_1 : A$	by assumption
$\Delta;\Gamma \vdash \lambda x{:}A_1.M_2 : \Pi x{:}B_1.B_2$	by inversion lemma 12
$\Delta;\Gamma \vdash M_1 : B_1$ and $\Delta;\Gamma \vdash [\mathsf{id}_\Gamma, M_1/x]B_2 \equiv A : \mathsf{type}$	
$\Delta;\Gamma \vdash A_1 : \mathsf{type}$ and $\Delta;\Gamma, x{:}A_1 \vdash M_2 : A_2$	by inversion lemma 12
$\Delta;\Gamma \vdash \Pi x{:}A_1.A_2 \equiv \Pi x{:}B_1.B_2 : \mathsf{type}$	
$\Delta;\Gamma \vdash A_1 \equiv B_1 : \mathsf{type}$	by injectivity of products (lemma 15)
$\Delta;\Gamma, x{:}A_1 \vdash A_2 \equiv B_2 : \mathsf{type}$	
$\Delta;\Gamma \vdash M_1 : A_1$	by type conversion
$\Delta;\Gamma \vdash \mathsf{id}_\Gamma : \Gamma$	by definition
$\Delta;\Gamma \vdash (\mathsf{id}_\Gamma, M_1/x) : (\Gamma, x{:}A_1)$	by substitution rule
$\Delta;\Gamma \vdash [\mathsf{id}_\Gamma, M_1/x]A_2 \equiv [\mathsf{id}_\Gamma, M_1/x]B_2 : \mathsf{type}$	by substitution lemma 6
$\Delta;\Gamma \vdash [\mathsf{id}_\Gamma, M_1/x]A_2 \equiv A : \mathsf{type}$	transitivity
$\Delta;\Gamma, x{:}A_1 \vdash M_2 \equiv M_2 : A_2$	by reflexivity
$\Delta;\Gamma \vdash M_1 \equiv M_1 : A_1$	by reflexivity
$\Delta;\Gamma \vdash (\lambda x{:}A_1.M_2)\ M_1 \equiv [\mathsf{id}_\Gamma, M_1/x]M_2 : [\mathsf{id}_\Gamma, M_1/x]A_2$	by rule
$\Delta;\Gamma \vdash (\lambda x{:}A_1.M_2)\ M_1 \equiv [\mathsf{id}_\Gamma, M_1/x]M_2 : A$	by type conversion
$\Delta;\Gamma \vdash (\lambda x{:}A_1.M_2)\ M_1 \equiv [\mathsf{id}_{\Gamma^-}, M_1/x]M_2 : A$	by erase lemma 16
$\Delta;\Gamma \vdash [\mathsf{id}_\Gamma, M_1/x]M_2 : [\mathsf{id}_\Gamma, M_1/x]A_2$	by subst. property lemma 6
$\Delta;\Gamma \vdash [\mathsf{id}_\Gamma, M_1/x]M_2 : A$	by type conversion
$\Delta;\Gamma \vdash [\mathsf{id}_{\Gamma^-}, M_1/x]M_2 : A$	by erase lemma 16

Case $\mathcal{W} = \dfrac{\Delta^-;\Gamma^- \vdash M \overset{whr}{\longrightarrow} N}{\Delta^-;\Gamma^- \vdash M\ M_2 \overset{whr}{\longrightarrow} N\ M_2}$

$\Delta;\Gamma \vdash M\ M_2 : A$	by assumption
$\Delta;\Gamma \vdash [\mathsf{id}_\Gamma, M_2/x]A_1 \equiv A : \mathsf{type}$	by inversion lemma 12
$\Delta;\Gamma \vdash M : \Pi x{:}A_2.A_1$ and $\Delta;\Gamma \vdash M_2 : A_2$	
$\Delta;\Gamma \vdash N : \Pi x{:}A_2.A_1$	by i.h.
$\Delta;\Gamma \vdash M \equiv N : \Pi x{:}A_2.A_1$	by i.h.
$\Delta;\Gamma \vdash M_2 \equiv M_2 : A_2$	by reflexivity
$\Delta;\Gamma \vdash M\ M_2 \equiv N\ M_2 : [\mathsf{id}_\Gamma, M_2/x]A_1$	by rule
$\Delta;\Gamma \vdash M\ M_2 \equiv N\ M_2 : A$	by type conversion
$\Delta;\Gamma \vdash N\ M_2 : [\mathsf{id}_\Gamma, M_2/x]A_1$	by rule
$\Delta;\Gamma \vdash N\ M_2 : A$	by type conversion

□

Theorem 31 (Soundness of algorithmic equality)

1. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta;\Gamma \vdash \sigma' : \Psi$ *and* $\Delta^-;\Gamma^- \vdash \sigma \longleftrightarrow \sigma' : \Psi^-$ *then* $\Delta;\Gamma \vdash \sigma \equiv \sigma' : \Psi$

2. *If* $\Delta;\Gamma \vdash M : A$ *and* $\Delta;\Gamma \vdash N : A$ *and* $\Delta^-;\Gamma^- \vdash M \Longleftrightarrow N : A^-$ *then* $\Delta;\Gamma \vdash M \equiv N : A$.

3. *If* $\Delta;\Gamma \vdash M : A$ *and* $\Delta;\Gamma \vdash N : B$ *and* $\Delta^-;\Gamma^- \vdash M \longleftrightarrow N : \tau$ *then* $\Delta;\Gamma \vdash M \equiv N : A$ *and* $\Delta;\Gamma \vdash A \equiv B : \mathsf{type}$ *and* $A^- = B^- = \tau$.

4. *If* $\Delta;\Gamma \vdash A : K$ *and* $\Delta;\Gamma \vdash B : K$ *and* $\Delta^-;\Gamma^- \vdash A \Longleftrightarrow B : K^-$ *then* $\Delta;\Gamma \vdash A \equiv B : K$.

5. *If* $\Delta;\Gamma \vdash A : K$ *and* $\Delta;\Gamma \vdash B : L$ *and* $\Delta^-;\Gamma^- \vdash A \longleftrightarrow B : \kappa$ *then* $\Delta;\Gamma \vdash A \equiv B : K$ *and* $\Delta;\Gamma \vdash K \equiv L : \mathsf{kind}$ *and* $K^- = L^- = \kappa$.

6. *If* $\Delta;\Gamma \vdash K : \mathsf{kind}$ *and* $\Delta;\Gamma \vdash L : \mathsf{kind}$ *and* $\Delta^-;\Gamma^- \vdash K \Longleftrightarrow L : \mathsf{kind}^-$ *then* $\Delta;\Gamma \vdash K \equiv L : \mathsf{kind}$.

Proof: By induction on the structure of the given derivations for algorithmic equality. Proof for (1).

Case $\mathcal{T} = \dfrac{}{\Delta^-; \Gamma^- \vdash \cdot \longleftrightarrow \cdot : \cdot}$

$\Delta; \Gamma \vdash \cdot \equiv \cdot : \cdot$	by rule

Case $\mathcal{T} = \dfrac{\Delta^-; \Gamma^- \vdash M \Longleftrightarrow N : A^- \qquad \Delta^-; \Gamma^- \vdash \sigma \longleftrightarrow \sigma' : \Psi^-}{\Delta^-; \Gamma^- \vdash (\sigma, M/x) \longleftrightarrow (\sigma', N/x) : (\Psi^-, x{:}A^-)}$

$\Delta; \Gamma \vdash (\sigma, M/x) : (\Psi, x{:}A)$	by assumption
$\Delta; \Gamma \vdash \sigma : \Psi$ and $\Delta; \Gamma \vdash M : [\sigma]A$	by inversion
$\Delta; \Gamma \vdash (\sigma', N/x) : (\Psi, x{:}A)$	by assumption
$\Delta; \Gamma \vdash \sigma : \Psi$ and $\Delta; \Gamma \vdash N : [\sigma']A$	by inversion
$\Delta; \Gamma \vdash \sigma \equiv \sigma' : \Psi$	by i.h.
$\Delta; \Gamma \vdash A : \mathsf{type}$	by validity of ctx Ψ
$\Delta; \Gamma \vdash [\sigma]A \equiv [\sigma']A : \mathsf{type}$	by functionality lemma 9
$\Delta; \Gamma \vdash [\sigma']A \equiv [\sigma]A : \mathsf{type}$	by symmetry
$\Delta; \Gamma \vdash N : [\sigma]A$	by type conversion
$\Delta^-; \Gamma^- \vdash M \Longleftrightarrow N : ([\sigma]A)^-$	by erase lemma 16
$\Delta; \Gamma \vdash M \equiv N : [\sigma]A$	by i.h.
$\Delta; \Gamma \vdash (\sigma, M/x) \equiv (\sigma', N/x) : (\Psi, x{:}A)$	by rule

Proof for (2)

Case $\mathcal{T} = \dfrac{\overset{\mathcal{W}_1}{\Delta^-; \Gamma^- \vdash N \xrightarrow{whr} M'} \qquad \overset{\mathcal{T}_2}{\Delta^-; \Gamma^- \vdash M' \Longleftrightarrow M : A^-}}{\Delta^-; \Gamma^- \vdash N \Longleftrightarrow M : A^-}$

$\Delta; \Gamma \vdash N : A$	by assumption
$\Delta; \Gamma \vdash M : A$	by assumption
$\Delta; \Gamma \vdash N \equiv M' : A$	by subject reduction on $\mathcal{W}_1$
$\Delta; \Gamma \vdash M' : A$	by validity

$\Delta; \Gamma \vdash M' \equiv M : A$ by i.h. on $\mathcal{T}_2$
$\Delta; \Gamma \vdash N \equiv M : A$ by transitivity

Case $\mathcal{T} = \dfrac{\overset{\mathcal{W}_1}{\Delta^-; \Gamma^- \vdash M \xrightarrow{whr} M'} \qquad \overset{\mathcal{T}_2}{\Delta^-; \Gamma^- \vdash N \Longleftrightarrow M' : A^-}}{\Delta^-; \Gamma^- \vdash N \Longleftrightarrow M : A^-}$

similar to the one above (symmetric).

Case $\mathcal{T} = \dfrac{\Delta^-; \Gamma^- \vdash M \longleftrightarrow N : A^-}{\Delta^-; \Gamma^- \vdash M \Longleftrightarrow N : A^-}$

$\Delta; \Gamma \vdash M : A$ by assumption
$\Delta; \Gamma \vdash N : A$ by assumption
$\Delta; \Gamma \vdash M \equiv N : A$ by i.h.

Case $\mathcal{T} = \dfrac{\overset{\mathcal{T}_1}{\Delta^-; \Gamma^-, x{:}\tau_1 \vdash M\,x \Longleftrightarrow N\,x : \tau_2}}{\Delta^-; \Gamma^- \vdash M \Longleftrightarrow N : \tau_1 \to \tau_2}$

$\Delta; \Gamma \vdash M : \Pi x{:}A_1.A_2$ by assumption
$\Delta; \Gamma \vdash N : \Pi x{:}A_1.A_2$ by assumption
$(A_1)^- = \tau_1$ and $(A_2)^- = \tau_2$ by erasure def.
$\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 : \mathsf{type}$ by validity
$\Delta; \Gamma \vdash A_1 : \mathsf{type}$ and $\Delta; \Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$ by inversion lemma 12
$\Delta; \Gamma, x{:}A_1 \vdash x : A_1$ by var rule
$\Delta; \Gamma, x{:}A_1 \vdash M : \Pi x{:}A_1.A_2$ by weakening
$\Delta; \Gamma, x{:}A_1 \vdash M\,x : A_2$ by rule (app)
$\Delta; \Gamma, x{:}A_1 \vdash N\,x : A_2$ by rule (app) and weakening
$\Delta; \Gamma, x{:}A_1 \vdash M\,x \equiv N\,x : A_2$ by i.h. on $\mathcal{T}_1$
$\Delta; \Gamma \vdash M \equiv N : \Pi x{:}A_1.A_2$ by rule (extensionality)

Proof (3), we only give the case for modal variable.

Case $\mathcal{S} = \dfrac{u{::}(\Psi)^- \vdash (A_1)^- \in \Delta^- \qquad \overset{\mathcal{S}_1}{(\Delta)^-; \Gamma^- \vdash \sigma \longleftrightarrow \sigma' : \Psi^-}}{(\Delta)^-; \Gamma^- \vdash u[\sigma] \longleftrightarrow u[\sigma'] : \tau}$

$\Delta; \Gamma \vdash u[\sigma] : A$	by assumption
$\Delta; \Gamma \vdash u[\sigma'] : B$	by assumption
$\Delta; \Gamma \vdash A \equiv [\sigma]A_1 : \mathsf{type}$ and $\Delta = \Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2$	by inversion lemma 12
$\Delta; \Gamma \vdash \sigma : \Psi$	
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Gamma \vdash B \equiv [\sigma']A_1 : \mathsf{type}$	by inversion lemma 12
$\Delta; \Gamma \vdash \sigma' : \Psi$	
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Gamma \vdash [\sigma']A_1 \equiv B : \mathsf{type}$	by symmetry
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Gamma \vdash \sigma \equiv \sigma' : \Psi$	by i.h. on $\mathcal{S}_1$
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Psi \vdash A_1 : \mathsf{type}$	by validity and weakening
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Psi \vdash A_1 \equiv A_1 : \mathsf{type}$	by reflexivity
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Gamma \vdash [\sigma]A_1 \equiv [\sigma']A_1 : \mathsf{type}$	by functionality lemma 9
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Gamma \vdash A \equiv [\sigma']A_1 : \mathsf{type}$	by transitivity
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Gamma \vdash A \equiv B : \mathsf{type}$	by transitivity
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Gamma \vdash u[\sigma] \equiv u[\sigma'] : [\sigma]A_1$	by rule
$\Delta_1, u{::}\Psi{\vdash}A_1, \Delta_2; \Gamma \vdash u[\sigma] \equiv u[\sigma'] : A$	by type conversion
$(A)^- = ([\sigma]A_1)^- = (A_1)^- = \tau$	by erasure property
$(B)^- = ([\sigma']A_1)^- = (A_1)^- = \tau$	by erasure property

□

Corollary 32 (Logically related terms are definitionally equal)

1. *If* $\Delta; \Gamma \vdash M : A$, $\Delta; \Gamma \vdash N : A$, *and* $\Delta^-; \Gamma^- \vdash M = N \in [\![A^-]\!]$ *then* $\Delta; \Gamma \vdash M \equiv N : A$.

2. *If* $\Delta; \Gamma \vdash A : K$, $\Delta; \Gamma \vdash B : K$, *and* $\Delta^-; \Gamma^- \vdash A = B \in [\![K^-]\!]$ *then* $\Delta; \Gamma \vdash A \equiv B : K$.

Proof: Direct from assumptions and prior theorems.

$\Delta^-; \Gamma^- \vdash M = N \in [\![A^-]\!]$	by assumption
$\Delta^-; \Gamma^- \vdash M \Longleftrightarrow N : A^-$	by completeness theorem 22
$\Delta; \Gamma \vdash M \equiv N : A$	by soundness theorem 31

□

We can now show that the inference rules for algorithmic equality constitute a decision procedure. We say an object is *normalizing* iff it is related to some term by the type-directed equivalence algorithm. More precisely, M is *normalizing* at simple type τ iff $\Lambda; \Omega \vdash M \Longleftrightarrow M' : \tau$ for some term M'. By transitivity and symmetry, this implies that $\Lambda; \Omega \vdash M \Longleftrightarrow M : \tau$. A term M is *structurally normalizing* iff it is related to some term by the structural equivalence algorithm, i.e. $\Lambda; \Omega \vdash M \longleftrightarrow M' : \tau$ for some M'. Equality is decidable on normalizing terms.

Lemma 33 (Decidability for normalizing terms)

1. *If* $\Lambda; \Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi$ *and* $\Lambda; \Omega \vdash \tau \longleftrightarrow \tau' : \Phi$ *then it is decidable whether* $\Lambda; \Omega \vdash \sigma \longleftrightarrow \tau : \Phi$.

2. *If* $\Lambda; \Omega \vdash M \Longleftrightarrow M' : \tau$ *and* $\Lambda; \Omega \vdash N \Longleftrightarrow N' : \tau$ *then it is decidable whether* $\Lambda; \Omega \vdash M \Longleftrightarrow N : \tau$.

3. *If* $\Lambda; \Omega \vdash M \longleftrightarrow M' : \tau_1$ *and* $\Lambda; \Omega \vdash N \longleftrightarrow N' : \tau_2$ *then it is decidable whether* $\Lambda; \Omega \vdash M \longleftrightarrow N : \tau_3$ *for some* τ_3.

4. *If* $\Lambda; \Omega \vdash A \Longleftrightarrow A' : \kappa$ *and* $\Lambda; \Omega \vdash B \Longleftrightarrow B' : \kappa$ *then it is decidable whether* $\Lambda; \Omega \vdash A \Longleftrightarrow B : \kappa$.

5. *If* $\Lambda; \Omega \vdash A \longleftrightarrow A' : \kappa_1$ *and* $\Lambda; \Omega \vdash B \longleftrightarrow B' : \kappa_2$ *then it is decidable whether* $\Lambda; \Omega \vdash A \longleftrightarrow B : \kappa_3$ *for some* κ_3.

6. *If* $\Lambda; \Omega \vdash K \Longleftrightarrow K' : \mathsf{kind}^-$ *and* $\Lambda; \Omega \vdash L \Longleftrightarrow L' : \mathsf{kind}^-$ *then it is decidable whether* $\Lambda; \Omega \vdash K \Longleftrightarrow L : \mathsf{kind}^-$.

Proof: By structural induction on the derivations □

Theorem 34 (Decidability of algorithmic equality)

1. *If* $\Delta; \Gamma \vdash M : A$ *and* $\Delta; \Gamma \vdash N : A$ *then it is decidable whether* $\Delta^-; \Gamma^- \vdash M \Longleftrightarrow N : A^-$.

2. *If* $\Delta; \Gamma \vdash A : K$ *and* $\Delta; \Gamma \vdash B : K$ *then it is decidable whether* $\Delta^-; \Gamma^- \vdash A \Longleftrightarrow B : K^-$.

3. *If* $\Delta;\Gamma \vdash K : \mathsf{kind}$ *and* $\Delta;\Gamma \vdash L : \mathsf{kind}$ *then it is decidable whether* $\Delta^-;\Gamma^- \vdash K \Longleftrightarrow L : \mathsf{kind}^-$.

Proof: By reflexivity of definitional equality and completeness of algorithmic equality, both M and N are normalizing. Therefore, by the previous lemma, the algorithmic equivalence is decidable. □

Theorem 35 (Decidability of definitional equality)

1. *If* $\Delta;\Gamma \vdash M : A$ *and* $\Delta;\Gamma \vdash N : A$ *then it is decidable whether* $\Delta;\Gamma \vdash M \equiv N : A$.

2. *If* $\Delta;\Gamma \vdash A : K$ *and* $\Delta;\Gamma \vdash B : K$ *then it is decidable whether* $\Delta;\Gamma \vdash A \equiv B : K$.

3. *If* $\Delta;\Gamma \vdash K : \mathsf{kind}$ *and* $\Delta;\Gamma \vdash L : \mathsf{kind}$ *then it is decidable whether* $\Delta;\Gamma \vdash K \equiv L : \mathsf{kind}$.

Proof: By soundness and completeness it suffices to check algorithmic equality which is decidable by the previous lemma. □

2.11 Decidability of type-checking

Next, we give a type-checking algorithm which uses algorithmic equality. It is bi-directional and uses the following two judgments:

$\Delta;\Gamma \vdash \sigma \Leftarrow \Psi$	Check σ against Ψ
$\Delta;\Gamma \vdash M \Rightarrow A$	Synthesize a type A for M

While we can synthesize a type A for an object M, we need to check the substitution σ against a context Ψ. We assume that the modal context Δ and the context Γ and Ψ are valid.

Substitutions

$$\frac{}{\Delta;\Gamma \vdash \cdot \Leftarrow \cdot} \qquad \frac{\Delta;\Gamma \vdash \sigma \Leftarrow \Psi \quad \Delta;\Gamma \vdash M \Rightarrow A' \quad \Delta;\Gamma \vdash A' \Longleftrightarrow [\sigma]A}{\Delta;\Gamma \vdash (\sigma, M/x) \Leftarrow (\Psi, x{:}A)}$$

Note that we synthesize a type A' for M, and then check separately that A' is definitional equal to $[\sigma]A$.

Objects

$$\frac{x{:}A \text{ in } \Gamma}{\Delta;\Gamma \vdash x \Rightarrow A} \qquad \frac{c{:}A \text{ in } \Sigma}{\Delta;\Gamma \vdash c \Rightarrow A} \qquad \frac{\Delta, u{::}(\Psi{\vdash}A), \Delta';\Gamma \vdash \sigma \Leftarrow \Psi}{\Delta; u{::}(\Psi{\vdash}A), \Delta';\Gamma \vdash u[\sigma] \Rightarrow [\sigma]A}$$

$$\frac{\Delta;\Gamma \vdash A_1 \Rightarrow \mathsf{type} \quad \Delta;\Gamma, x{:}A_1 \vdash M \Rightarrow A_2}{\Delta;\Gamma \vdash \lambda x{:}A_1.\, M \Rightarrow \Pi x{:}A_1.\, A_2}$$

$$\frac{\Delta;\Gamma \vdash M_1 \Rightarrow \Pi x{:}A_2.\, A_1 \quad \Delta;\Gamma \vdash M_2 \Rightarrow A_2' \quad \Delta;\Gamma \vdash A_2' \Longleftrightarrow A_2 : \mathsf{type}}{\Delta;\Gamma \vdash M_1\, M_2 \Rightarrow [\mathsf{id}_\Gamma, M_2/x]A_1}$$

Families

$$\frac{a \Rightarrow K \text{ in signature}}{\Delta;\Gamma \vdash a \Rightarrow K} \qquad \frac{\Delta;\Gamma \vdash A_1 \Rightarrow \mathsf{type} \quad \Delta;\Gamma, x{:}A_1 \vdash A_2 \Rightarrow \mathsf{type}}{\Delta;\Gamma \vdash \Pi x{:}A_1.\, A_2 \Rightarrow \mathsf{type}}$$

$$\frac{\Delta;\Gamma \vdash A \Rightarrow \Pi x{:}B'.\, K \quad \Delta;\Gamma \vdash M \Rightarrow B \quad \Delta;\Gamma \vdash B \Longleftrightarrow B' : \mathsf{type}}{\Delta;\Gamma \vdash A\, M \Rightarrow [\mathsf{id}_\Gamma, M/x]K}$$

Kinds

$$\frac{}{\Delta;\Gamma \vdash \mathsf{type} \Rightarrow \mathsf{kind}} \qquad \frac{\Delta;\Gamma, x{:}A \vdash K \Rightarrow \mathsf{kind} \quad \Delta;\Gamma \vdash A \Rightarrow \mathsf{type}}{\Delta;\Gamma \vdash \Pi x{:}A.\, K \Rightarrow \mathsf{kind}}$$

Similar rules exist for checking validity of context and signatures.

Theorem 36 (Correctness of algorithmic type-checking)

1. *If* $\Delta;\Gamma \vdash \sigma \Leftarrow \Psi$ *then* $\Delta;\Gamma \vdash \sigma : \Psi$.
2. *If* $\Delta;\Gamma \vdash M \Rightarrow A$ *then* $\Delta;\Gamma \vdash M : A$.
3. *If* $\Delta;\Gamma \vdash \sigma : \Psi$ *then* $\Delta;\Gamma \vdash \sigma \Leftarrow \Psi$.
4. *If* $\Delta;\Gamma \vdash M : A$ *then* $\Delta;\Gamma \vdash M \Rightarrow A'$ *for some* A' *such that* $\Delta;\Gamma \vdash A \equiv A' : \mathsf{type}$.

Proof: Part (1) and (2) follows by simultanous induction on the first derivation using validity, soundness of algorithmic equality and the rule of type conversion. Part (3) and (4) follows by induction on the first derivation using transitivity of equality, inversion on type equality and completeness of algorithmic equality. □

Since the algorithmic typing rules are syntax-directed and algorithmic equality is decidable, there either exists a unique A' s.t. $\Delta; \Gamma \vdash M \Rightarrow A'$ or there is no such A'. By correctness of algorithmic type-checking, we have $\Delta; \Gamma \vdash M : A$ iff $\Delta; \Gamma \vdash A' \equiv A : \mathsf{type}$ which can be decided by checking $\Delta^-; \Gamma^- \vdash A' \Longleftrightarrow A : \mathsf{type}$.

Theorem 37 (Decidability of type-checking)

1. *It is decidable if Δ is valid.*
2. *Given a valid Δ, it is decidable if Γ is valid.*
3. *Given a valid Δ, Γ, M and A, it is decidable whether $\Delta; \Gamma \vdash M : A$.*
4. *Given a valid Δ, Γ, A and K, it is decidable whether $\Delta; \Gamma \vdash A : K$.*
5. *Given a valid Δ, Γ, and K, it is decidable whether $\Delta; \Gamma \vdash K : \mathsf{kind}$.*

Proof: We note that the algorithmic typing rules are syntax-directed and algorithmic equality is decidable (see theorem 34). Hence, there either exists a unique A' s.t. $\Delta; \Gamma \vdash M \Rightarrow A'$ or there is no such A'. By correctness of algorithmic type-checking, we then have $\Delta; \Gamma \vdash M : A$ iff $\Delta; \Gamma A' \equiv A : \mathsf{type}$, which can be decided by checking $\Delta^-; \Gamma^- \vdash A' \Longleftrightarrow A : \mathsf{type}$. □

The correctness of algorithmic type-checking allows us to show strengthening.

Lemma 38 (Strengthening)

1. *If $\Delta; \Gamma, x{:}A, \Gamma' \vdash J$ and x does not occur in the free ordinary variables of J and does not occur in the free ordinary variables of Γ', then $\Delta; \Gamma, \Gamma' \vdash J$.*
2. *If $\Delta, u{::}(\Psi{\vdash}A), \Delta'; \Gamma \vdash J$ and u does not occur in the free modal variables of J and u does not occur in the free modal variables of Δ',*
   then $\Delta, \Delta'; \Gamma \vdash J$.

Proof: Strengthening for the algorithmic version of type-checking follows by structural induction over the structure of the given derivation, taking advantage of the obvious strengthening for algorithmic equality. Strengthening for the original typing rules then follows by soundness and completeness of algorithmic typing. Strengtheing for equality judgments follows from completeness (theorem 29, soundness (theorem 31), and strengthening for the typing judgments. □

2.12 Canonical forms

In this section, we discuss canonical forms where canonical forms are η-long and β-normal forms. We will start by instrumenting the algorithmic equality judgements to produce the mediating object for two equal objects. These mediating objects will be unique. Since algorithmic equality uses simplified types, no typing information is available on the bound variables in lambda-abstractions. Fortunately, we can reconstruct the typing information and the type labels are uniquely determined.

We formalize this idea using quasi-canonical and quasi-atomic forms, in which type labels have been deleted.

$$\begin{array}{rrcl} \text{Quasi-canonical Objects} & U & ::= & \lambda x.\, U \mid R \\ \text{Quasi-atomic Objects} & R & ::= & c \mid x \mid u[\eta] \mid R\,U \\ \text{Quasi-canonical Substitutions} & \eta & ::= & \cdot \mid \eta, U/x \end{array}$$

Now, we instrument the algorithmic equality relations to extract a common quasi-canonical or quasi-atomic form for the terms being compared.

Instrumented type-directed object equality

$$\frac{\Lambda;\Omega \vdash M \xrightarrow{whr} M' \quad \Lambda;\Omega \vdash M' \Longleftrightarrow N : \alpha \Uparrow R}{\Lambda;\Omega \vdash M \Longleftrightarrow N : \alpha \Uparrow R}$$

$$\frac{\Lambda;\Omega \vdash N \xrightarrow{whr} N' \quad \Lambda;\Omega \vdash M \Longleftrightarrow N' : \alpha \Uparrow R}{\Lambda;\Omega \vdash M \Longleftrightarrow N : \alpha \Uparrow R}$$

$$\frac{\Lambda;\Omega \vdash M \longleftrightarrow N : \alpha \downarrow R}{\Lambda;\Omega \vdash M \Longleftrightarrow N : \alpha \Uparrow R} \qquad \frac{\Lambda;\Omega, x{:}\tau_1 \vdash M\,x \Longleftrightarrow N\,x : \tau_2 \Uparrow U}{\Lambda;\Omega \vdash M \Longleftrightarrow N : \tau_1 \to \tau_2 \Uparrow \lambda x.U}$$

Instrumented structural object equality

$$\frac{x{:}\tau \text{ in } \Omega}{\Lambda;\Omega \vdash x \longleftrightarrow x : \tau \downarrow x} \qquad \frac{c{:}\tau \text{ in signature}\Sigma}{\Lambda;\Omega \vdash c \longleftrightarrow c : \tau \downarrow c}$$

$$\frac{u{::}(\Phi \vdash \tau) \text{ in } \Lambda \quad \Lambda;\Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi \Uparrow \eta}{\Lambda;\Omega \vdash u[\sigma] \longleftrightarrow u[\sigma'] : \tau \downarrow u[\eta]}$$

$$\frac{\Lambda;\Omega \vdash M_1 \longleftrightarrow N_1 : \tau_2 \to \tau_1 \downarrow R \quad \Lambda;\Omega \vdash M_2 \Longleftrightarrow N_2 : \tau_2 \Uparrow U}{\Lambda;\Omega \vdash M_1\, M_2 \longleftrightarrow N_1\, N_2 : \tau_1 \downarrow R\, U}$$

Instrumented substitution equality

$$\frac{}{\Lambda;\Omega \vdash \cdot \longleftrightarrow \cdot : \cdot \Uparrow \cdot} \qquad \frac{\Lambda;\Omega \vdash \sigma \longleftrightarrow \sigma' : \Phi \Uparrow \eta \quad \Lambda;\Omega \vdash M \Longleftrightarrow N : \tau \Uparrow U}{\Lambda;\Omega \vdash (\sigma, M/x) \longleftrightarrow (\sigma', N/x) : (\Phi, x{:}\tau) \Uparrow (\eta, U/x)}$$

We say that a term M has a quasi-canonical form U if $\Lambda;\Omega \vdash M \Longleftrightarrow M : \tau \Uparrow U$ for appropriate Λ, Ω and τ, and similarly for quasi-canonical forms. From the previous development, it follows that every well-formed term has a unique quasi-canonical form.

Theorem 39 (Quasi-canonical and quasi-atomic forms)

1. *If* $\Delta;\Gamma \vdash M_1 : A$ *and* $\Delta;\Gamma \vdash M_2 : A$ *and* $\Delta^-;\Gamma^- \vdash M_1 \Longleftrightarrow M_2 : A^- \Uparrow U$ *then there is an* N *such that* $|N| = U$, $\Delta;\Gamma \vdash N : A$, $\Delta;\Gamma \vdash M_1 \equiv N : A$ *and* $\Delta;\Gamma \vdash M_2 \equiv N : A$.

2. *If* $\Delta;\Gamma \vdash M_1 : A$ *and* $\Delta;\Gamma \vdash M_2 : A$ *and* $\Delta^-;\Gamma^- \vdash M_1 \longleftrightarrow M_2 : A^- \downarrow R$ *then* $\Delta;\Gamma \vdash A_1 \equiv A_2 : \mathsf{type}$, $A^- = B^- = \tau$ *and there exists a* N *such that* $|N| = R$, $\Delta;\Gamma \vdash N : A$, $\Delta;\Gamma \vdash M_1 \equiv N : A$ *and* $\Delta;\Gamma \vdash M_2 \equiv N : A$

3. *If* $\Delta;\Gamma \vdash \sigma_1 : \Psi$ *and* $\Delta;\Gamma \vdash \sigma_2 : \Psi$ *and* $\Delta^-;\Gamma^- \vdash \sigma_1 \longleftrightarrow \sigma_2 : \Psi^- \Uparrow \eta$ *then there exists a* τ *such that* $|\tau| = \eta$, $\Delta;\Gamma \vdash \tau : \Psi$, $\Delta;\Gamma \vdash \sigma_1 \equiv \tau : \Psi$ *and* $\Delta;\Gamma \vdash \sigma_2 \equiv \tau : \Psi$

Proof: By simultaneous induction on the instrumented equality derivations, using validity of Γ, functionality lemma 9, type conversion and symmetry for the substitution case (part 3). $\square$

In the implementation, we use a stronger normal form where existential variables (represented here by modal variables) must also be of atomic type. This is accomplished by a technique called *lowering*. Lowering replaces a variable $u{::}(\Psi{\vdash}\Pi x{:}A_1.A_2)$ by a new variable $u'{:}(\Psi, x{:}A_1{\vdash}A_2)$. This process is repeated until all existential variables have a type of the form $\Psi \vdash b\, N_1 \ldots N_k$. This operation has been proved correct for the simply-typed case by Dowek et al. [22], but remains somewhat mysterious. Here, it is justified by the modal substitution principle.

Lemma 40

1. *(Lowering) If* $\Delta, u{::}(\Psi{\vdash}\Pi x{:}A_1.\ A_2), \Delta'; \Gamma \vdash M : A$
then $\Delta, u'{::}(\Psi, x{:}A_1{\vdash}A_2), \Delta'^{\#}; \Gamma^{\#} \vdash M^{\#} : A^{\#}$
where $(P)^{\#} = [\![(\lambda x{:}A_1.u'[\mathsf{id}_\Psi, x/x])/u]\!]P$.

2. *(Raising) If* $\Delta, u'{::}(\Psi, x{:}A_1{\vdash}A_2), \Delta'; \Gamma \vdash M : A$
then $\Delta, u{::}(\Psi{\vdash}\Pi x{:}A_1.\ A_2), \Delta'^{+}; \Gamma^{+} \vdash M^{+} : A^{+}$
where $(P)^{+} = [\![(u[\mathsf{id}_\Psi]\, x)/u']\!]P$.

3. $()^{+}$ *and* $()^{\#}$ *are inverse substitutions (modulo definitional equality).*

Proof: Direct, by weakening and the modal substitution principle. For part (1) we observe that $\Delta, u'{::}(\Psi, x{:}A_1{\vdash}A_2); \Psi \vdash \lambda x{:}A_1.u'[\mathsf{id}_\Psi, x/x] : \Pi x{:}A_1.\ A_2$.
For part (2) we use instead that $\Delta, u{::}(\Psi{\vdash}\Pi x{:}A_1.\ A_2); \Psi, x{:}A_1 \vdash u[\mathsf{id}_\Psi]\, x : A_2$. Part (3) is direct by calculation. □

Since we can lower all modal variables, we can change the syntax of canonical forms so that terms $u[\eta]$ are also canonical objects of base type, rather than atomic objects. This is, in fact, what we chose in the implementation.

For deciding definitional equality between canonical objects, we can use syntactic equality modulo renaming of bound variables, as usual. For syntactic equality between two canonical objects U_1 and U_2, we write $U_1 = U_2$. The algorithm is similar to the structural equality for objects given earlier, except we are now allowed to descend into the body of λ-abstractions and compare the bodies directly. Since we know, the objects are in canonical form, we eliminate the weak head reduction rules and the eta-rule. In other words, type-directed equality can be replaced by syntactic equality for λ-abstractions.

Finally, we would like to point out that for canonical forms we can refine the bi-directional type checking rules given earlier on page 56 in such a way that we check quasi-canonical (normal) terms agains a type A while we infer a type A for quasi-atomic (neutral) terms.

2.13 Implementation of existential variables

In the implementation the modal variables in Δ are used to represent existential variables (also known as meta-variables), while the variables in Γ are universal variables (also known as parameters).

Existential variables are created in an ambient context Ψ and then lowered. We do not explicitly maintain a context Δ of these existential variables, but it is important that a proper order for them exists. Existential variables are created with a mutable reference, which is updated with an assignment when we need to carry out a substitution $[\![M/u]\!]$.

In certain operations, and particularly after type reconstruction, we need to abstract over the existential variables in a term. Since the LF type theory provides no means to quantify over $u{::}(\Psi {\vdash} A)$ we raise such variables until they have the form $u'{::}(\cdot {\vdash} A')$. It turns out that in the context of type reconstruction we can now quantify over them as ordinary variables $x'{:}A'$. However, this is not satisfactory as it requires first raising the type of existential variables for abstraction, and later again lowering the type of existential variables during unification to undo the effect of raising. To efficiently treat existential variables, we would like to directly quantify over modal variables u.

The judgmental reconstruction in terms of modal logic suggests two ways to incorporate modal variables. One way is via a new quantifier $\Pi^{\Box} u{::}(\Psi {\vdash} A_1).\ A_2$, the other is via a general modal operator $\Box_{\Psi}$. Proof-theoretically, the former is slightly simpler, so we will pursue this here. The new operator then has the form $\Pi^{\Box} u{::}(\Psi {\vdash} A_1).\ A_2$ and is defined by the following rules.

$$\frac{\Delta; \Psi \vdash A : \mathsf{type} \quad \Delta, u{::}(\Psi {\vdash} A); \Gamma \vdash B : \mathsf{type}}{\Delta; \Gamma \vdash \Pi^{\Box} u{::}(\Psi {\vdash} A).\ B : \mathsf{type}}$$

$$\frac{\Delta, u{::}(\Psi {\vdash} A); \Gamma \vdash M : B}{\Delta; \Gamma \vdash \lambda^{\Box} u.\ M : \Pi^{\Box} u{::}(\Psi {\vdash} A).\ B} \qquad \frac{\Delta; \Gamma \vdash N : \Pi^{\Box} u{::}(\Psi {\vdash} A).\ B \quad \Delta; \Psi \vdash M : A}{\Delta; \Gamma \vdash N \mathbin{\scriptstyle\Box} M : [\![M/u]\!]B}$$

The main complication of this extension is that variables u can now be bound and substitution must be capture avoiding. In the present implementation, this is handled by de Bruijn indices. In this thesis, we do not develop a theory where lambda-box and pi-box are first class abstractions.

2.14 Conclusion and related work

In this chapter, we presented an abstract view of existential variables based on modal type theory and showed that type-checking remains decidable and canonical forms exist. Central to the development is the clear distinction between existential variables declared in the modal context Δ and bound variables declared in the context Γ. Unlike calculi of explicit substitutions [1, 22], our system does not require de Bruijn indices nor does it require closure $M[\sigma]$ as first-class terms. Although the use of de Bruijn indices in calculi of explicit substitutions leads to a simple formal system, the readability may be obstructed and critical principles are obfuscated by the technical notation. In addition, some techniques like pre-cooking of terms [22] and optimizations such as lowering and grafting remain ad hoc. This makes it more difficult to transfer these optimizations to other calculi.

Viewing existential variables as modal variables leads to a simple clean framework which allows us to explain techniques such as lowering and raising logically by substitution principles. Surprisingly, the presented framework justifies with hindsight logically many techniques and decisions in the Twelf implementation [53].

Chapter 3

Toward efficient higher-order pattern unification

Unification lies at the heart of automated reasoning systems, logic programming and rewrite systems. Thus its performance affects in a crucial way the global efficiency of each of these applications. This need for efficient unification algorithms has led to many investigations in the first-order setting. However, the efficient implementation of higher-order unification, especially for dependently typed λ-calculus, is still a central open problem limiting the potential impact of higher-order reasoning systems such as Twelf [53], Isabelle [45], or λProlog [39].

The most comprehensive study on efficient and robust implementation techniques for higher-order unification so far has been carried out by Nadathur and colleagues for the simply-typed λ-calculus in the programming language λProlog [37, 38]. The Teyjus compiler [40] embodies many of the insights found, in particular an adequate representation of lambda terms and mechanisms to delay and compose substitutions. Higher-order unification is implemented via Huet's algorithm [31] and special mechanisms are incorporated into the WAM instruction set to support branching and postponing unification problems. To only perform an occurs-check when necessary, the compiler distinguishes between the first occurrence and subsequent occurrences of a variable and compiles them into different WAM instructions. While for the first occurrence of a variable the occurs-check may be omitted, full unification is used for all subsequent variables. This approach seems to work well in the simply-typed setting, however it is not clear how to generalize it to dependent types.

In this chapter, we discuss the efficient implementation of higher-order pattern unification for the dependently typed lambda-calculus. Unlike Huet's general higher-order unification algorithm which involves branching and backtracking, higher-order pattern unification [34, 48] is deterministic and decidable. An important step toward the efficient implementation of higher-order pattern unification was the development based on explicit substitutions and de Bruijn indices [22] for the simply-typed lambda-calculus. This allows a clear distinction between bound and existential variables and reduces the problem to essentially first-order unification. Although the use of de Bruijn indices leads to a simple formal system, the readability may be obstructed and critical principles are obfuscated by the technical notation. In addition, as we have argued in Chapter 2, some techniques like pre-cooking of terms and optimizations such as lowering and grafting remain ad hoc.

In this chapter, we will particularly focus on one optimization called linearization, which eliminates many unnecessary occurs-checks and delays any computationally expensive higher-order parts. Based on the modal dependently typed lambda calculus, we present a higher-order unification algorithm for linear patterns. We will assume that all terms are in canonical form and all existential variables must be of atomic type. As pointed out previously, this can be achieved by lowering and raising (see Chapter 2). We have implemented this optimization of higher-order unification as part of the Twelf system. Experimental results which we discuss at the end of this chapter demonstrate significant performance improvement, including those in the area of proof-carrying code.

3.1 Higher-order pattern unification

In the following, we will consider the pattern fragment of the modal lambda-calculus. Higher-order patterns are terms where existential variables must be applied to distinct bound variables. This fragment was first identified by Miller [34] for the simply-typed lambda-calculus, and later extended by Pfenning [48] to the dependently typed and polymorphic case. We enforce that all terms are in canonical form, and the type of existential variables has been lowered and is atomic. We call a normal term U an *atomic pattern*, if all the subterms of the form $u[\sigma]$ are such that $\sigma = y_1/x_1, \ldots y_k/x_k$ where $y_1, \ldots, y_k$ are distinct bound variables. This is already implicitly assumed for $x_1, \ldots, x_k$ because all variables defined by a substitution must be distinct. Such a substitution is called a *pattern substitution*. Moreover the type of any occurrence of $u[\sigma]$ is atomic, i.e., of the form $a\ M_1\ \ldots M_n$. We will write Q for atomic types. As a consequence, any instantiation of u must be a neutral object R, but can never be a lambda-abstraction. We write θ for simultaneous substitutions $[\![R_1/u_1, \ldots R_n/u_n]\!]$ for existential variables. As a consequence, canonical forms are preserved even when applying the modal substitution to instantiate modal variables.

We emphasize that throughout this chapter we only deal with canonical terms denoted by U for normal and R for neutral objects. Moreover, we only consider existential variables of atomic type and consider them as neutral objects. If the distinction is not important we may still use M and N to denote objects. Similarly, we will continue to denote the declarations of modal variables in a modal context Δ as $u{::}\Psi{\vdash}Q$, where $Q = a\ M_1\ \ldots M_n$. Where the distinction is not important, we may still use A to denote arbitrary types.

Before we describe higher-order pattern unification, we give some details on modal substitutions. Recall the definition of modal substitutions from Chapter 2.

$\Delta \;\vdash\; \theta : \Delta'$ Modal substitution θ matches context Δ'

Substitutions

$$\begin{array}{rcl}
[\![\theta]\!](\cdot) & = & \cdot \\
[\![\theta]\!](\sigma, U/y) & = & ([\![\theta]\!]\sigma, [\![\theta]\!]U/y)
\end{array}$$

Objects

$$\begin{array}{rcl}
[\![\theta]\!]c & = & c \\
[\![\theta]\!]x & = & x \\
[\![\theta_1, U/u, \theta_2]\!](u[\sigma]) & = & [[\![\theta_1, U/u, \theta_2]\!]\sigma]U \\
[\![\theta]\!](R\,U) & = & ([\![\theta]\!]R)\,([\![\theta]\!]U) \\
[\![\theta]\!](\lambda y{:}A.\ U) & = & \lambda y{:}[\![\theta]\!]A.\ [\![\theta]\!]U
\end{array}$$

Context

$$\begin{array}{rcl}
[\![\theta]\!]\cdot & = & \cdot \\
[\![\theta]\!](\Gamma, x{:}A) & = & [\![\theta]\!]\Gamma, x{:}[\![\theta]\!]A
\end{array}$$

Typing rules for modal substitutions are given next:

Modal substitutions

$$\frac{}{\Delta \vdash (\cdot) : (\cdot)} \qquad \frac{\Delta \vdash \theta : \Delta' \quad \Delta; [\![\theta]\!]\Psi \vdash U : [\![\theta]\!]A}{\Delta \vdash (\theta, U/u) : (\Delta', u{::}\Psi{\vdash}A)}$$

We write id_Δ for the identity modal substitution $[\![u_1[\mathsf{id}_{\Psi_1}]/u_1, \dots, u_n[\mathsf{id}_{\Psi_n}]/u_n]\!]$ for a modal context $\Delta = (\cdot, u_1{::}(\Psi_1{\vdash}A_1), \dots, u_n{::}(\Psi_n{\vdash}A_n))$. By definition, we then have $\Delta \vdash \mathsf{id}_\Delta : \Delta$. In an implementation, where existential variables are realized via pointers, this essentially means the store of pointers for existential variables does not change. Next, we show some properties of modal substitutions.

Lemma 41 (Modal substitution properties)
If $\Delta' \vdash \theta : \Delta$ and $\Delta; \Gamma \vdash J$ then $\Delta'; [\![\theta]\!]\Gamma \vdash [\![\theta]\!]J$.

Proof: By simultaneous structural induction on the second derivation. □

Similar to composition of ordinary substitution, we prove some composition for modal substitutions.

Lemma 42 (Composition of modal substitutions)

1. $[\![\theta_2]\!]([\![\theta_1]\!][\sigma]) = [\![[\![\theta_2]\!]\theta_1]\!][\sigma]$
2. $[\![\theta_2]\!]([\![\theta_1]\!]U) = [\![[\![\theta_2]\!]\theta_1]\!]U$
3. $[\![\theta_2]\!]([\![\theta_1]\!]R) = [\![[\![\theta_2]\!]\theta_1]\!]R$
4. $[\![\theta_2]\!]([\![\theta_1]\!]A) = [\![[\![\theta_2]\!]\theta_1]\!]A$
5. $[\![\theta_2]\!]([\![\theta_1]\!]K) = [\![[\![\theta_2]\!]\theta_1]\!]K$
6. $[\![\theta_2]\!]([\![\theta_1]\!]\Gamma) = [\![[\![\theta_2]\!]\theta_1]\!]\Gamma$

Proof: By simultaneous induction on the definition of modal substitutions. □

In addition, we show the following property relating modal and ordinary substitutions.

Lemma 43 *Assume* $\Delta; \Gamma \vdash \sigma : \Psi$ *and* $\Delta' \vdash \theta : \Delta$.

1. $[\![\theta]\!]([\sigma]\tau) = [[\![\theta]\!]\sigma]([\![\theta]\!]\tau)$
2. $[\![\theta]\!]([\sigma]U) = [[\![\theta]\!]\sigma]([\![\theta]\!]U)$
3. $[\![\theta]\!]([\sigma]R) = [[\![\theta]\!]\sigma]([\![\theta]\!]R)$
4. $[\![\theta]\!]([\sigma]A) = [[\![\theta]\!]\sigma]([\![\theta]\!]A)$
5. $[\![\theta]\!]([\sigma]K) = [[\![\theta]\!]\sigma]([\![\theta]\!]K)$

Proof: By simultaneous induction on the definition of ordinary substitutions. □

Finally, we note that applying a modal substitution θ to a pattern substitution σ does not change σ itself, since the range of σ refers only to bound variables, while θ refers to modal variables.

Lemma 44
If $\Delta' \vdash \theta : \Delta$ *and* σ *is a pattern substitution, s.t.* $\Delta; \Gamma \vdash \sigma : \Psi$ *then* $[\![\theta]\!](\sigma) = \sigma$

Proof: Induction on the structure of σ □

Now, we give the judgments for describing higher-order pattern unification.

$\Delta; \Gamma \vdash U_1 \doteq U_2 \;/\; (\Delta', \theta)$	Unification of normal atomic patterns
$\Delta; \Gamma \Vdash R_1 \doteq R_2 \;/\; (\Delta', \theta)$	Unification of neutral atomic objects

We assume U_1 (resp. R_1) and U_2 (resp. R_2) are normal atomic patterns, are well-typed in the context Γ and have the same type A. Since their type does not play a role during unification, we omit it here. θ is a modal substitution for the modal variables in Δ s.t.

$$\Delta' \vdash \theta : \Delta \quad \text{and} \quad \Delta'; [\![\theta]\!]\Gamma \vdash [\![\theta]\!]U_1 \equiv [\![\theta]\!]U_2 : [\![\theta]\!]A$$

Since we require that all existential variables have atomic type the modal context Δ has the following form:

$$\Delta = u_1{::}\Psi_1{\vdash}Q_1, \ldots, u_n{::}\Psi_n{\vdash}Q_n$$

Moreover, we will never instantiate an existential variable with a lambda-abstraction. Hence, the modal substitution θ has the following form:

$$\theta = (R_1/u_1, \ldots, R_n/u_n)$$

where R denotes a neutral atomic object which can be a bound variable, a constant, an application, or another existential variable which has atomic type. Moreover, since we require that U_1 (resp. R_1) and U_2 (resp. R_2) are in canonical form, and all existential variables are of atomic type, canonical forms are preserved when applying the modal substitution θ and in fact we can check if $[\![\theta]\!]U_1$ is definitionally equal to $[\![\theta]\!]U_2$ by checking syntactic equality between these two objects. Similar statments hold for types, kinds, and the bound variable context Γ.

Theorem 45

1. *If U is a normal term and all modal variables are of atomic type and $\Delta' \vdash \theta : \Delta$ and $\Delta; \Gamma \vdash U : A$ then $[\![\theta]\!]U$ is a normal term.*

2. *If R is a neutral term and all modal variables are of atomic type and $\Delta' \vdash \theta : \Delta$ and $\Delta; \Gamma \vdash R : A$ then $[\![\theta]\!]R$ is neutral.*

Proof: Simultaneous structural induction on the structure of normal and neutral terms. The key observation is that θ only substitutes neutral objects R for modal variables. $\Box$

Theorem 46 *Let U and U' be normal terms and R and R' be neutral terms and all modal variables are of atomic type and $\Delta \vdash \theta : \Delta'$.*

1. *$[\![\theta]\!]U = [\![\theta']\!]U'$ iff $\Delta; [\![\theta]\!]\Gamma \vdash [\![\theta]\!]U \equiv [\![\theta]\!]U' : [\![\theta]\!]A$.*

2. *$[\![\theta]\!]R = [\![\theta']\!]R'$ iff $\Delta; [\![\theta]\!]\Gamma \vdash [\![\theta]\!]R \equiv [\![\theta]\!]R' : [\![\theta]\!]A$.*

Proof: By the previous theorem, we know $[\![\theta]\!]U$ and $[\![\theta]\!]U'$ are canonical. As discussed in Chapter 2 on page 61, definitionally equality reduces to syntactic equality for canonical forms. $\Box$

Higher-order pattern unification can be done in two phases (see [48, 22] for another account). During the first phase, we decompose the terms until one of the two terms we unify is an existential variable $u[\sigma]$. This decomposition phase is straightforward and resembles first-order unification closely.

$$\frac{\Delta; \Gamma, x{:}A \vdash U_1 \doteq U_2 \;/\; (\Delta', \theta)}{\Delta; \Gamma \vdash \lambda x{:}A.U_1 \doteq \lambda x{:}A.U_2 \;/\; (\Delta', \theta)}\, lam \qquad \frac{\Delta; \Gamma \Vdash R_1 \doteq R_2 \;/\; (\Delta', \theta)}{\Delta; \Gamma \vdash R_1 \doteq R_2 \;/\; (\Delta', \theta)}\, coerce$$

$$\frac{}{\Delta; \Gamma \Vdash x \doteq x \;/\; (\Delta, \mathsf{id}_\Delta)}\, var \qquad \frac{}{\Delta; \Gamma \Vdash c \doteq c \;/\; (\Delta, \mathsf{id}_\Delta)}\, const$$

$$\frac{\Delta; \Gamma \Vdash R_1 \doteq R_2 \;/\; (\Delta_1, \theta_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \vdash [\![\theta_1]\!]U_1 \doteq [\![\theta_1]\!]U_2 \;/\; (\Delta_2, \theta_2)}{\Delta; \Gamma \Vdash R_1\, U_1 \doteq R_2\, U_2 \;/\; (\Delta_2, [\![\theta_1]\!](\theta_2))}\, app$$

Note that we do not need to worry about capture in the rule *lam*, since existential variables and bound variables are defined in different contexts. During the second phase, we need to find an actual instantiation for the existential variable u. There are two main cases to distinguish: (1) when we unify two existential variables, $u[\sigma] \doteq v[\sigma']$, and (2) when we unify an existential variable with another kind of term, $u[\sigma] \doteq U$. In the following, we consider each of these two cases separately.

3.1.1 Unifying an existential variable with another term

The case for unifying an existential variable $u[\sigma]$ with another term U can be transformed into $u \doteq [\sigma]^{-1} U$ assuming u does not occur in U and all variables $v[\tau]$ are *pruned* so that the free variables in τ all occur in the image of σ (see below or [34, 22] for details). Before considering the pruning operation, we define the inverse substitution. Note that we view $[\sigma]^{-1} U$ as a new meta-level operation such as substitution, because it may be defined even if σ is not invertible in full generality. This meta-level operation is defined as follows[1].

[1]We do not omit the type label on the lambda-abstraction here, since it is sometimes convenient to have access to the type of the bound variable x. This is not strictly necessary, since we require all terms are canonical and type labels can be omitted.

$$
\begin{array}{rcll}
[\sigma]^{-1}\, c & = & c & \\
[\sigma]^{-1}\, x & = & y & \text{if } x/y \in \sigma \text{, undefined otherwise} \\
[\sigma]^{-1}\, (v[\tau]) & = & v[[\sigma]^{-1}\, \tau] & \\
[\sigma]^{-1}\, (R\, U) & = & ([\sigma]^{-1}\, R)\, ([\sigma]^{-1}\, U) & \\
[\sigma]^{-1}\, (\lambda x{:}A.\, U) & = & \lambda x{:}([\sigma]^{-1}\, A).\, [\sigma, x/x]^{-1}\, U & \text{if } x \text{ not declared or free in } \sigma \\
 & & & \\
[\sigma]^{-1}\, (\cdot) & = & (\cdot) & \\
[\sigma]^{-1}\, (\tau, U/x) & = & ([\sigma]^{-1}\, \tau, [\sigma]^{-1}\, U/x) &
\end{array}
$$

Note that the defined meta-level operation of inverting substitutions is only defined for a variable x if x/y is in σ and undefined otherwise, thus enforcing that the resulting term $[\sigma]^{-1}\, U$ is total and exists. It is important to note that the meta-level operation of inverting substitutions and applying the modal substitution commute. This property only holds if the inverse substitution in fact exists.

Lemma 47 (Property of inverting substitutions) *Let τ and σ be pattern substitutions.*

1. *If $[\sigma]^{-1}\, \tau$ and $[\sigma]^{-1}\, ([\![\theta]\!]\tau)$ exist then $[\sigma]^{-1}\, ([\![\theta]\!](\tau)) = [\![\theta]\!]([\sigma]^{-1}\, \tau)$.*
2. *If $[\sigma]^{-1}\, U$ and $[\sigma]^{-1}\, ([\![\theta]\!]U)$ exist then $[\sigma]^{-1}\, ([\![\theta]\!]U) = [\![\theta]\!]([\sigma]^{-1}\, U)$.*
3. *If $[\sigma]^{-1}\, R$ and $[\sigma]^{-1}\, ([\![\theta]\!]R)$ exist then $[\sigma]^{-1}\, ([\![\theta]\!]R) = [\![\theta]\!]([\sigma]^{-1}\, R)$.*

Proof: By simultaneous induction on the structure of τ, U, and R. □

Lemma 48

1. *If $[\sigma]^{-1}\, \tau$ exists then $[\sigma]([\sigma]^{-1}\, \tau) = \tau$.*
2. *If $[\sigma]^{-1}\, U$ exists then $[\sigma]([\sigma]^{-1}\, U) = U$.*
3. *If $[\sigma]^{-1}\, R$ exists then $[\sigma]([\sigma]^{-1}\, R) = R$.*

Proof: The lemmas are proven by simultaneous structural induction on τ, U, and R. □

Lemma 49

1. *If σ is a pattern substitution and $[\sigma]U = U'$ then $U = [\sigma]^{-1} U'$*

2. *If σ is a pattern substitution and $[\sigma]R = R'$ then $R = [\sigma]^{-1} R'$*

Proof: The statement is proven by a simultaneous induction on the structure of U and R. □

Lemma 50 *Let τ and σ be pattern substitutions.*

1. *If $[\sigma]^{-1}\tau$ exists and $\Delta; \Gamma \vdash \tau : \Psi_1$ and $\Delta; \Gamma \vdash \sigma : \Psi_2$, then $\Delta; \Psi_2 \vdash [\sigma]^{-1}\tau : \Psi_1$.*

2. *If $[\sigma]^{-1} U$ exists and $\Delta; \Gamma \vdash U : [\sigma]Q$ and $\Delta; \Gamma \vdash \sigma : \Psi_2$, then $\Delta; \Psi_2 \vdash [\sigma]^{-1} U : Q$.*

3. *If $[\sigma]^{-1} R$ exists and $\Delta; \Gamma \vdash R : [\sigma]Q$ and $\Delta; \Gamma \vdash \sigma : \Psi_2$, then $\Delta; \Psi_2 \vdash [\sigma]^{-1} U : Q$.*

Proof: By simultaneous structural induction on the definition of inverse substitutions for τ, U, R. □

Next, we discuss pruning for normal and neutral objects.

$$\begin{array}{ll} \Delta; \Gamma \vdash U \mid\ [\sigma]^{-1} \Rightarrow (\Delta', \rho) & \text{Prune } U \text{ with respect to } \sigma \\ \Delta; \Gamma \Vdash R \mid\ [\sigma]^{-1} \Rightarrow (\Delta', \rho) & \text{Prune } R \text{ with respect to } \sigma \end{array}$$

Let Δ be the modal variables in U (and R resp). Then pruning with respect to the substitution σ, will return a modal substitution ρ s.t. $\Delta' \vdash \rho : \Delta$ and $[\sigma]^{-1}([\![\rho]\!]U)$ exists. The modal substitution ρ replaces modal variables u_i in Δ, where $u_i[\tau]$ occurs as a subterm in U (or R resp.) and $[\sigma]^{-1}(u_i[\tau])$ is not defined, with new modal variables $v[\tau']$, s.t. τ' prunes the substitution τ and $[\sigma]^{-1}(v[[\tau]\tau'])$ exists. In other words, pruning ensures that all bound variables occurring in $[\![\rho]\!]U$ occur in the range of σ which is achieved by applying the modal pruning substitution ρ to U.

$$\frac{\Delta; \Gamma, x{:}A \vdash U \mid\ [\sigma, x/x]^{-1} \Rightarrow (\Delta_1, \rho_1)}{\Delta; \Gamma \vdash \lambda x{:}A.U \mid\ [\sigma]^{-1} \Rightarrow (\Delta_1, \rho_1)} \qquad \frac{\Delta; \Gamma \Vdash R \mid\ [\sigma]^{-1} \Rightarrow (\Delta', \rho)}{\Delta; \Gamma \vdash R \mid\ [\sigma]^{-1} \Rightarrow (\Delta', \rho)}$$

$$\frac{(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2); \Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2 \quad \Delta_2' = [\![\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v]\!]\Delta_2}{(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2); \Gamma \Vdash v[\tau] \mid\ [\sigma]^{-1} \Rightarrow ((\Delta_1, v'{::}\Psi_2{\vdash}Q, \Delta_2'), (\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v, \mathsf{id}_{\Delta_2}))}$$

$$\frac{\Delta;\Gamma \Vdash R \mid\ [\sigma]^{-1} \Rightarrow (\Delta_1,\,\rho_1) \quad \Delta_1;[\![\rho_1]\!]\Gamma \vdash [\![\rho_1]\!]U_1 \mid\ [\sigma]^{-1} \Rightarrow (\Delta_2,\,\rho_2)}{\Delta;\Gamma \Vdash R\,U \mid\ [\sigma]^{-1} \Rightarrow (\Delta_2,\,[\![\rho_2]\!](\rho_1))}$$

$$\frac{[\sigma]^{-1}\,x \text{ exists}}{\Delta;\Gamma_1,x{:}A,\Gamma_2 \Vdash x \mid\ [\sigma]^{-1} \Rightarrow (\Delta,\,\mathsf{id}_\Delta)} \qquad \frac{}{\Delta;\Gamma \Vdash c \mid\ [\sigma]^{-1} \Rightarrow (\Delta,\,\mathsf{id}_\Delta)}$$

Applying a modal substitution θ to a modal context Δ is defined as follows:

$$\begin{array}{rcl} [\![\theta]\!](\cdot) & = & \cdot \\ [\![\theta]\!](u{::}\Psi{\vdash}A,\Delta) & = & (u{::}[\![\theta]\!]\Psi{\vdash}[\![\theta]\!]A,\ [\![\theta,u[\mathsf{id}_\Psi]/u]\!]\Delta) \\ & & \text{if } u \text{ does not occur in } [\![\theta]\!]\Psi \text{ and } [\![\theta]\!]A \end{array}$$

The application of θ to some modal context Δ is only defined if we do not create any circularities in the resulting context $[\![\theta]\!]\Delta$. This ensures that the resulting modal context $[\![\theta]\!]\Delta$ is well-formed. In the previous rule for pruning a modal variable $v[\tau]$ with respect to $[\sigma]^{-1}$, we create a new context $\Delta_2' = [\![\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v]\!]\Delta_2$. Since the range of $[\![\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v]\!]$ is different from the modal variables declared in Δ_2, we always will obtain a well-formed Δ_2'.

Note that pruning of an object U (resp. R) does not always succeed. In particular, if U contains a bound variable x which does not occur in the image of the substitution σ pruning will fail. When we prune an object U (resp. R), we assume that $\Delta;\Gamma \vdash U : A$ and if pruning succeeds it will return a new context Δ' and a modal substitution ρ such that $\Delta' \vdash \rho : \Delta$.

Note that τ is a pattern substitution and therefore we do not need to recursively prune the substitutions. However, we may need to prune some bound variables because $[\sigma]^{-1}\,\tau$ may not necessarily exist. Next, we describe the pruning of a substitution τ with respect to a substitution $[\sigma]^{-1}$.

Given a substitution τ such that $\Delta;\Gamma \vdash \tau : \Psi_1$, pruning τ with respect to $[\sigma]^{-1}$, where $\Delta;\Gamma \vdash \sigma : \Psi$ returns a context Ψ_2 such that $[\sigma]^{-1}\,([\tau]\mathsf{id}_{\Psi_2})$ exists. The following judgment describes the pruning of a substitution τ.

$$\Delta;\Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2$$

Ψ_2 is well-formed context such that $\Psi_2 \leq \Psi_1$ and $\Delta; \Psi_1 \vdash \mathsf{id}_{\Psi_2} : \Psi_2$. Pruning the substitution τ with respect to $[\sigma]^{-1}$ will always succeed and return a well-formed context Ψ_2.

$$\frac{}{\Delta; \Gamma \vdash \cdot : \cdot \mid [\sigma]^{-1} \Rightarrow \cdot} \qquad \frac{\Delta; \Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2 \quad [\sigma]^{-1}(y) \text{ exists}}{\Delta; \Gamma \vdash (\tau, y/x) : (\Psi_1, x{:}A) \mid [\sigma]^{-1} \Rightarrow (\Psi_2, x{:}A)}$$

$$\frac{\Delta; \Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2 \quad [\sigma]^{-1}(y) \text{ does not exists}}{\Delta; \Gamma \vdash (\tau, y/x) : (\Psi_1, x{:}A) \mid [\sigma]^{-1} \Rightarrow \Psi_2}$$

Before we show some properties about pruning, we summarize some of the considerations in defining pruning judgments. Recall that we call pruning when unifying an object $u[\sigma]$ with another object U. Since we only unify terms of the same type, we know that $\Delta; \Gamma \vdash u[\sigma] : Q'$ and $\Delta; \Gamma \vdash U : Q'$. We also know by typing invariant that $u{::}\Psi{\vdash}Q$ in Δ and in fact $\Delta; \Gamma \vdash u[\sigma] : [\sigma]Q$ and $\Delta; \Gamma \vdash \sigma : \Psi$. By the previous two assumptions, we know that in fact $Q' = [\sigma]Q$. Moreover, $\Delta = \Delta_1, u{::}\Psi{\vdash}Q, \Delta_2$ and $\Delta_1; \Psi \vdash Q : \mathsf{type}$. Since σ is a pattern substitution, we must have $\Delta_1; \Gamma \vdash [\sigma]Q : \mathsf{type}$.

We first argue why we do not need to prune the types when pruning an existential variable $v[\tau]$ with respect to $[\sigma]^{-1}$ and when pruning a substitution τ with respect to $[\sigma]^{-1}$. Moreover, we show when pruning τ with respect to $[\sigma]^{-1}$ the resulting context Ψ_2 must be well-formed. We say a variable y is *shared* between two pattern substitutions τ and σ if y is in the image of both τ and σ. It is easy to see that if $\Delta; \Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2$, then Ψ_2 contains exactly those variables x in Ψ_1 such that $[\tau]x$ is shared between τ and σ.

Lemma 51 (Well-formedness of pruned context)
If $\Delta; \Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2$, $\Delta; \Gamma \vdash \tau : \Psi_1$ *and* $\Delta; \Gamma \vdash \sigma : \Psi$ *then* $\Delta \vdash \Psi_2$ ctx *and* $\Psi_2 \leq \Psi_1$.

Proof: Proof by structural induction on the first derivation.

Case $\mathcal{D} = \dfrac{}{\Delta; \Gamma \vdash \cdot : \cdot \mid [\sigma]^{-1} \Rightarrow \cdot}$

$\Delta \vdash \cdot$ ctx by rule
$\cdot \leq \cdot$

Case $\mathcal{D} = \dfrac{\Delta;\Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2 \qquad [\sigma]^{-1}(y) \text{ exists}}{\Delta;\Gamma \vdash (\tau, y/x) : (\Psi_1, x{:}A) \mid [\sigma]^{-1} \Rightarrow (\Psi_2, x{:}A)}$

$\Delta;\Gamma \vdash (\tau, y/x) : (\Psi_1, x{:}A)$ — by assumption
$\Delta;\Gamma \vdash \tau : \Psi_1$ and $\Delta;\Gamma \vdash x{:}[\tau]A$ (*) — by inversion
$\Delta;\Gamma \vdash \sigma : \Psi$ — by assumption
$\Delta \vdash \Psi_2\ \mathsf{ctx}$ and $\Psi_2 \leq \Psi_1$ — by i.h.
$\Gamma(y) = B = [\tau]A$ — by inversion on (*)
$\sigma = (\sigma_1, y/z, \sigma_2)$ and $\Psi = (\Psi', z{:}A', \Psi'')$ — since $[\sigma]^{-1}(y)$ exists
$\Gamma(y) = B[\sigma_1]A'$ — since $\Delta;\Gamma \vdash \sigma : \Psi$
$[\tau]A = [\sigma_1]A'$ — transitivity of equality
$[\tau]A$ can only depend on variables shared between τ and σ_1
— τ and σ_1 are pattern substitutions
A can only depend on variables from Ψ_2 — see remark before this proof
$\Delta;\Psi_1 \vdash A : \mathsf{type}$ — from above
$\Delta \vdash \Psi_2, x{:}A\ \mathsf{ctx}$ — by rule
$(\Psi_2, x{:}A) \leq (\Psi_1, x{:}A)$ — by i.h.

Case $\mathcal{D} = \dfrac{\Delta;\Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2 \qquad [\sigma]^{-1}(y) \text{ does not exist}}{\Delta;\Gamma \vdash (\tau, y/x) : (\Psi_1, x{:}A) \mid [\sigma]^{-1} \Rightarrow \Psi_2}$

$\Delta;\Gamma \vdash (\tau, y/x) : (\Psi_1, x{:}A)$ — by assumption
$\Delta;\Gamma \vdash \tau : \Psi_1$ and $\Delta;\Gamma \vdash y{:}[\tau]A$ — by inversion
$\Delta;\Gamma \vdash \sigma : \Psi$ — by assumption
$\Delta \vdash \Psi_2\ \mathsf{ctx}$ and $\Psi_2 \leq (\Psi_1, x{:}A)$ — by i.h.

□

Lemma 52 (Well-formed pruning substitution)

1. *If* $\Delta;\Gamma \vdash U \mid [\sigma]^{-1} \Rightarrow (\Delta', \rho)$ *and* $\Delta;\Gamma \vdash U : [\sigma]A$ *and* $\Delta;\Gamma \vdash \sigma : \Psi$ *then* $\Delta' \vdash \rho : \Delta$.

2. *If* $\Delta;\Gamma \Vdash R \mid [\sigma]^{-1} \Rightarrow (\Delta', \rho)$ *and* $\Delta;\Gamma \vdash R : [\sigma]A$ *and* $\Delta;\Gamma \vdash \sigma : \Psi$ *then* $\Delta' \vdash \rho : \Delta$.

Proof: Simultaneous structural induction on the first derivation. We show a few cases of the proof.

Case $\mathcal{D} = \dfrac{\Delta;\Gamma, x{:}A \vdash U \mid \; [\sigma, x/x]^{-1} \Rightarrow (\Delta',\, \rho)}{\Delta;\Gamma \vdash \lambda x{:}A.U \mid \; [\sigma]^{-1} \Rightarrow (\Delta',\, \rho)}$

$\Delta;\Gamma \vdash \lambda x{:}A.U : [\sigma]B$ by assumption
$B = \Psi x{:}B_1.B_2$ for some B_1 and B_2
$\Delta;\Gamma \vdash [\sigma]B \equiv \Pi x{:}[\sigma]B_1.[\sigma, x/x]B_2 : \mathsf{type}$ and
$\Delta;\Gamma, x{:}A \vdash U : [\sigma, x/x]B_2$ by typing inversion lemma 12
$\Delta;\Gamma \vdash [\sigma]B \equiv \Pi x{:}[\sigma]B_1.[\sigma, x/x]B_2 : \mathsf{type}$ and by equality inversion 14
$\Delta;\Gamma \vdash [\sigma]B_1 \equiv A : \mathsf{type}$ and $\Delta;\Gamma \vdash [\sigma, x/x]B_2 \equiv [\sigma, x/x]B_2 : \mathsf{type}$
$\Delta;\Gamma \vdash \sigma : \Psi$ by assumption
$\Delta;\Gamma, x{:}A \vdash \sigma : \Psi$ by weakening
$\Delta;\Gamma, x{:}A \vdash x : A$ by rule
$\Delta;\Gamma, x{:}A \vdash x : [\sigma]B_1$ by rule
$\Delta;\Gamma, x{:}A \vdash (\sigma, x/x) : (\Psi, x{:}B_1)$ by rule
$\Delta' \vdash \rho : \Delta$ by i.h.

Case $\mathcal{D} = \dfrac{(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2);\Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2}{(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2);\Gamma \Vdash v[\tau] \mid \; [\sigma]^{-1} \Rightarrow (\Delta',\, \rho)}$

$\Delta' = (\Delta_1, v'{::}\Psi_2{\vdash}Q, \Delta_2')$ and $\Delta_2' = [\![\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v]\!]\Delta_2$ and $\rho = (\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v, \mathsf{id}_{\Delta_2})$
$(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2);\Gamma \vdash v[\tau] : A$ by assumption
$(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2);\Gamma \vdash \tau : \Psi_1$ and
$(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2);\Gamma \vdash [\sigma]A \equiv [\tau]Q : \mathsf{type}$ by typing inversion 12
$(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2);\Gamma \vdash \sigma : \Psi$ by assumption
$\Delta \vdash \Psi_2\ \mathsf{ctx}$ and $\Psi_2 \leq \Psi_1$ by lemma 51
$\Delta_1 \vdash \Psi_1\ \mathsf{ctx}$ by well-typedness of $(\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2)$
$\Delta_1 \vdash \Psi_2\ \mathsf{ctx}$ since $\Psi_2 \leq \Psi_1$
$A = [\sigma]^{-1}([\tau]Q)$ and $[\sigma]^{-1}([\tau]Q)$ exists by lemma 49
so pruning τ with respect to $[\sigma]^{-1}$ should not affect any variables in Q. Therefore, $\Delta_1;\Psi_2 \vdash Q : \mathsf{type}$. Moreover,
$(\Delta_1, v'{::}\Psi_2{\vdash}Q, \Delta_2') \vdash (\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v, \mathsf{id}_{\Delta_2}) : (\Delta_1, v{::}\Psi_1{\vdash}Q, \Delta_2)$

Case $\mathcal{D} = \dfrac{\Delta;\Gamma \Vdash R \mid \ [\sigma]^{-1} \Rightarrow (\Delta_1,\, \rho_1) \qquad \Delta_1; [\![\rho_1]\!]\Gamma \vdash [\![\rho_1]\!]U_1 \mid \ [\sigma]^{-1} \Rightarrow (\Delta_2,\, \rho_2)}{\Delta;\Gamma \Vdash R\, U \mid \ [\sigma]^{-1} \Rightarrow (\Delta_2,\, [\![\rho_2]\!](\rho_1))}$

$\Delta;\Gamma \vdash R\,U : [\sigma]A$ by assumption
$\Delta;\Gamma \vdash R : [\sigma](\Pi x{:}A_1.A_2)$ and $\Delta;\Gamma \vdash U : [\sigma]A_1$ and
$\Delta;\Gamma \vdash [\sigma, U/x]A_2 \equiv [\sigma]A : \mathsf{type}$ by typing inversion 12
$\Delta_1 \vdash \rho_1 : \Delta$ by i.h.
$\Delta_1; [\![\rho_1]\!]\Gamma \vdash [\![\rho_1]\!]U : [\![\rho_1]\!]([\sigma]A_1)$ by substitution property
$\Delta_1; [\![\rho_1]\!]\Gamma \vdash [\![\rho_1]\!]U : [\sigma]([\![\rho_1]\!]A_1)$ since ρ_1 is a pattern substitution
$\Delta_2 \vdash \rho_2 : \Delta_1$ by i.h.
$\Delta_2 \vdash [\![\rho_2]\!]\rho_1 : \Delta$ by composition of modal substitutions

□

Moreover, we show that if the modal variable u does not occur in R, then pruning R with respect to some σ, results in the modal context Δ' and the pruning substitution ρ, where u will be mapped to itself.

Lemma 53

1. *If the modal variable u does not occur in U and $\Delta;\Gamma \vdash U \mid [\sigma]^{-1} \Rightarrow (\Delta^*, \rho)$ and $\Delta = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2)$, then $\Delta^* = (\Delta_1^*, u{::}[\![\rho]\!]\Psi{\vdash}[\![\rho]\!]Q, \Delta_2^*)$ and $\rho = (\rho_1, u[\mathsf{id}_{[\![\rho]\!]\Psi}]/u, \rho_1)$.*
2. *If the modal variable u does not occur in R and $\Delta;\Gamma \Vdash R \mid [\sigma]^{-1} \Rightarrow (\Delta^*, \rho)$ and $\Delta = \Delta_1, u{::}\Psi{\vdash}Q, \Delta_2$, then $\Delta^* = (\Delta_1^*, u{::}[\![\rho]\!]\Psi{\vdash}[\![\rho]\!]Q, \Delta_2^*)$ and $\rho = (\rho_1, u[\mathsf{id}_{[\![\rho]\!]\Psi}]/u, \rho_1)$.*

Proof: Simultaneous structural induction on the pruning derivation. □

Next, we show correctness of pruning.

Lemma 54 (Pruning – Soundness)

1. *If $\Delta;\Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2$ and $\Delta;\Gamma \vdash \tau : \Psi_1$ and $\Delta;\Gamma \vdash \sigma : \Psi$ then $[\sigma]^{-1}([\tau](\mathsf{id}_{\Psi_2}))$ exists.*
2. *If $\Delta;\Gamma \vdash U \mid \ [\sigma]^{-1} \Rightarrow (\Delta',\, \rho)$ and $\Delta;\Gamma \vdash U{:}A$ and $\Delta;\Gamma \vdash \sigma : \Psi$ then $[\sigma]^{-1}([\![\rho]\!]U)$ exists.*

3. *If* $\Delta; \Gamma \Vdash R \mid \; [\sigma]^{-1} \Rightarrow (\Delta',\, \rho)$ *and* $\Delta; \Gamma \vdash R{:}A$ *and* $\Delta; \Gamma \vdash \sigma : \Psi$ *then* $[\sigma]^{-1}\,(\llbracket\rho\rrbracket R)$ *exists.*

Proof: The lemmas are proven by simultaneous structural induction on the first derivation.

Case $\Delta; \Gamma \Vdash R\,U \mid \sigma^{-1} \Rightarrow (\Delta_2, \llbracket\rho_2\rrbracket\rho_1)$

$\Delta; \Gamma \Vdash R \mid \sigma^{-1} \Rightarrow (\Delta_1, \rho_1)$	by premise
$\Delta_1; \llbracket\rho_1\rrbracket\Gamma \vdash \llbracket\rho_1\rrbracket U \mid \sigma^{-1} \Rightarrow (\Delta_2, \rho_2)$	by premise
$\Delta; \Gamma \vdash R\,U : A$	by assumption
$\Delta; \Gamma \vdash R : \Pi x{:}A_1.A_2$	
$\Delta; \Gamma \vdash U : A_1$ and	
$\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, U/x]A_2 \equiv A : \mathsf{type}$	by typing inversion lemma 12
$\Delta_1 \vdash \rho_1 : \Delta$	by lemma 52
$\Delta_1; \llbracket\rho_1\rrbracket\Gamma \vdash \llbracket\rho_1\rrbracket U : \llbracket\rho_1\rrbracket A_1$	by substitution property
$\Delta_2 \vdash \rho_2 : \Delta_1$	by lemma 52
$[\sigma]^{-1}\,(\llbracket\rho_1\rrbracket R)$ exists	by i.h.
$[\sigma]^{-1}\,(\llbracket\rho_2\rrbracket(\llbracket\rho_1\rrbracket U))$ exists	by i.h.
$[\sigma]^{-1}\,(\llbracket\llbracket\rho_2\rrbracket\rho_1\rrbracket U)$ exists	by substitution property
$\llbracket\rho_2\rrbracket[\sigma]^{-1}\,(\llbracket\rho_1\rrbracket R)$ exists	by substitution property
$[\sigma]^{-1}\,(\llbracket\rho_2\rrbracket\llbracket\rho_1\rrbracket R)$ exists	by previous lines
$[\sigma]^{-1}\,(\llbracket\llbracket\rho_2\rrbracket\rho_1\rrbracket R)$ exists	by substitution definition

Case $\Delta; \Gamma \vdash v[\tau] \mid \sigma^{-1} \Rightarrow (\Delta', \rho)$

$\Delta = \Delta_1, v{::}\Psi{\vdash}Q, \Delta_2$	
$\Delta' = (\Delta_1, v'{::}\Psi_2{\vdash}Q, \Delta_2')$ where and $\Delta_2' = \llbracket\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v\rrbracket\Delta_2$	
$\rho = (\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v, \mathsf{id}_{\Delta_2})$	by premise
$\Delta; \Gamma \vdash \tau \mid [\sigma]^{-1} \Rightarrow \Psi_2$	by premise
$\Delta; \Gamma \vdash v[\tau] : A$	by assumption
$\Delta; \Gamma \vdash \tau : \Psi$ and	
$\Delta; \Gamma \vdash A \equiv [\tau]Q : \mathsf{type}$	by typing inversion lemma 12
$[\sigma]^{-1}\,([\tau](\mathsf{id}_{\Psi_2})$ exists	by (1)

$v'[[\sigma]^{-1}([\tau](\mathsf{id}_{\Psi_2}))]$ exists
$[\sigma]^{-1}\,v'[[\tau](\mathsf{id}_{\Psi_2})]$ exists — by inverse substitution definition
$[\sigma]^{-1}(\llbracket\rho\rrbracket v[\tau])$ exists — by substitution definition

□

Lemma 55 (Pruning – Completeness)

1. *If* $\Delta;\Gamma \vdash \tau : \Psi$ *and* $\Delta;\Gamma \vdash \sigma : \Psi'$ *then* $\Delta;\Gamma \vdash \tau : \Psi \mid [\sigma]^{-1} \Rightarrow \Psi_2$ *for some* Ψ_2*.*

2. *If* $\Delta;\Gamma \vdash U : [\sigma]A$ *and* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta' \vdash \theta : \Delta$ *and* $[\sigma]^{-1}(\llbracket\theta\rrbracket U)$ *exists, then* $\Delta;\Gamma \vdash U \mid [\sigma]^{-1} \Rightarrow (\Delta'', \rho)$ *for some* Δ'' *and* ρ *and there exists a substitution* ρ' *such that* $\theta = \llbracket\rho'\rrbracket\rho$ *and* $\Delta' \vdash \rho' : \Delta''$*.*

3. *If* $\Delta;\Gamma \vdash R : [\sigma]A$ *and* $\Delta;\Gamma \vdash \sigma : \Psi$ *and* $\Delta' \vdash \theta : \Delta$ *and* $[\sigma]^{-1}(\llbracket\theta\rrbracket R)$ *exists, then* $\Delta;\Gamma \vdash R \mid [\sigma]^{-1} \Rightarrow (\Delta'', \rho)$ *for some* Δ'' *and* ρ *and there exists a substitution* ρ' *such that* $\theta = \llbracket\rho'\rrbracket\rho$ *and* $\Delta' \vdash \rho' : \Delta''$*.*

Proof: The statement is proven by simultaneous structural induction on the typing derivation. We show a few cases.

Case $\Delta;\Gamma \vdash v[\tau] : [\sigma]A$

$\Delta;\Gamma \vdash \tau : \Psi_1$ — by typing inversion lemma 12
$\Delta = (\Delta_1, v{::}\Psi_1{\vdash}Q_1, \Delta_2)$ and $\Delta;\Gamma \vdash [\sigma]A \equiv [\tau]Q_1 : \mathsf{type}$
$\Delta;\Gamma \vdash \sigma : \Psi$ — by assumption
$\Delta;\Gamma \vdash \tau : \Psi_1 \mid [\sigma]^{-1} \Rightarrow \Psi_2$ — by (1)
$\Delta;\Gamma \vdash v[\tau] \mid [\sigma]^{-1} \Rightarrow ((\Delta_1, v'{::}\Psi_2{\vdash}Q_1, \Delta_2'),\ (\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v, \mathsf{id}_{\Delta_2'}))$ — by rule
$\Delta_1; \Psi_2 \vdash Q_1 : \mathsf{type}$ — since $[\sigma]A = [\tau]Q_1$ and Ψ_2 contains exactly those variables shared between σ and τ
$\Delta' \vdash (\theta_1, U/v, \theta_2) : (\Delta_1, v{::}\Psi_1{\vdash}Q_1, \Delta_2)$ — by assumption
let $\rho' = (\theta_1, U/v', \theta_2)$ then we have $\theta = \llbracket\rho'\rrbracket((\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v, \mathsf{id}_{\Delta_2'})$.

Case $\Delta;\Gamma \vdash \lambda x{:}A.U : [\sigma]B$

$\Delta;\Gamma, x{:}A \vdash U : [\sigma, x/x]B_2$ and
$\Delta;\Gamma \vdash [\sigma]B \equiv \Pi x{:}A.[\sigma, x/x]B_2 : \mathsf{type}$ — by typing inversion 12

$\Delta;\Gamma \vdash [\sigma]B \equiv \Pi x{:}[\sigma]B_1.[\sigma, x/x]B_2 : \mathsf{type}$ and
$\Delta;\Gamma \vdash A \equiv [\sigma]B_1 : \mathsf{type}$ by equality inversion 14
$\Delta;\Gamma \vdash \sigma : \Psi$ by assumption
$\Delta;\Gamma, x{:}A \vdash x : A$ by rule
$\Delta;\Gamma, x{:}A \vdash x : [\sigma]B_1$ by rule
$\Delta;\Gamma, x{:}A \vdash (\sigma, x/x) : (\Psi, x{:}B_1)$ by rule
$[\sigma]^{-1}(\llbracket\theta\rrbracket(\lambda x{:}A.U))$ exists by assumption
$\lambda x{:}[\sigma]^{-1}(\llbracket\theta\rrbracket A).([\sigma, x/x]^{-1}\llbracket\theta\rrbracket U)$ exists by definition of inverse substitution
$([\sigma, x/x]^{-1}\llbracket\theta\rrbracket U)$ exists
$\exists\rho'.\theta = \llbracket\rho'\rrbracket\rho$ and $\Delta;\Gamma, x{:}A \vdash U \mid [\sigma, x/x]^{-1} \Rightarrow (\Delta', \rho)$ by i.h.
$\Delta;\Gamma \vdash \lambda x{:}A.U \mid [\sigma]^{-1} \Rightarrow (\Delta', \rho)$ by rule

□

Now, we can give the cases for unifying an existential variable with another kind of term. Since we are requiring that all existential variables are of atomic type, this term R can only be a modal variable $v[\tau']$, where v is different from u, an application, an ordinary variable or a constant.

$$\frac{\begin{array}{ll}\Delta;\Gamma \vdash R \mid [\sigma]^{-1} \Rightarrow (\Delta^*, \rho) & \Delta^* = (\Delta_1^*, u{::}\llbracket\rho\rrbracket\Psi{\vdash}\llbracket\rho\rrbracket Q, \Delta_2^*) \\ u \text{ does not occur in } R & \Delta_2' = \llbracket \mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}(\llbracket\rho\rrbracket R)/u\rrbracket\Delta_2^*\end{array}}{(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2);\Gamma \Vdash u[\sigma] \doteq R \;/\; ((\Delta_1^*, \Delta_2'), \llbracket\mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}(\llbracket\rho\rrbracket R)/u, \mathsf{id}_{\Delta_2^*}\rrbracket\rho)}\;ex1$$

$$\frac{\begin{array}{ll}\Delta;\Gamma \vdash R \mid [\sigma]^{-1} \Rightarrow (\Delta^*, \rho) & \Delta^* = (\Delta_1^*, u{::}\llbracket\rho\rrbracket\Psi{\vdash}\llbracket\rho\rrbracket Q, \Delta_2^*) \\ u \text{ does not occur in } R & \Delta_2' = \llbracket \mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}(\llbracket\rho\rrbracket R)/u\rrbracket\Delta_2^*\end{array}}{(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2);\Gamma \Vdash R \doteq u[\sigma] \;/\; ((\Delta_1^*, \Delta_2'), \llbracket\mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}(\llbracket\rho\rrbracket R)/u, \mathsf{id}_{\Delta_2^*}\rrbracket\rho)}\;ex2$$

The side condition "u does not occur in R" ensures that we obey the occurs-check. Note, that we do not need to ensure that u does not occur in the type of R. Recall that by invariant, we know that $\Delta;\Gamma \vdash u[\sigma] : Q'$ and $\Delta;\Gamma \vdash R : Q'$. We also know by typing invariant that $u{::}\Psi{\vdash}Q$ in Δ and $\Delta;\Gamma \vdash u[\sigma] : [\sigma]Q$ and $\Delta;\Gamma \vdash \sigma : \Psi$. By the previous two assumptions, we know that $Q' = [\sigma]Q$. Moreover, $\Delta = \Delta_1, u{::}\Psi{\vdash}Q, \Delta_2$ and $\Delta_1;\Psi \vdash Q : \mathsf{type}$. Since σ is a pattern substitution, we must have $\Delta_1; \Gamma \vdash [\sigma]Q : \mathsf{type}$. Since $[\sigma]Q = Q'$, we know that u cannot occur in the type Q'. Therefore we do not need to perform the occurs check on the type of R.

However, since there may be dependencies among the types and declarations in Δ_2 may depend on u, it is not necessarily obvious why traversing R and checking whether u occurs in R suffices in the higher-order dependently typed setting. Potentially we could be in the following situation: a modal variable $v[\tau']$ occurs in R, but $v{::}\Psi'{\vdash}Q'$ occurs in Δ_2 and depends on u. When we construct the new $\Delta_2' = [\mathsf{id}_{\Delta_1}, [\sigma]^{-1}\,(\llbracket\rho\rrbracket R)/u]\Delta_2$, we will replace occurrences of u in Ψ' and Q' with $[\sigma]^{-1}\,(\llbracket\rho\rrbracket R)$. But since R itself refers to v, we may be creating a circular new declaration for v and the resulting modal context Δ_2' may not be well-formed.

Lemma 56

1. *If $\Delta; \Gamma \vdash U \Leftarrow A$ and $\Delta = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2)$ and u does not occur in Γ, U, and A, then for any modal variable v where v is different from u that occurs in U and $v{::}\Psi'{\vdash}Q'$ is declared in Δ, u does not occur in Ψ' and Q'.*

2. *If $\Delta; \Gamma \vdash R \Rightarrow A$ and $\Delta = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2)$ and u does not occur in Γ, R, then u does not occur in A and for any modal variable v where v is different from u that occurs in R and $v{::}\Psi'{\vdash}Q'$ is declared in Δ, u does not occur in Ψ' and Q'.*

Proof: By simultaneous induction on the algorithmic typing derivation.

Case $\Delta; \Gamma \vdash R\,U \Rightarrow [\mathsf{id}_\Gamma, U/x]A_2$

$\Delta; \Gamma \vdash R \Rightarrow \Pi x{:}A_1'.A_2$

$\Delta; \Gamma \vdash U \Leftarrow A_1$ and $\Delta; \Gamma \vdash A_1 \Longleftrightarrow A_1' : \mathsf{type}$ — by inversion

$A_1 = A_1'$ — since A_1, A_1' are canonical

u does not occur in $R\,U$ — by assumption

u does not occur in R

u does not occur in U

u does not occur in G — by assumption

u does not occur in $\Pi x{:}A_1'.A_2$ and for any modal variable $v{::}\Psi'{\vdash}Q'$ occurring in R,
    u does not occur in Ψ' and Q' — by i.h.

u does not occur in A_1'

u does not occur in A_1 since $A_1 \Longleftrightarrow A_1'$
for any modal variable $v{::}\Psi'{\vdash}Q'$ occurring in U,
    u does not occur in Ψ' and Q' — by i.h.
for any modal variable $v{::}\Psi'{\vdash}Q'$ occurring in $R\,U$,
    u does not occur in Ψ' and Q'

Case $(\Delta', v{::}\Psi'{\vdash}Q', \Delta''); \Gamma \vdash v[\tau] \Rightarrow [\tau]Q'$

$\Delta = (\Delta', v{::}\Psi'{\vdash}Q', \Delta'')$ and $\Delta; \Gamma \vdash \tau \Leftarrow \Psi'$ — by inversion
u does not occur in Γ — by assumption
u does not occur in Ψ' — since τ is a pattern substitution
u does not occur in $[\tau]Q'$ — by assumption
u does not occur in Q' — since τ is a pattern substitution

Case $\Delta; \Gamma \vdash c \Rightarrow A$

c is defined in the signature Σ. $\Sigma(c) = A$. Moreover, A must be closed and cannot refer to any modal variables. Therefore, u cannot occur in A.

Case $\Delta; \Gamma \vdash x \Rightarrow A$
$\Gamma(x) = A$ — by inversion
u does not occur in A — since u does not occur in Γ

Case $\Delta; \Gamma \vdash \lambda x{:}A_1.U \Leftarrow \Pi x{:}A_1.A_2$

$\Delta; \Gamma, x{:}A_1 \vdash U \Leftarrow A_2$ — by inversion
u does not occur in Γ — by assumption
u does not occur in $\Pi x{:}A_1.A_2$ — by assumption
u does not occur in A_1 and A_2
u does not occur in $\Gamma, x{:}A_1$
for any $v{::}\Psi'{\vdash}Q'$ occuring in U
    u does not occur in Ψ' and Q' — by i.h.
for any $v{::}\Psi'{\vdash}Q'$ occuring in $\lambda x{:}A_1.U$
    u does not occur in Ψ' and Q'

□

The last lemma justifies why a simple occurs check also suffices in the higher-order dependently typed setting. In an implementation, we may combine computing the pruning substitution with the occurs check. Since we already traverse the term R to check whether $[\sigma]^{-1} R$ exists, we may simultaneously check whether u occurs in R. However, the previous lemma shows that for higher-order patterns, we do not need to prune types and the bound variable context. Moreover, we do not need to perform an occurs check on the types of modal variables which may occur in R[2] .

3.1.2 Unifying two identical existential variables

For unifying the two existential variables which are the same, we need an additional auxiliary judgment, which computes the intersection of two substitutions.

$$\Delta; \Gamma \vdash \sigma \cap \tau : \Psi \Rightarrow \Psi'$$

Given the substitution $\Delta; \Gamma \vdash \sigma : \Psi$ and $\Delta; \Gamma \vdash \tau : \Psi$, we construct a context Ψ' which is well-formed and for the substitution $\mathsf{id}_{\Psi'}$ such that $\Delta; \Psi \vdash \mathsf{id}_{\Psi'} : \Psi'$ we have $[\sigma](\mathsf{id}_{\Psi'}) = [\tau](\mathsf{id}_{\Psi'})$.

$$\frac{}{\Delta; \Gamma \vdash \cdot \cap \cdot : \cdot \Rightarrow \cdot}$$

$$\frac{\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi'}{\Delta; \Gamma \vdash (\tau, y/x) \cap (\sigma, y/x) : (\Psi, x{:}A) \Rightarrow (\Psi', x{:}A)}$$

$$\frac{\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi' \quad z \neq y}{\Delta; \Gamma \vdash (\tau, z/x) \cap (\sigma, y/x) : (\Psi, x{:}A) \Rightarrow \Psi'}$$

We say a variable y is *strictly shared* between two pattern substitutions τ and σ if there is an x such that $y = [\tau]x = [\sigma]x$. It is easy to see that if $\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi'$ then Ψ' contains exactly those variables x in Ψ such that $y = [\tau]x = [\sigma]x$ (in other words, y is strictly shared between τ and σ). The resulting context Ψ' will always be well-formed. To illustrate the difference between the notion of "shared" and "strictly

[2]It seems possible that an extended occurs check would lead to incompleteness, no?

shared" consider the substitution $[y/x1, z/x2]$ and $[z/x1, y/x2]$. Here, y and z are shared, but not strictly shared. y and z will be retained under inverse substitution, but they are pruned under intersection.

Lemma 57 *If* $\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi'$, $\Delta; \Gamma \vdash \tau : \Psi$ *and* $\Delta; \Gamma \vdash \sigma : \Psi$ *then* $\Delta \vdash \Psi'$ ctx.

Proof: Structural induction on the first derivation. We consider the interesting case, where

$$\frac{\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi'}{\Delta; \Gamma \vdash (\tau, y/x) \cap (\sigma, y/x) : (\Psi, x{:}B) \Rightarrow \Psi', x{:}B}$$

$\Delta; \Gamma \vdash (\tau, y/x) : (\Psi, x{:}B)$ by assumption
$\Delta; \Gamma \vdash \tau : \Psi$ by inversion
$\Delta; \Gamma \vdash y : [\tau]B$ by inversion
$\Gamma(y) = B' = [\tau]B$ by inversion
$\Delta; \Gamma \vdash (\sigma, y/x) : (\Psi, x{:}B)$ by assumption
$\Delta; \Gamma \vdash \sigma : \Psi$ by inversion
$\Delta; \Gamma \vdash y : [\sigma]B$ by inversion
$\Gamma(y) = B' = [\sigma]B$ by inversion
$B' = [\tau]B = [\sigma]B$ by previous lines
B can only depend on variables strictly shared between τ and σ
$\Delta \vdash \Psi'$ ctx by i.h.
$\Delta; \Psi' \vdash B : \mathsf{type}$ since Ψ' contains exactly those variables strictly shared by τ and σ
$\Delta \vdash (\Psi', x{:}B)$ ctx □

Next, we show correctness of computing the intersection between two substitutions.

Lemma 58

1. *If* $\Delta; \Gamma \vdash \tau : \Psi$ *and* $\Delta; \Gamma \vdash \sigma : \Psi$ *and* $\Delta = \Delta_1, u{::}\Psi{\vdash}Q, \Delta_2$ *then*
   $\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi'$

2. *If* $\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi'$ *then*
   $[\sigma](\mathsf{id}_{\Psi'}) = [\tau](\mathsf{id}_{\Psi'})$ *and* $\Delta; \Psi \vdash \mathsf{id}_{\Psi'} : \Psi'$.

Proof: The first statement is proven by structural induction on the first derivation. The second statement is proven by induction on the first derivation using lemma 4. □

Now we can give the rule for unifying two modal variables which are the same.

$$\frac{(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi_2 \quad \Delta_2' = [\![\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v]\!]\Delta_2}{(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \Vdash u[\sigma] \doteq u[\tau] \;/\; ((\Delta_1, v{::}\Psi_2{\vdash}Q, \Delta_2'),\, (\mathsf{id}_{\Delta_1}, v[\mathsf{id}_{\Psi_2}]/u, \mathsf{id}_{\Delta_2}))}\, ex3$$

We have shown that the resulting Ψ_2 of computing the intersection of τ and σ, is indeed well-formed (lemma 57). We can also justify why in the unification rule itself the type Q of two existential variables must be well-typed in the pruned context Ψ_2. Recall that by typing invariant, we know that $\Delta; \Gamma \vdash \tau : \Psi$ and $\Delta; \Gamma \vdash \sigma : \Psi$ and $[\sigma]Q = [\tau]Q = Q'$. But this mean that Q can only depend on the variables strictly shared by τ and σ. Since Ψ_2 is exactly the context, which captures the shared variables, Q must also be well-typed in Ψ_2.

During unification, we ensure that the resulting modal substitution θ maps all the modal variables in Δ to a new modal context Δ'.

Lemma 59

1. *If* $\Delta; \Gamma \vdash U_1 \doteq U_2 \;/\; (\Delta', \theta)$ *and* $\Delta; \Gamma \vdash U_1 : A$ *and* $\Delta; \Gamma \vdash U_2 : A$
   then $\Delta' \vdash \theta : \Delta$*.*

2. *If* $\Delta; \Gamma \Vdash R_1 \doteq R_2 \;/\; (\Delta', \theta)$ *and* $\Delta; \Gamma \vdash U_1 : A$ *and* $\Delta; \Gamma \vdash U_2 : A$
   then $\Delta' \vdash \theta : \Delta$*.*

Proof: Structural induction on the first derivation. We show the cases for unifying an existential variable with another term and unifying two identical existential variables.

Case $\mathcal{D} = \Delta; \Gamma \vdash u[\sigma] \doteq R \;/\; ((\Delta_1^*, \Delta_2'), [\![\mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}([\![\rho]\!]R)/u, \mathsf{id}_{\Delta_2^*}]\!]\rho)$

$\Delta = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2)$	
$\Delta; \Gamma \vdash R \mid [\sigma]^{-1} \Rightarrow (\Delta^*, \rho)$ and u does not occur in R	by premises
$\Delta; \Gamma \vdash u[\sigma] : A$	by assumption
$\Delta; \Gamma \vdash \sigma : \Psi$ and $\Delta; \Gamma \vdash A \equiv [\sigma]Q : \mathsf{type}$	by typing inversion lemma 12
u does not occur in Ψ and Q	by well-formedness of Δ
$\Delta; \Gamma \vdash R : [\sigma]Q$	by assumption

$\Delta^* \vdash \rho : \Delta$ by lemma 52

$[\sigma]^{-1} [\![\rho]\!]R$ exists by lemma 54

$\Delta^* = (\Delta_1^*, u{::}[\![\rho]\!]\Psi{\vdash}[\![\rho]\!]Q, \Delta_2^*)$ and $\rho = (\rho_1^*, u[\mathsf{id}_{[\![\rho]\!]\Psi}]/u, \rho_2^*)$ by lemma 53

$\Delta^*; [\![\rho]\!]\Gamma \vdash [\![\rho]\!]R : [\![\rho]\!]([\sigma]Q)$ by substitution property (lemma 41)

$\Delta^*; [\![\rho]\!]\Gamma \vdash [\![\rho]\!]R : [\sigma]([\![\rho]\!]Q)$ since σ is a pattern substitution

$\Delta^*; [\![\rho]\!]\Gamma \vdash \sigma : [\![\rho]\!]\Psi$ by substitution property

$\Delta^*; [\![\rho]\!]\Psi \vdash [\sigma]^{-1} ([\![\rho]\!]R) : [\![\rho]\!]Q$ by lemma 50

u does not occur in $[\![\rho]\!]\Psi$ and $[\![\rho]\!]Q$ and $[\sigma]^{-1} ([\![\rho]\!]R$ because ρ only replaces modal variables with new variables

for any modal variable v occuring in $[\sigma]^{-1} ([\![\rho]\!]R)$ where $v{::}\Psi'{\vdash}Q'$,
u cannot occur in Ψ' and Q' by lemma 56

$\Delta_2' = [\![\mathsf{id}_{\Delta_1^*}, [\sigma]^{-1} ([\![\rho]\!]R)/u]\!]\Delta_2^*$ and $\vdash \Delta_2'$ mctx by previous lines
i.e. no circularities are created in Δ_2'

$\theta = [\![\mathsf{id}_{\Delta_1^*}, [\sigma]^{-1} ([\![\rho]\!]R)/u, \mathsf{id}_{\Delta_2^*}]\!]\rho = (\rho_1^*, [\sigma]^{-1} ([\![\rho]\!]R)/u, \rho_2^*)$ by substitution definition

$(\Delta_1^*, \Delta_2'); [\![\rho]\!]\Psi \vdash [\sigma]^{-1} ([\![\rho]\!]R) : [\![\rho]\!]Q$ since u does not occur in Ψ, R or Q

$(\Delta_1^*, \Delta_2'); [\![\rho_1^*]\!]\Psi \vdash [\sigma]^{-1} ([\![\rho]\!]R) : [\![\rho_1^*]\!]Q$ since $\Delta_1; \Psi \vdash Q : \mathsf{type}$

$(\Delta_1^*, \Delta_2') \vdash \rho_1^* : \Delta_1$ since the pruning substitution ρ^* only instantiates modal variables from Δ with new modal variables

$(\Delta_1^*, \Delta_2') \vdash (\rho_1^*, [\sigma]^{-1} ([\![\rho]\!]R)/u) : (\Delta_1, u{::}\Psi{\vdash}Q)$ by previous lines

$(\Delta_1^*, \Delta_2') \vdash (\rho_1^*, [\sigma]^{-1} ([\![\rho]\!]R)/u, \rho_2^*) : \Delta$ by typing rules

Case $\mathcal{D} = \Delta; \Gamma \vdash u[\sigma] \doteq u[\tau] \;/\; ((\Delta_1, v{::}\Psi_2{\vdash}Q, \Delta_2'), [\![\mathsf{id}_{\Delta_1}, v[\mathsf{id}_{\Psi_2}]/u, \mathsf{id}_{\Delta_2}]\!])$

$\Delta = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2)$

$\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi_2$ by premise

$\Delta; \Gamma \vdash u[\tau] : A$ by assumption

$\Delta; \Gamma \vdash \tau : \Psi$ and $\Delta; \Gamma \vdash A \equiv [\tau]Q : \mathsf{type}$ by typing inversion 12

$\Delta; \Gamma \vdash u[\sigma] : A$ by assumption

$\Delta; \Gamma \vdash \sigma : \Psi$ and $\Delta; \Gamma \vdash A \equiv [\sigma]Q : \mathsf{type}$ by typing inversion lemma 12

$\Delta; \Gamma \vdash [\sigma]Q \equiv [\tau]Q : \mathsf{type}$ by transitivity

Q can only depend on those bound variables which are strictly shared by τ and σ

Ψ_2 exactly contains those shared variables

$\Delta_1; \Psi \vdash Q : \mathsf{type}$ by well-typedness of Δ

$\Delta_1; \Psi_2 \vdash Q : \mathsf{type}$ by previous lines

$(\Delta_1, v{::}\Psi_2{\vdash}Q, \Delta_2') \vdash (\mathsf{id}_{\Delta_1}, v[\mathsf{id}_{\Psi_2}]/u, \mathsf{id}_{\Delta_2}) : \Delta$ by previous lines and typing rules

$\square$

Recall, we assume that the two objects we unify are in canonical form, in particular we may omit the type label on λ-abstractions.

For soundness, we show that if two objects U_1 and U_2 unify under the modal substitution θ, then $[\![\theta]\!]U_1$ is definitional equal to $[\![\theta]\!]U_2$. Since we concentrate on canonical forms, definitional equality can be established by checking whether two objects are syntactically equal. Now we show that if the object U_1 and U_2 unify under the modal substitution θ, then $[\![\theta]\!]U_1$ is syntactically equal to $[\![\theta]\!]U_2$, thereby showing indirectly that $[\![\theta]\!]U_1$ and $[\![\theta]\!]U_2$ must also be definitional equal.

Theorem 60 (Soundness of higher-order pattern unification)

1. *If* $\Delta; \Gamma \vdash U_1 \doteq U_2 \,/\, (\Delta', \theta)$ *and* $\Delta; \Gamma \vdash U_1 : A$, *and* $\Delta; \Gamma \vdash U_2 : A$ *then* $[\![\theta]\!]U_1 = [\![\theta]\!]U_2$.

2. *If* $\Delta; \Gamma \Vdash R_1 \doteq R_2 \,/\, (\Delta', \theta)$ *and* $\Delta; \Gamma \vdash R_1 : A$, *and* $\Delta; \Gamma \vdash R_2 : A$ *then* $[\![\theta]\!]R_1 = [\![\theta]\!]R_2$.

Proof: By simultaneous structural induction on the first derivation. Below, we give a few cases.

Case $\mathcal{D} = \dfrac{\Delta; \Gamma, x{:}A \vdash U_1 \doteq U_2/(\Delta', \theta)}{\Delta; \Gamma \vdash \lambda x{:}A.U_1 \doteq \lambda x{:}A.U_2/(\Delta', \theta)}$

$\Delta; \Gamma \vdash \lambda x{:}A.U_1 : A'$ by assumption

$\Delta; \Gamma, x{:}A \vdash U_1 : B$ by inversion lemma 12

$\Delta; \Gamma \vdash A : \mathsf{type}$ and $\Delta; \Gamma \vdash A' \equiv \Pi x{:}A.B : \mathsf{type}$

$\Delta; \Gamma \vdash \lambda x{:}A.U_2 : A'$ by assumption

$\Delta; \Gamma, x{:}A \vdash U_2 : B$ by inversion lemma 12

$[\![\theta_1]\!]U_1 = [\![\theta]\!]U_2$ by i.h.

$\Delta' \vdash \theta : \Delta$ by lemma 52
$[\![\theta]\!]A = [\![\theta]\!]A$ by reflexivity
$\lambda x{:}[\![\theta]\!]A.[\![\theta]\!]U_1 = \lambda x{:}[\![\theta]\!]A.[\![\theta]\!]U_2$ by syntactic equality
$[\![\theta]\!](\lambda x{:}A.U_1) = [\![\theta]\!](\lambda x{:}A.U_2)$ by modal substitution definition

Case $\mathcal{D} = \dfrac{\Delta; \Gamma \Vdash R_1 \overset{\mathcal{D}_1}{\doteq} R_2/(\Delta_1,\, \theta_1) \qquad \Delta_1; [\![\theta_1]\!]\Gamma \vdash [\![\theta_1]\!]U_1 \overset{\mathcal{D}_2}{\doteq} [\![\theta_1]\!]U_2/(\Delta_2,\, \theta_2)}{\Delta; \Gamma \Vdash R_1\ U_1 \doteq R_2\ U_2/(\Delta_2,\, [\![\theta_2]\!](\theta_1))}$

$\Delta; \Gamma \vdash R_1\ U_1 : A$ by assumption
$\Delta; \Gamma \vdash R_2\ U_2 : A$ by assumption
$\Delta; \Gamma \vdash R_1 : \Pi x{:}A_2.A_1$ by inversion lemma 12
$\Delta; \Gamma \vdash U_1 : A_2$ and $\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, U_1/x]A_1 \equiv A : \mathsf{type}$
$\Delta; \Gamma \vdash R_2 : \Pi x{:}A_2.A_1$ by inversion lemma 12
$\Delta; \Gamma \vdash U_2 : A_2$ and $\Delta; \Gamma \vdash [\mathsf{id}_\Gamma, U_2/x]A_1 \equiv A : \mathsf{type}$
$[\![\theta_1]\!]R_1 = [\![\theta_1]\!]R_2$ by i.h.
$\Delta_2 \vdash \theta_2 : \Delta_1$ by lemma 52
$[\![\theta_2]\!]([\![\theta_1]\!]R_1) = [\![\theta_2]\!]([\![\theta_1]\!]R_2)$ by modal substitution lemma 41
$[\![[\![\theta_2]\!]\theta_1]\!](R_1) = [\![[\![\theta_2]\!]\theta_1]\!](R_2)$ by modal comp. lemma 42
$\Delta_1 \vdash \theta_1 : \Delta$ by lemma 52
$\Delta_1; [\![\theta_1]\!]\Gamma \vdash [\![\theta_1]\!]U_1 : [\![\theta_1]\!]A_2$ by modal substitution lemma 41
$\Delta_1; [\![\theta_1]\!]\Gamma \vdash [\![\theta_1]\!]U_2 : [\![\theta_1]\!]A_2$ by modal substitution lemma 41
$[\![\theta_2]\!]([\![\theta_1]\!]U_1) = [\![\theta_2]\!]([\![\theta_1]\!]U_2)$ by i.h.
$[\![[\![\theta_2]\!]\theta_1]\!]U_1 = [\![[\![\theta_2]\!]\theta_1]\!]U_2$ by modal comp. lemma 42
$[\![[\![\theta_2]\!]\theta_1]\!](R_1\ U_1) = [\![[\![\theta_2]\!]\theta_1]\!](R_2\ U_2)$ by rule and modal sub. def.

Case $\mathcal{D} = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \Vdash u[\sigma] \doteq R\ /\ (\Delta',\, \theta)$

$\Delta' = (\Delta_1^*, \Delta_2')$ and $\theta = [\![\mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}([\![\rho]\!]R)/u, \mathsf{id}_{\Delta_2^*}]\!]\rho$ by premise
$\Delta; \Gamma \vdash R \mid\ [\sigma]^{-1} \Rightarrow (\Delta^*,\, \rho)$ and by premise
$\Delta^* = (\Delta_1^*, u{::}[\![\rho]\!]\Psi{\vdash}[\![\rho]\!]Q, \Delta_2^*)$ and u does not occur in R by inversion
$\Delta^* \vdash \rho : \Delta$ by lemma 51
$\rho = (\rho_1, u[\mathsf{id}_{[\![\rho]\!]\Psi}]/u, \rho_2)$ by lemma 53

$\Delta' \vdash \theta : (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2)$	by lemma 59
$([\sigma]^{-1}\,(\llbracket\rho\rrbracket R))$ exists	by lemma 54
$\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2; \Gamma \vdash u[\sigma] : Q'$	by assumption
$\Delta; \Gamma \vdash Q' \equiv [\sigma]Q : \mathsf{type}$	by typing inversion lemma 12
$\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2; \Gamma \vdash R : Q'$	by assumption
$\Delta'; \llbracket\theta\rrbracket\Gamma \vdash \llbracket\theta\rrbracket R : \llbracket\theta\rrbracket Q'$	by modal substitution lemma 41
$\llbracket\theta\rrbracket(u[\sigma]) = [\sigma]([\sigma]^{-1}\,\llbracket\rho\rrbracket R)$	by substitution definition
$= \llbracket\rho\rrbracket R$	by lemma 48
$= \llbracket\theta\rrbracket R$	because u does not occur in R

Case $\mathcal{D} = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \Vdash u[\sigma] \doteq u[\tau] \,/\, (\Delta', \theta)$

$\Delta' = (\Delta_1, v{::}\Psi_2{\vdash}Q, \Delta_2')$ and $\theta = (\mathsf{id}_{\Delta_1}, v[\mathsf{id}_{\Psi_2}]/u, \mathsf{id}_{\Delta_2})$	by premise
$(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi_2$ and $\Delta_2' = \llbracket\mathsf{id}_{\Delta_1}, v[\mathsf{id}_{\Psi_2}]/u\rrbracket\Delta_2$	by premise
$[\sigma]\mathsf{id}_{\Psi_2} = [\tau]\mathsf{id}_{\Psi_2}$	by lemma 58
$v[[\sigma]\mathsf{id}_{\Psi_2}] = v[[\tau]\mathsf{id}_{\Psi_2}]$	by syntactic equality
$[\sigma](v[\mathsf{id}_{\Psi_2}]) = [\tau](v[\mathsf{id}_{\Psi_2}])$	by substitution definition
$\llbracket\theta\rrbracket(u[\sigma]) = \llbracket\theta\rrbracket(u[\tau])$	by substitution definition

□

For completeness, we again assume that the objects are in canonical form and show that if two objects are syntactically equal under some modal substitution θ then they are unifiable and the unification algorithm returns a modal substitution θ' s.t. $\theta = \llbracket\rho\rrbracket\theta'$.

Theorem 61 (Completeness of higher-order pattern unification)

1. *If $\Delta; \Gamma \vdash U_1 : A$ and $\Delta; \Gamma \vdash U_2 : A$ and $\Delta' \vdash \theta : \Delta$ and $\llbracket\theta\rrbracket U_1 = \llbracket\theta\rrbracket U_2$ then for some Δ'' and θ', we have $\Delta; \Gamma \vdash U_1 \doteq U_2 \,/\, (\Delta'', \theta')$ and there exists a modal substitution ρ, s.t. $\Delta' \vdash \rho : \Delta''$ and $\theta = \llbracket\rho\rrbracket(\theta')$.*

2. *If $\Delta; \Gamma \vdash R_1 : A_1$ and $\Delta; \Gamma \vdash R_2 : A_2$ and $\Delta' \vdash \theta : \Delta$ and $\llbracket\theta\rrbracket R_1 = \llbracket\theta\rrbracket R_2$ then for some Δ'' and θ', we have $\Delta; \Gamma \Vdash R_1 \doteq R_2 \,/\, (\Delta'', \theta')$, $\llbracket\theta\rrbracket A_1 = \llbracket\theta\rrbracket A_2$ and there exists a modal substitution ρ, s.t. $\Delta' \vdash \rho : \Delta''$ and $\theta = \llbracket\rho\rrbracket(\theta')$.*

Proof: Proof by simultaneous induction on the structure of U_1 and U_2 (and R_1, R_2 respectively). Most cases are straightforward and we give a few cases to illustrate.

Case $U_1 = \lambda x.U'$ and $U_2 = \lambda x.U''$

$[\![\theta]\!](\lambda x.U') = [\![\theta]\!](\lambda x.U'')$ by assumption
$\lambda x.[\![\theta]\!](U') = \lambda x.[\![\theta]\!]U''$ by substitution definition
$[\![\theta]\!](U') = [\![\theta]\!](U'')$ by definition of syntactic equality
$\Delta; \Gamma \vdash \lambda x.U' : A$ by assumption
$\Delta; \Gamma, x{:}A_1 \vdash U' : A_2$ by inversion lemma 12
$\Delta; \Gamma \vdash A_1 : \mathsf{type}$ and $\Delta; \Gamma \vdash A \equiv \Pi x{:}A_1.A_2 : \mathsf{type}$
$\Delta; \Gamma \vdash \lambda x.U'' : A$ by assumption
$\Delta; \Gamma, x{:}A_1' \vdash U'' : A_2'$ by inversion lemma 12
$\Delta; \Gamma \vdash A_1' : \mathsf{type}$ and $\Delta; \Gamma \vdash A \equiv \Pi x{:}A_1'.A_2' : \mathsf{type}$
$\Delta; \Gamma \vdash \Pi x{:}A_1.A_2 \equiv \Pi x{:}A_1'.A_2' : \mathsf{type}$ by transitivity
$\Delta; \Gamma \vdash A_1 \equiv A_1' : \mathsf{type}$ and $\Delta; \Gamma, x{:}A_1 \vdash A_2 \equiv A_2' : \mathsf{type}$ by injectivity of products (lemma 15)
$\Delta; \Gamma, x{:}A_1 \vdash U' \doteq U'' / (\Delta'', \theta')$ by i.h.
$\theta = [\![\rho]\!](\theta')$ and $\Delta'' \vdash \theta' : \Delta$ and $\theta = [\![\rho]\!](\theta')$
$\Delta; \Gamma \vdash \lambda x.U' \doteq \lambda x.U'' / (\Delta'', \theta')$ by rule

Case $U_1 = R_1\ U_1'$ and $U_2 = R_2\ U_2'$

$\Delta; \Gamma \vdash R_1\ U_1' : A$ by assumption
$\Delta; \Gamma \vdash R_1 : \Pi x{:}A_1.A_2$ by typing inversion lemma 12
$\Delta; \Gamma \vdash U_1' : A_1$ and $\Delta; \Gamma \vdash A \equiv [\mathsf{id}_\Gamma, U_1/x]A_2 : \mathsf{type}$
$\Delta; \Gamma \vdash R_2\ U_2' : A$ by assumption
$\Delta; \Gamma \vdash R_2 : \Pi x{:}A_1'.A_2'$ by typing inversion lemma 12
$\Delta; \Gamma \vdash U_2' : A_1'$ and $\Delta; \Gamma \vdash A \equiv [\mathsf{id}_\Gamma, U_2/x]A_2' : \mathsf{type}$
$\Delta' \vdash \theta : \Delta$ by assumption
$[\![\theta]\!](R_1\ U_1') = [\![\theta]\!](R_2\ U_2')$ by assumption
$([\![\theta]\!]R_1)\ ([\![\theta]\!]U_1') = ([\![\theta]\!]R_2)\ ([\![\theta]\!]U_2')$ by substitution definition
$([\![\theta]\!]R_1) = ([\![\theta]\!]R_2)$ and $([\![\theta]\!]U_1') = ([\![\theta]\!]U_2')$ by definition of syntactic equality

$\Delta; \Gamma \Vdash R_1 \doteq R_2/(\Delta_1, \theta_1)$ and $\llbracket\theta\rrbracket(\Pi x{:}A_1.A_2) = \llbracket\theta\rrbracket(\Pi x{:}A'_1.A'_2)$	by i.h.
$\exists \rho.\Delta' \vdash \rho : \Delta_1$ and $\theta = \llbracket\rho\rrbracket\theta_1$	
$\Delta_1 \vdash \theta_1 : \Delta$	by lemma 59
$\Delta_1; \llbracket\theta_1\rrbracket\Gamma \vdash \llbracket\theta_1\rrbracket U'_1 : \llbracket\theta_1\rrbracket A_1$	by substitution property (lemma 41)
$\Delta_1; \llbracket\theta_1\rrbracket\Gamma \vdash \llbracket\theta_1\rrbracket U'_2 : \llbracket\theta_1\rrbracket A'_1$	by substitution property (lemma 41)
$\Pi x{:}\llbracket\theta\rrbracket A_1.\llbracket\theta\rrbracket A_2 = \Pi x{:}\llbracket\theta\rrbracket A'_1.\llbracket\theta\rrbracket A'_2$	by substitution property
$\llbracket\theta\rrbracket A_1 = \llbracket\theta\rrbracket A'_1$	by syntactic equality
$\llbracket\llbracket\rho\rrbracket\theta_1\rrbracket A_1 = \llbracket\llbracket\rho\rrbracket\theta_1\rrbracket A'_1$ and $\llbracket\theta_1\rrbracket A_1 = \llbracket\theta_1\rrbracket A'_1$	by previous lines
$\llbracket\llbracket\rho\rrbracket\theta_1\rrbracket U'_1 = \llbracket\llbracket\rho\rrbracket\theta_1\rrbracket U'_2$	by previous lines
$\llbracket\rho\rrbracket(\llbracket\theta_1\rrbracket U'_1) = \llbracket\rho\rrbracket(\llbracket\theta_1\rrbracket U'_2)$	by substitution definition
$\Delta_1; \llbracket\theta_1\rrbracket\Gamma \vdash \llbracket\theta_1\rrbracket U'_1 \doteq \llbracket\theta_1\rrbracket U'_2/(\Delta_2, \theta_2)$	by i.h.
$\exists \rho'.\rho = \llbracket\rho'\rrbracket\theta_2$ and $\Delta' \vdash \rho' : \Delta_2$	
$\Delta; \Gamma \Vdash (R_1\ U'_1) \doteq (R_2\ U'_2)/(\Delta_2, \llbracket\theta_2\rrbracket\theta_1)$	by rule
$\theta = \llbracket\llbracket\rho'\rrbracket\theta_2\rrbracket\theta_1 = \llbracket\rho'\rrbracket(\llbracket\theta_2\rrbracket\theta_1)$	by substitution definition

Case $U_1 = u[\sigma]$ and u does not occur in U_2 By assumption, the type of U_2 must be atomic, that is, $U_2 = R_2$.

$\llbracket\theta\rrbracket(u[\sigma]) = \llbracket\theta\rrbracket R_2$	by assumption
$\theta = (\theta_1, R/u, \theta_2)$	by assumption
$\Delta = \Delta_1, u{::}\Psi{\vdash}Q, \Delta_2$	by assumption
$[\sigma]R = \llbracket\theta\rrbracket R_2$	by substitution definition
$R = [\sigma]^{-1}\,\llbracket\theta\rrbracket R_2$	by lemma 49
$\theta = (\theta_1, R/u, \theta_2) = (\theta_1, [\sigma]^{-1}\,\llbracket\theta\rrbracket R_2/u, \theta_2)$	by previous assumptions
$\Delta; \Gamma \vdash u[\sigma] : [\sigma]Q$	by assumption
$\Delta; \Gamma \vdash \sigma : \Psi$	by typing inversion
$\Delta' \vdash (\theta_1, [\sigma]^{-1}\,\llbracket\theta\rrbracket R_2/u, \theta_2) : \Delta$	by assumption
$\Delta'; \llbracket\theta_1\rrbracket\Psi \vdash [\sigma]^{-1}\,\llbracket\theta\rrbracket R_2 : \llbracket\theta_1\rrbracket Q$	by typing rules
$\Delta'; \llbracket\theta\rrbracket\Psi \vdash [\sigma]^{-1}\,\llbracket\theta\rrbracket R_2 : \llbracket\theta\rrbracket Q$	by weakening
$\Delta'; \llbracket\theta\rrbracket\Gamma \vdash \sigma : \llbracket\theta\rrbracket\Psi$	by modal substitution property
$\Delta'; \llbracket\theta\rrbracket\Gamma \vdash [\sigma][\sigma]^{-1}\,\llbracket\theta\rrbracket R_2 : [\sigma]\llbracket\theta\rrbracket Q$	by substitution property
$\Delta'; \llbracket\theta\rrbracket\Gamma \vdash \llbracket\theta\rrbracket R_2 : \llbracket\theta\rrbracket[\sigma]Q$	by lemma 44 and 48

$\Delta;\Gamma \vdash R_2 : [\sigma]Q$ by modal substitution property

$\Delta;\Gamma \vdash R_2 \mid [\sigma]^{-1} \Rightarrow (\Delta^*,\, \rho)$ and

$\exists \rho'.\, \theta = [\![\rho']\!]\rho$ and $\Delta' \vdash \rho' : \Delta^*$ by lemma 55

$\Delta^* \vdash \rho : \Delta$ by lemma 52

$\rho = (\rho_1, u[\mathsf{id}_{[\![\rho]\!]\Psi}]/u, \rho_2)$ and $\Delta^* = (\Delta_1^*, u{::}[\![\rho]\!]\Psi{\vdash}[\![\rho]\!]Q, \Delta_2^*)$ by lemma 53

$\Delta;\Gamma \Vdash u[\sigma] \doteq R_2 \;/\; ((\Delta_1^*, \Delta_2'),\, [\![\mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}([\![\rho]\!]R_2)/u, \mathsf{id}_{\Delta_2^*}]\!]\rho)$

with $\Delta_2' = [\![\mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}([\![\rho]\!]R_2)/u, \mathsf{id}_{\Delta_2^*}]\!]\Delta_2^*$ by rule

$\theta = (\theta_1, R/u, \theta_2) = (\theta_1, [\sigma]^{-1}([\![\theta]\!]R_2)/u, \theta_2)$ by previous lines

$= (\theta_1, [\sigma]^{-1}([\![[\![\rho']\!]\rho]\!]R_2)/u, \theta_2)$ by previous lines

$= (\theta_1, [\sigma]^{-1}([\![\rho']\!][\![\rho]\!]R_2)/u, \theta_2)$ by substitution definition

$= (\theta_1, [\![\rho']\!][\sigma]^{-1}([\![\rho]\!]R_2)/u, \theta_2)$ since by lemma 54 $[\sigma]^{-1}([\![\rho]\!]R_2)$ exists and by lemma 47, $[\sigma]^{-1}([\![\rho']\!][\![\rho]\!]R_2) = [\![\rho']\!][\sigma]^{-1}([\![\rho]\!]R_2)$

$= [\![\rho']\!]\rho$

$= [\![\rho']\!](\rho_1, u[\mathsf{id}_{[\![\rho]\!]\Psi}]/u, \rho_2)$

$= ([\![\rho']\!]\rho_1, [\![\rho']\!](u[\mathsf{id}_{[\![\rho]\!]\Psi}])/u, [\![\rho']\!]\rho_2)$ by substitution definition

Since $(\theta_1, [\![\rho']\!][\sigma]^{-1}([\![\rho]\!]R_2)/u, \theta_2) = ([\![\rho']\!]\rho_1, [\![\rho']\!](u[\mathsf{id}_{[\![\rho]\!]\Psi}])/u, [\![\rho']\!]\rho_2)$,

we must have the following:

$[\![\rho']\!](u[\mathsf{id}_\Psi]) = [\![\rho']\!]([\sigma]^{-1}([\![\rho]\!]R_2))$ and by previous lines

$\rho' = (\rho_1', [\![\rho']\!]([\sigma]^{-1}([\![\rho]\!]R_2))/u, \rho_2')$ by previous lines

$= [\![\rho_1', \rho_2']\!](\mathsf{id}_{\Delta_1^*}, ([\sigma]^{-1}([\![\rho]\!]R_2))/u, \mathsf{id}_{\Delta_2^*})$ by substitution definition and the fact that u does not occur in R itself

$\theta = [\![[\![\rho_1', \rho_2']\!](\mathsf{id}_{\Delta_1^*}, ([\sigma]^{-1}([\![\rho]\!]R_2))/u, \mathsf{id}_{\Delta_2^*})]\!]\rho$ by previous lines

$= [\![\rho_1', \rho_2']\!]([\![\mathsf{id}_{\Delta_1^*}, ([\sigma]^{-1}([\![\rho]\!]R_2))/u, \mathsf{id}_{\Delta_2^*}]\!]\rho)$ by substitution definition

Case $U_1 = u[\sigma]$ and $U_2 = u[\tau]$

$\Delta;\Gamma \vdash u[\sigma] : Q'$ by assumption

$\Delta;\Gamma \vdash \sigma : \Psi$ and $\Delta;\Gamma \vdash Q' \equiv [\sigma]Q : \mathsf{type}$ by typing inversion lemma 12

$\Delta = \Delta_1, u{::}\Psi{\vdash}Q, \Delta_2$

$\Delta;\Gamma \vdash u[\tau] : Q'$ by assumption

$\Delta;\Gamma \vdash \tau : \Psi$ and

$\Delta;\Gamma \vdash Q' \equiv [\tau]Q : \mathsf{type}$ by typing inversion lemma 12

$\Delta; \Gamma \vdash [\sigma]Q \equiv [\tau]Q : \mathsf{type}$ by transitivity
$\Delta; \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi_2$ by lemma 58
$(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \Vdash u[\sigma] \doteq u[\tau] \; / \; ((\Delta_1, v{::}\Psi_2{\vdash}Q, \Delta_2'), \; (\mathsf{id}_{\Delta_1}, v[\mathsf{id}_{\Psi_2}]/u, \mathsf{id}_{\Delta_2}))$
with $\Delta_2' = [\![\mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v]\!]\Delta_2$ by rule
$\theta = (\theta_1, U/u, \theta_2)$ and $\Delta; \Psi \vdash U : Q$ by assumption
$[\![\theta]\!](u[\sigma]) = [\![\theta]\!](u[\tau])$ by assumption
$[\sigma]U = [\tau]U$ by substitution definition
$([\mathsf{id}_{\Psi_2}]U) = U$ by definition
let be $\rho' = (\theta_1, U/v, \theta_2)$
$\theta = [\![\rho']\!](\mathsf{id}_{\Delta_1}, v[\mathsf{id}_{\Psi_2}]/u, \mathsf{id}_{\Delta_2})$

□

The main efficiency problem in pattern unification lies in treating the case for existential variables. In particular, we must traverse the term U. First of all, we must perform the occurs-check to prevent cyclic terms. Second, we may need to prune the substitutions associated with existential variables occurring in U. Third, we need to ensure that all bound variables occurring in U do occur in the range of σ, otherwise $[\sigma]^{-1}U$ does not exist.

In the next section, we will show how linearization can be used to enforce the two criteria which eliminates the need to traverse U. First, we will enforce that all existential variables occur only once, thereby eliminating the occurs-check. Second, we will require that the substitution σ associated with existential variables is always a permutation π, thereby restricting higher-order patterns further. This ensures that the substitutions are always invertible and eliminates the need for pruning. Restricting higher-order patterns even further to patterns where existential variables must be applied to all bound variables has also been used by Hanus and Prehofer [26] in the context of simply-typed higher-order functional logic programming. While Hanus and Prehofer syntactically disallow terms which are not fully applied, we translate any term into a linear higher-order pattern thereby factoring out and delaying subterms which are not applied to all bound variables. The resulting assignment algorithm is a specialization of the presented higher-order pattern unification algorithm.

3.2 Rules for higher-order pattern unification

$$\frac{\Delta; \Gamma, x{:}A \vdash U_1 \;\doteq\; U_2 \;/\; (\Delta', \theta)}{\Delta; \Gamma \vdash \lambda x{:}A.U_1 \;\doteq\; \lambda x{:}A.U_2 \;/\; (\Delta', \theta)}\; lam \qquad \frac{\Delta; \Gamma \Vdash R_1 \doteq R_2 \;/\; (\Delta', \theta)}{\Delta; \Gamma \vdash R_1 \;\doteq\; R_2 \;/\; (\Delta', \theta)}\; coerce$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash R \mid\; [\sigma]^{-1} \Rightarrow (\Delta^*, \rho) \qquad \Delta^* \;=\; (\Delta_1^*, u{::}\llbracket\rho\rrbracket\Psi{\vdash}\llbracket\rho\rrbracket Q, \Delta_2^*) \\ u \text{ does not occur in } R \qquad \Delta_2' = \llbracket \mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}\,(\llbracket\rho\rrbracket R)/u, \mathsf{id}_{\Delta_2^*}\rrbracket \Delta_2^*\end{array}}{(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \Vdash u[\sigma] \;\doteq\; R \;/\; ((\Delta_1^*, \Delta_2'),\; \llbracket \mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}\,(\llbracket\rho\rrbracket R)/u, \mathsf{id}_{\Delta_2^*}\rrbracket \rho)}\; ex1$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash R \mid\; [\sigma]^{-1} \Rightarrow (\Delta^*, \rho) \qquad \Delta^* \;=\; (\Delta_1^*, u{::}\llbracket\rho\rrbracket\Psi{\vdash}\llbracket\rho\rrbracket Q, \Delta_2^*) \\ u \text{ does not occur in } R \qquad \Delta_2' = \llbracket \mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}\,(\llbracket\rho\rrbracket R)/u, \mathsf{id}_{\Delta_2^*}\rrbracket \Delta_2^*\end{array}}{(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \Vdash R \;\doteq\; u[\sigma] \;/\; ((\Delta_1^*, \Delta_2'),\; \llbracket \mathsf{id}_{\Delta_1^*}, [\sigma]^{-1}\,(\llbracket\rho\rrbracket R)/u, \mathsf{id}_{\Delta_2^*}\rrbracket \rho)}\; ex2$$

$$\frac{(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \vdash \tau \cap \sigma : \Psi \Rightarrow \Psi_2 \quad \Delta_2' = \llbracket \mathsf{id}_{\Delta_1}, v'[\mathsf{id}_{\Psi_2}]/v\rrbracket \Delta_2}{(\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2); \Gamma \Vdash u[\sigma] \;\doteq\; u[\tau] \;/\; ((\Delta_1, v{::}\Psi_2{\vdash}Q, \Delta_2'),\; (\mathsf{id}_{\Delta_1}, v[\mathsf{id}_{\Psi_2}]/u, \mathsf{id}_{\Delta_2}))}\; ex3$$

$$\frac{\Delta; \Gamma \Vdash R_1 \;\doteq\; R_2 \;/\; (\Delta_1, \theta_1) \quad \Delta_1; \llbracket\theta_1\rrbracket\Gamma \vdash \llbracket\theta_1\rrbracket U_1 \;\doteq\; \llbracket\theta_1\rrbracket U_2 \;/\; (\Delta_2, \theta_2)}{\Delta; \Gamma \Vdash R_1\, U_1 \;\doteq\; R_2\, U_2 \;/\; (\Delta_2, \llbracket\theta_1\rrbracket(\theta_2))}\; app$$

$$\frac{}{\Delta; \Gamma \Vdash x \;\doteq\; x \;/\; (\Delta, \mathsf{id}_\Delta)}\; var \qquad \frac{}{\Delta; \Gamma \Vdash c \;\doteq\; c \;/\; (\Delta, \mathsf{id}_\Delta)}\; const$$

3.3 Linearization

One critical optimization in unification is to perform the occurs-check only when necessary. While unification with the occurs-check is at best linear in the sum of the sizes of the terms being unified, unification without the occurs-check is linear in the smallest term being unified. In fact the occurs-check can be omitted if the terms are linear, i.e., every existential variable occurs only once.

Let us consider a program which evaluates expressions from a small functional language Mini-ML. First, we give the grammar for expressions.

$$\text{expressions } e \;\;::=\;\; \mathsf{z} \mid \mathsf{s}\; e \mid \mathsf{lam}\; x.e \mid (\mathsf{app}\; e_1\; e_2) \mid \ldots$$

To describe that expression e evaluates to some value v, we use the following judgment: $e \hookrightarrow v$.

$$\frac{}{\mathsf{z} \hookrightarrow \mathsf{z}} \qquad \frac{e \hookrightarrow v}{(\mathsf{s}\ e) \hookrightarrow (\mathsf{s}\ v)} \qquad \frac{}{\mathsf{lam}\ x.e \hookrightarrow \mathsf{lam}\ x.e}$$

$$\frac{e_1 \hookrightarrow \mathsf{lam}\ x.e' \qquad e_2 \hookrightarrow v_2 \qquad [v_2/x]e' \hookrightarrow v}{\mathsf{app}\ e_1\ e_2 \hookrightarrow v}$$

The grammar of expressions and the evaluation judgment are represented in pure LF the following way. For a more detailed general introduction, we refer the reader to [49].

$$\begin{array}{l}
\mathsf{exp} : \mathsf{type}.\\
\mathsf{z} : \mathsf{exp}.\\
\mathsf{s}\ : \mathsf{exp} \rightarrow \mathsf{exp}.\\
\mathsf{app}\ : \mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp}.\\
\mathsf{lam} : (\mathsf{exp} \rightarrow \mathsf{exp}) \rightarrow \mathsf{exp}.\\
\\
\mathsf{eval} : \mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{type}.
\end{array}$$

Next, we represent the evaluation inference rules in pure LF. We concentrate here on the rule for functions.

$$\mathsf{ev\_lam} : \Pi e{:}(\mathsf{exp} \rightarrow \mathsf{exp}).\mathsf{eval}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e\ x))\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e\ x)).$$

The variable e is quantified outside with a Π. When executing this signature as a logic program, we view the variable e as a logic (existential) variable, which will be instantiated using unification during the execution of the logic program. A more detailed description of the operational semantics and interpreting LF types as logic programs is given later in Chapter 4. Here we just note that existential variables arise during logic programming execution and need to be instantiated via unification. Note that e has function type $\mathsf{exp} \rightarrow \mathsf{exp}$. The instantiation for e will be some λ-abstraction where x needs to be applied to. β-reduction needs to be applied to reduce the resulting expression.

The modal term language introduced earlier in Chapter 2, allows a more compact description.

$$\mathsf{ev\_lam} : \Pi^{\Box} e{::}(y{:}\mathsf{exp}{\vdash}\mathsf{exp}).\mathsf{eval}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e[x/y]))\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e[x/y])).$$

The existential variable e in the clause ev_lam is quantified by $\Pi^{\Box} e{::}(y{:}\mathsf{exp}{\vdash}\mathsf{exp})$. Note that here the variable e is already lowered and has atomic type. We can now use just first-order replacement (or grafting) to instantiate the variable e. To enforce that every existential variable occurs only once, the clause head of ev_lam can be translated into the linear type

$$\Pi^{\Box} e{::}(y{:}\mathsf{exp}{\vdash}\mathsf{exp}).\Pi^{\Box} e'{::}(z{:}\mathsf{exp}{\vdash}\mathsf{exp}).\mathsf{eval}\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e[x/y]))\ (\mathsf{lam}\ (\lambda x{:}\mathsf{exp}.e'[x/z]))$$

together with the variable definition

$$\forall x{:}\mathsf{exp}.e'[x/z] \overset{D}{=} e[x/y])$$

where e' is a new existential variable. Variable definitions are defined as follows:

$$\text{Residuals defining variables} \quad D \quad ::= \quad \mathsf{true} \mid u[\mathsf{id}_{\Psi}] \overset{D}{=} U \mid D_1 \wedge\ D_2 \mid \forall x{:}A.D$$

Then we can use a constant time assignment algorithm for assigning a linear clause head to a goal, and the residuating variable definitions are solved later by conventional unification. As a result, the occurs-check is only performed if necessary. Note that the $\Pi^{\Box}$-quantifier will only appear at the outside, and we can view the clause in $\Pi^{\Box}$-form as the compiled form of the original pure LF clause.

In the dependently typed lambda-calculus, there are several difficulties in performing this optimization. First of all, all existential variables carry their context Ψ and type A. If we introduce a new existential variable, then the question arises what type should be assigned to it. As type inference is undecidable in the dependently typed case, this may be expensive. In general, we may even obtain a term which is not necessarily well-typed.

To illustrate, let us modify the previous example, and annotate the expressions with their type thus enforcing that any evaluation of a Mini-ML expression will be well-typed. First, we give the grammar for Mini-ML types which includes the natural numbers nat and the function type $t_1 \rightarrow t_2$.

$$\text{Mini-ML types } t \quad ::= \quad \mathsf{nat} \mid t_1 \rightarrow t_2$$

Next, we give typing rules for Mini-ML expressions. The main typing judgment is the following:

$$\Gamma \vdash e : t$$

which means the Mini-ML expression e has Mini-ML type t in the typing context Γ. The typing rules are straightforward:

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathsf{nat}} \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash (\mathsf{s}\ e) : \mathsf{nat}}$$

$$\frac{\Gamma, x{:}t_1 \vdash e : t_2}{\Gamma \vdash \mathsf{lam}\ x.e : t_1 \rightarrow t_2} \qquad \frac{\Gamma \vdash e_1 : t_2 \rightarrow t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (\mathsf{app}\ e_1\ e_2) : t}$$

To allow for Mini-ML types also in our LF specification, we declare a type family tp for Mini-ML types, which has as elements the Mini-ML type nat and $\rightarrow$.

$$\begin{aligned}&\mathsf{tp} : \mathsf{type}.\\ &\mathsf{nat} : \mathsf{tp}.\\ &\mathsf{arr} : \mathsf{tp} \rightarrow \mathsf{tp} \rightarrow \mathsf{tp}.\end{aligned}$$

Instead of implementing the typing rules for Mini-ML separately to the evaluation rules, we annotate the expressions with their corresponding Mini-ML types by using the dependent types available in LF. The dependent types then ensures that any evaluation of Mini-ML expressions is well-typed. This has the advantage that a type-preservation theorem does not need to be established separately. We proceed as follows. The type family exp for Mini-ML expressions is indexed by Mini-ML types tp.

$$\begin{aligned}&\mathsf{exp} : \mathsf{tp} \rightarrow \mathsf{type}.\\ &\mathsf{z} : \mathsf{exp}\ \mathsf{nat}.\\ &\mathsf{s} : \mathsf{exp}\ \mathsf{nat} \rightarrow \mathsf{exp}\ \mathsf{nat}.\\ &\mathsf{app} : \Pi t_1{:}\mathsf{tp}.\Pi t_2{:}\mathsf{tp}.\mathsf{exp}\ (\mathsf{arr}\ t_1\ t_2) \rightarrow \mathsf{exp}\ t_1 \rightarrow \mathsf{exp}\ t_2.\\ &\mathsf{lam} : \Pi t_1{:}\mathsf{tp}.\Pi t_2{:}\mathsf{tp}.((\mathsf{exp}\ t_1) \rightarrow (\mathsf{exp}\ t_2)) \rightarrow (\mathsf{exp}\ (\mathsf{arr}\ t_1\ t_2)).\end{aligned}$$

The LF-types of the constants z, s , app , and lam directly encode the previous Mini-ML typing rules and ensure that only expressions which are well-typed according to the Mini-ML typing rules can be stated. In the modal LF calculus, the LF-types for app and lam are written as follows.

$$\begin{array}{ll} \mathsf{app} & : \Pi^{\Box} t_1{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_2{::}(\cdot \vdash \mathsf{tp}).\mathsf{exp}\ (\mathsf{arr}\ t_2\ t) \to \mathsf{exp}\ t_2 \to \mathsf{exp}\ t. \\ \mathsf{lam} & : \Pi^{\Box} t_1{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_2{::}(\cdot{\vdash}\mathsf{tp}).((\mathsf{exp}\ t_2) \to (\mathsf{exp}\ t)) \to (\mathsf{exp}\ (\mathsf{arr}\ t_2\ t)). \end{array}$$

In the following discussion, we will directly write the LF signatures using the $\Pi^{\Box}$-notation. The evaluation judgment, now takes an expression of Mini-ML type t and returns an expression of the same type. This can be enforced using dependent types of LF as follows:

$$\mathsf{eval} : \Pi^{\Box} t{::}(\cdot{\vdash}\mathsf{tp}).\mathsf{exp}\ t\ \to \mathsf{exp}\ t\ \to \mathsf{type}.$$

Next, we reformulate the evaluation rule for lambda-expressions give it in fully explicit form.

$$\begin{array}{l} \mathsf{ev\_lam} : \Pi^{\Box} t_1{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_2{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} e{::}(y{:}\mathsf{exp}\ t_1{\vdash}\mathsf{exp}\ t_2). \\ \qquad \mathsf{eval}\ (\mathsf{arr}\ \ t_1\ t_2)\ (\mathsf{lam}\ t_1\ t_2\ (\lambda x{:}\mathsf{exp}\ t_1.e[x/y]))\ (\mathsf{lam}\ t_1\ t_2\ (\lambda x{:}\mathsf{exp}\ t_1.e[x/y])). \end{array}$$

Note that the constructor lam is indexed by the Mini-ML types t_1 and t_2. Similarly eval is indexed by the Mini-ML type $(\mathsf{arr}\ \ t_1\ t_2)$. Due to these dependencies, we now have multiple occurrences of t_1 and t_2, in addition to the duplicate occurrence of e. During linearization, the clause head of $\mathsf{ev\_lam}$ will now be translated into

$$\begin{array}{l} \Pi^{\Box} t_1{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_2{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} e{::}(y{:}\mathsf{exp}\ t_1{\vdash}\mathsf{exp}\ t_2). \\ \Pi^{\Box} t_3{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_4{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_5{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_6{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} e'{::}(z{:}\mathsf{exp}\ t_1{\vdash}\mathsf{exp}\ t_2). \\ \mathsf{eval}\ (\mathsf{arr}\ \ t_1\ t_2)(\mathsf{lam}\ t_3\ t_4\ (\lambda x{:}\mathsf{exp}\ t_1.e[x/y]))\ (\mathsf{lam}\ t_5\ t_6\ (\lambda x{:}\mathsf{exp}\ t_1.e'[x/z])). \end{array}$$

and the following residuals for defining variables

$$t_1 \overset{D}{=} t_3 \wedge\ t_1 \overset{D}{=} t_5 \wedge\ t_2 \overset{D}{=} t_4 \wedge\ t_2 \overset{D}{=} t_6 \wedge\ \forall x{:}\mathsf{exp}\ t_1.e'[x/z] \overset{D}{=} e[x/y]$$

Note, that due to the linearization, the linear clause head is clearly not well-typed. However, the clause head is well-typed after carrying out the substitution

$$[\![t_1/t_3,\ t_1/t_5,\ t_2/t_4,\ t_2/t_6, e[z/y]/e']\!]$$

This relies on the fact, that we can find an appropriate ordering of the existential variables in Δ such that dependencies among existential variables and variable definitions are obeyed. It is worth pointing out that these complications are due to dependent types in our framework and do not arise in the simply typed case. Note that some of these variable definitions are in fact redundant, which is another orthogonal optimization (see [43] for an analysis on a fragment of LF).

Another approach would be to interpret the new existential variables modulo variable definitions, by introducing a new quantifier $\Pi^{\Box} u = U{::}\Psi{\vdash}A$ which corresponds to the variable definition $u[\mathsf{id}_\Psi] \overset{D}{=} U$. This would allow us to type-check clauses which may contain variable definitions directly, without applying the variable definitions. Using this new notation the linear clause head becomes

$$\begin{array}{l}
\Pi^{\Box} t_1{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_2{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} e{::}(y{:}\mathsf{exp}\ t_1{\vdash}\mathsf{exp}\ t_2).\\
\Pi^{\Box} t_3 = t_1{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_4 = t_2{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_5 = t_1{::}(\cdot{\vdash}\mathsf{tp}).\Pi^{\Box} t_6 = t_2{::}(\cdot{\vdash}\mathsf{tp}).\\
\Pi^{\Box} e' = e[z/y]{::}(z{:}\mathsf{exp}\ t_1{\vdash}\mathsf{exp}\ t_2).\\
\mathsf{eval}\ (\mathsf{arr}\ \ t_1\ t_2)(\mathsf{lam}\ t_3\ t_4\ (\lambda x{:}\mathsf{exp}\ t_1.e[x/y]))\ (\mathsf{lam}\ t_5\ t_6\ (\lambda x{:}\mathsf{exp}\ t_1.e'[x/z])).
\end{array}$$

Here, we do not pursue this idea further, since the variable definitions are treated in a very special manner in the implementation.

The idea of factoring out duplicate existential variables can be generalized to replacing arbitrary subterms by new existential variables and creating variable definitions. In particular, the process of linearization also replaces any existential variables $v[\sigma]$ where σ is *not* a permutation by a new variable $u[\mathsf{id}_\Psi]$ and a variable definition $u[\mathsf{id}_\Psi] \overset{D}{=} v[\sigma]$.

Thus, our linear term language can be characterized as follows:

$$\begin{array}{lrcl}
\text{Normal Linear Objects} & L & ::= & \lambda x{:}A.\ L \mid u[\pi_\Gamma] | P\\
\text{Neutral Linear Objects} & P & ::= & c \mid x \mid P\, L
\end{array}$$

We assume that all normal linear objects are in canonical form and all existential variables are of atomic type. As shown in Chapter 2, this can be achieved by lowering and raising operations which are justified by lemma 40. The linearization itself is quite straightforward and we will omit the details here. In the actual implementation, we do not generate types A and contexts Ψ for the new, linearly occurring existential variables, but ensure that all such variables are instantiated and disappear by the time the variable definitions have been solved.

3.4 Assignment for higher-order patterns

In this section, we give a refinement of the general higher-order pattern unification which exploits the presented ideas. The algorithm proceeds in three phases. First we will unify a linear atomic higher-order pattern L with a normal object U. Note that U may not be linear and does not refer to any variable definitions. During the first phase, we decompose the objects L and U and assign terms to the existential variables occurring in L. The following judgments capture the assignment between a linear atomic higher-order pattern L and a normal object U. We write θ for simultaneous substitutions $[\![U_1/u_1, \ldots U_n/u_n]\!]$ for existential variables.

$$\begin{array}{ll} \Delta; \Gamma \vdash L \doteq U/(\theta, \Delta', E) & \text{assignment for normal objects} \\ \Delta; \Gamma \Vdash P \doteq R/(\theta, \Delta', E) & \text{assignment for atomic objects} \end{array}$$

where P is a linear neutral object and L is a normal linear object. In contrast, U is a normal object and R is a neutral object, both of which are not necessarily linear. Similar to higher-order pattern unification, the assignment algorithm returns a substitution θ and a modal context Δ'. In addition, we may generate some residual equations E. As discussed earlier, linearization of a term may in general lead to linear term which is not necessarily dependently typed. However, the term is still approximately well-typed when all the dependencies among types are erased (for more on erasing types see 2). Here, we assume that both L (resp. P) and U (resp. R) are approximately well-typed in context Δ and Γ, i.e. when we erase the type-dependencies then L^- (resp. P^-)is approximately well-typed in the erased modal context Δ^- and erased context Γ^-. Moreover, approximate typing invariants are preserved during linear higher-order pattern unification.

There are two possible kinds of residual equations we need to solve after assignment succeeds, residuals due to variable definitions and residuals due to the fact that one of the terms in the unification is non-linear. So in addition to the residuals for defining variables, we define the residuals E as follows:

$$\text{Residual Equation} \quad E \ := \ \mathsf{true} \mid U_1 \ \doteq \ U_2 \mid E_1 \wedge \ E_2 \mid \forall x{:}A.E$$

The assignment algorithm itself is given below is a specialization of the previous higher-order pattern unification algorithm.

$$\frac{\Delta; \Gamma, x{:}A \vdash L \doteq U \;/\; (\theta, \Delta', E)}{\Delta; \Gamma \vdash \lambda x{:}A.L \doteq \lambda x{:}A.U \;/\; (\theta, \Delta', \forall x{:}A.E)}\; lam$$

$$\frac{\Delta; \Gamma \Vdash P \doteq R \;/\; (\theta, \Delta, E)}{\Delta; \Gamma \vdash P \doteq R \;/\; (\theta, \Delta, E)}\; coerce$$

$$\frac{\Delta = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2) \quad \Delta_2' = [\![\mathsf{id}_{\Delta_1}, [\pi]^{-1}\, R/u]\!]\Delta_2}{\Delta; \Gamma \vdash u[\pi] \doteq R \;/\; ((\mathsf{id}_{\Delta_1}, [\pi]^{-1}\, R/u, \mathsf{id}_{\Delta_2}), (\Delta_1, \Delta_2'), \mathsf{true})}\; existsL$$

$$\frac{\Delta = (\Delta_1, u{::}\Psi{\vdash}Q, \Delta_2)}{\Delta; \Gamma \vdash L \doteq u[\sigma] \;/\; (\mathsf{id}_{\Delta}, \Delta, L \;\doteq\; u[\sigma])}\; existsR$$

$$\frac{}{\Delta; \Gamma \Vdash x \doteq x \;/\; (\mathsf{id}_{\Delta}, \Delta, \mathsf{true})}\; const \qquad \frac{}{\Delta; \Gamma \Vdash c \doteq c \;/\; (\mathsf{id}_{\Delta}, \Delta, \mathsf{true})}\; var$$

$$\frac{\Delta; \Gamma \Vdash P \doteq R \;/\; (\theta_1, \Delta_1, E_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \vdash [\![\theta_1]\!]L \doteq [\![\theta_1]\!]U \;/\; (\theta_2, \Delta_2, E_2)}{\Delta; \Gamma \Vdash P\; L \doteq R\; U \;/\; ([\![\theta_2]\!]\theta_1,\; \Delta_2, E_1 \wedge\; E_2)}\; app$$

Note again that we do not need to worry about capture in the rule *lam*, since existential variables and bound variables are defined in different contexts. In the rule *app*, we give the same rule as previously. Applying θ_1 to the second premise is only necessary to maintain the invariant that the resulting substitution θ_1 maps all existential variables from Δ to some new context Δ_1. Since all modal variables in the linear term occur uniquely, θ_1 only instantiates modal variables from P and has no "effect" on the object L. Note that the case for unifying an existential variable $u[\pi]$ with another term R is now simpler and more efficient than in the general higher-order pattern case. In particular, it does not require a traversal of R (see rule *existsL*). Since the inverse of the substitution π can be computed directly and will be total, we know $[\pi]^{-1}\, R$ exists and can simply generate a substitution $[\pi]^{-1}\, R/u$. Finally, we may need to postpone solving some unification problems and generate some residual equations if the non-linear term is an existential variable (see *existsR*).

The rules for solving the residual equations are straightforward. First, the rules to solve the remaining variable definitions.

$$\Delta; \Gamma \vdash D/(\Delta', \theta)$$

The variable definitions D are well-typed in the context Γ and modal context Δ.

If the variable definitions can be solved, then a modal substitution θ is returned, s.t. $\Delta' \vdash \theta : \Delta$. Note, that we assume that the modal substitution θ_1, which is the result of the previous assignment algorithm, has been already applied, when we use the following rules to solve the residual variable definitions.

$$\frac{\Delta; \Gamma, x{:}A \Vdash D/(\Delta', \theta)}{\Delta; \Gamma \Vdash \forall x{:}A.D/(\Delta', \theta)} \qquad \frac{\Delta; \Gamma \Vdash D_1/(\Delta_1, \theta_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \Vdash [\![\theta_1]\!]D_2/(\Delta_2, \theta_2)}{\Delta; \Gamma \Vdash D_1 \wedge\ D_2/(\Delta_2, [\![\theta_2]\!](\theta_1)}$$

$$\frac{\Delta_2' = [\![\mathsf{id}_{\Delta_1}, U/u]\!]\Delta_2}{\Delta_1, u{::}(\Gamma{\vdash}A), \Delta_2; \Gamma \Vdash u[\mathsf{id}_\Gamma] \overset{D}{=} U/(\Delta_1, \Delta_2'),\ (\mathsf{id}_{\Delta_1}, U/u, \mathsf{id}_{\Delta_2}))} (*)$$

$$\frac{\Delta; \Gamma \vdash U' \ \doteq\ U/(\Delta', \theta)}{\Delta; \Gamma \Vdash U' \overset{D}{=} U/(\Delta', \theta)}$$

$$\frac{}{\Delta; \Gamma \Vdash \mathsf{true}/(\Delta, \mathsf{id}_\Delta)}$$

Solving variable definitions is then straightforward. The only case we need to pay attention is the rule (*). This case may arise if $u[\mathsf{id}_\Gamma]$ is a modal variable which has not been instantiated yet during assignment. In this case, we simply assign u the object U. This ensures that after the variable definitions are solved, no modal variables which are used in variable definitions are left uninstantiated. In other words, all the intermediate modal variables which have been introduced during linearization are now instantiated.

Next, we show the rules for solving the residual equations, which have been generated during assignment. The algorithm is straightforward and very similar to solving variable definitions, except that no special provisions have to be made.

$$\Delta; \Gamma \Vdash^{e} E\ /\ (\Delta', \theta)$$

The residual equations E are well-typed in the context Γ and modal context Δ. If the residual equations can be solved, then a modal substitution θ is returned, s.t.

$\Delta' \vdash \theta : \Delta$.

$$\frac{\Delta; \Gamma, x{:}A \Vdash E/(\Delta', \theta)}{\Delta; \Gamma \Vdash \forall x{:}A.E/(\Delta', \theta)} \quad \frac{\Delta; \Gamma \Vdash E_1/(\Delta_1, \theta_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \Vdash [\![\theta_1]\!]E_2/(\Delta_2, \theta_2)}{\Delta; \Gamma \Vdash E_1 \wedge\ E_2/(\Delta_2, [\![\theta_2]\!](\theta_1))}$$

$$\frac{\Delta; \Gamma \vdash L \ \doteq\ U/(\Delta', \theta)}{\Delta; \Gamma \Vdash L \ \doteq\ U/(\Delta', \theta)} \quad \frac{}{\Delta; \Gamma \Vdash \mathsf{true}/(\Delta, \mathsf{id}_\Delta)}$$

The overall algorithm can be summarized as follows:

1. Higher-order assignment: $\Delta; \Gamma \vdash L \doteq U/(\theta_1, \Delta_1, E)$
2. Solving variable definitions: $\Delta_1; \cdot \vdash [\![\theta_1]\!]D/(\theta_2, \Delta_2)$
3. Solving residual equations: $\Delta_2; \cdot \vdash [\![[\![\theta_2]\!]\theta_1]\!]E/(\Delta_3, \theta_3)$

The final result of the overall algorithm is the modal substitution $[\![\theta_3]\!]([\![[\![\theta_2]\!]\theta_1]\!])$. For the higher-order pattern fragment, we can now show that assignment algorithm is sound and complete for linear higher-order patterns. In principle, it is possible to generalize these statements further to include variable definitions and residual equations E and show that linearization together with linear higher-order pattern unification is sound and complete. This would requires a more precise description of the linearization process and how variable definitions can be eliminated. In particular the completeness theorem would be more complicated

3.5 Experiments

We have implemented the presented algorithm as part of the Twelf system. The instantiation is applied eagerly by using destructive updates. As mentioned before, lowering and grafting are also done eagerly using destructive updates. In this section, we discuss some experimental results with different programs written in Twelf. All experiments are done on a machine with the following specifications: 1.60GHz Intel Pentium Processor, 256 MB cache. We are using SML of New Jersey 110.0.3 under Linux Red Hat 7.1. Times are measured in seconds. In the tables below, the column "opt" refers to the optimized version with linearization and assignment, while the column "stand" refers to the standard implementation using general higher-order pattern unification.

The column "trivial" indicates the percentage of problems which are already linear and have no residuating variable definitions. The column "fail" indicates how many variable definitions were not solvable, if there were any. The last column gives the overall success rate with respect to the total number of calls to assignment made. The column "time reduction" refers to how much time is saved. This is calculated as follows: $(time_{old} - time_{new})/time_{old}$. In addition, we give the factor of improvement, which can be calculated $time_{old}/time_{new}$[3].

3.5.1 Higher-order logic programming

In this section, we present two experiments with higher-order logic programming. The first one uses an implementation of a meta-interpreter for ordered linear logic by Polakow and Pfenning [58]. In the second experiment we evaluate our unification algorithm using an implementation of foundational proof-carrying code developed at Princeton University [2].

Meta-interpreter for ordered linear logic

example	opt	stand	reduction in time	improve	variable def trivial	variable def fail	assign succeed
sqnt (bf)	0.84	2.09	60%	2.49	44%	46%	23%
sqnt (dfs)	0.93	2.35	60%	2.52	44%	47%	22%
sqnt (perm)	4.44	7.11	38%	1.60	44%	52%	20%
sqnt (rev)	1.21	1.70	29%	1.40	45%	48%	21%
sqnt (mergesort)	2.26	3.39	33%	1.50	46%	53%	20%

As the results for the meta-interpreter demonstrate, performance is improved by to a factor of 2.5. Roughly, 45% of the time there were no variable definitions at all. From the non-trivial equations roughly 45% were not unifiable. This means overall, in approx. 20% - 30% of the cases the assignment algorithm succeeded and the failure of unification was delayed. It is worth noting that 77% to 80% of the calls to assignment presented in Section 3.4 fail immediately.

[3]Note that from the factor of improvement one can compute the time-reduction automatically. If the factor of improvement is k then this results in $1 - (1/k)$ reduction in time, since $(time_{old} - time_{new})/time_{old} = 1 - (time_{new}/time_{old}) = 1 - (1/k)$.

Recently, Twelf has been applied to mobile code security in the foundational proof-carrying code project at Princeton [3]. To provide guarantees about the behavior of mobile code, programs are equipped with a certificate (proof) that asserts certain safety properties. These safety properties are represented as higher-order logic programs. Twelf's logic programming interpreter then executes the specification and generates a certificate (proof) that a given program fulfills a specified safety policy. We have had the opportunity to evaluate the presented assignment algorithm using programs from the proof-carrying code benchmark provided by Andrew Appel's group at Princeton University.

Foundational proof-carrying code

example	opt	stand	reduction in time	improve	variable def trivial	variable def fail	assign succeed
inc	5.8	9.19	36.7%	1.58	64%	46%	18%
switch	36.00	49.69	27.54%	1.38	64%	48%	19%
mul2	5.51	9.520	42.86%	1.72	64%	46%	18%
div2	121.96	153.610	20.63%	1.26	63%	48%	20%
divx	333.69	1133.150	70.50%	3.39	63%	50%	21%
listsum	1073.33	∞	∞	∞	65%	45%	18%
polyc	2417.85	∞	∞	∞	65%	41%	17%
pack	197.07	1075.610	81.65%	5.45	66%	45%	19%

In the table above we show the performance on programs from the proof-carrying code benchmark. Performance is improved by up to a factor of 5.45 and some examples are not executable without linearization and assignment. The results clearly demonstrate that an efficient unification algorithm is critical in large-scale examples.

There are several reasons for this substantial performance improvement. 1) We reduced the need for trailing instantiations of meta-variables in the implementation. As all meta-variables in the clause head have been created since the last choice point, no trailing should be necessary. As a consequence, less space is needed for storing instantiation of variables. However, to enforce this trailing policy in a high-level implementation this may require additional bookkeeping mechanism. The presented simple assignment algorithm opened another possibility to reduce the need for trailing without modifying the actual trailing mechanism. 2) We not only factor out duplicate meta-variables, but also expressions which call external constraint solvers. Calling the external constraint solvers may be expensive, especially if unification fails at a later point anyway. Factoring out constraint expressions and delaying their solution, allows us to fail quickly and disregard a clause as not applicable. As over 80% of the clauses tried fail, this is particularly important.

Remark: There are several other standard optimizations possible such as for example first-argument indexing. Maybe surprisingly, first-argument indexing did not improve performance. Although the number of calls to assignment were reduced by roughly 10%, this did not seem to influence the overall performance. One reason we found was that unification and weak head normal form dominate the run-time dramatically.

3.5.2 Higher-order theorem proving

Besides a logic programming engine, the Twelf system also provides a theorem prover which is based on iterative deepening search. In this section, we consider two examples, theorem proving in an intuitionistic sequent calculus and theorem proving in the classical natural deduction calculus.

As the results demonstrate, the performance of the theorem prover is not greatly influenced by the optimized unification algorithm. The main reason is that we have many dynamic assumptions, which need to be unified with the current goal. However, we use the standard higher-order pattern unification algorithm for this operation and use the optimized algorithm only for selecting a clause. For dynamic assumptions we cannot maintain the linearity requirement and linearizing the dynamic assumptions at run-time seems too expensive.

Proof search in the intuitionistic sequent calculus

example	opt	stand	reduction	improve	variable def		assign
			in time		trivial	fail	success
dist-1	53.00	57.11	7%	1.08	100%	0%	52%
distImp	0.40	0.44	9%	1.10	100%	0%	53%
pierce	1520.77	1563.35	3%	1.03	100%	0%	52%
trans	0.13	0.13	0%	1.00	100%	0%	53%

Proof search in NK (and, impl, neg)

example	opt	stand	reduction in time	improve	variable def trivial	fail	assign success
andEff1-nk	7.67	13.14	42%	1.71	100%	0%	80%
andEff2-nk	3.86	6.58	41%	1.70	100%	0%	81%
assocAnd-nk	2.24	3.74	40%	1.67	100%	0%	81%
combS-nk	3.85	6.64	42%	1.72	100%	0%	81%

The second example is theorem proving in the natural deduction calculus. In contrast to the previous experiments with the sequent calculus, there is a substantial performance improvement by factor 1.7. It is worth pointing out that in both examples, all clause heads can be solved by the optimized unification algorithm and no non-trivial variable definitions arise. Moreover, most of the time unification succeeds. This is not surprising when considering the formulation of the natural deduction system as any elimination rule will be applicable at any given time. Although linear head compilation substantially improves performance, more optimizations, such as tabling and indexing, are needed to solve more complex theorems.

3.5.3 Constraint solvers

Finally, we consider problems using constraint solvers.

Examples using constraint solvers

example	opt	stand	reduction in time	improve	variable def trivial	fail	assign success
solve	1.73	1.11	-56%	0.36	7.8%	42%	82%
represent	0.16	0.16	0%	1.00	85%	33%	93%
cont_fraction	0.61	0.64	33%	1.50	79%	12%	82%
mortgage	4.88	6.84	29%	1.40	66%	0%	100%
sieve 500	2.61	4.88	46%	1.86	50%	47%	79%

In our implementation, we delay solving any sub-terms which call constraint solvers. As the results below demonstrate this may be beneficial as in the sieve 500 problem, but it may also degrade performance in rare case as the first problem demonstrates. We left out the reduction of time, since there is none – in fact time increases. This has been the only time we found that performance deteriorated. The reason for performance degradation in this example is that most unification problems were outside the fragment of linear higher-order patterns.

3.6 Related work and conclusion

The language most closely related to our work is λProlog. Two main different implementations of λProlog exist, Prolog/Mali and Teyjus. In Prolog/Mali implementation, the occurs-check is left out entirely [6]. While Teyjus [38, 37] also eliminates some unnecessary occurs-checks statically during compilation. However, in addition to the presence of dependencies in Twelf, there are several other differences between our implementation and Teyjus. 1) Teyjus compiles first and subsequent occurrences of existential variables into different instructions. Therefore, assignment and unification are freely mixed during the execution. This may lead to expensive failure in some cases, since unification is still called. In our approach, we perform a simple fast assignment check and delay unification entirely. As the experimental results demonstrate, only a small percentage of the cases fails after it already passed the assignment test and most cases benefit from a fast simple assignment check. 2) We always assume that the types of existential variables are lowered. This can be done at compile time and incurs no run-time overhead. In Huet's unification algorithm, projection and imitation rules are applied at run-time to construct the correct prefixes of λ-abstractions. 3) Our approach can easily incorporate definitions and constraint domains. This is important since unifying definitions and constraint expressions may potentially be expensive. In fact, we generalize and extend the idea of linearization in the implementation and factor out not only duplicate existential variables but also any difficult sub-expressions such as definitions and constraint expressions. Therefore, our approach seems more general than the one adopted in Teyjus.

As experiments show, performance is improved substantially. This is especially important in large-scale applications such as proof-carrying code and allows us explore the full potential of logical frameworks in real-world applications. This may however only be considered as a first step towards reducing the performance gap between higher-order and first-order system. Another optimization which is particularly important to sustain performance in large-scale examples is term indexing. We will present a higher-order term indexing technique based on the ideas of linearization and assignment later in chapter 5,

Chapter 4

Tabled higher-order logic programming

Tabled first-order logic programming has been successfully applied to solve complex problems such as implementing recognizers and parsers for grammars [68], representing transition systems CCS and writing model checkers [16]. The idea behind it is to eliminate redundant computation by memoizing sub-computations and re-using their results later. The resulting search procedure is complete and terminates for programs with the bounded-term size property. The XSB system [62], a tabled logic programming system, demonstrates impressively that tabled together with non-tabled programs can be executed efficiently.

Higher-order logic programming languages such as *Elf* [47] extend first-order logic programming in three ways: first, they allow the programmer to define his or her own higher-order data-types together with constructors. In particular, constructors may take functions as arguments to denote the scope of bound variables. Dependent types allows us to model dependencies among the constructor arguments. Second, we may generate and use dynamic assumptions during proof search. Third, Elf's logic programming engine not only succeeds or fails, but also returns a proof giving evidence why it succeeded.

In this chapter, we give a motivating example and explained the basic idea behind *tabled higher-order logic programming* where some redundant computation is eliminated by memoizing sub-computations and re-using their result later. In particular, we will focus on some of the higher-order issues. As we may have dynamic assumptions in higher-order logic programming, goals might depend on a context of dynamic assumptions. We also have dependencies among terms, as the term language is derived from the dependently typed λ-calculus. Finally, in contrast to tabled first-order logic programming, where the result of a computation is an answer substitution for the existential variables in the query, tabled higher-order logic programming needs to also supply the actual proof as an object in the dependently typed lambda calculus. This means we need to store in the memo-table not only answer substitutions to memoized goals, but also the actual proof. The combination of these three features requires careful design of the table and table operations and poses several challenges in implementing a tabled higher-order logic programming interpreter.

4.1 A motivating example: subtyping

As a running example we consider a type system for a restricted functional language Mini-ML, which includes subtyping. We include an example datatype bits, along with two subtypes nat for natural numbers (bit-strings without leading zeroes), pos for positive numbers, and zero for the number zero.

$$\begin{array}{lll} \text{types} & A & ::= \ \mathsf{zero} \mid \mathsf{pos} \mid \mathsf{nat} \mid \mathsf{bits} \mid A_1 \Rightarrow A_2 \mid \ldots \\ \text{expressions} & M & ::= \ \epsilon \mid M\ \mathsf{0} \mid M\ \mathsf{1} \mid \mathsf{lam}\ x.M \mid \mathsf{app}\ M_1\ M_2 \end{array}$$

We represent natural numbers as bit-strings in standard form, with the least significant bit rightmost and no leading zeroes. We view 0 and 1 as constructors written in postfix form and ϵ stands for the empty string. For example 6 is represented as ϵ 1 1 0.

Next, we give the subtyping relation for types, including reflexivity and transitivity.

$$\frac{}{\mathsf{zero} \preceq \mathsf{nat}}\,\mathsf{zn} \qquad \frac{}{\mathsf{pos} \preceq \mathsf{nat}}\,\mathsf{pn} \qquad \frac{}{\mathsf{nat} \preceq \mathsf{bits}}\,\mathsf{nb}$$

$$\frac{}{A \preceq A}\,\mathsf{refl} \qquad \frac{A \preceq B \qquad B \preceq C}{A \preceq C}\,\mathsf{tr}$$

The typing rules our Mini-ML language are described as follows:

$$\frac{}{\Gamma \vdash \epsilon : \mathsf{bits}}$$

$$\frac{\Gamma \vdash M : \mathsf{pos}}{\Gamma \vdash M\ \ \mathsf{0} : \mathsf{pos}}\ \mathsf{tp\_z1} \qquad \frac{\Gamma \vdash M : \mathsf{bits}}{\Gamma \vdash M\ \ \mathsf{0} : \mathsf{bits}}\ \mathsf{tp\_z2}$$

$$\frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash M\ \ \mathsf{1} : \mathsf{pos}}\ \mathsf{tp\_o1} \qquad \frac{\Gamma \vdash M : \mathsf{bits}}{\Gamma \vdash M\ \ \mathsf{1} : \mathsf{bits}}\ \mathsf{tp\_o2}$$

$$\frac{\Gamma \vdash M : A' \qquad A' \preceq A}{\Gamma \vdash M : A}\ \mathsf{tp\_sub} \qquad \frac{\Gamma, x{:}A_1 \vdash M : A_2}{\Gamma \vdash \mathsf{lam}\ x.M : A_1 \Rightarrow A_2}\ \mathsf{tp\_lam}$$

The subtyping relation is directly translated into *Elf* using logic programming notation. We first declare a type family `tp` for Mini-ML types with constants `bit`, `zero`, `pos`, and `nat` represent the basic types and the function type is denoted by `T1 => T2`. Similarly, we declare a type family `exp` for Mini-ML expressions.

```
tp    :  type.                exp  :  type.
zero  :  tp.                  e    :  exp.
pos   :  tp.                  0    :  exp -> exp.
bits  :  tp.                  1    :  exp -> exp.
=>    :  tp -> tp -> tp.      lam  :  (exp -> exp) -> exp.
                              app  :  exp -> exp -> exp.
```

To ease readability, we assume the function type `=>` is declared as an infix operator, and the constructors `0` and `1` are declared as postfix operators.

Next, we give an implementation of the subtyping relation and the typing rules as higher-order logic programs. Throughout this example, we reverse the arrow $A_1 \rightarrow A_2$ writing instead $A_2 \leftarrow A_1$. From a logic programming point of view, it might be more intuitive to think of the clause $H \leftarrow A_1 \leftarrow \ldots \leftarrow A_n$ as $H \leftarrow A_1\ ,\ \ldots\ ,\ A_n$.

```
sub   :  tp -> tp -> type.
refl  :  sub A A.          zn   :  sub zero nat.
nb    :  sub nat bits.     pn   :  sub pos nat.
tr    :  sub A C           arr  :  sub (A1 => A2) (B1 => B2)
          <- sub A B                <- sub B1 A1
          <- sub B C.               <- sub A2 B2.
```

```
of      :  exp -> tp -> type.
tp_sub  :  of M A            tp_lam  :  of (lam ([x] M x)) (A1 => A2)
           <- of M A'                   <- ({x:exp} of x A1 -> of (M x) A2).
           <- sub A' A.
                             tp_app  :  of (app M1 M2) T
                                        <- of M1 (T2 => T)
                                        <- of M2 T2.
```

Note that the capital variables `A`, `B`, `C` etc. are implicitly quantified at the outside. Twelf internally reconstructs the types of the logic variables `A`, `B`, etc. In LF, the transitivity clause is written as follows:

$$\mathsf{tr} : \Pi a{:}\mathsf{tp}.\Pi b{:}\mathsf{tp}.\Pi c{:}\mathsf{tp}.\mathsf{sub}\ b\ c \rightarrow \mathsf{sub}\ a\ b \rightarrow \mathsf{sub}\ a\ c.$$

Under the type-theoretic view, $\Pi a{:}\mathsf{tp}.\Pi b{:}\mathsf{tp}.\Pi c{:}\mathsf{tp}.\mathsf{sub}\ b\ c \rightarrow \mathsf{sub}\ a\ b \rightarrow \mathsf{sub}\ a\ c$ is the type of the constant tr. Under the logic programming interpretation, we interpret types as formulas and we refer to $\Pi a{:}\mathsf{tp}.\Pi b{:}\mathsf{tp}.\Pi c{:}\mathsf{tp}.\mathsf{sub}\ b\ c \rightarrow \mathsf{sub}\ a\ b \rightarrow \mathsf{sub}\ a\ c$ as a clause and tr as the name of the clause. and In the following presentation, we often use types and formulas interchangeably.

The proof for $\mathsf{zero} \preceq \mathsf{bits}$ which is given by the following derivation

$$\dfrac{\dfrac{}{\mathsf{zero} \preceq \mathsf{nat}}\ zn \qquad \dfrac{}{\mathsf{nat} \preceq \mathsf{bits}}\ nb}{\mathsf{zero} \preceq \mathsf{bits}}\ tr$$

is represented as a proof term ($\mathsf{tr\ zero\ nat\ bits\ nb\ zn}$) in LF. The constant tr is a function where the existential variables a, b, c are instantiated to zero, nat, bits. In addition we pass in as arguments the proof term nb for $\mathsf{nat} \preceq \mathsf{bits}$ and the proof term zn for $\mathsf{zero} \preceq \mathsf{nat}$. Note that we usually omit the implicit arguments zero and nat, which denote the instantiation of transitivity rule and just write ($\mathsf{tr\ nb\ zn}$) for the proof of $\mathsf{zero} \preceq \mathsf{bits}$. In Twelf notation, this corresponds to the proof term `(tr bn zn)` for the proof of the query `(sub zero bit)`.

Although the specification and implementation of the typing rules including subtyping is straightforward, the implementation is not directly executable. Computation may be trapped in infinite paths and performance may be hampered by redundant computation. For example, the execution of the query `sub zero T` will end in an infinite branch trying to apply the transitivity rule. Similarly, the execution of the query `of (lam [x] x) T` will not terminate and fail to enumerate all possible types. In addition, we repeatedly type-check sub-expressions, which occur more than once. To eliminate redundancy, some sophisticated type checkers for example for refinement types memoize the result of sub-computations to obtain more efficient implementations. In this chapter, we extend higher-order logic programming languages such as *Elf* with generic memoization techniques, called *tabled higher-order logic programming.* This has several advantages. Although it is possible to derive an algorithmic subtyping relation for the given example, this might not be trivial in general. One could argue that the user can refine the implementation further and add an extra argument to the type family `of`, which can be used to store intermediate results, thereby developing a customized type-checker with explicit support for memoization. However, this complicates the type checker substantially and proving the correctness of the type-checker with special memoization support will be hard, because we need to reason explicitly about the structure of memoization. Moreover, the certificates, which are produced as a result of the execution, are larger and contain references to the explicit memoization data-structure. This is especially undesirable in the context of certified code where certificates are transmitted to and checked by a consumer, as sending larger certificates takes up more bandwidth and checking them takes more time. Finally, tabled logic programming provides a complete proof search strategy. For programs with the bounded-term size property, tabled logic programming terminates and we are able to disprove certain statements[1]. This in turn helps the user to debug the specification and implementations.

4.2 Tabled logic programming: a quick review

Tabling methods evaluate programs by maintaining tables of subgoals and their answers and by resolving repeated occurrences of subgoals against answers from the table.

[1]The size of a term is defined recursively where the size of a bound or modal variable or constant is 1 and the size of $R\,U$ is the size of R plus the size of U. In the higher-order setting, we can in addition define the size of $\lambda x{:}A.U$, as 1 plus the size of U and assume that all terms must be eta-expanded. A finite program P has the bounded-term size property if an atomic query Q has no arguments whose size exceeds n and there is a function $f(n)$, then there is no atomic subgoal in the derivation whose an argument size exceeds $f(n)$. (see [12], section 5.4).

We review briefly Tamaki and Sato's multi-stage strategy [65], which differs only insignificantly from SLG resolution [12] for programs without negation. To demonstrate tabled computation, we consider the evaluation of the query sub zero T in more detail.

The search proceeds in multiple stages. The table serves two purposes: 1) We record all sub-goals encountered during search. If the current goal is not in the table or more precisely there is no entry A s.t. A is an alpha-variant of the current goal, then we add it to the table and proceed with the computation. We consider a goal A an alpha-variant of another goal A' if there exists a bijection between the modal variables and ordinary variables.

Computation at a node is suspended, if the current goal is a variant of a table entry. 2) In addition to the sub-goals we are trying to solve, we also store the result of the computation in the table as a list of answers to the sub-goals. An answer is a pair of a substitution θ for the existential variables in the current goal A together with the actual proof P.

In each stage we apply program clauses and answers from the table. Figure 4.1 illustrates the search process.

The root of the search tree is labeled with the goal sub zero A. Each node is labeled with a goal statement and each child node is the result of applying a program clause or an answer from the table to the *leftmost* atom of the parent node. Applying a clause $H \leftarrow A_1 \leftarrow A_2 \ldots \leftarrow A_n$ results in the subgoals $A_1, A_2, \ldots, A_n$ where all of these subgoals need to be satisfied. We will then expand the first subgoal A_1 carrying the rest of the subgoals $A_2, \ldots, A_n$ along. If a branch is successfully solved, we show the obtained answer. To distinguish between program clause resolution and re-using of answers, we have two different kinds of edges in the tree. The edges obtained by program clause resolution are solid annotated with the clause name used to derive the child node. Edges obtained by reusing answers from the table are dashed and annotated with the answer used to derived the child node. Using the labels at the edges, we can reconstruct the proof term for a given query. In general, we will omit the actual substitution under that the parent node unifies with the program clause to avoid cluttering the example. To ensure we generate all possible answers for the query, we restrict the re-use of answers from the table. In each stage, we are only allowed to re-use answers that were generated in previous stages. Answers from previous stages (available for answer resolution) are marked gray, while current answers (not available

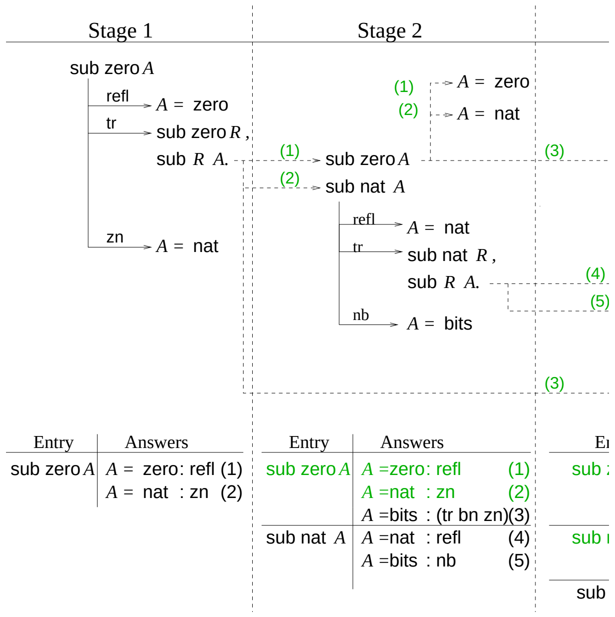


Figure 4.1: Staged computation

yet) are black. In the first stage no answers are available yet. In the second stage, we are only allowed to use answers (1) and (2). In the third stage, we may use all answers from (1) up to (5). In the final stage, we are allowed to use all answers.

4.3 Tabled higher-order logic programming

In tabled higher-order logic programming, we extend tabling to handle sub-goals that may contain implications and universal quantification and our term language is the dependently typed λ-calculus. The table entries are no longer atomic goals, but atomic goals A together with a context Γ of assumptions. In addition, terms might depend on assumptions on Γ. To highlight some of the challenges we present the evaluation of

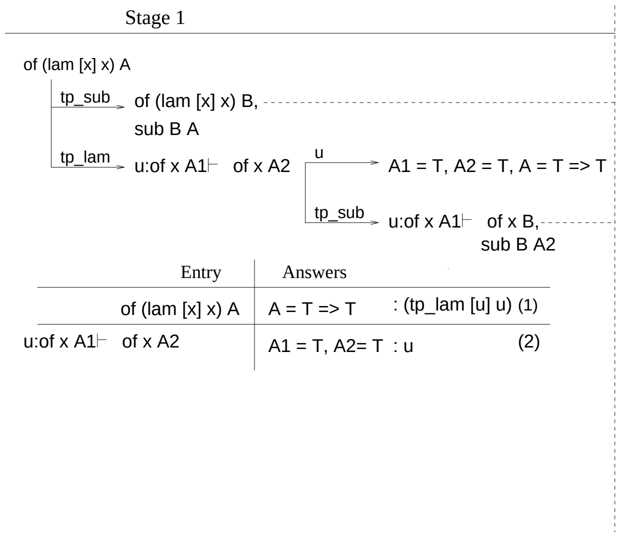


Figure 4.2: Staged computation for identity function

the query of (lam $[x]$ x) T in Figure 4.2.

The possibility of nested implications and universal quantifiers adds a new degree of complexity to memoization-based computation. Retrieval operations on the table need to be redesigned. One central question is how to look up whether a goal $\Gamma \vdash P$ is already in the table. There are two options: In the first option we only retrieve answers for a goal P given a context Γ, if the goal together with the context matches an entry $\Gamma' \vdash P'$ in the table. In the second option we match the subgoal P against the goal P' of the table entry $\Gamma' \vdash P'$, and treat the assumptions in Γ' as additional subgoals, thereby delaying satisfying these assumptions. We choose the first option of retrieving goals together with their dynamic context Γ'. One reason is that it restricts the number of possible retrievals early on in the search. For example, to solve subgoal u:of x $A_1 \vdash$

$\mathsf{of}\ x\ B, \mathsf{sub}\ B\ A_2$, we concentrate on solving the left-most goal $u{:}\mathsf{of}\ x\ A_1 \vdash \mathsf{of}\ x\ B$ keeping in mind that we still need to solve $u{:}\mathsf{of}\ x\ A_1 \vdash \mathsf{sub}\ B\ A_2$[2]. As there exists a table entry $u{:}\mathsf{of}\ x\ A_1 \vdash \mathsf{of}\ x\ A_2$, which is a variant of the current goal $u{:}\mathsf{of}\ x\ A_1 \vdash \mathsf{of}\ x\ B$, computation is suspended. Note, we adopted a common convention in logic, where we omitted the explicit declaration of the parameter x in the context. To be more precise, we write in the following development $x{:}\mathsf{exp},\ u{:}\mathsf{of}\ x\ A_1$ for the context of ordinary variables instead of just $u{:}\mathsf{of}\ x\ A_1$.

4.4 Subordination

Due to the higher-order setting, the predicates and terms might depend on assumptions in the context Γ. Virga [67] developed in his PhD thesis techniques, called subordination, to analyze dependencies in *Elf* programs statically before execution. In the Mini-ML example, the terms of type `exp` and `tp` are independent of each other. On the level of predicates, the type checker `of` depends on the subtyping relation `sub`, but not vice versa. When checking whether a subgoal $\Gamma \vdash P$ is already in the table, we exploit the subordination information in two ways. First, we use it to analyze the context Γ and determine which assumptions might contribute to the proof of P. For example the proof for $x{:}\mathsf{exp}, u{:}\mathsf{of}\ x\ A_1 \vdash \mathsf{of}\ x\ A_2$ depends on the assumption u. However, the predicate sub does not refer to the predicate of. Similarly, the predicate sub does not refer to expressions exp. Therefore the proof for $x{:}\mathsf{exp},\ u{:}\mathsf{of}\ x\ T \vdash \mathsf{sub}\ A\ A_2$ cannot depend on the assumption u or x. When checking whether $u{:}\mathsf{of}\ x\ T \vdash \mathsf{sub}\ T\ A_2$ is already in the table, it suffices to look for a variant of $\mathsf{sub}\ T\ A_2$. In the given example, computation at subgoal $x{:}\mathsf{exp}, u{:}\mathsf{of}\ x\ T \vdash \mathsf{sub}\ T\ A_2$ is suspended during stage 2 as the table already contains $\mathsf{sub}\ B_1\ T$. If we for example first discover $x{:}\mathsf{exp}, u{:}\mathsf{of}\ x\ T \vdash \mathsf{sub}\ T\ A_2$, then we store the strengthened goal $\mathsf{sub}\ T\ A_2$ in the table with an empty context.

Second, subordination provides information about terms. As we are working in a higher-order setting, solutions to new existential variables, which are introduced during execution, might depend on assumptions from Γ. For example, applying the subtyping

[2]Note that u in the declaration $u{:}\mathsf{of}\ x\ A_1$ denotes an ordinary bound variable, not a modal variable. Recall that to denote the declaration of the modal variable u we write $u{::}\Psi{\vdash}A$

rule to $x{:}\mathsf{exp}, u{:}\mathsf{of}\ x\ A_1 \vdash \mathsf{of}\ x\ A_2$ yields the following new goal

$$x{:}\mathsf{exp},\ u{:}\mathsf{of}\ x\ A_1 \vdash \mathsf{of}\ x\ (B\ x\ u)\ ,\ \mathsf{sub}\ (B\ x\ u)\ A_2$$

where the solution for the new variable B might depend on the new variable $x{:}\mathsf{exp}$ and the assumptions $u{:}\mathsf{of}\ x\ A_1$. However, we know that the solution must be an object of type tp and that objects of type tp are independent of Mini-ML expressions exp and the Mini-ML typing rules of. Hence, we can omit x and u and write $x{:}\mathsf{exp}, u{:}\mathsf{of}\ x\ A_1 \vdash \mathsf{of}\ x\ B,\ \mathsf{sub}\ B\ A_2$. Before comparing goals with table entries and adding new table entries, we eliminate unnecessary dependencies from the subgoal $\Gamma \vdash P$. This allows us to detect more loops in the search tree and eliminate more redundant computation. We come back to this issue of subordination and other optimizations to detect more loops in the higher-order setting later in the chapter on implementation of a tabled higher-order logic programming interpreter.

4.5 A foundation for tabled higher-order logic programming

In this section, we give a proof-theoretic characterization of tabled higher-order logic programming based on uniform proofs [36] and show soundness of the resulting interpreter. This work forms the basis of the implemented tabled interpreter for the language *Elf*. Although we concentrate on the logical framework LF, which is the basis of *Elf*, it seems possible to apply the presented approach to λProlog [39] or Isabelle [45], which are based on hereditary Harrop formulas and simply typed terms.

4.5.1 Uniform proofs

Computation in logic programming is achieved through proof search. Given a goal (or query) A and a program Γ, we derive A by successive application of clauses of the program Γ. Miller *et al* [36] propose to interpret the connectives in a goal A

as *search instructions* and the clauses in Γ as specifications of how to continue the search when the goal is atomic. A proof is *goal-oriented* if every compound goal is immediately decomposed and the program is accessed only after the goal has been reduced to an atomic formula. A proof is *focused* if every time a program formula is considered, it is processed up to the atoms it defines without need to access any other program formula. A proof having both these properties is *uniform* and a formalism such that every provable goal has a uniform proof is called an *abstract logic programming language*.

Elf is one example of an abstract logic programming language, which is based on the LF type theory. $\Pi x{:}A_1.A_2$ denotes the dependent function type where the type A_2 may depend on an object x of type A_1. Whenever x does not occur free in A_2 we may abbreviate $\Pi x{:}A_1.A_2$ as $A_1 \rightarrow A_2$. Using the types as formulas interpretation, we refer to $\Pi x{:}A_1.A_2$ as the universal quantifier and treat $A_1 \rightarrow A_2$ as an implication. Moreover, the typing context Γ is viewed as a context of formulas which represents the program. We will use types and formulas interchangeably in the presentation. We give the fragment used for logic programming in LF below. Types and programs are defined as follows:

$$\begin{array}{lcl} \text{Types } A & ::= & P \mid A_1 \rightarrow A_2 \mid \Pi x : A_1.A_2 \\ \text{Programs } \Gamma & ::= & \cdot \mid \Gamma, x{:}A \end{array}$$

P ranges over atomic formulas i.e. $a\ M_1\ \ldots\ M_n$. The clause

```
tr:sub A C <- sub A B <- sub B C.
```

is interpreted as

$$\mathsf{tr}{:}\Pi t{:}\mathsf{tp}.\Pi s{:}\mathsf{tp}.\Pi r{:}\mathsf{tp}.\mathsf{sub}\ r\ s \rightarrow (\mathsf{sub}\ t\ r \rightarrow \mathsf{sub}\ t\ s)$$

Every type (or formula) has a corresponding proof term U. We assume that all proof terms are in canonical form. Recall that applications which are in canonical form are of the following form: $(((H\ U_1)\ U_2)\ldots U_n)$ where head H is either a constant c or a variable x. To allow direct head access, we flatten the representation of applications and decompose the previous application into its head c and the arguments U_1, $U_2, \ldots U_n$. The spine-notation as introduced in [11] essentially achieves this idea.

$$\begin{array}{lcl} \text{Terms } U & ::= & H \cdot S \mid \lambda x{:}A.U \\ \text{Spines } S & ::= & \mathsf{nil} \mid U; S \\ \text{Heads } H & ::= & c \mid x \end{array}$$

In the example from Sec. 4.1, the proof term corresponding to sub zero bits is given as tr zn refl. Note that we actually omitted the implicit arguments zero and nat, which denote the instantiation of transitivity rule. In the following discussion, we will include implicit arguments in the proof term representation. To give an intuition for this notation we give a few examples.

$$(((((\mathsf{tr\ zero})\ \mathsf{nat})\ \mathsf{bits})\ \mathsf{nb})\ \mathsf{zn})$$

is denoted using spine notation by

$$\mathsf{tr} \cdot (\mathsf{zero}\ ;\ \mathsf{nat}\ ;\ \mathsf{bits}\ ;\ \mathsf{nb}\ ;\ \mathsf{zn}\ ;\ \mathsf{nil}).$$

The proof term

$$((((\mathsf{tp\_lam}\ T)\ (\lambda x{:}\mathsf{exp}.x))\ T)\ (\lambda x{:}\mathsf{exp}.\lambda u{:}\mathsf{of}\ x\ T.\ u))$$

is denoted by

$$\mathsf{tp\_lam} \cdot (T\ ;\ (\lambda x{:}\mathsf{exp}.x \cdot \mathsf{nil})\ ;\ T\ ;\ (\lambda x{:}\mathsf{exp}.\lambda u{:}\mathsf{of}\ x\ T.\ u \cdot \mathsf{nil})\ ;\ \mathsf{nil}).$$

where T denotes a term of type tp for example the term $(\mathsf{nat} \cdot \mathsf{nil})$. We can characterize uniform proofs by two main judgments:

$\Gamma \xrightarrow{u} U : A$	U is a uniform proof for A from Γ
$\Gamma \gg A \xrightarrow{f} S : P$	S is a focused proof for the atom P by focusing on clause A

Taking a type-theoretic view, we can interpret the first judgment as U has type A in the context Γ and the later as S has type P in context Γ. Note that we assume all operations are capture-avoiding and we can rename bound variables if necessary. Inference rules describing uniform and focused proofs are given in Figure 4.3.

In the rule f$\forall$, we instantiate the bound variable x with a term U. As x has type A_1, we check that U has type A_1 in Γ. The proof term represents the witness of the proof. When searching for a uniform proof, the proof term is constructed simultaneously. In the following discussion, we will often ignore proof terms, but keep in mind that they are silently generated as a result of the proof.

$$\frac{\Gamma, x : A, \Gamma' \gg A \xrightarrow{f} S : P}{\Gamma, x : A, \Gamma' \xrightarrow{u} x \cdot S : P}\ \mathsf{u\_atom} \qquad \frac{}{\Gamma \gg P \xrightarrow{f} \mathsf{nil} : P}\ \mathsf{f\_atom}$$

$$\frac{\Gamma, x : A_1 \xrightarrow{u} U : A_2}{\Gamma \xrightarrow{u} \lambda x : A_1.U : \Pi x : A_1.A_2}\ \mathsf{u}\forall \qquad \frac{\Gamma \gg [\mathsf{id}_\Gamma, U/x]A_2 \xrightarrow{f} S : P \quad \Gamma \vdash U : A_1}{\Gamma \gg \Pi x : A_1.A_2 \xrightarrow{f} U; S : P}\ \mathsf{f}\forall$$

$$\frac{\Gamma, c : A_1 \xrightarrow{u} U : A_2}{\Gamma \xrightarrow{u} \lambda c : A_1.U : A_1 \to A_2}\ \mathsf{u} \to^c \qquad \frac{\Gamma \gg A_1 \xrightarrow{f} S : P \qquad \Gamma \xrightarrow{u} U : A_2}{\Gamma \gg A_2 \to A_1 \xrightarrow{f} U; S : P}\ \mathsf{f} \to$$

Figure 4.3: Uniform deduction system for $\mathcal{L}$

4.5.2 Uniform proofs with answer substitutions

The result of a computation in logic programming is generally an answer substitution θ for the existentially quantified variables in a goal A. To obtain an algorithm that computes answer substitutions, we substitute existential variables u for the bound variable x in the $\mathsf{f}\forall$ rule. We will interpret the existential variables as modal variables again and generalize the previous judgment to include a context Δ of modal variables.

Existential variables are later instantiated later during unification yielding a substitution θ. We will concentrate here on the case of higher-order pattern unification.

There are different alternative approaches to treat existential variables. One we have explored in [55] relies on the following fact: If U is a solution for x in the context Γ then there exists a solution U' of type $\Pi\Gamma.A_1$ such that $U' \cdot \Gamma$ is also a solution for x and U' is well-typed in the empty context. We write $\Pi\Gamma.A_1$ for the type $\Pi x_1{:}B_1.\ldots\Pi x_n{:}B_n.A_1$ where Γ is a context $x_1{:}B_1, \ldots, x_n{:}B_n$ and $U' \cdot \Gamma$ as an abbreviation for $U' \cdot x_1; \ldots; x_n; \mathsf{nil}$. Moreover, there is a one-to-one correspondence between these two solutions, as $U' \cdot \Gamma$ reduces to U. This has been shown by Miller [35] for the simply-typed case and by Pfenning in the dependently typed and polymorphic case [48]. Following Miller's terminology, we say U' is the result of raising U. Intuitively, U depends globally on the assumptions in Γ. Raising allows us to localize dependencies by replacing U with $U' \cdot \Gamma$. Then we can translate the $\mathsf{f}\forall$ rule to

$$\frac{\Gamma \gg [\mathsf{id}_\Gamma, U' \cdot \Gamma/x]A_2 \xrightarrow{f} S : P \qquad \cdot \xrightarrow{u} U' : \Pi\Gamma.A_1}{\Gamma \gg \Pi x : A_1.A_2 \xrightarrow{f} (U' \cdot \Gamma); S : P}\,\mathsf{f}\forall$$

To obtain a calculus with existential variables, we then replace U' with a new existential variable which is annotated with its type. Another alternative would be to use mixed-prefixes [35] to model dependencies. However this would complicate the presentation further. Here, we use the modal context and modal substitution as previously introduced in chapter 2. This eliminates the need of first translating the f$\forall$-rule. The two main judgments for computing answer substitutions are

$\Delta; \Gamma \xrightarrow{u} A/(\Delta', \theta)$	Uniform proof with answer substitution $\Delta' \vdash \theta : \Delta$
$\Delta; \Gamma \gg A \xrightarrow{f} P/(\Delta', \theta)$	Focused proof with answer substitution $\Delta' \vdash \theta : \Delta$

We assume that A is well-typed in the modal context Δ and bound variable context Γ. The inference rules are given in Figure 4.4.

To obtain an algorithm, we impose left-to-right order on the solution of the fs $\rightarrow$ rule. This matches our intuitive understanding of computation in logic programming. In the fs $\rightarrow$ rule for example we first decompose the focused clause until we reach the head of the clause. After we unified the head of the clause with our goal A on the right-hand side of the sequent and completed this branch, we proceed proving the subgoals. This left-to-right evaluation strategy only fixes a don't care non-deterministic choice in the inference system. In the fs$\forall$ rule we delay the instantiation of x by introducing a new modal variable $u[\mathsf{id}_\Gamma]$. Note that this variable is created with type A and context Γ. There is no condition needed to ensure that θ does not leave its scope in the conclusion of the rule fs$\forall$ since θ only depends on modal variables in Δ and is independent of the bound variables in Γ. In the fs_atom rule the instantiation for existentially quantified variables is obtained by unifying P with P' in the context Γ. θ is a solution to the unification problem $\Delta; \Gamma \vdash P \doteq P'$ where P and P' are higher-order patterns.

There is still some non-determinism left in $\mathcal{L}_\theta$, which needs to be resolved in an actual implementation. In the us_atom rule, we do not specify which clause from Γ we pick and focus on. Moreover, usually the order of clauses in Γ is determined by the order of the clauses in the program. Logic programming interpreters then usually try the clauses in Γ in order and do not backtrack over different choices since this is considered

Uniform Proof:

$$\frac{\Delta; \Gamma, x{:}A, \Gamma' \gg A \xrightarrow{f} P/(\Delta', \theta)}{\Delta; \Gamma, x{:}A, \Gamma' \xrightarrow{u} P/(\Delta', \theta)}\,\mathsf{us\_atom} \qquad \frac{\Delta; \Gamma, x{:}A_1 \xrightarrow{u} A_2/(\Delta, \theta)}{\Delta; \Gamma \xrightarrow{u} \Pi x : A_1.A_2/(\Delta, \theta)}\,\mathsf{us}\forall$$

$$\frac{\Delta; \Gamma, x : A_1 \xrightarrow{u} A_2/(\Delta, \theta)}{\Delta; \Gamma \xrightarrow{u} A_1 \rightarrow A_2/(\Delta, \theta)}\,\mathsf{us} \rightarrow^x$$

Focused Proof

$$\frac{\Delta; \Gamma \vdash P' \doteq P/(\Delta', \theta)}{\Delta; \Gamma \gg P' \xrightarrow{f} P/(\Delta', \theta)}\,\mathsf{fs\_atom}$$

$$\frac{\Delta, u{::}(\Gamma \vdash A_1);\ \Gamma \gg [\mathsf{id}_\Gamma, u[\mathsf{id}_\Gamma]/x]A_2 \xrightarrow{f} P/(\Delta', (\theta, U/u)) \quad u \text{ is a new modal variable}}{\Delta; \Gamma \gg \Pi x : A_1.A_2 \xrightarrow{f} P/(\Delta', \theta)}\,\mathsf{fs}\forall$$

$$\frac{\Delta; \Gamma \gg A_1 \xrightarrow{f} P/(\Delta_1, \theta_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \xrightarrow{u} [\![\theta_1]\!]A_2/(\Delta_2, \theta_2)}{\Delta; \Gamma \gg A_2 \rightarrow A_1 \xrightarrow{f} P/(\Delta_2, [\![\theta_2]\!]\theta_1)}\,\mathsf{fs} \rightarrow$$

Figure 4.4: Uniform deduction system for $\mathcal{L}_\theta$ with substitutions

too expensive. This choice renders the search strategy incomplete in practice. However, the presented deductive system $\mathcal{L}_\theta$, which generates answer substitutions, is sound and complete, as expressed by the following two theorems.

Theorem 62 (Soundness)

1. *If* $\Delta; \Gamma \xrightarrow{u} A/(\Delta', \theta)$ *then for any modal substitution* $\cdot \vdash \rho : \Delta'$ *we have* $\cdot; [\![[\![\rho]\!]\theta]\!]\Gamma \xrightarrow{u} [\![[\![\rho]\!]\theta]\!]A$.
2. *If* $\Delta; \Gamma \gg A \xrightarrow{f} P/(\Delta', \theta)$ *then for any modal substitution* $\cdot \vdash \rho : \Delta'$ *we have* $\cdot; [\![[\![\rho]\!]\theta]\!]\Gamma \gg [\![[\![\rho]\!]\theta]\!]A \xrightarrow{f} [\![[\![\rho]\!]\theta]\!]P$.

Proof: Structural simultaneous induction on the derivation $\mathcal{D}$.

Case $\mathcal{D} = \dfrac{\Delta; \Gamma, x{:}A_1 \xrightarrow{u} A_2/(\Delta, \theta)}{\Delta; \Gamma \xrightarrow{u} \Pi x : A_1.A_2/(\Delta, \theta)}\ \mathsf{us}\forall$

$[\![[\![\rho]\!]\theta]\!](\Gamma, x{:}A_1) \xrightarrow{u} [\![[\![\rho]\!]\theta]\!]A_2$ by i.h.
$[\![[\![\rho]\!]\theta]\!](\Gamma), x{:}[\![[\![\rho]\!]\theta]\!](A_1) \xrightarrow{u} [\![[\![\rho]\!]\theta]\!]A_2$ by substitution definition
$[\![[\![\rho]\!]\theta]\!](\Gamma) \xrightarrow{u} \Pi x{:}[\![[\![\rho]\!]\theta]\!](A_1).[\![[\![\rho]\!]\theta]\!]A_2$ by rule
$[\![[\![\rho]\!]\theta]\!](\Gamma) \xrightarrow{u} [\![[\![\rho]\!]\theta]\!]\Pi x{:}A_1.A_2$ by substitution definition

Case $\mathcal{D} = \dfrac{\Delta; \Gamma, c : A_1 \xrightarrow{u} A_2/(\Delta, \theta)}{\Delta; \Gamma \xrightarrow{u} A_1 \to A_2/(\Delta, \theta)}\ \mathsf{us} \to^c$

$[\![[\![\rho]\!]\theta]\!](\Gamma, c : A_1) \xrightarrow{u} [\![[\![\rho]\!]\theta]\!]A_2$ by i.h.
$[\![[\![\rho]\!]\theta]\!](\Gamma), c : [\![[\![\rho]\!]\theta]\!](A_1) \xrightarrow{u} [\![[\![\rho]\!]\theta]\!]A_2$ by substitution definition
$[\![[\![\rho]\!]\theta]\!](\Gamma) \xrightarrow{u} [\![[\![\rho]\!]\theta]\!](A_1) \to [\![[\![\rho]\!]\theta]\!]A_2$ by rule
$[\![[\![\rho]\!]\theta]\!](\Gamma) \xrightarrow{u} [\![[\![\rho]\!]\theta]\!](A_1 \to A_2)$ by substitution definition

Case $\mathcal{D} = \dfrac{\Delta; \Gamma \gg A_1 \xrightarrow{f} P/(\Delta_1, \theta_1) \quad \Delta_1; [\![\theta_1]\!]\Gamma \xrightarrow{u} [\![\theta_1]\!]A_2/(\Delta_2, \theta_2)}{\Delta; \Gamma \gg A_2 \to A_1 \xrightarrow{f} P/(\Delta_2, [\![\theta_2]\!]\theta_1)}\ \mathsf{fs} \to$

$[\![[\![\rho]\!]\theta_1]\!]\Gamma \gg [\![[\![\rho]\!]\theta_1]\!]A_1 \xrightarrow{f} [\![[\![\rho]\!]\theta_1]\!]P$ by i.h.
let $\rho = [\![\rho']\!]\theta_2$
$[\![[\![\rho']\!]\theta_2]\!][\![\theta_1]\!]\Gamma \xrightarrow{u} [\![[\![\rho']\!]\theta_2]\!][\![\theta_1]\!]A_2$ by i.h.
$[\![[\![\rho']\!]\theta_2]\!][\![\theta_1]\!]\Gamma \gg [\![[\![\rho']\!]\theta_2]\!][\![\theta_1]\!]A_1 \to [\![[\![\rho']\!]\theta_2]\!][\![\theta_1]\!]A_2 \xrightarrow{f} [\![[\![\rho']\!]\theta_2]\!][\![\theta_1]\!]P$ by rule
$[\![[\![\rho']\!]\theta_2]\!][\![\theta_1]\!]\Gamma \gg [\![[\![\rho']\!]\theta_2]\!][\![\theta_1]\!](A_1 \to A_2) \xrightarrow{f} P$ by substitution definition

Case $\mathcal{D} = \dfrac{\Delta, u{::}(\Gamma{\vdash}A_1);\ \Gamma \gg [\mathsf{id}_\Gamma, u[\mathsf{id}_\Gamma]/x]A_2 \xrightarrow{f} P/(\Delta', (\theta, U/u))}{\Delta; \Gamma \gg \Pi x : A_1.A_2 \xrightarrow{f} P/(\Delta', \theta)}\ \mathsf{fs}\forall$

u is a new modal variable by assumption
$[\![[\![\rho]\!](\theta, U/u)]\!]\Gamma \gg [\![[\![\rho]\!](\theta, U/u)]\!]([\mathsf{id}_\Gamma, u[\mathsf{id}_\Gamma]/x]A_2) \xrightarrow{f} [\![[\![\rho]\!](\theta, U/u)]\!]P$ by i.h.
$\cdot \vdash \rho : \Delta'$

$[\![[\![\rho]\!](\theta, U/u)]\!]\Gamma = [\![[\![\rho]\!]\theta]\!]\Gamma$ by strengthening

$[\![[\![\rho]\!](\theta, U/u)]\!]P = [\![[\![\rho]\!]\theta]\!]P$ by strengthening

$[\![[\![\rho]\!](\theta, U/u)]\!] = [\![[\![\rho]\!]\theta, [\![\rho]\!]U/u]\!]$

$[\![[\![\rho]\!](\theta)]\!]\Gamma \gg ([\mathsf{id}_{[\![[\![\rho]\!]\theta]\!]\Gamma}, [\![\rho]\!]U/x]A_2) \xrightarrow{f} [\![[\![\rho]\!](\theta)]\!]P$ by substitution definition

$\Delta' \vdash (\theta, U/u) : (\Delta, u{::}\Gamma{\vdash}A_1)$ by invariant

$\Delta' \vdash \theta : \Delta$ by inversion

$\Delta'; [\![\theta]\!]\Gamma \vdash U : [\![\theta]\!]A_1$ by inversion

$\cdot; [\![\rho]\!][\![\theta]\!]\Gamma \vdash [\![\rho]\!]U : [\![\rho]\!][\![\theta]\!]A_1$ by substitution property (lemma 41)

$\cdot; [\![[\![\rho]\!]\theta]\!]\Gamma \vdash [\![\rho]\!]U : [\![[\![\rho]\!]\theta]\!]A_1$ by composition lemma 42

$\cdot; [\![[\![\rho]\!]\theta]\!]\Gamma \gg \Pi x{:}[\![[\![\rho]\!]\theta]\!]A_1.[\![[\![\rho]\!]\theta]\!]A_2 \xrightarrow{f} [\![[\![\rho]\!]\theta]\!]P$ by rule

$\cdot; [\![[\![\rho]\!]\theta]\!]\Gamma \gg [\![[\![\rho]\!]\theta]\!](\Pi x{:}A_1.A_2) \xrightarrow{f} [\![[\![\rho]\!]\theta]\!]P$ by substitution definition

Case $\mathcal{D} = \dfrac{\Delta; \Gamma \vdash P' \doteq P/(\Delta', \theta)}{\Delta; \Gamma \gg P' \xrightarrow{f} P/(\Delta', \theta)}$ $\mathsf{fs\_atom}$

$[\![\theta]\!]P' = [\![\theta]\!]P$ by soundness of pattern unification 60

$\Delta' \vdash \theta : \Delta$ by lemma 59

for any modal substitution ρ such that $\cdot \vdash \rho : \Delta'$

$[\![\rho]\!][\![\theta]\!]P' = [\![\rho]\!][\![\theta]\!]P$ by substitution property (lemma 41)

$[\![[\![\rho]\!]\theta]\!]P' = [\![[\![\rho]\!]\theta]\!]P$ by substitution property (lemma 41)

$\Gamma \gg [\![[\![\rho]\!]\theta]\!]P' \xrightarrow{f} [\![[\![rho]\!]\theta]\!]P$ by rule

□

Theorem 63 (Completeness)

1. *If* $\cdot; [\![\rho]\!]\Gamma \xrightarrow{u} [\![\rho]\!]A$ *for a modal substitution* ρ*, s.t.* $\cdot \vdash \rho : \Delta$
then $\Delta; \Gamma \xrightarrow{u} A/(\Delta', \theta)$ *for some* θ *and* $\rho = [\![\rho']\!]\theta$ *for some* ρ' *s.t.* $\cdot \vdash \rho' : \Delta'$.

2. *If* $\cdot; [\![\rho]\!]\Gamma \gg [\![\rho]\!]A \xrightarrow{f} [\![\rho]\!]P$ *for a modal substitution* ρ *s.t.* $\cdot \vdash \rho : \Delta$
then $\Delta; \Gamma \gg A \xrightarrow{f} P/(\Delta', \theta)$ *for some* θ *and* $\rho = [\![\rho']\!]\theta$ *for some* ρ'*, s.t.* $\cdot \vdash \rho' : \Delta'$.

Proof: Simultaneous structural induction on the first derivation.

Case $\mathcal{D} = \Gamma \xrightarrow{u} \Pi x : A_1.A_2$

$[\![\rho]\!]\Gamma \xrightarrow{u} [\![\rho]\!](\Pi x : A_1.A_2)$	by assumption
$[\![\rho]\!]\Gamma \xrightarrow{u} \Pi x : [\![\rho]\!]A_1.[\![\rho]\!]A_2$	by substitution definition
$[\![\rho]\!]\Gamma, x : [\![\rho]\!]A_1 \xrightarrow{u} [\![\rho]\!]A_2$	by inversion
$[\![\rho]\!](\Gamma, x : A_1) \xrightarrow{u} [\![\rho]\!]A_2$	by substitution definition
$\Delta; \Gamma, x : A_1 \xrightarrow{u} A_2/(\Delta', \theta)$ and $\rho = [\![\rho']\!]\theta$	by i.h.
$\Delta; \Gamma \xrightarrow{u} \Pi x : A_1.A_2/(\Delta', \theta)$	by rule

Case $\mathcal{D} = \Gamma \xrightarrow{u} A_1 \to A_2$

$[\![\rho]\!]\Gamma \xrightarrow{u} [\![\rho]\!](A_1 \to A_2)$	by assumption
$[\![\rho]\!]\Gamma \xrightarrow{u} [\![\rho]\!](A_1) \to [\![\rho]\!](A_2)$	by substitution definition
$[\![\rho]\!]\Gamma, c{:}[\![\rho]\!]A_1 \xrightarrow{u} [\![\rho]\!]A_2$	by inversion
$[\![\rho]\!](\Gamma, c{:}A_1) \xrightarrow{u} [\![\rho]\!]A_2$	by substitution definition
$\Delta; \Gamma, c{:}A_1 \xrightarrow{u} A_2/(\Delta', \theta')$ and $\rho = [\![\rho']\!]\theta$	by i.h.
$\Delta; \Gamma \xrightarrow{u} A_1 \to A_2/(\Delta', \theta')$	by rule

Case $\mathcal{D} = \Gamma \gg A_2 \to A_1 \xrightarrow{f} P$

$[\![\rho]\!]\Gamma \gg [\![\rho]\!](A_2 \to A_1) \xrightarrow{f} [\![\rho]\!]P$	by assumption
$[\![\rho]\!]\Gamma \gg [\![\rho]\!](A_2) \to [\![\rho]\!](A_1) \xrightarrow{f} [\![\rho]\!]P$	by substitution definition
$[\![\rho]\!]\Gamma \gg [\![\rho]\!]A_1 \xrightarrow{f} [\![\rho]\!]P$	by inversion
$[\![\rho]\!]\Gamma \xrightarrow{u} [\![\rho]\!]A_2$	by inversion
$\Delta; \Gamma \gg A_1 \xrightarrow{f} P/(\Delta_1, \theta_1)$ and $\rho = [\![\rho']\!]\theta_1$	by i.h.
$[\![[\![\rho']\!]\theta_1]\!]\Gamma \xrightarrow{u} [\![[\![\rho']\!]\theta_1]\!]A_2$	
$[\![\rho']\!]([\![\theta_1]\!]\Gamma) \xrightarrow{u} [\![\rho']\!]([\![\theta_1]\!]A_2)$	by composition lemma 42
$\Delta_1; [\![\theta_1]\!]\Gamma \xrightarrow{u} [\![\theta_1]\!]A_2/(\Delta_1, \theta_2)$ and $\rho' = [\![\rho'']\!](\theta_2)$	by i.h.
$\Delta; \Gamma \gg (A_2 \to A_1) \xrightarrow{f} P/(\Delta_2, [\![\theta_2]\!]\theta_1)$	by rule
$\rho = [\![\rho'']\!][\![\theta_2]\!]\theta_1$	

Case $\mathcal{D} = \Gamma \gg \Pi x{:}A_1.A_2 \xrightarrow{f} P$

$\cdot; [\![\rho]\!]\Gamma \gg [\![\rho]\!](\Pi x{:}A_1.A_2) \xrightarrow{f} [\![\rho]\!]P$	by assumption
$\cdot; [\![\rho]\!]\Gamma \gg \Pi x{:}[\![\rho]\!]A_1.[\![\rho]\!]A_2 \xrightarrow{f} [\![\rho]\!]P$	by substitution definition

$\cdot; [\![\rho]\!]\Gamma \gg [\mathsf{id}_{[\![\rho]\!]\Gamma}, U/x]([\![\rho]\!]A_2) \xrightarrow{f} [\![\rho]\!]P$ for some U by inversion

$\cdot; [\![\rho]\!]\Gamma \vdash U : [\![\rho]\!]A_1$ by inversion

$\cdot \vdash \rho : \Delta$ by assumption

$\cdot \vdash (\rho, U/u) : (\Delta, u{::}\Gamma{\vdash}A_1$ by rule

$[\mathsf{id}_{[\![\rho]\!]\Gamma}, U/x][\![\rho]\!]A_2 = [\![\rho]\!]([\mathsf{id}_\Gamma, U/x]A_2) = [\![\rho, U/u]\!]([\mathsf{id}_\Gamma, u[\mathsf{id}_\Gamma]/x]A_2)$ where u is new
by substitution property

$\cdot; [\![\rho, U/u]\!]\Gamma \gg [\![\rho, U/u]\!]([\mathsf{id}_\Gamma, u[\mathsf{id}_\Gamma]/x]A_2) \xrightarrow{f} [\![\rho, U/u]\!]P$ by substitution definition

$\Delta, u{::}\Gamma{\vdash}A_1; \Gamma \gg [\mathsf{id}_\Gamma, u[\mathsf{id}_\Gamma]/x]A_2 \xrightarrow{f} P/(\Delta', \theta')$ and
$(\rho, U/u) = [\![\rho']\!]\theta'$ for some θ' and ρ' by i.h.

let $\theta' = (\theta, U'/u)$.

$(\rho, U/u) = [\![\rho']\!](\theta, U'/u) = ([\![\rho']\!]\theta,\ [\![\rho]\!]U'/u)$ by substitution definition

$\rho = [\![\rho']\!]\theta$ and $U = [\![\rho]\!]U'$ by syntactic equaltiy

$\Delta; \Gamma \gg \Pi x{:}A_1.A_2 \xrightarrow{f} P/(\Delta', \theta)$ by rule

Case $\mathcal{D} = \Gamma \gg P \xrightarrow{f} P$

$\cdot; [\![\rho]\!]\Gamma \gg [\![\rho]\!]P \xrightarrow{f} [\![\rho]\!]P$ by assumption

$[\![\rho]\!]P = [\![\rho]\!]P$ and $\cdot \vdash \rho : \Delta$

$\Delta; \Gamma \vdash P \doteq P \ / \ (\Delta', \theta)$ and there exists a ρ' s.t. $\cdot \vdash \rho' : \Delta'$ and $\rho = [\![\rho']\!]\theta$
by completeness of higher-order pattern unification (lemma 61)

$\Delta; \Gamma \gg P \xrightarrow{f} P \ / \ (\Delta', \theta)$ by rule

□

In the next section, we extend this system $\mathcal{L}_\theta$ to include memoization.

4.5.3 Tabled uniform proofs

The idea behind tabled uniform proofs is to extend our two basic judgments with a table $\mathcal{T}$ in which we record atomic sub-goals and the corresponding answer substitutions and proof terms. A subgoal is a sequent $\Delta; \Gamma \xrightarrow{u} P$ where Δ describes the existential variables, Γ is a program context and P is an atomic goal, which we need to derive from Γ. When we discover the sub-goal $\Delta; \Gamma \xrightarrow{u} P$ for the first time, we memoize this goal in the table. Note that the sequent $\Delta; \Gamma \xrightarrow{u} P$ might potentially contain existential variables. Once we have proven the sub-goal $\Delta; \Gamma \xrightarrow{u} P$, we add the answer substitution $\Delta' \vdash \theta : \Delta$ to the table. We keep in mind that we are silently generating proof terms together with answer substitutions. We assume that some predicates are designated as tabled predicates where we record subgoals and corresponding answers. For predicates not designated as tabled the $\mathsf{us\_atom}$ rule still applies.

Definition 64 (Table) *A table entry consists of two parts: a goal $\Delta; \Gamma \xrightarrow{u} P$ and a list $\mathcal{A}$ of pairs, answer substitutions $\Delta' \vdash \theta : \Delta$ and proof terms U, such that $\Delta'; [\![\theta]\!]\Gamma \xrightarrow{u} [\![\theta]\!]U : [\![\theta]\!]P$ is a solution. A table $\mathcal{T}$ is a collection of table entries.*

The table is a store of proven and still open conjectures. The open conjectures are the table entries that have an empty list of answers. The proven conjectures (lemmas) are the table entries that have a list of answer substitutions associated with them. As proof terms are generated and stored together with answer substitutions, we also have the actual proof for the given conjecture. We will design the inference rules in such a way that for any solution in the table $\Delta'; [\![\theta]\!]\Gamma \xrightarrow{u} [\![\theta]\!]U : [\![\theta]\!]P$ there exists a derivation $\Delta; \Gamma \xrightarrow{u} U : P/(\Delta', \theta)$. We will keep all the previous inference rules, but keep in mind that we are silently passing around a table $\mathcal{T}$. Any substitution we apply to Γ and P (see for example the $\mathsf{fs} \rightarrow$ rule) will not effect the table. This is important because we do want to have explicit control over the table. The application of inference rules should not have any undesired effects on the table. The main judgments are

$$\begin{array}{rcl} \mathcal{T}; \Delta; \Gamma & \xrightarrow{u} & U : A/(\Delta', \theta, \mathcal{T}') \\ \mathcal{T}; \Delta; \Gamma \gg A & \xrightarrow{f} & S : P/(\Delta', \theta, \mathcal{T}') \end{array}$$

In addition to the $\mathsf{us\_atom}$ inference rule, we will have the rules extend and $\mathsf{retrieve}$. The extend rule adds a subgoal and its answer to the table. When we encounter a new subgoal, we add a new entry with an empty answer list to the table. Once we have proven this subgoal, we add the answer substitution and proof term to its answer list, and we can later use it as a lemma. $\mathsf{retrieve}$ allows us to close a branch by applying a lemma from the table. If we are proving $\Delta; \Gamma \xrightarrow{u} P$, where Γ and P may contain existential variables and we have a proof for $\Delta'; [\![\theta]\!]\Gamma \xrightarrow{u} [\![\theta]\!]U : [\![\theta]\!]P$ in the table then we can just re-use it by applying the answer substitution θ and substituting the proof term $[\![\theta]\!]U$ for it. Applying the $\mathsf{retrieve}$ rule corresponds to introducing an analytic cut in the proof using a lemma from the table.

Before we present these two rules, we define when a goal is α-variant to a table entries. We consider $\Delta; \Gamma \xrightarrow{u} P$ a variant of $\Delta'; \Gamma' \xrightarrow{u} P'$ if there exists a renaming of the ordinary variables in Γ and the modal variables in Δ such that $\Delta; \Gamma \xrightarrow{u} P$ is equal to $\Delta'; \Gamma' \xrightarrow{u} P'$.

Definition 65 (Variant) *The goal $\Delta; \Gamma \xrightarrow{u} P$ is a variant of $\Delta'; \Gamma' \xrightarrow{u} P'$ if*

- *there exists a bijection ρ between the modal variables in Δ and Δ' such that $\Delta' \vdash \rho : \Delta$.*
- *there exists a bijection σ between the ordinary variables in $[\![\rho]\!]\Gamma$ and Γ' such that $\Delta'; \Gamma' \vdash \sigma : [\![\rho]\!]\Gamma$*
- *and $[\sigma][\![\rho]\!]P = P'$.*

This definition of variant can be extended to terms. We say a term $\Delta; \Gamma \vdash U$ is a variant of term $\Delta'; \Gamma' \vdash U'$ if there exists a bijection between the modal variables in Δ and Δ' and a bijection between the ordinary variables in Γ and Γ'. Next, we define when two modal substitutions are variants of each other.

Definition 66 *The modal substitution $\Delta'_1 \vdash \theta_1 : \Delta_1$ is a variant of $\Delta'_2 \vdash \theta_2 : \Delta_2$*

- *$\Delta_1 \vdash \cdot : \cdot$ is a variant of $\Delta_2 \vdash \cdot : \cdot$.*
- *$\Delta'_1 \vdash (\theta_1, U/u) : (\Delta_1, u{::}(\Psi{\vdash}A))$ is a variant of $\Delta'_2 \vdash (\theta_2, N/u) : (\Delta_2, u{::}(\Psi'{\vdash}A')$ iff*
    1. *$\Delta'_1 \vdash \theta_1 : \Delta_1$ is a variant of $\Delta'_2 \vdash \theta_2 : \Delta_2$*
    2. *$\Delta'_1; \Psi \vdash U$ is a variant of $\Delta'_2; \Psi' \vdash N$*

Now we can define the three main operations on the table, extending the table, inserting an answer in the table and retrieving an answer from the table.

Definition 67 (extend) $\mathsf{extend}(\mathcal{T}, \Delta; \Gamma \xrightarrow{u} P) = \mathcal{T}'$
Let $\mathcal{T}$ be a table, $\Delta; \Gamma \xrightarrow{u} P$ be a goal.

- *If there exists a table entry $(\Delta'; \Gamma' \xrightarrow{u} P', \mathcal{A})$ in $\mathcal{T}$ and $\mathcal{A}$ is non-empty such that $\Delta'; \Gamma' \xrightarrow{u} P'$ is a variant of $\Delta; \Gamma \xrightarrow{u} P$ then return $\mathcal{T}$.*

- *If there exists* **no** *table entry* $(\Delta'; \Gamma' \xrightarrow{u} P', \mathcal{A})$ *in* $\mathcal{T}$ *such that* $\Delta'; \Gamma' \xrightarrow{u} P'$ *is a variant of* $\Delta; \Gamma \xrightarrow{u} P$*, then we obtain the extended table* $\mathcal{T}'$ *by adding* $\Delta; \Gamma \xrightarrow{u} P$ *to the table* $\mathcal{T}$ *with an empty solution list.*

Definition 68 (insert) $\mathsf{insert}(\mathcal{T}, \Delta; \Gamma \xrightarrow{u} P, U, \theta) = \mathcal{T}'$
Let $\mathcal{T}$ *be a table,* $\Delta; \Gamma \xrightarrow{u} P$ *be a goal and* $\Delta' \vdash \theta : \Delta$ *be a corresponding answer substitution and* U *a proof term such that* $\Delta; \Gamma \vdash U : P$*. Let* $(\Delta_i; \Gamma_i \xrightarrow{u} P_i, \mathcal{A})$ *be in the table* $\mathcal{T}$ *and* $\Gamma_i \xrightarrow{u} P_i$ *is a variant of* $\Delta; \Gamma \xrightarrow{u} P$*. If there exists no* $\Delta'_i \vdash \theta_i : \Delta_i$ *in the answer substitution list* $\mathcal{A}$*, such* θ_i *is a variant of* θ*, then we add* $\Delta' \vdash \theta : \Delta$ *together with the proof term* U *to the the answer list* $\mathcal{A}$*.*

If we generate and try to insert an answer substitution θ which already exists in the answer list, then we fail.

Definition 69 (retrieve) $\mathsf{retrieve}(\mathcal{T}, \Delta; \Gamma \xrightarrow{u} P) = (\Delta', \theta, U)$
Let $\mathcal{T}$ *be a table and* $\Delta; \Gamma \xrightarrow{u} P$ *be a goal. If there exists a table entry* $(\Delta_i; \Gamma_i \xrightarrow{u} P_i, \mathcal{A}_i)$ *such that* $\Delta_i; \Gamma_i \xrightarrow{u} P_i$ *is variant of* $\Delta; \Gamma \xrightarrow{u} P$ *and there exists an answer substitution* $\Delta'_i \vdash \theta_i : \Delta$ *together with a proof term* U *in* $\mathcal{A}_i$ *then return* $(\Delta'_i,\ \theta_i,\ U)$*.*

Now we can give the additional rules extend and retrieve.

$$\frac{\begin{array}{l}\mathsf{extend}(\mathcal{T},\ \Delta; (\Gamma, x{:}A, \Gamma') \xrightarrow{u} P) \ = \ \mathcal{T}_1 \\ \mathcal{T}_1; \Delta; (\Gamma, x{:}A, \Gamma') \gg A \xrightarrow{f} S : P/(\Delta', \theta, \mathcal{T}_2) \\ \mathsf{insert}(\mathcal{T}_2, \Delta; (\Gamma, x{:}A, \Gamma') \xrightarrow{u} P, U, \Delta', \theta) = \mathcal{T}_3\end{array}}{\mathcal{T}; \Delta; (\Gamma, x{:}A, \Gamma') \xrightarrow{u} x \cdot S : P/(\Delta', \theta, \mathcal{T}_3)}\ \mathsf{extend}$$

$$\frac{\mathsf{retrieve}(\mathcal{T}; \Delta; \Gamma \xrightarrow{u} P) = (\Delta', \theta, U)}{\mathcal{T}; \Delta; \Gamma \xrightarrow{u} U : P/(\Delta', \theta, \mathcal{T})}\ \mathsf{retrieve}$$

The rule extend is applicable, even if a table entry already exists. In this case, the goal $\Delta; \Gamma \xrightarrow{u} P$ will not be added to the table, but we will still try to find a solution for the modal variables in Δ. This is correct, since we may need to generate more than one answer substitution θ which needs to be added to the the table.

If we discover a sub-goal P in a context Γ, which is already in the table $\mathcal{T}$ but with an empty answer substitution list, then we have discovered a loop in the computation.

No inference rule is applicable, and therefore computation fails. The extend rule is not applicable, since we extend will only succeed if the subgoal P in the context Γ is not already in the table or it is in the table, but it must have a non-empty answerlist. The $\mathsf{retrieve}$ rule is also not applicable, since we have an empty answer list. The definitions of insert prevents us from inferring the same solution twice. If a sub-goal P is already in the table, but has some answers in the answer list $\mathcal{A}$, then we retrieve the answers. As we might need additional answers for P which are not already in the table yet, we need to still be able to apply extend rule.

The presented inference rules leave several choices undetermined. For example, we do not specify the order in which we use program clauses. This choice was already present in the non-tabled system. Similarly, the rules to allow memoization leave open in what order we retrieve answers, when to retrieve answers and when to apply the extend rule. In an implementation, we need to resolve these choices. We have chosen a high-level declarative description of the memoization based proof search. An advantage is that is is abstract and clearly illustrates the use of memoization as analytic cut. On the other hand, failure in such a proof system is implicit, i.e. if no inference rule is applicable, then we fail, “backtrack”, and try another rule. Upon failure, we backtrack and reset the state of the table. In an implementation, the table $\mathcal{T}$ is usually implemented via a global variable, and persists, even if proof search fails. Unfortunately, there is no way to express the persistence of the table in the proof system without making failure explicit in the deductive system.This is beyond the scope of this thesis. We are still able to prove soundness of memoization-based search, however completeness and termination are harder to establish.

Theorem 70 (Soundness)

1. *If* $\mathcal{D} : \mathcal{T}; \Delta; \Gamma \xrightarrow{u} A/(\Delta', \theta, \mathcal{T}')$ *then we have* $\mathcal{E} : \Delta; \Gamma \xrightarrow{u} A/(\Delta', \theta)$.

2. *If* $\mathcal{D} : \mathcal{T}; \Delta; \Gamma \gg A \xrightarrow{f} P/(\Delta', \theta, \mathcal{T}')$ *then we have* $\mathcal{E} : \Delta; \Gamma \gg A \xrightarrow{f} P/(\Delta', \theta)$.

Proof: Structural simultaneous induction on the derivation $\mathcal{D}$. The proof is straightforward using the fact that for every answer in the table we have a proof term. this proof term can be expanded into a proof. $\Box$

As we mentioned earlier, there are several choices left undetermined in the given proof system which we need to resolve in an actual implementation. The multi-stage strategy discussed earlier is one possible solution. In this strategy we proceed in lock-steps. First, we apply the extend rule until all clauses from Γ have been tried, and then allow the application of the retrieve rule. The strategy also restricts the retrieve rule, i.e. only answers from previous stages can be retrieved. Alternatively, we could use SCC scheduling (strongly connected components), which allows us to consume answers as soon as they are available [62]. As mentioned earlier, the proof system does not give us an explicit notion of failure and the table is not persistent across branches. In a real implementation, we make the table globally available. Moreover, we do not want to retract all our steps, when we discover a loop. Instead, we want to freeze the current proof state, suspend solving the goal and later resume computation. After some answers have been generated for the sequent $\Gamma \xrightarrow{u} P$, we awaken the suspended goal and resume computation of the pending sub-goals. Finally, we want to point out that although we used variant checking in the definitions, it is possible to extend and modify them to allow subsumption checking.

4.5.4 Related work and conclusion

This proof-theoretic view on computation based on memoization provides a high-level description of a tabled logic programming interpreter and separates logical issues from procedural ones leaving maximum freedom to choose particular control mechanisms. In fact, it is very close to our prototype implementation for *Elf*. So far all descriptions of tabling are highly procedural, either designed as an extension of SLD resolution [65] or to the WAM abstract machine[62]. Certificates, which provide evidence for the existence of a proof, have been added to tabled logic programming by Roychoudhury [61] and are called justifiers. The relationship between the certificate and SLD resolution is extrinsic rather then intrinsic and needs to be established separately. The proof-theoretical characterization offers a uniform framework for describing and reasoning about program clauses, goals and certificates (proof terms). It seems possible to apply the techniques described to other logic programming languages such as λProlog. Linear logic programming [30, 8] has been proposed as an extension of higher-order logic programming to model imperative state changes in a declarative (logical) way.

We believe our techniques can be extended to cover this case, but it requires some new considerations. In particular, we plan to investigate the interaction between resource management strategies [10] or constraints [27] with tabling.

With tabled uniform proof search we will find fewer proofs than with uniform proofs. For example in the subtyping example given in Sec. 4.1 the query sub zero zero has infinitely many proofs under the traditional logic programming interpretation while we find only one proof under the tabled logic programming interpretation. However, we often do not want and need to distinguish between different proofs for a formula A, but only care about the existence of a proof for A together with a proof term. In [50] Pfenning develops a dependent type theory for proof irrelevance and discusses potential applications in the logical framework. This allows us to treat all proofs for A as equal if they produce the same answer substitution. In this setting, it seems possible to show that search based on tabled uniform proofs is also non-deterministically complete, i.e. if computation fails, then there exists no proof.

4.6 Case studies

In this section, we discuss experiments with tabled higher-order logic programming. We compare the performance of proof search based memoization, depth-first search, iterative deepening using two applications: 1) bi-directional type checking using subtyping and intersection types and 2) parsing into higher-order abstract syntax. As the experiments demonstrate proof search based on memoization can lead to substantial performance improvements, making the execution of some queries feasible.

While computation using memoization yields better performance for programs with transitive closure or left-recursion, Prolog-style evaluation is more efficient for right recursion. For example, Prolog has linear complexity for a simple right recursive grammar, but with memoization the evaluation could be quadratic as calls need to be recorded in the tables using explicit copying. Therefore it is important to allow tabled and non-tabled predicates to be freely intermixed and be able to choose the strategy that is most efficient for the situation at hand. Hence in the current prototype, the user declares predicates to be tabled, if he/she wishes to use memoization for it. Mixing tabled and non-tabled predicates is essential, in order to obtain an efficient proof search engine.

4.6.1 Bidirectional typing (depth-first vs memoization)

In this section, we discuss experiments with a bi-directional type-checking algorithm for a small functional language with intersection types which has been developed by Davies and Pfenning [17]. Type inference in a functional language with subtyping and intersection types is usually considered impractical, as no principal types exist. The idea behind bi-directional type-checking is to distinguish expressions for which a type can be synthesized from expressions which can be checked against a given type. The programmer specifies some types to guide inferring a type for certain expressions.

$$\begin{array}{llll} \text{Inferable} & I & ::= & x \mid \epsilon \mid I\ \mathsf{0} \mid I\ \mathsf{1} \mid \mathsf{app}\ IC \mid C : A \\ \text{Checkable} & C & ::= & I \mid \mathsf{lam}\ x.C \mid \mathsf{let}\ u = I\ \mathsf{in}\ C \mid \\ & & & \mathsf{case}\ I\ \mathsf{o}f\ \epsilon\ \Rightarrow C_1 | x\ \mathsf{0} \Rightarrow C_2 | x\mathsf{1} \Rightarrow C_3 \end{array}$$

The intention is that given a context Γ and an expression I, we use type-inference to show expression I has type A and type-checking for verifying that expression C has type A. In an implementation of the bi-directional type checking algorithm, there may be many ways to derive that I has a type A and similarly there are more than one way to check that C has a given type A. To discuss the full bi-directional type-checking algorithm is beyond the scope of this paper and the interested reader is referred to the original paper by Davies and Pfenning [17].

We use an implementation of the bi-directional type-checker in *Elf* by Pfenning. The type-checker is executable with the original logic programming interpreter, which performs a depth-first search. However, redundant computation may severely hamper its performance as there are several derivations for proving that a program has a specified type. For example, there are approximately 20,000 ways to show that the program plus has the intersection type $(\mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{nat}) \wedge (\mathsf{nat} \rightarrow \mathsf{pos} \rightarrow \mathsf{pos}) \wedge (\mathsf{pos} \rightarrow \mathsf{nat} \rightarrow \mathsf{pos}) \wedge (\mathsf{pos} \rightarrow \mathsf{pos} \rightarrow \mathsf{pos})$. It might be argued that we are only interested in one proof for showing that plus has the specified type and not in all of them. However, it indicates that already fairly small programs involve a lot of redundant computation. More importantly than succeeding quickly may be to fail quickly, if plus has not the specified type. Failing quickly is essential in debugging programs and response times of several minutes are unacceptable. In the experiments, we are measuring the time it takes to explore the whole proof tree. This gives us an indication how much redundancy is in the search space. When checking a term C against a type A, we use memoization. The proof search based on memoization uses strengthening and variant checking. Using the refinement type-checker, we check programs for addition, subtraction, and multiplication. Note that in order to for type-check, for example, the multiplication program, we need to also type-check any auxiliary programs used such as addition and shift. Table 4.1 summarizes the results for finding all solutions to a type-checking problem. The number associated with each program name denotes the depth of the intersection type associated with it. For example, plus4 means we assigned 4 types by using intersections to plus. mult1a indicates we associated one possible type to the multiplication program. If a program label is marked with (np), it means this query does not have a solution and the type-checker should reject the query.

Program	Depth-First	Memoization	#Entries	#SuspGoals
plus'4	483.070 sec	2.330 sec	151	48
plus4	696.730 sec	3.150 sec	171	74
plus4(np)	22.770 sec	1.95 sec	143	56
sub'1a	0.070 sec	0.240 sec	58	11
sub'3a	0.130 sec	0.490 sec	92	20
sub1a	3.52 sec	7.430 sec	251	135
sub1b	3.88 sec	7.560 sec	252	138
sub3a	10.950 sec	9.970 sec	277	167
sub3b	10.440 sec	11.200 sec	278	170
mult1(np)	1133.490 sec	4.690 sec	217	83
mult1a	807.730 sec	4.730 sec	211	78
mult1b	2315.690 sec	6.050 sec	226	101
mult1c	2963.370 sec	5.310 sec	226	107
mult4	∞	17.900 sec	298	270
mult4(np)	∞	13.140 sec	275	194

Table 4.1: Finding all solutions: depth-first vs memoization-based search

For type-checking programs plus and multiplication proofs search based on memoization outperforms depth-first search. This is surprising, as no sophisticated indexing is used for accessing the table. It indicates that already simple memoization mechanism can substantially improve performance. In the case of multiplication, it makes type-checking possible. We stopped the depth-first search procedure after 10h. Of course proof search based on memoization has some overhead in storing and accessing goals in the table. As indicated with subtraction programs, this overhead might degrade the performance of memoization based proof search. When type-checking subtraction program depth-first search performs better than memoization-based search for the first 5 sample programs. In the last example sub3b however memoization-based search wins over depth-first search.

Program	Depth-First	Memoization	#Entries	#SuspGoals
plus'4	0.08 sec	0.180 sec	54	0
plus4	0.1 sec	0.430 sec	72	0
plus4(np)	22.770 sec	1.95 sec	143	56
sub'1a	0.050 sec	0.240 sec	64	11
sub'3a	0.110 sec	0.410 sec	92	20
sub1a	0.250 sec	6.210 sec	251	135
sub1b	0.250 sec	5.020 sec	242	121
sub3a	0.280 sec	7.80 sec	277	161
sub3b	0.350 sec	8.160 sec	278	164
mult1(np)	1133.490 sec	4.690 sec	217	83
mult1a	0.160 sec	2.900 sec	201	60
mult1b	0.180 sec	4.090 sec	222	90
mult1c	0.170 sec	2.930 sec	211	60
mult4	0.250 sec	7.150 sec	272	181
mult4(np)	∞	13.020 sec	275	194

Table 4.2: Finding the first solution: depth-first vs memoization-based search

We include also a comparison between the two search strategies, when we stop after the first answer has been found (see table 4.2). It is apparent that currently finding the first solution to a solvable type-checking problem (i.e., it is provable that the program has the specified type) always takes longer with proof search based on memoization – in some cases considerably longer (see subtraction and multiplication). This is an indication that accessing the table is still quite expensive in the current implementation. This comes as no real surprise, because no sophisticated techniques such as term indexing are used. This problem will be addressed in Chapter 5. The other reason for the poor performance of memoization-based search is due the multi-stage search strategy. Although this strategy is relatively easy to implement and understand, it restricts retrieval of answers present in the table to answers generated in previous stages. This causes subgoals to be suspended, although answers might be available, and solving those subgoals is delayed. For this reason, XSB uses the SCC (strongly connected component) scheduling strategy, which allows to re-use answers from the table as soon as they are available. But the benefits of memoization-based search are apparent when comparing the time it takes to reject a program by the type-checker as the examples plus4(np) and mult1(np) indicate. Overall, the performance of the memoization based search is much more consistent, i.e., it takes approximately the same time to accept or reject a program.

To make bi-directional type-checking reasonably efficient in practice, Davies and Pfenning currently investigate an algorithm which synthesizes all types of an inferable term and tracks applicable ones through the use of boolean constraints. This is however far from trivial and refining the *Elf* implementation by adding an extra argument to the type-checker to memoize solutions, complicates the type checker substantially. As a consequence, the certificates, which are produced as a result of the execution, are larger and contain references to the explicit memoization data-structure. This is especially undesirable in the context of certified code where certificates are transmitted to and checked by a consumer, as sending larger certificates takes up more bandwidth and checking them takes more time. Moreover, proving the correctness of the type-checker with special memoization support will be hard, because we need to reason explicitly about the structure of memoization. The experiments demonstrate that proof search based on memoization has the potential to turn the bi-directional type-checking algorithm into a quite efficient type-checker without any extra effort on behalf of the user.

4.6.2 Parsing (iterative deepening vs memoization)

Recognition algorithms and parsers for grammars are an excellent way to illustrate the benefits of tabled evaluation. Warren [68] notes that implementations of context-free grammars in Prolog result in a recursive descent recognizer, while tabled logic programming turns the same grammar into a variant of Early's algorithm (also known as active chart recognition algorithm) whose complexity is polynomial. Moreover, tabled logic programming allows us to execute left and right recursive grammars that would otherwise loop under Prolog-style execution (e.g. left recursive ones). We illustrate tabled computation with parsing of first-order formulas into higher-order abstract syntax. First-order formulas are defined as usual.

$$\begin{array}{lll}\text{Propositions} & A ::= & \mathsf{atom}\ P \mid \neg A \mid A\ \&\ A \mid A\ \mathsf{v}\ A \mid A \Rightarrow A \mid \mathsf{true} \mid \mathsf{false} \mid \\ & & \mathsf{forall}\ x.A \mid \mathsf{exists}\ x.A \mid (A)\end{array}$$

Terms are either constants or variables or functions with arguments. Atomic propositions are either propositional constant or a predicate with terms as arguments. In addition to the given grammar, we impose the following precedence ordering: $\neg >\ \& > \mathsf{v} > \Rightarrow$. Conjunction and disjunction are left associative, while implication is right associative.

```
fall: fq C ('forall' ; I ; F) F' (forall P)}
      <- ({x:id} fq ((bvar I x) # C) F F' (P x)).
fex:  fq C ('exist' ; I ; F) F' (exist P)}
      <- ({x:id} fq ((bvar I x) # C) F F' (P x)).
cq :  fq C F F' P
      <- fi C F F' P.
% implication -- right associative
fimp: fi C F F' (P1 => P2)
      <- fo C F ('imp' ; F1) P1
      <- fi C F1 F' P2.
ci:   fi C F F' P
      <- fo C F F' P.
% disjunction -- left associative
for: fo C F F' (P1 v P2)
     <- fo C F ('or' ; F1) P1
     <- fa C F1 F' P2.
```

The parser takes a context for the bound variables, a lists of tokens and returns a list of tokens and a valid formula represented in higher-order abstract syntax [52]. For example, in the predicate `fi C F F' P`, `C` represents the context for bound variables, `F` denotes the input stream, `F'` is a sub-list of the input stream s.t. `(F1 ; F')` = `F` and `F1` is translated into the formula `P`. The other predicates work similarly. The complete implementation of the parser is given in the appendix. Initially the context of bound variables is empty, the first list of tokens represents the input stream, the second list is empty and P will eventually contain the result. Using higher-order abstract syntax, variables bound in constructors such as forall and exists will be bound with λ in *Elf*. The simplest way to implement left and right associativity properties of implications, conjunction and disjunction is to mix right and left recursive program clauses. Clauses for conjunction and disjunction are left recursive, while the program clause for implication is right recursive.

Such an implementation of the grammar is straightforward mirroring the defined properties such as left and right associativity and precedence ordering. However, the execution of the grammar will loop infinitely when executed with a traditional logic programming interpreter. Hence we compare execution of some sample programs using proof search based on memoization and with proof search based on iterative deepening, as performed by the current theorem prover *Twelf* [64]. Iterative deepening search requires the user to provide a depth-bound. It is worth pointing out that iterative deepening search will stop after it found its first solution or it hits a depth-bound, while search based on memoization will stop after it found a solution (and showed that no other solution exists) or proved no other solution exists. This means, we cannot use iterative deepening search to decide whether a given stream of tokens should be accepted or not, while the memoization-based search yields a decision procedure.

The experiments indicate that proof search based on memoization provides a more efficient way to decide whether a given stream of tokens belongs to the language of formulas. In fact for input streams whose length is greater than 50 tokens, we stopped the iterative deepening procedure after several hours.

Remark 1: It is worth noting that in general the straightforward approach of adding an extra argument to the non-terminals of the input grammar – representing the portion of the parse tree that each rule generates – and naturally to also add the necessary code that constructs the parse tree can be extremely unsatisfactory from a complex-

Length of input	Iter. deepening	Memoization	#Entries	#SuspGoals
5	0.020 sec	0.010 sec	15	11
20	1.610 sec	0.260 sec	60	54
32	208.010 sec	2.020 sec	176	197
56	∞	7.980 sec	371	439
107	∞	86.320 sec	929	1185

Table 4.3: Comparison between iterative deepening and memoization based search

ity standpoint. Polynomial recognition algorithms might be turned into exponential algorithm since there may be exponentially many parse trees for a given input string. However in this example, there is exactly one parse trace associated to each formula, therefore adding an extra argument to the recognizer only adds a constant factor.

Remark 2: Instead of representing the input string as a list, we can store it in the database as a set of facts. We can think of each token in the input stream being numbered starting from 1. Then we will store the string as a set of facts of the form `word 1 'forall'. word 2 'x'.` etc. where `word i tok_i` represents the ith token of the input stream.

4.7 Related work and conclusion

A number of different frameworks similar to *Elf* have been proposed such as λProlog [39, 24] or Isabelle [45, 46]. While *Elf* is based on the LF type theory, λProlog and Isabelle are based on hereditary Harrop formulas. The traditional approach for supporting theorem proving in these frameworks is to guide proof search using tactics and tacticals. Tactics transform a proof structure with some unproven leaves into another. Tacticals combine tactics to perform more complex steps in the proof. Tactics and tacticals are written in ML or some other strategy language. To reason efficiently about some specification, the user implements specific tactics to guide the search. This means that tactics have to be rewritten for different specifications. Moreover, the user has to understand how to guide the prover to find the proof, which often requires expert knowledge about the systems. Proving the correctness of the tactic is itself a complex theorem proving problem. The approach taken in *Elf* is to endow the framework with the operational semantics of logic programming and design general proof search strategies for it. The user can concentrate on developing the high-level specification rather than getting the proof search to work. The correctness of the implementation is enforced by type-checking alone. The preliminary experiments demonstrate that proof search based on memoization offers a powerful search engine.

Chapter 5

Higher-order term indexing

Efficient term indexing techniques have resulted in dramatic speed improvements in all major first-order logic programming and theorem proving systems and have been crucial to their success. Broadly speaking, term indexing techniques facilitate the rapid retrieval of a set of candidate terms satisfying some property (e.g. unifiability, instance, variant etc.) from a large collection of terms. In logic programming, for example, we need to select all clauses from the program which unify with the current goal. In tabled logic programming we memoize intermediate goals in a table and reuse their results later in order to eliminate redundant and infinite computation. Here we need to find all entries in the table such that the current goal is a variant or an instance of a table entry and re-use the associated answers. If there is no such table entry, we need to add the current goal to the table. Since rapid retrieval and efficient storage of large collection of terms plays a central role in logic programming and in proof search in general, a variety of indexing techniques have been proposed for first-order terms (see [60] for a survey). However, indexing techniques for higher-order terms are missing thereby severely hampering the performance of higher-order systems such as Twelf [53], λProlog [39] or Isabelle [45].

In this chapter, we present a higher-order term indexing technique based on substitution trees. Substitution tree indexing [25] is a highly successful first-order term indexing strategy which allows the sharing of common sub-expressions via substitutions. Given the following two terms:

$$\mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{a}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{a} \text{ and } \mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{a}$$

we can compute the most specific linear generalization (mslg) of both terms, which is

$$\mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ *))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{a}$$

where $*$ is a placeholder. Then we obtain the first term by substituting a for the placeholder and the second term by substituting b for it.

In this chapter we present an indexing technique for terms of the dependently typed lambda-calculus. on computing the most specific linear generalization of two terms. However in the higher-order setting, the most specific linear generalization of two terms does not exist in general. Second, retrieving all terms, which unify or match, needs to be efficient – but higher-order unification is undecidable in general. As discovered by Miller [34], there exists a decidable fragment, called higher-order patterns. For this fragment, unification and computing the most specific linear generalization is decidable even in rich type theories with dependent types and polymorphism as shown by Pfenning [48]. However, these algorithms may not be efficient in practice [57] and hence it is not obvious that they are suitable for higher-order term indexing techniques.

In this chapter, we present substitution tree indexing for higher-order terms based on linear higher-order patterns (see Chapter 3). Linear higher-order patterns refine the notion of higher-order patterns further and factor out any computationally expensive parts. As we have shown in Chapter 3, many terms encountered fall into this fragment and linear higher-order pattern unification performs well in practice. In this chapter, we give algorithms for computing the most specific linear generalization of two linear higher-order patterns, for inserting terms in the index and for retrieving a set of terms from the index such that the query is an instance of the term in the index. This indexing structure is implemented as part of the Twelf system [53] to speed-up the execution of the tabled logic programming interpreter [55]. Experimental results show substantial performance improvements, between 100% and over 800%.

The chapter is organized as follows: In Section 5.1, we present the general idea of higher-order substitution trees. In Section 5.2, we give algorithms for computing the most specific linear generalization of two terms and inserting terms into the index. Retrieval is discussed in Section 5.3. In Section 5.4, we present some experimental results comparing the tabled higher-order logic programming interpreter with and without indexing.

5.1 Higher-order substitution trees

The general indexing problem can be described as follows: Given a large set $\mathcal{S}$ of indexed terms and a single term U called the *query term*, we have to retrieve quickly each term $V \in \mathcal{S}$ such that a *retrieval condition* R holds between U and V. In general the retrieval condition can be unifiability, instance or variant checking, or finding generalizations. The term indexing problem consists of finding a data-structure, called the *index*, which allows one to perform efficiently the following operations: 1) *index maintenance*, i.e. changing the index when terms are inserted (or deleted) from the set of indexed terms. 2) *term retrieval* i.e. finding some (or all) terms $V \in \mathcal{S}$ s.t. V and the query term U are in relation to each other.

To illustrate the notation, we give a sample signature together with a set of terms we want to index. We define a type family exp for expressions. In addition, we give constants a, b, c and constructors f, g, h for building expressions of this small language.

```
exp: type.
a: exp.                     h: exp -> exp.
b: exp.                     g: exp -> exp.
c: exp.                     f: (exp -> exp) -> exp.
```

Finally, we define a predicate pred which takes in three arguments, where each argument is an expression

```
pred: exp -> exp -> exp -> type.
```

Next, we give four examples of the predicate pred:

$$\begin{array}{rll} & \mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{a} & (1) \\ & \mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ \mathsf{b} & (2) \\ \Pi^{\Box} v{::}\cdot \vdash \mathsf{exp}. & \mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.v[\cdot]) & (3) \\ \Pi^{\Box} u{::}x{:}\mathsf{exp} \vdash \mathsf{exp}. & \mathsf{pred}\ (\mathsf{h}\ (\mathsf{h}\ \mathsf{c}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.u[\mathsf{id}]) & (4) \end{array}$$

The first two predicates are closed, while predicate (3) and (4) contain existential variables v and u respectively which are bound by $\Pi^{\Box}$. Using these examples, we will briefly highlight some of the subtle issues concerning the interplay of bound and existential variables. The third predicate (3) refers to the existential variable v. Note that v is associated with the empty substitution, although it occurs in the context of a bound variable x. This means that any instantiation we find for v is not allowed to depend on the bound variable x. In contrast, predicate (4) refers to an existential variable u, which is associated with the identity substitution. This means that any instantiation we find for u may depend on the bound variable x. We come back to this point in Section 5.1.1. All these terms share some structural information. For example, the third argument is same in all the terms. When storing these terms, we would like to store the subterm $(\mathsf{g}\ \mathsf{b})$ only once. Similarly, when looking up if a query U is already in the index, we only want to check once if the third argument of U is $(\mathsf{g}\ \mathsf{b})$. In other words, an indexing data-structure should allow us to share common structure and common operation.

In this chapter, we will focus on substitution tree indexing. We start by giving the general idea higher-order substitution tree. To build a higher-order substitution tree, we proceed in two steps: First, we standardize the terms and convert terms into linear higher-order patterns. Second, we represent terms as a sequence of substitutions, which are stored in a tree.

5.1.1 Standardization: linear higher-order patterns

To get the maximum structure sharing across different indexed terms it is important to use variables in a consistent manner. In first-order term indexing we therefore standardize the term before inserting a term into an index. In addition, first-order term indexing strategies often employ linearization. When converting the term into standard form every occurrence of an existential variable is represented as a distinct standardized existential variable. Together with the linear term, we then also store variable definitions, which establish the equality between these two variables. One of the reasons for using linearization is efficiency. Nonlinear terms may have multiple occurrences of the same existential variable and therefore requiring to check whether different substitutions for the same variable are consistent. Since such consistency checks can be potentially expensive and may lead to performance degradation, most first-order indexing techniques rely on a post-processing step to carry out the consistency checks.

To design a higher-order indexing technique, we will extend this notion of linearization and standardization. We will require that terms are linear higher-order patterns.

Higher-order patterns [34, 48] are terms where every existential variable must be applied to *some* distinct bound variables. Linear higher-order patterns (see Chapter 3) impose some further restrictions on the structure of terms: First, all existential variables must occur only once. This allows us to delay any expensive consistency checks. Second, all existential variables must be applied to *all* distinct bound variables. This eliminates any computationally expensive checks involving bound variables. This observation to restrict higher-order patterns even further to patterns where existential variables must be applied to all bound variables has also been made by Hanus and Prehofer [26] in the context of higher-order functional logic programming. While Hanus and Prehofer syntactically disallow terms which are not fully applied, we translate any term into a linear higher-order pattern together with some variable definitions. As we have shown in Chapter 3, performance of unification is improved substantially by this technique.

To illustrate, let us consider the previous term $\mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.v[\cdot])$. The existential variable $v[\cdot]$ occurs only once, but is not applied to all distinct bound variables, since v is not allowed to depend on the bound variable x. Therefore $v[\cdot]$ is a higher-order pattern, but it is not linear. We can enforce that every existential variable occurs only once and is applied to *all* bound variables, by translating it into a linear higher-order pattern:

$$\mathsf{pred}\ (\mathsf{h}\ (\mathsf{g}\ \mathsf{b}))\ (\mathsf{g}\ \mathsf{b})\ (\mathsf{f}\ \lambda x.u[x/x])$$

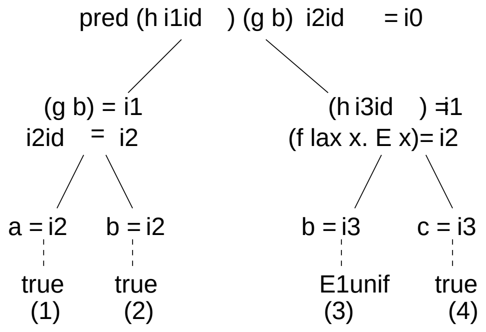
together with a variable definition,

$$\forall x{:}\mathsf{exp}.u[x/x] \overset{D}{=} v[\cdot]$$

where u is a new existential variable which is applied to the bound variable x. Similarly to first-order standardization and linearization, linearization in the higher-order case remains quite straightforward and can be done by traversing the term once. The main difference is that the postponed variable definitions may be more complex, as we have discussed earlier in Chapter 3.

5.1.2 Example

In this section, we show the substitution tree for the set of terms given earlier. In each node, we store a set of substitutions, which we write here as $U = i$ where i is an internal modal variable and i will always be applied to all bound variables it may depend on. Therefore it will always be associated with the identity substitution id.



Note that variable definitions are stored at the leafs. By composing the substitutions in the right-most branch for example we get

$$[\![\mathsf{c}/i_3]\!][\![\mathsf{f}\ \lambda x.\ u[\mathsf{id}]/i_2,\ (\mathsf{h}\ i_3[\mathsf{id}])/i_1]\!][\![\mathsf{pred}\ (\mathsf{h}\ i_1[\mathsf{id}])\ (\mathsf{g}\ \mathsf{b})\ i_2[\mathsf{id}]/i_0]\!]i_0[\mathsf{id}]$$

which represents the term (4) pred (h (h c)) (g b) (f $\lambda x.\ u$[id]). Note that in the implementation we can omit the identity substitution $i_2[\mathsf{id}]/i_2$ and maintain as an invariant that the substitutions in each node can be extended appropriately.

Higher-order substitution trees are designed for linear higher-order patterns. As discussed in Chapter 3, linearization of a term may lead to a linear term which is not necessarily dependently typed. However, the term is approximately well-typed when all the dependencies among types are erased. In Chapter 2, we have used erasure of type dependencies to describe a definitional equality algorithm. Here, we apply this idea of approximate types to define higher-order substitution trees. The algorithms for insertion and retrieval in substitution trees are based on mslg and unifiability, but types itself do not play a role when computing the mslg and unifiers. We know the term is well-typed before it is inserted into the substitution tree, and it will be well-typed, once all the linear substitutions on one path are composed to obtain the original term. We may think of linear terms as a representation which is only used internally within substitution trees, but not externally during proof search. It suffices to show that approximate types are preserved in substitution trees, and all intermediate variables introduced are only used within this data-structure, but do not leak outside. We will write Λ for the modal context Δ where all the type dependencies have been erased. Similarly, we write Ω for the ordinary context Γ where all the dependencies have been erased.

In the definition of higher-order substitution trees we will distinguish between a modal context Λ which denotes the original modal variables and a modal context Σ for the internal modal variables. Similarly to Λ, Σ denotes a modal context where all the dependencies have been erased. Note that the modal variables in Σ have no dependencies among each other and can be arbitrarily re-ordered. Moreover, every internal modal variable i will be created in such a way that it is applied to all bound variables, i.e. it will be associated with the identity substitution id. A higher-order substitution tree is a node with substitution ρ such that $(\Lambda, \Sigma) \vdash \rho : (\Lambda, \Sigma')$ and every child node has a substitution ρ_i such that $(\Lambda, \Sigma_i) \vdash (\mathsf{id}_\Lambda, \rho_i) : (\Lambda, \Sigma)$. At the leaf, we have a substitution ρ such that $\Lambda \vdash (\mathsf{id}_\Lambda, \rho) : (\Lambda, \Sigma)$.

Moreover, for every path from the top node ρ_0 where $(\Lambda, \Sigma_1) \vdash \rho_0 : (\Lambda, \Sigma_0)$ to the leaf node ρ_n, we have $\Lambda \vdash [\![\rho_n]\!]([\![\rho_{n-1}]\!] \ldots \rho_0) : (\Lambda, \Sigma_0)$. In other words, there are no internal modal variables left after we compose all the substitutions ρ_n up to ρ_0. Moreover, let δ be the modal substitution corresponding to the variable definitions, then the term, which we obtain after composing the modal substitution δ with all the substitutions ρ_n up to ρ_0, will be well-typed.

Inserting a new term U into the substitution tree corresponds to inserting the substitution $U = i_0$. Before presenting the algorithms for building substitution trees, we discuss our reasons for adapting substitution tree indexing to the higher-order setting. First, we note that the order of term traversal is not fixed in advance. For example, in the substitution tree given earlier we compare the substitution for the third argument before the substitutions for the first argument when looking up the term (3) and term (4). While we traverse the term (1) and (2) from left to right. This feature leads to very compact substitution trees and better memory usage and retrieval times. However, in general there may be multiple ways to insert a term and no optimal substitution trees exist. In contrast to other indexing techniques such as discrimination tries, substitution trees allows the sharing of common sub-expressions rather than common prefixes.

This is especially important for indexing dependently typed terms. To illustrate this point, we define a data-structure for lists consisting of characters and we keep track of the size of the list by using dependent types.

$$\begin{array}{ll}
\mathsf{char} : \mathsf{type}. & \mathsf{list} : \mathsf{char} \to \mathsf{type}. \\
\mathsf{a} : \mathsf{char}. & \mathsf{nil} : \mathsf{list}\ 0. \\
\mathsf{b} : \mathsf{char}. & \mathsf{cons} : \Pi^{\Box} n{::}\mathsf{int}\,.\mathsf{char} \to \mathsf{list}\ n \to \mathsf{list}\ (n+1). \\
\mathsf{test} : \Pi^{\Box} n{::}\mathsf{int}\,.\mathsf{list}\ n \to \mathsf{type}. &
\end{array}$$

The size of lists is an explicit argument to the predicate test. Hence test takes in two arguments, the first one is the size of the list and the second one is the actual list. The list constructor cons takes in three arguments. The first one denotes the size of the list, the second argument denotes the head and the third one denotes the tail. To illustrate, we give a few examples. We use gray color for the explicit arguments.

$$\begin{array}{l}
\mathsf{test}\ 4\ (\mathsf{cons}\ 4\ a\ (\mathsf{cons}\ 3\ a\ (\mathsf{cons}\ 2\ a\ (\mathsf{cons}\ 1\ b\ \mathsf{nil})))) \\
\mathsf{test}\ 5\ (\mathsf{cons}\ 5\ a\ (\mathsf{cons}\ 4\ a\ (\mathsf{cons}\ 3\ a\ (\mathsf{cons}\ 2\ a\ (\mathsf{cons}\ 1\ b\ \mathsf{nil}))))) \\
\mathsf{test}\ 6\ (\mathsf{cons}\ 6\ a\ (\mathsf{cons}\ 5\ a\ (\mathsf{cons}\ 4\ a\ (\mathsf{cons}\ 3\ b\ (\mathsf{cons}\ 2\ a\ (\mathsf{cons}\ 1\ b\ \mathsf{nil}))))))
\end{array}$$

If we use non-adaptive indexing techniques such as discrimination tries, we process the term from left to right and we will be able to share common prefixes. In the given example, such a technique discriminates on the first argument, which denotes the size of the list and leads to no sharing between the second argument. The substitution tree on the other hand allows us to share the structure of the second argument. The most specific linear generalization in this example is

$$\mathsf{test}\ i_1[\mathsf{id}]\ (\mathsf{cons}\ i_2[\mathsf{id}]\ a\ (\mathsf{cons}\ i_3[\mathsf{id}]\ a\ (\mathsf{cons}\ i_4[\mathsf{id}]\ a\ (\mathsf{cons}\ i_5[\mathsf{id}]\ i_6[\mathsf{id}]\ \mathsf{nil})))).$$

This allows us to skip over the implicit first argument denoting the size and indexing on the second argument, the actual list. It has been sometimes argued that it is possible to retain the flexibility in non-adaptive indexing techniques by reordering the arguments to test. However, this only works easily in an untyped setting and it is not clear how to maintain typing invariants in a dependently typed setting if we allow arbitrary reordering of arguments. Hence higher-order substitution trees offer a adaptive compact indexing data-structure while maintaining typing invariants.

5.2 Insertion

Insertion of a term U into the index is viewed as insertion of the substitution U/i_0. Assuming that U has type α in a modal context Λ and a bound variable context Ω, U/i_0 is a modal substitution such that $\Lambda \vdash U/i_0 : i_0{::}(\Omega{\vdash}\alpha)$. We will call the modal substitution which is going to be inserted into the index ρ. It is often convenient to consider the extended modal substitution $\Lambda \vdash \mathsf{id}_\Lambda, U/i_0 : (\Lambda, i_0{::}(\Omega{\vdash}\alpha))$. This will simplify the following theoretical development. Again we note that in an implementation, we do not need to carry around explicitely the modal substitution id_Λ, but can always assume that any substitution can be extended appropriately.

The insertion process works by following down a path in the tree that is *compatible* with the modal substitution ρ. To formally define insertion, we show how to compute the most specific linear generalization (mslg) of two modal substitutions and describe the most specific linear generalization of two terms. We start by giving the judgments for computing the most specific linear generalization of two modal substitutions.

$$(\Lambda, \Sigma_1) \vdash \rho_1 \sqcup \rho_2 : \Sigma_2 \Longrightarrow \rho/(\Sigma, \theta_1, \theta_2) \quad \rho \text{ is the mslg of } \rho_1 \text{ and } \rho_2$$

If $(\Lambda, \Sigma_1) \vdash (\mathsf{id}_\Lambda, \rho_1) : (\Lambda, \Sigma_2)$ and $(\Lambda, \Sigma_1) \vdash (\mathsf{id}_\Lambda, \rho_2) : (\Lambda, \Sigma_2)$ then ρ is the mslg of ρ_1 and ρ_2. Moreover, ρ is a modal substitution such that $[\![\mathsf{id}_\Lambda, \theta_1]\!]\rho$ is structurally equal to ρ_1 and $[\![\mathsf{id}_\Lambda, \theta_2]\!]\rho$ is structurally equal to ρ_2 and $(\Lambda, \Sigma) \vdash \rho : \Sigma_2$. We think of ρ_1 as the modal substitution which is already in the index, while the modal substitution ρ_2 is to be inserted. As a consequence, only ρ_1 will refer to the internal modal variables in Σ_1, while ρ_2 only depends on the modal variables in Λ. The result of the mslg are the modal substitution θ_1 and θ_2, where $\Lambda, \Sigma_1 \vdash \theta_1 : \Sigma$ and $\Lambda, \Sigma_1 \vdash \theta_2 : \Sigma$. In other words, θ_1 (resp. θ_2) only replaces internal modal variables in Σ. Note that any modal substitution ρ or θ with domain Σ, can be extended to a modal substitution $(\mathsf{id}_\Lambda, \rho)$ (or $(\mathsf{id}_\Lambda, \theta)$ resp.) with domain (Λ, Σ).

First, we give the rules for computing the most specific linear generalization of two modal substitutions.

$$\frac{}{(\Lambda, \Sigma) \vdash \cdot \sqcup \cdot : \cdot \Longrightarrow \cdot/(\cdot, \cdot, \cdot)}$$

$$\frac{(\Lambda, \Sigma_1) \vdash \rho_1 \sqcup \rho_2 : \Sigma_2 \Longrightarrow \rho/(\Sigma, \theta_1, \theta_2) \quad (\Lambda, \Sigma_1); \Omega \vdash L_1 \sqcup L_2 : \tau \Longrightarrow L/(\Sigma', \theta_1', \theta_2')}{(\Lambda, \Sigma_1) \vdash (\rho_1, L_1/i) \sqcup (\rho_2, L_2/i) : (\Sigma_2, i{::}(\Omega{\vdash}\tau)) \Longrightarrow (\rho, L/i)/((\Sigma, \Sigma'), (\theta_1, \theta_1'), (\theta_2, \theta_2'))}$$

Note, that we are allowed to just combine the modal substitutions θ_1 (θ_2 resp.) and θ_1' (θ_2' resp.) since we require that they refer to distinct modal variables and all the modal variables occur uniquely. Computing the most specific linear generalization of two modal substitutions relies on finding the most specific linear generalization of two objects. Note that we require that all objects are linear higher-order patterns and are in quasi-canonical form (see Chapter 2). Quasi-canonical forms are $\beta\eta$-normal forms where the type information on the bound variables in lambda-abstraction may be omitted. Moreover, we assume that all modal variables are lowered and have atomic type. We will use the spine notation introduced earlier in Chapter 4.

$$\begin{array}{lll} (\Lambda, \Sigma); \Omega \vdash L_1 \sqcup L_2 : \tau & \Longrightarrow \quad L/(\Sigma',\ \theta_1, \theta_2) & L \text{ is the mslg of } L_1 \text{ and } L_2 \\ (\Lambda, \Sigma); \Omega \vdash S_1 \mathrel{\underline{\sqcup}} S_2 : \tau > \alpha & \Longrightarrow \quad S/(\Sigma',\ \theta_1, \theta_2) & S \text{ is the mslg of } S_1 \text{ and } S_2 \end{array}$$

If the terms L_1 and L_2 have type τ in modal context (Λ, Σ) and bound variable context Ω, then L is the most specific linear generalization of L_1 and L_2 such that $[\![\mathsf{id}_\Lambda, \theta_1]\!]L$ is structurally equal to L_1 and $[\![\mathsf{id}_\Lambda, \theta_2]\!]L$ is structurally equal to L_2. Moreover, θ_1 and θ_2 are modal substitutions which map modal variables from Σ' to the modal context (Λ, Σ). Finally, $(\Lambda, \Sigma'); \Omega \vdash L : \tau$. For spines a similiar invariant holds. If S_1 and S_2 are spines from heads of type τ to terms of type α, then S is the mslg of S_1 and S_2 such that $[\![\mathsf{id}_\Lambda, \theta_1]\!]S$ is structurally equal to S_1 and $[\![\mathsf{id}_\Lambda, \theta_2]\!]S$ is structurally equal to S_2. θ_1 and θ_2 are modal substitutions which map modal variables from Σ' to the modal context (Λ, Σ).

We think of L_1 (or S_1) as an object which is already in the index and L_2 (or S_2) is the object to be inserted. As a consequence, only L_1 (and S_1) may refer to the internal variables in Σ, while L_2 (and S_2) only depends on Λ. The inference rules for computing the mslg are given next.

Normal linear objects

$$\frac{(\Lambda,\Sigma);\Omega, x{:}\tau_1 \vdash L_1 \sqcup L_2 : \tau_2 \Longrightarrow L/(\Sigma', \theta_1, \theta_2)}{(\Lambda,\Sigma);\Omega \vdash \lambda x.L_1 \sqcup \lambda x.L_2 : \tau_1 \to \tau_2 \Longrightarrow \lambda x.L/(\Sigma',\ \theta_1, \theta_2)}$$

$$\frac{u{::}(\Phi{\vdash}\alpha) \in \Lambda}{(\Lambda,\Sigma);\Omega \vdash u[\pi] \sqcup u[\pi] : \alpha \Longrightarrow u[\pi]/(\cdot,\cdot,\cdot)}\ (*)$$

$$\frac{u{::}\Phi{\vdash}\alpha \in \Lambda \quad i \text{ must be new} \quad L \neq u[\pi]}{(\Lambda,\Sigma);\Omega \vdash u[\pi] \sqcup L : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega{\vdash}\alpha,\ u[\pi]/i,\ L/i)}\ (1a)$$

$$\frac{i{::}\Omega{\vdash}\alpha \in \Sigma}{(\Lambda,\Sigma);\Omega \vdash i[\mathsf{id}_\Omega] \sqcup L : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega{\vdash}\alpha,\ i[\mathsf{id}_\Omega]/i,\ L/i)}\ (1b)$$

$$\frac{u{::}(\Phi{\vdash}\alpha) \in \Lambda \quad L \neq i[\mathsf{id}] \quad L \neq u[\pi]}{(\Lambda,\Sigma);\Omega \vdash L \sqcup u[\pi] : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega{\vdash}\alpha,\ L/i,\ u[\pi]/i)}\ (2)$$

$$\frac{(\Lambda,\Sigma);\Omega \vdash S_1 \sqcup\!\!\!\!\_\ S_2 : \tau > \alpha \Longrightarrow S/(\Sigma', \theta_1, \theta_2)}{(\Lambda,\Sigma);\Omega \vdash H \cdot S_1 \sqcup H \cdot S_2 : \alpha \Longrightarrow H \cdot S/(\Sigma', \theta_1, \theta_2)}\ (3a)$$

$$\frac{H_1 \neq H_2 \quad i \text{ must be new}}{(\Lambda,\Sigma);\Omega \vdash H_1 \cdot S_1 \sqcup H_2 \cdot S_2 : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/((i{::}\Omega{\vdash}\alpha), (H_1 \cdot S_1/i), (H_2 \cdot S_2/i))}\ (3b)$$

Normal linear spines

$$\frac{}{(\Lambda,\Sigma);\Omega \vdash \mathsf{nil} \sqcup\!\!\!\!\_\ \mathsf{nil} : \alpha > \alpha \Longrightarrow \mathsf{nil}/(\cdot,\cdot,\cdot)}$$

$$\frac{\begin{array}{c}(\Lambda,\Sigma);\Omega \ \vdash\ L_1 \sqcup L_2 \ :\ \tau_1 \ \Longrightarrow\ L/(\Sigma_1,\ \ \theta_1, \theta_1') \\ (\Lambda,\Sigma);\Omega \ \vdash\ S_1 \sqcup\!\!\!\!\_\ S_2 : \tau_2 > \alpha \Longrightarrow S/(\Sigma_2,\ \theta_2, \theta_2')\end{array}}{(\Lambda,\Sigma);\Omega \vdash (L_1; S_1) \sqcup\!\!\!\!\_\ (L_2; S_2) : \tau_1 \to \tau_2 > \alpha \Longrightarrow (L; S)/((\Sigma_1, \Sigma_2), (\theta_1, \theta_2), (\theta_1', \theta_2'))}\ (4)$$

Note in the rule for lambda, we do not need to worry about capture, since modal variables and bound variables are defined in different context. Rule (*) treats the case where both terms are existential variables. Note that we require that both existential variables must be the same and their associated substitutions must also be equal. In rule (1) and (2), we just create the substitution $u[\pi]/i$. In general, we would need to create $[\mathsf{id}_\Omega]^{-1}(u[\pi])$, but since we know that π is a permutation substitution, we know that $[\mathsf{id}_\Omega]^{-1}(\pi)$ exists. In addition, the inverse substitution of the identity is the identity. Note that we distinguish between the internal modal variables i and the "global" modal variables u in the rules (1a) and (1b). The key distinction is that we pick a new internal modal variable i in rule (1a) while we re-use the internal modal variable i in rule (1b). This is important for maintaining the invariant that any child of $(\Lambda, \Sigma_2) \vdash (\mathsf{id}_\Lambda, \rho) : (\Lambda, \Sigma_1)$ has the form $(\Lambda, \Sigma_3) \vdash (\mathsf{id}_\Lambda, \rho') : (\Lambda, \Sigma_2)$ during insertion (see the insertion algorithm later on).

In rule (3a) and (3b) we distinguish on the head symbol H and compute the most specific linear generalization of two objects $H_1 \cdot S_1$ and $H_1 \cdot S_1$. If H_1 and H_2 are not equal, then we generate a new internal modal variable $i[\mathsf{id}_\Omega]$ together with the substitutions $H_1 \cdot S_1/i$ and $H_2 \cdot S_2/i$. Otherwise, we traverse the spines S_1 and S_2 and compute the most specific linear generalization of them. In rule (4), we can just combine the substitution θ_1 and θ_2, as we require that all modal variables occur uniquely, and hence there are no dependencies among Σ_1 and Σ_2.

Definition 71 (Compatibility of normal objects) *Two normal objects L_1 and L_2 are incompatible, if $(\Lambda, \Sigma); \Omega \vdash L_1 \sqcup L_2 : \tau \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega{\vdash}\tau, L_1/i, L_2/i)$. Otherwise, we call L_1 and L_2 compatible.*

In other words, we call two terms compatible, if they share at least the head symbol or a λ-prefix. Moreover, if two terms are incompatible then they must be syntactially different. Similarly, we can define the compatibility of two substitutions.

Definition 72 (Compatibility of modal substitutions)
Two modal substitutions $(\Lambda, \Sigma_1) \vdash \rho_1 : \Sigma_2$ and $(\Lambda, \Sigma_1) \vdash \rho_2 : \Sigma_2$ are incompatible, if $(\Lambda, \Sigma_1) \vdash \rho_1 \sqcup \rho_2 : \Sigma_2 \Longrightarrow \mathsf{id}_\Sigma/(\Sigma, \rho_1, \rho_2)$. Otherwise, we call ρ_1 and ρ_2 compatible.

As a consequence, if ρ_1 and ρ_2 are incompatible, then for any $L_1/i \in \rho_1$ and $L_2/i \in \rho_2$, we know that L_1 and L_2 are incompatible. Next, we prove that we can always extend modal substitutions.

Lemma 73

1. *Let $\Sigma \vdash \sigma : \Sigma'$. If $\Lambda \vdash \theta : \Sigma$, and $\Lambda \vdash (\theta_1, \theta) : (\Sigma_1, \Sigma)$ and $\Lambda \vdash (\theta, \theta_2) : (\Sigma, \Sigma_2)$ then $[\![\theta]\!]\sigma = [\![\theta_1, \theta]\!]\sigma = [\![\theta, \theta_2]\!]\sigma$.*

2. *If* $\Lambda \vdash \theta : \Sigma$, *and* $\Lambda \vdash (\theta_1, \theta) : (\Sigma_1, \Sigma)$ *and* $\Lambda \vdash (\theta, \theta_2) : (\Sigma, \Sigma_2)$ *then* $[\![\theta]\!]L = [\![\theta_1, \theta]\!]L = [\![\theta, \theta_2]\!]L$.

3. *If* $\Lambda \vdash \theta : \Sigma$ *and* $\Lambda \vdash (\theta_1, \theta) : (\Sigma_1, \Sigma)$ *and* $\Lambda \vdash (\theta, \theta_2) : (\Sigma, \Sigma_2)$ *then* $[\![\theta]\!]S = [\![\theta_1, \theta]\!]S = [\![\theta, \theta_2]\!]S$.

Proof: Induction on the structure of σ, L and S. □

Theorem 74 (Soundness of mslg for objects)

1. *If* $(\Lambda, \Sigma); \Omega \vdash L_1 \sqcup L_2 : \tau \Longrightarrow L/(\Sigma', \theta_1, \theta_2)$ *and* $(\Lambda, \Sigma); \Omega \vdash L_1 : \tau$ *and* $(\Lambda, \Sigma); \Omega \vdash L_2 : \tau$ *then* $(\Lambda, \Sigma) \vdash \theta_1 : \Sigma'$ *and* $(\Lambda, \Sigma) \vdash \theta_2 : \Sigma'$ *and* $L_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]L$ *and* $L_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!]L$ *and* $(\Lambda, \Sigma'); \Omega \vdash L : \tau$.

2. *If* $(\Lambda, \Sigma); \Omega \vdash S_1 \sqcup\!\!\!\!\_ \ S_2 : \tau > \alpha \Longrightarrow S/(\Sigma', \theta_1, \theta_2)$ *and* $(\Lambda, \Sigma); \Omega \vdash S_1 : \tau > \alpha$ *and* $(\Lambda, \Sigma); \Omega \vdash S_2 : \tau > \alpha$ *then* $(\Lambda, \Sigma) \vdash \theta_1 : \Sigma'$ *and* $(\Lambda, \Sigma) \vdash \theta_2 : \Sigma'$ *and* $S_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]S$ *and* $S_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!]S$ *and and* $(\Lambda, \Sigma'); \Omega \vdash S : \tau > \alpha$.

Proof: Simultanous induction on the structure of the first derivation. We give here a few cases.

Case $\mathcal{D} = (\Lambda, \Sigma); \Omega \vdash \lambda x.L_1 \sqcup \lambda x.L_2 : \tau_1 \to \tau_2 \Longrightarrow \lambda x.L/(\Sigma', \theta_1, \theta_2)$

$(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L_1 \sqcup L_2 : \tau_2 \Longrightarrow L/(\Sigma', \theta_1, \theta_2)$	by premise
$(\Lambda, \Sigma); \Omega \vdash \lambda x.L_1 : \tau_1 \to \tau_2$	by assumption
$(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L_1 : \tau_2$	by inversion
$(\Lambda, \Sigma); \Omega \vdash \lambda x.L_2 : \tau_1 \to \tau_2$	by assumption
$(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L_2 : \tau_2$	by inversion
$(\Lambda, \Sigma) \vdash \theta_1 : \Sigma'$	by i.h.
$(\Lambda, \Sigma) \vdash \theta_2 : \Sigma'$	by i.h.
$L_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]L$	by i.h.
$\lambda x.L_1 = \lambda x.[\![\mathsf{id}_\Lambda, \theta_1]\!]L$	by rule
$\lambda x.L_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!](\lambda x.L)$	by modal substitution definition
$L_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!]L$	by i.h.

$\lambda x.L_2 = \lambda x.[\![\mathsf{id}_\Lambda, \theta_2]\!]L$ by rule
$\lambda x.L_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!](\lambda x.L)$ by modal substitution definition
$(\Lambda; \Sigma'); \Omega, x{:}\tau_1 \vdash L : \tau_2$ by i.h.
$(\Lambda; \Sigma'); \Omega \vdash \lambda x.L : \tau_1 \to \tau_2$ by rule

Case $\mathcal{D} = (\Lambda, \Sigma); \Omega \vdash u[\pi] \sqcup u[\pi] : \alpha \Longrightarrow u[\pi]/(\cdot, \cdot, \cdot)$

$u{::}(\Phi{\vdash}\alpha) \in \Lambda$ by premise
$(\Lambda, \Sigma); \Omega \vdash u[\pi] : \alpha$ by assumption
$u[\pi] = u[\pi]$ by reflexivity
$(\Lambda, \Sigma) \vdash \cdot : \cdot$ by rule
$\Lambda; \Omega \vdash u[\pi] : \alpha$ by strengthening

Case $\mathcal{D} = (\Lambda, \Sigma); \Omega \vdash u[\pi] \sqcup L : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega{\vdash}\alpha,\ u[\pi]/i,\ L/i)$

$u{::}(\Phi{\vdash}\alpha) \in \Lambda$ by premise
$(\Lambda, \Sigma); \Omega \vdash u[\pi] : \alpha$ by assumption
$(\Lambda, \Sigma); \Omega \vdash L : \alpha$ by assumption
$u[\pi] = [\![\mathsf{id}_\Lambda, u[\pi]/i]\!]i[\mathsf{id}_\Omega]$
$u[\pi] = u[\pi]$ by reflexivity
$L = [\![\mathsf{id}_\Lambda, L/i]\!]i[\mathsf{id}_\Omega]$
$L = L$ by reflexivity
$(\Lambda, \Sigma) \vdash L/i : i{::}\Omega{\vdash}\alpha$ by rule using assumption
$(\Lambda, \Sigma) \vdash u[\pi]/i : i{::}\Omega{\vdash}\alpha$ by rule using assumption
$(\Lambda, i{::}\Omega{\vdash}\alpha); \Omega \vdash \mathsf{id}_\Omega : \Omega$ by definition
$(\Lambda, i{::}\Omega{\vdash}\alpha); \Omega \vdash i[\mathsf{id}_\Omega] : \alpha$ by rule

Case $\mathcal{D} = (\Lambda, \Sigma); \Omega \vdash H \cdot S_1 \sqcup H \cdot S_2 : \alpha \Longrightarrow H \cdot S/(\Sigma',\ \theta_1,\ \theta_2)$

$(\Lambda, \Sigma); \Omega \vdash S_1 \underline{\sqcup} S_2 : \tau > \alpha \Longrightarrow S/(\Sigma', \theta_1, \theta_2)$ by premise
$(\Lambda, \Sigma); \Omega \vdash H \cdot S_1 : \alpha$ by assumption
$(\Lambda, \Sigma); \Omega \vdash S_1 : \tau > \alpha$ by inversion
$(\Lambda, \Sigma); \Omega \vdash H \cdot S_2 : \alpha$ by assumption
$(\Lambda, \Sigma); \Omega \vdash S_2 : \tau > \alpha$ by inversion
$S_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]S$ by i.h.

$S_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!]S$ by i.h.
$(\Lambda, \Sigma) \vdash \theta_1 : \Sigma_1$ by i.h.
$(\Lambda, \Sigma) \vdash \theta_2 : \Sigma_1$ by i.h.
$H \cdot S_1 = H \cdot\ [\![\mathsf{id}_\Lambda, \theta_1]\!]S$ by rule
$H \cdot S_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!](H \cdot S)$ by modal substitution definition
$H \cdot S_2 = H \cdot\ [\![\mathsf{id}_\Lambda, \theta_2]\!]S$ by rule
$H \cdot S_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!](H \cdot\ S)$ by modal substitution definition
$(\Lambda, \Sigma'); \Omega \vdash S : \tau > \alpha$ by i.h.
$(\Lambda, \Sigma'); \Omega \vdash H \cdot S : \alpha$ by rule

Case $\mathcal{D} = (\Lambda, \Sigma); \Omega \vdash H_1 \cdot S_1 \sqcup H_2 \cdot S_2 : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega \vdash \alpha,\ H_1 \cdot S_1/i,\ H_2 \cdot S_2/i)$

$(\Lambda, \Sigma); \Omega \vdash H_1 \cdot S_1 : \alpha$ by assumption
$(\Lambda, \Sigma); \Omega \vdash H_2 \cdot S_2 : \alpha$ by assumption
$H_1 \cdot S_1 = [\![(H_1 \cdot S_1)/i]\!](i[\mathsf{id}_\Omega])$ by modal substitution definition
$H_1 \cdot S_1 = H_1 \cdot S_1$ by reflexivity
$H_2 \cdot S_2 = [\![H_2 \cdot S_2/i]\!](i[\mathsf{id}_\Omega])$ by modal substitution definition
$H_2 \cdot S_2 = H_2 \cdot S_2$ by reflexivity
$(\Lambda, i{::}(\Omega{\vdash}\alpha)); \Omega \vdash \mathsf{id}_\Omega : \Omega$ by definition
$(\Lambda, i{::}(\Omega{\vdash}\alpha)); \Omega \vdash i[\mathsf{id}_\Omega] : \alpha$ by rule

Case $\mathcal{D} = (\Lambda, \Sigma); \Omega \vdash (L_1; S_1) \underline{\sqcup} (L_2; S_2) : (\tau_1 \to \tau_2) > \alpha$
$\Longrightarrow (L; S)/((\Sigma_1, \Sigma_2),\ (\theta_1, \theta_1'),\ (\theta_2, \theta_2'))$

$(\Lambda, \Sigma); \Omega \vdash L_1 \sqcup L_2 : \tau_1 \Longrightarrow L/(\Sigma_1,\ \theta_1,\ \theta_2)$ by premise
$(\Lambda, \Sigma); \Omega \vdash S_1 \underline{\sqcup} S_2 : \tau_2 > \alpha \Longrightarrow S/(\Sigma_2,\ \theta_1',\ \theta_2')$
$(\Lambda, \Sigma); \Omega \vdash (L_1; S_1) : \tau_1 \to \tau_2 > \alpha$ by assumption
$(\Lambda, \Sigma); \Omega \vdash L_1 : \tau_1$ by inversion
$(\Lambda, \Sigma); \Omega \vdash S_1 : \tau_2 > \alpha$
$(\Lambda, \Sigma); \Omega \vdash (L_2; S_2) : \tau_1 \to \tau_2 > \alpha$ by assumption
$(\Lambda, \Sigma); \Omega \vdash L_2 : \tau_1$ by inversion
$(\Lambda, \Sigma); \Omega \vdash S_2 : \tau_2 > \alpha$
$L_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]L$ by i.h.

$L_2 = \llbracket \mathsf{id}_\Lambda, \theta_2 \rrbracket L$ by i.h.
$S_1 = \llbracket \mathsf{id}_\Lambda, \theta_1' \rrbracket S$ by i.h.
$S_2 = \llbracket \mathsf{id}_\Lambda, \theta_2' \rrbracket S$ by i.h.
$L_1 = \llbracket \mathsf{id}_\Lambda, \theta_1, \theta_1' \rrbracket L$ by lemma 73
$L_2 = \llbracket \mathsf{id}_\Lambda, \theta_2, \theta_2' \rrbracket L$ by lemma 73
$S_1 = \llbracket \mathsf{id}_\Lambda, \theta_1, \theta_1' \rrbracket S$ by lemma 73
$S_2 = \llbracket \mathsf{id}_\Lambda, \theta_2, \theta_2' \rrbracket S$ by lemma 73
$(L_1; S_1) = (\llbracket \mathsf{id}_\Lambda, \theta_1, \theta_1' \rrbracket L; \llbracket \mathsf{id}_\Lambda, \theta_1, \theta_1' \rrbracket S)$ by rule
$(L_1; S_1) = \llbracket \mathsf{id}_\Lambda, \theta_1, \theta_1' \rrbracket (L; S)$ by modal substitution definition
$(L_2; S_2) = (\llbracket \mathsf{id}_\Lambda, \theta_2, \theta_2' \rrbracket L; \llbracket \mathsf{id}_\Lambda, \theta_2, \theta_2' \rrbracket S)$ by rule
$(L_2; S_2) = \llbracket \mathsf{id}_\Lambda, \theta_2, \theta_2' \rrbracket (L; S)$ by modal substitution definition
$(\Lambda, \Sigma) \vdash (\theta_1, \theta_1') : (\Sigma_1, \Sigma_2)$ θ_1 and θ_1' refer to distinct modal variables (lemma 73)
$(\Lambda, \Sigma) \vdash (\theta_2, \theta_2') : (\Sigma_1, \Sigma_2)$ θ_2 and θ_2' refer to distinct modal variables (lemma 73)
$(\Lambda, \Sigma_1); \Omega \vdash L : \tau_1$ by i.h.
$(\Lambda, \Sigma_2); \Omega \vdash S : \tau_2 > \alpha$ by i.h.
$(\Lambda, \Sigma_1, \Sigma_2); \Omega \vdash L : \tau_1$ by weakening
$(\Lambda, \Sigma_1, \Sigma_2); \Omega \vdash S : \tau_2 > \alpha$ by weakening
$(\Lambda, \Sigma_1, \Sigma_2); \Omega \vdash (L \ ; \ S) : \alpha$ by rule

$\square$

Theorem 75 (Completeness of mslg of terms)

1. *If $\Lambda, \Sigma \vdash \theta_1 : \Sigma'$ and $\Lambda, \Sigma \vdash \theta_2 : \Sigma'$ and θ_1 and θ_2 are incompatible and $L_1 = \llbracket \mathsf{id}_\Lambda, \theta_1 \rrbracket L$ and $L_2 = \llbracket \mathsf{id}_\Lambda, \theta_2 \rrbracket L$ then there exists a modal substitution θ_1^*, θ_2^*, and a modal context Σ^*, such that $(\Lambda, \Sigma); \Omega \vdash L_1 \sqcup L_2 : A \Longrightarrow L/(\Sigma^*, \theta_1^*, \theta_2^*)$ and $\theta_1^* \subseteq \theta_1$, $\theta_2^* \subseteq \theta_2$ and $\Sigma^* \subseteq \Sigma'$*

2. *If $\Lambda, \Sigma \vdash \theta_1 : \Sigma'$ and $\Lambda, \Sigma \vdash \theta_2 : \Sigma'$ and θ_1 and θ_2 are incompatible and $S_1 = \llbracket \mathsf{id}_\Lambda, \theta_1 \rrbracket S$ and $S_2 = \llbracket \mathsf{id}_\Lambda, \theta_2 \rrbracket S$ then there exists a modal substitution θ_1^*, θ_2^*, and a modal context Σ^*, such that $(\Lambda, \Sigma); \Omega \vdash S_1 \underline{\sqcup} S_2 : A \Longrightarrow S/(\Sigma^*, \theta_1^*, \theta_2^*)$ and $\theta_1^* \subseteq \theta_1$, $\theta_2^* \subseteq \theta_2$ and $\Sigma^* \subseteq \Sigma'$.*

Proof: Simultanous induction on the structure of L and S.

Case $L = u[\pi]$ and $u{::}\Phi{\vdash}\alpha \in \Lambda$

$L_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!](u[\pi])$ by assumption

$L_1 = u[\pi]$ by modal substitution definition

$u[\pi] = u[\pi]$ by inversion

and $L_1 = u[\pi]$

$(\Lambda, \Sigma); \Omega \vdash u[\pi] \sqcup u[\pi] : \alpha \Longrightarrow u[\pi]/(\cdot, \cdot, \cdot)$ by rule

$\cdot \subseteq \Sigma'$, $\cdot \subseteq \theta_1$, $\cdot \subseteq \theta_2$

Case $L = \lambda x.L'$.

$L_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!](\lambda x.L')$ by assumption

$L_1 = \lambda x.[\![\mathsf{id}_\Lambda, \theta_1]\!]L'$ by modal substitution definition

$L_1' = [\![\mathsf{id}_\Lambda, \theta_1]\!]L'$ by inversion

and $L_1 = \lambda x.L_1'$

$L_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!](\lambda x.L')$ by assumption

$L_2 = \lambda x.[\![\mathsf{id}_\Lambda, \theta_2]\!]L'$ by modal substitution definition

$L_2' = [\![\mathsf{id}_\Lambda, \theta_2]\!]L'$ by inversion

and $L_2 = \lambda x.L_2'$

$(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L_1' \sqcup L_2' : \tau_2 \Longrightarrow L'/(\Sigma^*, \theta_1^*, \theta_2^*)$ by i.h.

$\Sigma^* \subseteq \Sigma'$, $\theta_1^* \subseteq \theta_1$, $\theta_2^* \subseteq \theta_2$

$(\Lambda, \Sigma); \Omega \vdash \lambda x.L_1' \sqcup \lambda x.L_2' : \tau_1 \to \tau_2 \Longrightarrow \lambda x.L'/(\Sigma^*, \theta_1^*, \theta_2^*)$ by rule

Case $L = i[\mathsf{id}_\Omega]$

$L_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!](i[\mathsf{id}_\Omega])$ by assumption

$L_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!](i[\mathsf{id}_\Omega])$ by assumption

$L'/i \in \theta_1$ and $L''/i \in \theta_2$ by assumption

L' and L'' are incompatible by assumption

$L_1 = L'$ by modal substitution definition

$L_2 = L''$ by modal substitution definition

Case: $L_1 = u[\pi]$ and $L_2 = L''$

$(\Lambda, \Sigma); \Omega \vdash u[\pi] \sqcup L'' : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega \vdash \alpha, u[\pi]/i, L''/i)$ by rule

$i{::}(\Omega \vdash \alpha) \subseteq \Sigma'$, $(u[\pi]/i) \subseteq \theta_1$, $(L''/i) \subseteq \theta_2$

Case: $L_1 = L'$ and $L_2 = u[\pi]$

$(\Lambda, \Sigma); \Omega \vdash L' \sqcup u[\pi] : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega \vdash \alpha, L'/i, u[\pi]/i)$ by rule

$(i{::}(\Omega \vdash \alpha)) \subseteq \Sigma'$, $(u[\pi]/i) \subseteq \theta_2$, $(L'/i) \subseteq \theta_1$

Case: $L_1 = H \cdot S_1$ and $L_2 = H_2 \cdot S_2$

$H_1 \cdot S_1$ is incompatible with $H_2 \cdot S_2$ and $H_1 \neq H_2$ by assumption

$(\Lambda, \Sigma); \Omega \vdash H_1 \cdot S_1 \sqcup H_2 \cdot S_2 : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega \vdash \alpha, H_1 \cdot S_1/i, H_2 \cdot S_2/i)$ by rule

$(i{::}(\Omega \vdash \alpha)) \subseteq \Sigma'$, $(H_1 \cdot S_1/i) \subseteq \theta_1$, $(H_2 \cdot S_2/i) \subseteq \theta_2$

□

Theorem 76 (Soundness for mslg of substitutions)
If $(\Lambda, \Sigma_1) \vdash \rho_1 \sqcup \rho_2 : \Sigma_2 \Longrightarrow \rho/(\Sigma, \theta_1, \theta_2)$ *and*
$(\Lambda, \Sigma_1) \vdash \rho_1 : \Sigma_2$ *and* $(\Lambda, \Sigma_1) \vdash \rho_2 : \Sigma_2$ *then* $[\![\theta_1]\!]\rho = \rho_1$ *and* $[\![\theta_2]\!]\rho = \rho_2$

Proof: Induction on the first derivation.

Case $\mathcal{D} = \dfrac{}{(\Lambda, \Sigma_1) \vdash \cdot : \cdot \Longrightarrow \cdot/(\cdot, \cdot, \cdot)}$

$\cdot = \cdot$ by syntactic equality

$\cdot = [\![\,\cdot\,]\!](\cdot)$ modal substitution definition

Case $\mathcal{D} = (\Lambda, \Sigma_1) \vdash (\rho_1, L_1/i) \sqcup (\rho_2, L_2/i) : (\Sigma_2, i{::}(\Phi \vdash \alpha))$
$\Longrightarrow (\rho, L/i)/((\Sigma, \Sigma'), (\theta_1, \theta_1'), (\theta_2, \theta_2'))$

$(\Lambda, \Sigma_1) \vdash \rho_1 \sqcup \rho_2 : \Sigma_2 \Longrightarrow \rho/(\Sigma, \theta_1, \theta_2)$ by premise

$(\Lambda, \Sigma_1); \Phi \vdash L_1 \sqcup L_2 : \alpha \Longrightarrow L/(\Sigma', \theta_1', \theta_2')$ by premise

$(\Lambda, \Sigma_1) \vdash (\rho_1, L_1/i) : (\Sigma_2, i{::}(\Phi \vdash \alpha))$ by assumption

$(\Lambda, \Sigma_1) \vdash \rho_1 : \Sigma_2$ by inversion

$(\Lambda, \Sigma_1); \Phi \vdash L_1 : \alpha$

$(\Lambda, \Sigma_1) \vdash (\rho_2, L_2/i) : (\Sigma_2, i{::}(\Phi \vdash \alpha))$ by assumption

$(\Lambda, \Sigma_1) \vdash \rho_2 : \Sigma_2$ by inversion

$(\Lambda, \Sigma_1); \Phi \vdash L_2 : \alpha$

$L_1 = [\![\mathsf{id}_\Lambda, \theta_1']\!]L$ by lemma 74

$L_2 = [\![\mathsf{id}_\Lambda, \theta_2']\!]L$ by lemma 74

$L_1 = [\![\mathsf{id}_\Lambda, \theta_1, \theta_1']\!]L$ by lemma 73

$L_2 = [\![\mathsf{id}_\Lambda, \theta_2, \theta_2']\!]L$ by lemma 73

$\rho_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]\rho$ by i.h.

$\rho_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!]\rho$ by i.h.

$\rho_1 = [\![\mathsf{id}_\Lambda, \theta_1, \theta_1']\!]\rho$ by lemma 73
$\rho_2 = [\![\mathsf{id}_\Lambda, \theta_2, \theta_2']\!]\rho$ by lemma 73
$(\rho_1, L_1/i) = ([\![\mathsf{id}_\Lambda, \theta_1, \theta_1']\!]\rho,\ [\![\mathsf{id}_\Lambda, \theta_1, \theta_1']\!]L/i)$ by rule
$(\rho_2, L_2/i) = ([\![\mathsf{id}_\Lambda, \theta_2, \theta_2']\!]\rho,\ [\![\mathsf{id}_\Lambda, \theta_2, \theta_2']\!]L/i)$ by rule
$(\rho_1, L_1/i) = [\![\mathsf{id}_\Lambda, \theta_1, \theta_1']\!](\rho, L/i)$ by modal substitution definition
$(\rho_2, L_2/i) = [\![\mathsf{id}_\Lambda, \theta_2, \theta_2']\!](\rho, L/i)$ by modal substitution definition
$(\Lambda, \Sigma') \vdash \rho : \Sigma_2$ by i.h.
$(\Lambda, \Sigma''); \Omega \vdash L : \alpha$ by i.h.
$(\Lambda, \Sigma', \Sigma'') \vdash \rho : \Sigma_2$ by weakening
$(\Lambda, \Sigma', \Sigma''); \Omega \vdash L : \alpha$ by weakening
$(\Lambda, \Sigma', \Sigma'') \vdash (\rho, L/i) : (\Sigma_2, i{::}\Omega{\vdash}\alpha)$ by rule

□

Theorem 77 (Completeness for mslg of modal substitutions)
If $(\Lambda, \Sigma) \vdash \theta_1 : \Sigma'$ and $(\Lambda, \Sigma) \vdash \theta_2 : \Sigma'$ and θ_1 and θ_2 are incompatible and $\rho_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]\rho$ and $\rho_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!]\rho$ then $(\Lambda, \Sigma) \vdash \rho_1 \sqcup \rho_2 : \Sigma_1 \Longrightarrow \rho/(\Sigma^*, \theta_1^*, \theta_2^*)$ such that $\Sigma^* \subseteq \Sigma'$, $\theta_1^* \subseteq \theta_1$, $\theta_2^* \subseteq \theta_2$.

Proof: Induction on the structure of ρ.

Case $\rho = \cdot$

$\rho_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!](\cdot)$ by assumption
$\rho_1 = \cdot$ and $\Sigma_1 = \cdot$ by inversion
$\rho_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!](\cdot)$ by assumption
$\rho_2 = \cdot$ and $\Sigma_1 = \cdot$ by inversion
$(\Lambda, \Sigma) \vdash \cdot \sqcup \cdot : \cdot \Longrightarrow \cdot/(\cdot, \cdot, \cdot)$ by rule
$\cdot \subseteq \Sigma_1,\ \cdot \subseteq \theta_1,\ \cdot \subseteq \theta_2$

Case $\rho = (\rho', L/i)$

$\rho_1' = [\![\mathsf{id}_\Lambda, \theta_1]\!](\rho', L/i)$
$\rho_1' = ([\![\mathsf{id}_\Lambda, \theta_1]\!](\rho'),\ [\![\mathsf{id}_\Lambda, \theta_1]\!]L/i)$ by modal substitution definition
$\rho_1' = (\rho_1, L_1/i)$
$\rho_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]\rho'$

$L_1 = [\![\mathsf{id}_\Lambda, \theta_1]\!]L$
$\rho_2' = [\![\mathsf{id}_\Lambda, \theta_2]\!](\rho', L/i)$
$\rho_2' = ([\![\mathsf{id}_\Lambda, \theta_2]\!](\rho'),\ [\![\mathsf{id}_\Lambda, \theta_2]\!]L/i)$ by modal substitution definition
$\rho_2' = (\rho_2, L_2/i)$
$\rho_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!]\rho'$
$L_2 = [\![\mathsf{id}_\Lambda, \theta_2]\!]L$
$(\Lambda, \Sigma); \Phi \vdash L_1 \sqcup L_2 : \alpha \Longrightarrow L/(\Sigma^*, \theta_1^*, \theta_2^*)$ by completeness lemma 75
$\Sigma^* \subseteq \Sigma',\ \theta_1^* \subseteq \theta_1,\ \theta_2^* \subseteq \theta_2$
$(\Lambda, \Sigma) \vdash \rho_1 \sqcup \rho_2 : \Sigma_1 \Longrightarrow \rho'/(\Sigma^{**}, \theta_1^{**}, \theta_2^{**})$ by i.h.
$\Sigma^{**} \subseteq \Sigma',\ \theta_1^{**} \subseteq \theta_1,\ \theta_2^{**} \subseteq \theta_2 (\Lambda, \Sigma) \vdash (\rho_1, L_1/i) \sqcup (\rho_2, L_2/i) : (\Sigma_1, i{::}\Phi{\vdash}\alpha)$
$\Longrightarrow (\rho', L/i)/((\Sigma^{**}, \Sigma^*),\ (\theta_1^{**}, \theta_1^*),\ (\theta_2^{**}, \theta_2^*))$ by rule
$(\Sigma^{**}, \Sigma^*) \subseteq \Sigma',\ (\theta_1^{**}, \theta_1^*) \subseteq \theta_1,\ (\theta_2^{**}, \theta_2^*) \subseteq \theta_2$

□

When inserting a substitution ρ_2 into a substitution tree, we need to traverse the index tree and compute at each node N with substitution ρ the mslg between ρ and ρ_2. Before we describe the traversal more formally, we give a more formal definition of substitution trees.

$$\begin{array}{lll} \text{Node N} & ::= & (\Sigma{\vdash}\rho \twoheadrightarrow C) \\ \text{Children C} & ::= & [N, C] \mid \mathsf{nil} \end{array}$$

A tree is a node N with a modal substitution ρ and a list of children C. In general, we will write $\Lambda \vdash N : \Sigma'$ where $N = (\Sigma{\vdash}\rho \twoheadrightarrow C)$ which means that $(\Lambda, \Sigma) \vdash \rho : \Sigma'$ and all the children N_i in C, $\Lambda \vdash N_i : \Sigma$.

To insert a new substitution ρ_2 in to the substitution tree N where $\Lambda \vdash N : \Sigma$, we proceed in two steps. First, we inspect all the children N_i of a parent node N, where $N_i = \Sigma_i{\vdash}\rho_i \twoheadrightarrow C_i$ and check if ρ_1 is compatible with ρ_2. This compatibility check has three possible results:

1. $(\Lambda, \Sigma_i) \vdash \rho_i \sqcup \rho_2 : \Sigma \Longrightarrow \mathsf{id}_\Sigma/(\Sigma, \rho_i, \rho_2)$:
   This means ρ_i and ρ_2 are not compatible

2. $(\Lambda, \Sigma_i) \vdash \rho_i \sqcup \rho_2 : \Sigma \Longrightarrow \rho_1/(\Sigma_i, \mathsf{id}_{\Sigma_i}, \theta_2)$
   This means ρ_2 is an instance of ρ_i and we will continue to insert θ_2 into the

children C_i. In this case $[\![\mathsf{id}_\Lambda, \theta_2]\!]\rho_2$ is structurally equivalent to ρ_i and we will call ρ_2 *fully compatible* with ρ_i.

3. $(\Lambda, \Sigma_i) \vdash \rho_i \sqcup \rho_2 : \Sigma \Longrightarrow \rho'/(\Sigma'', \theta_1, \theta_2)$
   ρ_i and ρ_2 are compatible, but we need to replace node N_i with a new node $\Sigma''{\vdash}\rho' \twoheadrightarrow C'$ where C' contains two children, the child node $\Sigma_i{\vdash}\theta_1 \twoheadrightarrow C_i$ and the child node $\Sigma_i{\vdash}\theta_2 \twoheadrightarrow \mathsf{nil}$. In this case we will call ρ_2 *partially compatible* with ρ_i.

This idea can be formalized by using the following judgment to filter out all children from C which are compatible with ρ_2. Moreover, we will distinguish between the fully compatible children, which we collect in V, and the partially compatible children, which we collect in S.

$$\Lambda \vdash C \sqcup \rho_2 : \Sigma \Longrightarrow (V, S)$$

$$\begin{array}{lrcl} \text{Fully compatible children} & V & ::= & \cdot \mid V, (N, \theta) \\ \text{Partially compatible children} & S & ::= & \cdot \mid S,\ (N, \Sigma{\vdash}\rho, \theta_1, \theta_2) \end{array}$$

δ is a modal substitution such that $\Lambda \vdash \delta : \Sigma$, and for all the children $C_i = (\Sigma_i{\vdash}\rho_i \twoheadrightarrow C')$ in C, we have $\Lambda, \Sigma_i \vdash \rho_i : \Sigma$. Then V and S describe all the children from C which are compatible with δ. We distinguish three cases.

$$\frac{}{\Lambda \vdash \mathsf{nil} \sqcup \delta : \Sigma \Longrightarrow (\mathsf{nil}, \mathsf{nil})}$$

$$\frac{\Lambda \vdash C \sqcup \delta : \Sigma \Longrightarrow (V, S) \quad \Lambda, \Sigma_1 \vdash \rho_1 \sqcup \delta : \Sigma \Longrightarrow \mathsf{id}_\Sigma/(\Sigma, \rho_1, \delta)}{\Lambda \vdash [(\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Sigma \Longrightarrow (V, S)}\ NC$$

$$\frac{\Lambda \vdash C \sqcup \delta : \Sigma \Longrightarrow (V, S) \quad \Lambda, \Sigma_1 \vdash \rho_1 \sqcup \delta : \Sigma \Longrightarrow \rho_1/(\Sigma_1, \mathsf{id}_{\Sigma_1}, \theta_2) \quad \rho_1 \neq \mathsf{id}_{\Sigma_1}}{\Lambda \vdash [(\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Sigma \Longrightarrow ((V,\ ((\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), \theta_2)),\ S)}\ FC$$

$$\frac{\Lambda \vdash C \sqcup \delta : \Sigma \Longrightarrow (V, S) \quad \Lambda, \Sigma_1 \vdash \rho_1 \sqcup \delta : \Sigma \Longrightarrow \rho/(\Sigma_2, \theta_1, \theta_2) \quad \rho \neq \rho_1 \neq \mathsf{id}_{\Sigma_2}}{\Lambda \vdash [(\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Sigma \Longrightarrow (V, (S\ ,\ ((\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), \Sigma_2{\vdash}\rho, \theta_1, \theta_2))}\ PC$$

The NC rule describes the case where the child C_i is not compatible with δ. Rule FC describes the case where δ is fully compatible with the child C_i and the rule PC describes the case where δ is partially compatible with C_i. Before we describe the traversal of the substitution tree, we prove some straightforward properties about these rules

Lemma 78
If $\Lambda \vdash C \sqcup \delta : \Sigma \Longrightarrow (V, S)$ and $\Lambda \vdash \delta : \Sigma$ and for any $(\Sigma_i {\vdash} \rho_i \twoheadrightarrow C') \in C$ with $\Lambda, \Sigma_i \vdash \rho_i : \Sigma$ then

1. *for any $(N_i, \theta_2) \in V$ where $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$, we have $[\![\theta_2]\!]\rho_i = \delta$.*
2. *for any $(N_i, \Sigma' \vdash \rho', \theta_1, \theta_2) \in S$ where $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$, we have $[\![\theta_2]\!]\rho' = \delta$ and $[\![\theta_1]\!]\rho' = \rho_i$.*

Proof: By structural induction on the first derivation and by using the previous soundness lemma for mslg of substitutions (lemma 76).

Case $\mathcal{D} = \dfrac{}{\Lambda \vdash \mathsf{nil} \sqcup \delta : \Sigma \Longrightarrow (\mathsf{nil}, \mathsf{nil})}$.

Trivially true.

Case $\mathcal{D} = \dfrac{\Lambda \vdash C \sqcup \delta : \Sigma \Longrightarrow (V, S) \qquad \Lambda, \Sigma_1 \vdash \rho_1 \sqcup \delta : \Sigma \Longrightarrow \mathsf{id}_\Sigma/(\Sigma, \rho_1, \delta)}{\Lambda \vdash [(\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Sigma \Longrightarrow (V, S)}\, NC$

By i.h., for any $(N_i, \theta_2) \in V$, $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$, we have $[\![\theta_2]\!]\rho_i = \delta$ and for any $(N_i, \Sigma' \vdash \rho', \theta_1', \theta_2') \in S$ where $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$, we have $[\![\theta_2']\!]\rho' = \delta$ and $[\![\theta_1']\!]\rho' = \rho_i$.

Case $\mathcal{D} = \dfrac{\Lambda \vdash C \sqcup \delta : \Sigma \Longrightarrow (V, S) \qquad \Lambda, \Sigma_1 \vdash \rho_1 \sqcup \delta : \Sigma \Longrightarrow \rho_1/(\Sigma_1, \mathsf{id}_{\Sigma_1}, \theta_2)}{\Lambda \vdash [(\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Sigma \Longrightarrow ((V\ ,\ (\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1)), S)}\, FC$

By i.h., for any $(N_i, \theta_2) \in V$, $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$, we have $[\![\theta_2]\!]\rho_i = \delta$ and for any $(N_i, (\Sigma' \vdash \rho', \theta_1', \theta_2')) \in S$ where $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$, we have $[\![\theta_2']\!]\rho' = \delta$ and $[\![\theta_1']\!]\rho' = \rho_i$. By soundness lemma 76, $[\![\theta_2]\!]\rho_1 = \delta$, therefore for any $(N_i, \theta') \in (V,\ ((\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), \theta_2))$, where $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$ we have $[\![\theta']\!]\rho_i = \delta$.

Case $\mathcal{D} = \dfrac{\Lambda \vdash C \sqcup \delta : \Sigma \Longrightarrow (V, S) \qquad \Lambda, \Sigma_1 \vdash \rho_1 \sqcup \delta : \Sigma \Longrightarrow \rho^*/(\Sigma_2, \theta_1, \theta_2)}{\Lambda \vdash [(\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), C] \sqcup \delta : \Sigma \Longrightarrow (V, (S\ ,\ ((\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), \Sigma_2{\vdash}\rho^*, \theta_1, \theta_2))}\, PC$

By i.h., for any $(N_i, \theta_2') \in V$, $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$, we have $[\![\theta_2']\!]\rho_i = \delta$ and for any $(N_i, (\Sigma' \vdash \rho', \theta_1', \theta_2')) \in S$ where $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$, we have $[\![\theta_2']\!]\rho' = \delta$ and $[\![\theta_1']\!]\rho' = \rho_i$. By soundness lemma 76, $[\![\theta_2]\!]\rho^* = \delta$ and $[\![\theta_1]\!]\rho^* = \rho_1$, therefore for any $(N_i, \Sigma' \vdash \rho', \theta_1', \theta_2') \in (S\ ,\ ((\Sigma_1{\vdash}\rho_1 \twoheadrightarrow C_1), \Sigma_2{\vdash}\rho^*, \theta_1, \theta_2))$, where $N_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C_i)$ we have $[\![\theta_1']\!]\rho' = \rho_i$ and $[\![\theta_2']\!]\rho' = \delta$.

□

Next, we show insertion of a substitution δ into a substitution tree N. The main judgment is the following:

$$\Lambda \vdash N \sqcup \delta : \Sigma \Longrightarrow N' \quad \text{Insert } \delta \text{ into the substitution tree } N$$

If N is a substitution tree and δ is not already in the tree, then N' will be the new substitution tree after inserting δ into N. We write $[N'/N_i]C$ to indicate that the i-th node N_i in the children C is replaced by the new node N'. Recall that the substitution δ which is inserted into the substitution tree N does only refer to modal variables in Λ and does not contain any internal modal variables. Therefore, a new leaf with substitution δ must have the following form: $\cdot\!\vdash\!\delta \twoheadrightarrow \mathsf{nil}$. Similarly, if we split the current node and create a new leaf $\cdot\!\vdash\!\theta_2 \twoheadrightarrow \mathsf{nil}$ (see rule "Split current node").

Create new leaf

$$\frac{\Lambda \vdash C \sqcup \delta : \Sigma' \Longrightarrow (\cdot, \cdot)}{\Lambda \vdash (\Sigma\!\vdash\!\rho \twoheadrightarrow C) \sqcup \delta : \Sigma' \Longrightarrow (\Sigma\!\vdash\!\rho \twoheadrightarrow (C, (\cdot\!\vdash\!\delta \twoheadrightarrow \mathsf{nil}))}$$

Recurse

$$\frac{\Lambda \vdash C \sqcup \delta : \Sigma' \Longrightarrow (V, S) \quad N_i \in C \quad (N_i, \theta_2) \in V \quad \Lambda \vdash N \sqcup \theta_2 \Longrightarrow N'}{\Lambda \vdash (\Sigma'\!\vdash\!\rho \twoheadrightarrow C) \sqcup \delta : \Sigma \Longrightarrow (\Sigma'\!\vdash\!\rho \twoheadrightarrow [N'/N_i]C}$$

Split current node

$$\frac{\Lambda \vdash C \sqcup \delta : \Sigma \Longrightarrow (\cdot, S) \quad N_i \in C \quad N_i = (\Sigma_i\!\vdash\!\rho_i \twoheadrightarrow C_i) \quad (N_i, \Sigma^*\!\vdash\!\rho, \theta_1, \theta_2) \in S}{\Lambda \vdash (\Sigma'\!\vdash\!\rho \twoheadrightarrow C) \sqcup \delta : \Sigma \Longrightarrow (\Sigma'\!\vdash\!\rho \twoheadrightarrow [(\Sigma^*\!\vdash\!\rho \twoheadrightarrow ((\Sigma_i\!\vdash\!\theta_1 \twoheadrightarrow C_i), (\cdot\!\vdash\!\theta_2 \twoheadrightarrow \mathsf{nil})))/N_i]C)}$$

The above rules always insert a substitution δ into the children C of a node $\Sigma\!\vdash\!\rho \twoheadrightarrow C$. We start inserting a substitution L/i_0 into the empty substitution tree which contains the identity substitution $i_0[\mathsf{id}]/i_0$ and has an empty list of children. After the first insertion, we obtain the substitution tree which contains the identity substitution $i_0[\mathsf{id}]/i_0$ and the child of this node contains the substitution L/i_0. In other words, we require that the top node of a substitution tree contains a redundant identity substitution which allows us to treat insertion of a substitution δ into a substiution tree uniformly. This leads us to the following soundness statement where we show that if we inserte a substiutition δ into the children C, then there exists a child $C_i = \Sigma_i\!\vdash\!\rho_i \twoheadrightarrow C_i'$ in C and a path from ρ_i to ρ_n, where ρ_n is at a leaf such that $[\![\rho_n]\!][\![\rho_{n-1}]\!] \ldots \rho_i = \delta$.

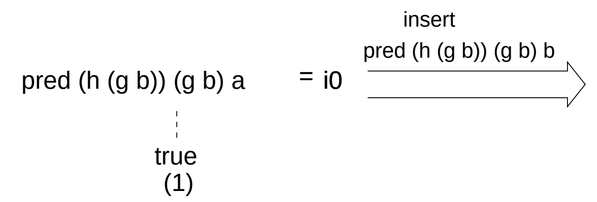
Theorem 79 (Soundness of insertion)
If $\Lambda \vdash (\Sigma' \vdash \rho' \twoheadrightarrow C) \sqcup \delta : \Sigma \Longrightarrow (\Sigma' \vdash \rho' \twoheadrightarrow C')$ then there exists a child $C_i = (\Sigma_i \vdash \rho_i \twoheadrightarrow C')$ and a path from ρ_i to ρ_n such that $[\![\rho_n]\!][\![\rho_{n-1}]\!] \dots \rho_i = \delta$.

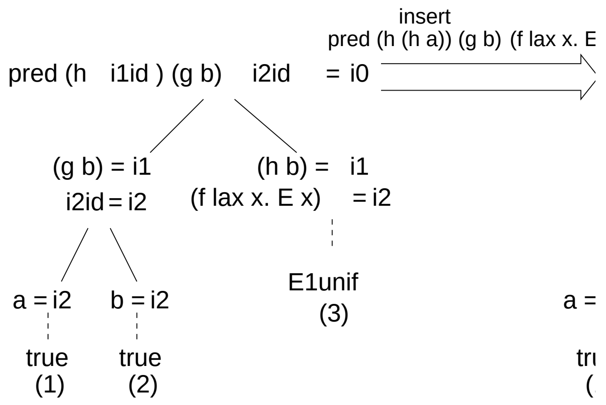
Proof: By induction on the first derivation using the previous lemma 78. □

We illustrate this process by an example. Assume we have the following two modal substitutions:

$$\rho_1 = [\![(g\ a)/i_1,\ f\ (\lambda x.u[\mathsf{id}])/i_2,\ (h\ c)/i_3]\!]$$
$$\delta = [\![(g\ b)/i_1,\ c/i_2,\ (h\ a)/i_3]\!]$$

Then, the most specific linear common generalization of the substitution is computed as follows. We compute the most specific linear generalization for each element. Hence the resulting most specific linear common substitution ρ of ρ_1 and δ is $[\![(g\ i_4[\mathsf{id}])/i_1,\ i_2[\mathsf{id}]/i_2,\ (h\ i_5[\mathsf{id}])/i_3]\!]$. In addition, ρ_1 will be split into the most specific linear common substitution ρ and the substitution $\theta_1 = [\![a/i_4,\ f\ (\lambda x.u[\mathsf{id}])/i_2,\ c/i_5]\!]$, s.t. $[\![\theta_1]\!]\rho = \rho_1$. Similarly δ will be split into ρ and the substitution $\theta_2 = [\![b/i_4,\ c/i_2,\ a/i_5]\!]$, s.t. $[\![\theta_2]\!]\rho = \delta$. Next, we illustrate the process of inserting the terms (1) to (4) from page 147 into a higher-order substitution tree.





In the second step, we can see why it is important to re-use the name of internal modal variables when computing the mslg of two terms. Here we need to compute the mslg of $\mathsf{pred\ (h\ (g\ b))\ (g\ b)}\ i_2[\mathsf{id}]$ and $\mathsf{pred\ (h\ (h\ b))\ (g\ b)\ (f}\ \lambda x.u[\mathsf{id}])$. The result is the substitution $\theta_1 = [\![(\mathsf{g\ b})/i_1, i_2[\mathsf{id}]/i_2]\!]$ and $\theta_2 = [\![(\mathsf{h\ b})/i_1, (\mathsf{f}\ \ \lambda x.u[\mathsf{id}])/i_2]\!]$. Since we chose to re-use the variable i_2 and created a substitution $i_2[\mathsf{id}]/i_2$, we can add a new node with the substitution θ_2 and children $a = i_2$ and $b = i_2$ into the tree without violating the stated invariant or renaming the children.

Finally, we note that we may need to apply weakening before inserting a substitution δ into the substitution tree N. Consider the first two insertions into the empty substitution tree in the previous example. Since the first two terms inserted, do not refer to any modal variables $\Lambda = \cdot$. In other words, if the top node in the substitution tree is well-defined in the empty modal context and $\Lambda = \cdot$. In the third step, we are inserting the object $\mathsf{pred\ (h\ (h\ b))\ (g\ b)\ (f}\ \lambda x.u[\mathsf{id}])$ which refers to the modal variable u. Hence before inserting the third term, we need to weaken the modal context Λ such that such that $\Lambda \vdash N : \Sigma$ and $\Lambda \vdash \delta : \Sigma$.

5.3 Retrieval

In general, we can retrieve all terms from the index which satisfy some property. This property may be finding all terms from the index which unify with a given term U or finding all terms V from the index, s.t. a given term U is an instance or variant of V. One advantage of substitution trees is that all these retrieval operations can be implemented with only small changes. A key challenge in the higher-order setting is to design efficient retrieval algorithms which will perform well in practice. In Chapter 3 we presented a unification algorithm for linear higher-order patterns. We specialize this algorithm to check whether the term L_2 is an instance of the term L_1. We treat again internal modal variables differently than global modal variables. In addition, both terms must be linear unlike the algorithm in Chapter 3 where only one of them has to be. The principal judgement are as follows:

$$\begin{array}{lll}
(\Lambda, \Sigma); \Omega \vdash L_1 & \doteq & L_2 : \tau/(\theta, \rho) \qquad L_2 \text{ is an instance of } L_1 \\
(\Lambda, \Sigma); \Omega \vdash S_1 & \doteq & S_2 : \tau > \alpha/(\theta, \rho) \quad S_2 \text{ is an instance of } S_2
\end{array}$$

Again we assume that L_1 and L_2 must be well-typed in the modal context Λ, Σ and the bound variable context Ω. We assume that only L_1 contains internal modal variables and is stored in the index, while L_2 is given, and that the modal variables occurring in L_1 are distinct from the modal variables occuring in L_2. This implies that $(\Lambda_1, \Sigma); \Omega \vdash L_1 : \tau$ and $\Lambda_2; \Omega \vdash L_2 : \tau$ and $\Lambda = (Lambda_1, \Lambda_2)$. ρ is the substitution for some internal modal variables in Σ while θ is the substitution for some modal variables in Λ_1. Both substitutions can always be extended with identity substitutions such that the extension of θ (e.g. ρ) covers all the variables in Λ (e.g. Σ). Moreover, $[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!] L_1$ is syntactically equal to L_2.

$$\frac{i{::}\Omega{\vdash}\alpha \in \Sigma}{(\Lambda, \Sigma); \Omega \vdash i[\mathsf{id}_\Omega] \doteq L : \alpha \;/\; (\cdot, (L/i))}\; existsL-1$$

$$\frac{u{::}\Phi{\vdash}\alpha \in \Lambda}{(\Lambda, \Sigma); \Omega \vdash u[\pi] \doteq L : \alpha \;/\; (([\pi]^{-1}\, L/u), \cdot)}\; existsL-2$$

$$\frac{(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L_1 \doteq L_2 : \tau_2 \;/\; (\theta, \rho)}{(\Lambda, \Sigma); \Omega \vdash \lambda x.L_1 \doteq \lambda x.L_2 : \tau_1 \to \tau_2 \;/\; (\theta, \rho)}\; lam$$

$$\frac{(\Lambda, \Sigma); \Omega \Vdash S_1 \doteq S_2 : \tau > \alpha \;/\; (\theta, \rho)}{(\Lambda, \Sigma); \Omega \vdash H \cdot S_1 \doteq H \cdot S_2 : a \;/\; (\theta, \rho)}$$

$$\frac{}{(\Lambda, \Sigma); \Omega \Vdash \mathsf{nil} \doteq \mathsf{nil} : \alpha > \alpha \;/\; (\cdot, \cdot)}$$

$$\frac{(\Lambda, \Sigma); \Omega \vdash L_1 \doteq L_2 : \tau_1 \;/\; (\theta_1, \rho_1) \quad (\Lambda, \Sigma); \Omega \Vdash S_1 \doteq S_2 : \tau_2 > \alpha \;/\; (\theta_2, \rho_2)}{(\Lambda, \Sigma); \Omega \Vdash (L_1; S_1) \doteq (L_2; S_2) : \tau_1 \to \tau_2 > \alpha \;/\; ((\theta_1, \theta_2), (\rho_1, \rho_2))}$$

Note that we need not worry about capture in the rule for lambda expressions since existential variables and bound variables are defined in different contexts. In the rule *app*, we are allowed to union the two substitutions θ_1 and θ_2, as the linearity requirement ensures that the domains of both substitutions are disjoint. Note that the case for matching an existential variable $u[\pi]$ with another term L is simpler and more efficient than in the general higher-order pattern case. In particular, it does not require a traversal of L (see rules *existsL-1* and *existsL-2*). Since the inverse of the substitution π can be computed directly and will be total, we know $[\pi]^{-1}\, L$ exists and can simply generate a substitution $[\pi]^{-1}\, L/u$. The algorithm can be easily specialized to retrieve variances by requiring in the *existsL-2* rule that L must be $u[\pi]$. To check unifiability we need to add the dual rule to *existsL-2* where we unify L with an existential variable $u[\pi]$. The only complication is that L may contain internal modal variables which are defined later on the path in the substitution tree.

The instance algorithm for terms can be straightforwardly extended to instances of substitutions. We define the following judgment for it:

$$\Lambda, \Sigma \vdash \rho_1 \;\doteq\; \rho_2 : \Sigma'/(\theta, \rho) \quad \rho_2 \text{ is an instance of } \rho_1$$

We assume that ρ_1 and ρ_2 are modal substitutions from a modal context Σ' to a modal context Λ, Σ. ρ is a modal substitution for the modal variables in Σ, while θ is the modal substitution for the modal variables in Λ such that $[\![\theta, \sigma]\!]\rho_1$ is syntactically equal to ρ_2.

$$\frac{}{(\Lambda, \Sigma) \vdash \cdot \ \doteq \ \cdot : \cdot/(\cdot, \cdot)}$$

$$\frac{(\Lambda, \Sigma_1) \vdash \rho_1 \ \doteq \ \rho_2 : \Sigma_2/(\theta, \rho) \quad (\Lambda, \Sigma_1); \Omega \vdash L_1 \ \doteq \ L_2 : \tau/(\theta', \rho')}{(\Lambda, \Sigma_1) \vdash (\rho_1, L_1/i) \ \doteq \ (\rho_2, L_2/i) : (\Sigma_2, i{::}(\Omega{\vdash}\tau))/((\theta, \theta'), (\rho, \rho')}$$

Now we show soundness and completeness of the retrieval algorithm. We first show soundness and completeness of the instance algorithm for terms.

Theorem 80 (Soundness)

1. *If* $(\Lambda, \Sigma); \Omega \vdash L_1 \ \doteq \ L_2 : \tau/(\theta, \rho)$ *for some* Λ_1 *and* Λ_2 *where* $\Lambda = (Lambda_1, \Lambda_2)$ *and* $(\Lambda_1, \Sigma); \Omega \vdash L_1 : \tau$ *and* $\Lambda_2; \Omega \vdash L_2 : \tau$ *then* $[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!]L_1 = L_2$.
2. *If* $(\Lambda, \Sigma); \Omega \vdash S_1 \ \doteq \ S_2 : \tau > \alpha/(\theta, \rho)$ *for some* Λ_1 *and* Λ_2 *where* $\Lambda = (Lambda_1, \Lambda_2)$ *and* $(\Lambda_1, \Sigma); \Omega \vdash S_1 : \tau > \alpha$ *and* $\Lambda_2; \Omega \vdash S_2 : \tau > \alpha$ *then* $[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!]S_1 = S_2$.

Proof: Simultanous structural induction on the first derivation. The proof is straighforward, and we give a few cases here.

Case $\mathcal{D} = \dfrac{i{::}\Omega{\vdash}\alpha}{(\Lambda, \Sigma); \Omega \vdash i[\mathsf{id}_\Omega] \doteq L : \alpha \ / \ (\cdot, (L/i))}\ existsL - 1$

$i{::}\Omega{\vdash}\alpha; \Omega \vdash i[\mathsf{id}_\Omega] : \alpha$ by assumption
$\Lambda_2; \Omega \vdash L : \alpha$ and $\Lambda = \Lambda_2$ by assumption
$L = L$ by reflexivity
$[\![\mathsf{id}_{\Lambda_2}, L/i]\!](i[\mathsf{id}_\Omega]) = L$ by substitution definition

Case $\mathcal{D} = \dfrac{u{::}\Phi{\vdash}\alpha \in \Lambda}{(\Lambda, \Sigma); \Omega \vdash u[\pi] \doteq L : \alpha \ / \ (([\pi]^{-1} L/u), \cdot)}\ existsL - 2$

$u{::}\Phi{\vdash}\alpha; \Omega \vdash u[\pi] : \alpha$ by assumption
$\Lambda_2; \Omega \vdash L : \alpha$ and $\Lambda = \Lambda_2, u{::}\Phi{\vdash}\alpha$ by assumption
$[\pi]([\pi]^{-1} L) = L$ by property of inversion
$[\![\mathsf{id}_{\Lambda_2}, [\pi]^{-1} L/u]\!](u[\pi]) = L$ by substitution definition

Case $\mathcal{D} = \dfrac{(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L_1 \doteq L_2 : \tau_2 \;/\; (\theta, \rho)}{(\Lambda, \Sigma); \Omega \vdash \lambda x.L_1 \doteq \lambda x.L_2 : \tau_1 \to \tau_2 \;/\; (\theta, \rho)}\, lam$

$(\Lambda_1, \Sigma); \Omega \vdash \lambda x.L_1 : \tau_1 \to \tau_2$ by assumption
$(\Lambda_1, \Sigma); \Omega, x{:}\tau_1 \vdash L_1 : \tau_2$ by inversion
$\Lambda_2; \Omega \vdash \lambda x.L_2 : \tau_1 \to \tau_2$ by assumption
$\Lambda_2, ; \Omega, x{:}\tau_1 \vdash L_2 : \tau_2$ by inversion
$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!] L_1 = L_2$ by i.h.

Case $\mathcal{D} = \dfrac{(\Lambda, \Sigma); \Omega \vdash L_1 \doteq L_2 : \tau_1 \;/\; (\theta_1, \rho_1) \quad (\Lambda, \Sigma); \Omega \Vdash S_1 \doteq S_2 : \tau_2 > \alpha \;/\; (\theta_2, \rho_2)}{(\Lambda, \Sigma); \Omega \Vdash (L_1; S_1) \doteq (L_2; S_2) : \tau_1 \to \tau_2 > \alpha \;/\; ((\theta_1, \theta_2),\, (\rho_1, \rho_2))}$

$(\Lambda_1; \Sigma); \Omega \vdash (L_1; S_1) : \tau > \alpha$ by assumption
$(\Lambda_1; \Sigma); \Omega \vdash L_1 : \tau_1$ by inversion
$(\Lambda_1; \Sigma); \Omega \vdash S_1 : \tau_1 \to \tau_2 > \alpha$
$\Lambda_2; \Omega \vdash L_2 : \tau_1$ by inversion
$\Lambda_2; \Omega \vdash S_2 : \tau_1 \to \tau_2 > \alpha$
$[\![\mathsf{id}_{\Lambda_2}, \theta_1, \rho_1]\!] L_1 = L_2$ by i.h.
$[\![\mathsf{id}_{\Lambda_2}, \theta_2, \rho_2]\!] S_1 = S_2$ by i.h.
$[\![\mathsf{id}_{\Lambda_2}, \theta_1, \theta_2, \rho_1, \rho_2]\!] L_1 = L_2$ by weakening
$[\![\mathsf{id}_{\Lambda_2}, \theta_1, \theta_2, \rho_1, \rho_2]\!] S_1 = S_2$ by weakening
$[\![\mathsf{id}_{\Lambda_2}, \theta_1, \theta_2, \rho_1, \rho_2]\!] (L_1\ S_1) = [\![\mathsf{id}_{\Lambda_2} \theta_1, \theta_2, \rho_1, \rho_2]\!] (L_2\ S_2)$ by rule
and substitution definition

□

Next, we show soundness of retrieval for substitions.

Theorem 81 (Soundness of retrieval for substitutions)
If $(\Lambda, \Sigma) \vdash \rho_1 \ \doteq\ \rho_2 : \Sigma'/(\theta, \rho)$ *and* $(\Lambda_1, \Sigma) \vdash \rho_1 : \Sigma'$ *and* $\Lambda_2 \vdash \rho_2 : \Sigma'$ *and* $(\Lambda_1, \Lambda_2) = \Lambda$ *and all the variables in* Σ, Λ_1 *and* Λ_2 *are distinct then* $[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!] \rho_1 = \rho_2$.

Proof: Structural induction on the first derivation and using previous lemma 80. □

For completeness we show that if the term L_2 is an instance of a linear term L then the given algorithm will succeed and return substitution θ^* for the modal variables and a substitution ρ^* for the internal modal variables ocurring in L. This establishes a form of local completeness of the given retrieval algorithm. We will show later a global completeness theorem, which states that any time we compute the msgl of a term L_1 and L_2 to be L, then we can show that L_2 is in fact an instance of L. More generally, we show that any time we insert a substiution L_2/i_0 we can also retrieve it.

Theorem 82 (Completeness of instance algorithm for terms)

1. *If* $(\Lambda_1, \Sigma); \Omega \vdash L : \tau$ *and* $\Lambda_2; \Omega \vdash L_2 : \tau$ *and* $\Lambda = (\Lambda_1, \Lambda_2)$ *and* $\Lambda \vdash \theta : \Lambda_1$ *and* $\Lambda \vdash \rho : \Sigma$ *and* $[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!]L = L_1$ *then* $(\Lambda, \Sigma); \Omega \vdash L \doteq L_2 : \tau/(\theta^*, \rho^*)$ *where* $\theta^* \subseteq \theta$ *and* $\rho^* \subseteq \rho$.

2. *If* $(\Lambda_1, \Sigma); \Omega \vdash S : \tau > \alpha$ *and* $\Lambda_2; \Omega \vdash S_2 : \tau > \alpha$ *and* $\Lambda = (\Lambda_1, \Lambda_2)$ *and* $\Lambda \vdash \theta : \Lambda_1$ *and* $\Lambda \vdash \rho : \Sigma$ *and* $[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!]S = S_2$ *then* $(\Lambda, \Sigma); \Omega \vdash S \doteq S_2 : \tau > \alpha/(\theta^*, \rho^*)$ *where* $\theta^* \subseteq \theta$ *and* $\rho^* \subseteq \rho$.

Proof: Simultanous structural induction on the first typing derivation.

Case $\mathcal{D} = \dfrac{(\Lambda_1, \Sigma); \Omega, x{:}\tau_1 \vdash L : \tau_2}{(\Lambda_1, \Sigma); \Omega \vdash \lambda x.L : \tau_1 \to \tau_2}$

$\Lambda_2; \Omega \vdash \lambda x.L_2 : \tau_1 \to \tau_2$	by assumption
$\Lambda_2; \Omega, x{:}\tau_1 \vdash L_2 : \tau_2$	by inversion
$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](\lambda x.L) = \lambda x.L_2$	by assumption
$\lambda x.[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](L) = \lambda x.L_2$	by substitution definition
$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](L) = L_2$	by syntactic equality
$(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L \doteq L_2 : \tau_2/(\theta^*, \rho^*)$	by i.h.
$\theta^* \subseteq \theta$ and $\rho^* \subseteq \rho$	
$(\Lambda, \Sigma); \Omega \vdash \lambda x.L \doteq \lambda x.L_2 : \tau_1 \to \tau_2/(\theta^*, \rho^*)$	by rule

Case $\mathcal{D} = \dfrac{i{::}\Omega{\vdash}\alpha \in \Sigma}{(\Lambda_1, \Sigma); \Omega \vdash i[\mathsf{id}_\Omega] : \alpha}$

$i{::}\Omega{\vdash}\alpha; \Omega \vdash i[\mathsf{id}_\Omega] : \alpha$	by rule
$\Sigma = \Sigma_1, i{::}\Omega{\vdash}\alpha, \Sigma_2$	
$\Lambda_2; \Omega \vdash L_2 : \alpha$	by assumption

$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](i[\mathsf{id}_\Omega]) = L_2$ by assumption
$L'/i \in \rho$ by assumption
$L' = L_2$ and $L_2/i \in \rho$ by substitution definition
$(\Lambda, i{::}\Omega{\vdash}\alpha); \Omega \vdash i[\mathsf{id}_\Omega] \doteq L_2 : \alpha/(\cdot, L_2/i)$ by rule
$\cdot \subseteq \mathsf{id}_\Lambda$ and $(L_2/i) \subset \rho$

Case $\mathcal{D} = \dfrac{u{::}\Phi{\vdash}\alpha \in \Lambda_1}{(\Lambda_1, \Sigma); \Omega \vdash u[\pi] : \alpha}$

$u{::}\Phi{\vdash}\alpha; \Omega \vdash u[\pi] : \alpha$ by rule
$\Lambda_1 = \Lambda_1', u{::}\Phi{\vdash}\alpha, \Lambda_1''$
$\Lambda_2; \Omega \vdash L_2 : \alpha$ by assumption
$\theta = (\theta_1, L/u, \theta_2)$ by assumption
$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](u[\pi]) = L_2$ by assumption
$[\pi]L = L_2$ by substitution definition
$L = [\pi]^{-1}\, L_2$ and $[\pi]([\pi]^{-1}\, L_2) = L_2$ by inverse substitution property
$\Lambda_2, u{::}\Phi{\vdash}\alpha; \Omega \vdash u[\pi] \doteq L_2 : \alpha/([\pi]^{-1}\, L_2/u, \cdot)$ by rule
$([\pi]^{-1}\, L_2/u) \subseteq \theta$ and $\cdot \subseteq \rho$

Case $\mathcal{D} = \dfrac{(\Lambda_1, \Sigma); \Omega \vdash L_1 : \tau_1 \qquad (\Lambda_1, \Sigma); \Omega \Vdash S_1 : \tau_1 \to \tau > \alpha}{(\Lambda_1, \Sigma); \Omega \Vdash (L_1; S_1) : \tau > \alpha}$

$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](L_1; S_1) = S'$ by assumption
$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](L_1)\ ;\ [\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](S_1) = S'$ by substitution definition
$S' = (L_2; S_2)$ by inversion
$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](L_1) = L_2$ by inversion
$[\![\mathsf{id}_{\Lambda_2}, \theta, \rho]\!](S_1) = S_2$ by inversion
$\Lambda_2; \Omega \vdash (L_2; S_2) : \tau > \alpha$ by assumption
$\Lambda_2; \Omega \vdash L_2 : \tau_1$ by inversion
$\Lambda_2; \Omega \vdash S_2 : \tau_1 \to \tau > \alpha$
$(\Lambda, \Sigma); \Omega \vdash L_1 \ \doteq\ L_2 : \tau_1/(\theta_1^*, \rho_1^*)$ and $\theta_1^* \subseteq \theta$ and $\rho_1^* \subseteq \rho$ by i.h.
$(\Lambda, \Sigma); \Omega \vdash S_1 \ \doteq\ S_2 : \tau_1 \to \tau > \alpha/(\theta_2^*, \rho_2^*)$ and $\theta_2^* \subseteq \theta$ and $\rho_2^* \subseteq \rho$ by i.h.

$(\Lambda, \Sigma); \Omega \vdash L_1 \doteq L_2 : \tau_1/(\theta_1^*, \rho_1^*)$ by weakening
$(\Lambda, \Sigma); \Omega \vdash S_1 \doteq S_2 : \tau_1 \rightarrow \tau > \alpha/(\theta_2^*, \rho_2^*)$ by weakening
$(\Lambda, \Sigma); \Omega \vdash (L_1; S_1) \doteq (L_2; S_2) : \tau > \alpha/((\theta_1^*, \theta_2^*), (\rho_1^*, \rho_2^*))$ by rule
$(\theta_1^*, \theta_2^*) \subseteq \theta$ and $(\rho_1^*, \rho_2^*) \subseteq \rho$ by subset property

□

Next, we show the global completeness of the mslg and instance algorithm. We show that if the mslg of object L_1 and L_2 returns the modal substitutions θ_1 and θ_2 together with the mslg L, then in fact the retrieval algorithm shows that L_1 is an instance of L under θ_1 and L_2 is an instance of L under θ_2. This guarantees that any time we insert a term L_2 we can in fact retrieve it.

Theorem 83 (Interaction between mslg and instance algorithm)

1. *If for some Λ_1 and Λ_2 where $(\Lambda_1, \Sigma); \Omega \vdash L_1 : \tau$ and $\Lambda_2; \Omega \vdash L_2 : \tau$ and $\Lambda = (\Lambda_1, \Lambda_2)$ and $(\Lambda, \Sigma); \Omega \vdash L_1 \sqcup L_2 : \tau \Longrightarrow L/(\Sigma', \theta_1, \theta_2)$ then*
   $((\Lambda, \Sigma), \Sigma'); \Omega \vdash L \doteq L_1 : \tau/(\theta', \theta_1)$ and
   $((\Lambda, \Sigma), \Sigma'); \Omega \vdash L \doteq L_2 : \tau/(\theta'', \theta_2)$ and $\theta' \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ and $\theta'' \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$.

2. *If for some Λ_1 and Λ_2 where $(\Lambda_1, \Sigma); \Omega \vdash S_1 > \alpha : \tau$ and $\Lambda_2; \Omega \vdash S_2 : \tau > \alpha$ and $\Lambda = (\Lambda_1, \Lambda_2)$ and $(\Lambda, \Sigma); \Omega \vdash S_1 \sqcup S_2 : \tau > \alpha \Longrightarrow S/(\Sigma', \theta_1, \theta_2)$ then*
   $((\Lambda, \Sigma), \Sigma'); \Omega \vdash S \doteq S_1 : \tau > \alpha/(\theta', \theta_1)$ and
   $((\Lambda, \Sigma), \Sigma'); \Omega \vdash S \doteq S_2 : \tau > \alpha/(\theta'', \theta_2)$ and $\theta' \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ and $\theta'' \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$.

Proof: Simultanous structural induction on the first derivation.

Case $\mathcal{D} = \dfrac{(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L_1 \sqcup L_2 : \tau_2 \Longrightarrow L/(\Sigma', \theta_1, \theta_2)}{(\Lambda, \Sigma); \Omega \vdash \lambda x.L_1 \sqcup \lambda x.L_2 : \tau_1 \rightarrow \tau_2 \Longrightarrow \lambda x.L/(\Sigma',\ \theta_1, \theta_2)}$

$(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L \doteq L_1 : \tau_2/(\theta', \theta_1)$ and $\theta' \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by i.h.
$(\Lambda, \Sigma); \Omega \vdash \lambda x.L \doteq \lambda x.L_1 : \tau_1 \rightarrow \tau_2/(\theta', \theta_1)$ by rule
$(\Lambda, \Sigma); \Omega, x{:}\tau_1 \vdash L \doteq L_2 : \tau_2/(\theta'', \theta_2)$ and $\theta'' \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by i.h.
$(\Lambda, \Sigma); \Omega \vdash \lambda x.L \doteq \lambda x.L_2 : \tau_1 \rightarrow \tau_2/(\theta'', \theta_2)$ by rule

Case $\mathcal{D} = \dfrac{u{::}(\Phi \vdash \alpha) \in \Lambda}{(\Lambda, \Sigma); \Omega \vdash u[\pi] \sqcup u[\pi] : \alpha \Longrightarrow u[\pi]/(\cdot, \cdot, \cdot)}$

$[\pi]^{-1}(u[\pi])/u) = (u[\mathsf{id}_\Phi]/u) = id_{\Lambda_1}$ where $\Lambda_1 = u{::}\Phi{\vdash}\alpha$

$\Lambda, \Sigma; \Omega \vdash u[\pi] \doteq u[\pi] : \alpha/(u[\mathsf{id}_\Phi]/u, \cdot)$ and $u[\mathsf{id}_\Phi/u] \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by rule $existsL - 2$

Case $\mathcal{D} = \dfrac{u{::}\Phi{\vdash}\alpha \in (\Lambda, \Sigma) \qquad i \text{ must be new}}{(\Lambda, \Sigma); \Omega \vdash u[\pi] \sqcup L : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/(i{::}\Omega{\vdash}\alpha,\ u[\pi]/i,\ L/i)}\ 1a$

$\Lambda, \Sigma; \Omega \vdash i[\mathsf{id}_\Omega] \doteq u[\pi] : \alpha/(\cdot, u[\pi]/i)$ and $\cdot \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by rule

$\Lambda, \Sigma; \Omega \vdash i[\mathsf{id}_\Omega] \doteq L : \alpha/(\cdot, L/i)$ and $\cdot \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by rule

Case Cases for rules (1b) and (2) follow the same pattern.

Case $\mathcal{D} = \dfrac{H_1 \neq H_2 \qquad i \text{ must be new}}{(\Lambda, \Sigma); \Omega \vdash H_1 \cdot S_1 \sqcup H_2 \cdot S_2 : \alpha \Longrightarrow i[\mathsf{id}_\Omega]/((i{::}\Omega{\vdash}\alpha), (H_1 \cdot S_1/i), (H_2 \cdot S_2/i))}$

$\Lambda, \Sigma; \Omega \vdash i[\mathsf{id}_\Omega] \doteq H_1 \cdot S_1 : \alpha/(\cdot, H_1 \cdot S_1/i)$ and $\cdot \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by rule

$\Lambda, \Sigma; \Omega \vdash i[\mathsf{id}_\Omega] \doteq H_2 \cdot S_2 : \alpha/(\cdot, H_2 \cdot S_2/i)$ and $\cdot \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by rule

Case $\mathcal{D} = (\Lambda, \Sigma); \Omega \vdash (L_1; S_1) \sqcup\!\!\!\sqcup (L_2; S_2) : \tau_1 \to \tau_2 > \alpha \Longrightarrow$
$(L; S)/((\Sigma_1, \Sigma_2), (\theta_1, \theta_2), (\theta'_1, \theta'_2))$

$(\Lambda, \Sigma); \Omega \vdash L_1 \sqcup L_2 : \tau_1 \Longrightarrow U/(\Sigma_1,\ \theta_1, \theta'_1)$ by inversion

$(\Lambda, \Sigma); \Omega \ \vdash\ S_1 \sqcup\!\!\!\sqcup S_2 : \tau_2 > \alpha \Longrightarrow S/(\Sigma_2,\ \theta_2, \theta'_2)$

$(\Lambda, \Sigma); \Omega \vdash L \sqcup L_1 : \tau_1/(\theta^*_1, \theta_1)$ and $\theta^*_1 \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by i.h.

$(\Lambda, \Sigma); \Omega \vdash L \sqcup L_2 : \tau_1/(\theta^*_2, \theta_2)$ and $\theta^*_2 \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by i.h.

$(\Lambda, \Sigma); \Omega \ \vdash\ S \sqcup\!\!\!\sqcup S_1 : \tau_2 > \alpha/(\theta^{**}_1, \theta'_1)$ and $\theta^{**}_1 \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by i.h.

$(\Lambda, \Sigma); \Omega \ \vdash\ S \sqcup\!\!\!\sqcup S_2 : \tau_2 > \alpha/(\theta^{**}_2, \theta'_2)$ and $\theta^{**}_2 \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ by i.h.

$(\Lambda, \Sigma); \Omega \vdash (L; S) \sqcup (L_1; S_1) : \tau_1/((\theta^*_1, \theta^{**}_1), (\theta_1, \theta'_1))$ by rule

$(\Lambda, \Sigma); \Omega \vdash (L; S) \sqcup (L_2; S_2) : \tau_1/((\theta^*_2, \theta^{**}_2), (\theta_2, \theta'_2))$ by rule

$(\theta^*_2, \theta^{**}_2) \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ and $(\theta^*_1, \theta^{**}_1) \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$

□

Theorem 84 (Interaction of insertion and retrieval for substitutions)
If $(\Lambda, \Sigma) \vdash \rho_1 \sqcup \rho_2 : \Sigma' \Longrightarrow \rho/(\Sigma'', \theta_1, \theta_2)$ *then* $(\Lambda, \Sigma), \Sigma'' \vdash \rho \doteq \rho_1 : \Sigma'/(\theta', \theta_1)$ *and* $(\Lambda, \Sigma), \Sigma'' \vdash \rho \doteq \rho_2 : \Sigma'/(\theta'', \theta_2)$ *and* $\theta' \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$ *and and* $\theta'' \subseteq \mathsf{id}_{(\Lambda,\Sigma)}$.

Proof: Structural induction on the first derivation and use of lemma 83. □

Next, we show how to traverse the tree, find a path $[\![\rho_n]\!][\![\rho_{n-1}]\!]\dots\rho_1$ such that ρ_2 is an instance of it and return a modal substitution θ such that $[\![\theta]\!][\![\rho_n]\!][\![\rho_{n-1}]\!]\dots\rho_1 = \rho_2$. Traversal of the tree is straightforward.

$$\frac{\Lambda, \Sigma \vdash \rho \doteq \rho_2 : \Sigma'/(\theta', \rho') \quad \Lambda \vdash C \doteq \rho' : \Sigma/\theta}{\Lambda \vdash [(\Sigma \vdash \rho \twoheadrightarrow C), C'] \doteq \rho_2 : \Sigma'/(\theta', \theta)}$$

$$\frac{\begin{array}{r}\text{there is no derivation such that} \Lambda, \Sigma \vdash \rho \doteq \rho_2 : \Sigma'/(\theta', \rho') \\ \Lambda \vdash C' \doteq \rho : \Sigma/\theta\end{array}}{\Lambda \vdash [(\Sigma \vdash \rho \twoheadrightarrow C), C'] \doteq \rho_2 : \Sigma'/\theta}$$

Theorem 85 (Soundness of retrieval)
If $\Lambda \vdash C \doteq \rho' : \Sigma'/\theta$ *then there exists a child* C_i *with substitution* ρ_i *in* C *such that the path* $[\![\theta]\!][\![\rho_n]\!][\![\rho_{n-1}]\!]\dots[\![\rho_i]\!] = \rho'$.

Proof: By structrural induction on the first derivation and use of lemma 81. □

Finally, we show that if we insert ρ into a substitution tree and obtain a new tree, then we are able to retrieve ρ from it.

Theorem 86 (Interaction between insertion and retrieval)
If $\Lambda \vdash (\Sigma \vdash \rho \twoheadrightarrow C) \sqcup \rho_2 : \Sigma \Longrightarrow (\Sigma \vdash \rho \twoheadrightarrow C')$ *then* $\Lambda \vdash C \doteq \rho_2/\mathsf{id}_\Delta$.

Proof: Structural induction on the derivation using lemma 84. □

5.4 Experimental results

In this section, we discuss examples from three different applications which use the tabled logic programming engine in Twelf. Here we focus on an evaluation of the indexing technique. For a more in depth-discussion of tabling and these examples we refer the reader to the previous chapter 4. All experiments are done on a machine with the following specifications: 1.60GHz Intel Pentium Processor, 256 MB RAM. We are using SML of New Jersey 110.0.3 under Linux Red Hat 7.1. Times are measured in seconds. All the examples use variant checking as a retrieval mechanism. Although we have implemented subsumption checking, we did not observe substantial performance improvements using subsumption. A similar observation has been made for tabling in the first-order logic programming engine XSB [59]. Potentially subsumption checking becomes more important in theorem proving, as the experience in the first-order setting shows.

5.4.1 Parsing of first-order formulae

Parsing and recognition algorithms for grammars are excellent examples for tabled evaluation, since we often want to mix right and left recursion (see also [68]). In this example, we adapted ideas from Warren [68] to implement a parser for first-order formulas using higher-order abstract syntax (see also Chapter 4).

#tok	noindex	index	reduction in time	improvement factor
20	0.13	0.07	46%	1.85
58	2.61	1.25	52%	2.08
117	10.44	5.12	51%	2.03
178	32.20	13.56	58%	2.37
235	75.57	26.08	66%	2.90

The first column denotes the number of tokens which are parsed. This example illustrates that indexing can lead to improvements by over a factor of 2.90. In fact, the more tokens need to be parsed and the longer the tabled logic programming engine runs, the larger the benefits of indexing. The table grows up to over 4000 elements in this example. This indicates that indexing prevents to some extent program degradation due to large tables and longer run-times.

5.4.2 Refinement type-checker

In this section, we discuss experiments with a bi-directional type-checking algorithm for a small functional language with intersection types which has been developed by Davies and Pfenning [17]. We use an implementation of the bi-directional type-checker in *Elf* by Pfenning. The type-checker is executable with the original logic programming interpreter, which performs a depth-first search. However, redundant computation may severely hamper its performance as there are several derivations for proving that a program has a specified type (see also Chapter 4).

We give several examples which are grouped in three categories. In the first category, we are interested in finding the first answer to a type checking problem and once we have found the answer execution stops. The second category contains example programs which are not well-typed and the implemented type-checker rejects these programs as not well-typed. The third category are examples where we are interested in finding all answer to the type-checking problem.

First answer

example	noindex	index	reduction	improvement
			time	factor
sub1	3.19	0.46	86%	6.93
sub2	4.22	0.55	87%	7.63
sub3	5.87	0.63	89%	9.32
mult	7.78	0.89	89%	8.74
square1	9.08	0.99	89%	9.17
square2	9.02	0.98	89%	9.20

Not provable

example	noindex	index	reduction	improvement
			time	factor
multNP1	2.38	0.38	84%	6.26
multNP2	2.66	0.51	81%	5.22
plusNP1	1.02	0.24	76%	4.25
plusNP2	6.48	0.85	87%	7.62
squareNP1	9.29	1.09	88%	8.52
squareNP2	9.26	1.18	87%	7.85

All answers

example	noindex	index	reduction time	improvement factor
sub1	6.88	0.71	90%	9.69
sub2	3.72	0.48	87%	7.75
sub3	4.99	0.59	88%	8.46
mult	9.06	0.98	89%	9.24
square1	10.37	1.11	89%	9.34
square2	10.30	1.08	90%	9.54

As the results demonstrate indexing leads to substantial improvements by over a factor of 9. Table sizes are around 500 entries.

5.4.3 Evaluating Mini-ML expression via reduction

In the third experiment we use an implementation which evaluates expressions of a small functional language via reduction. The reduction rules are highly non-deterministic containing reflexivity and transitivity rules.

example	noindex	index	reduction time	improvement factor
mult1	10.86	6.26	57%	1.73
mult2	39.13	18.31	47%	2.14
addminus1	54.31	14.42	73%	3.77
addminus2	57.34	15.66	73%	3.66
addminus3	55.23	25.45	54%	2.17
addminus4	144.73	56.63	61%	2.55
minusadd1	1339.16	462.83	65%	2.89

As the results demonstrate, performance is improved by up to 3.77. Table size was around 500 entries in the table. The limiting factor in this example is not necessarily the table size but the large number of suspended goals which is over 6000. This may be the reason why the speed-up is not as large as in the refinement type-checking example.

5.5 Related work and conclusion

We have presented a higher-order term indexing technique, called higher-order substitution trees. We only know of two other attempts to design and implement a higher-order term indexing technique. L. Klein [32] developed in his master's thesis a higher-order term indexing technique for simply typed terms where algorithms are based on a fragment of Huet's higher-order unification algorithm, the simplification rules. Since the most specific linear generalization of two higher-order terms does not exist in general, he suggests to maximally decompose a term into its atomic subterms. This approach result in larger substitution trees and stores redundant substitutions. In addition, he does not use explicit substitutions leading to further redundancy in the representation of terms. As no linearity criteria is exploited, the consistency checks need to be performed eagerly, potentially degrading the performance.

Necula and Rahul briefly discuss the use of automata driven indexing for higher-order terms in [42]. Their approach is to ignore all higher-order features when maintaining the index, and return an imperfect set of candidates. Then full higher-order unification on the original terms is used to filter out the ones which are in fact unifiable in a post-processing step. They also implemented Huet's unification algorithm, which is highly nondeterministic. Although they have achieved substantial speed-up for their application in proof-carrying code, it is not as general as the technique we have presented here. The presented indexing technique is designed as a perfect filter for linear higher-order patterns. For objects which are not linear higher-order patterns, we solve variable definitions via higher-order unification, but avoid calling higher-order unification on the original term.

So far, we have implemented and successfully used higher-order substitution trees in the context of higher-order tabled logic programming. The table is a dynamically built index, i.e. when evaluating a query we store intermediate goals encountered during proof search. One interesting use of indexing is in indexing the actual higher-order logic program. For this we can build the index statically. Although the general idea of substitution trees is also applicable in this setting there are several important optimizations. For example, we can compute an optimal substitution tree via unification factoring [18] for a static set of terms to get the best sharing among clause heads. In the future, we plan to adopt and optimize substitution tree indexing for indexing higher-order logic programming clauses.

Chapter 6

An implementation of tabling

Tabled logic programming has been successfully applied in a wide variety of applications, such as parsing, deductive databases, program analysis and more recently in verification through model checking. In previous chapters, we have shown that tabling techniques are also very useful and effective in the higher-order setting. Although the overall idea of tabling is easy to understand on an informal level, there is surprisingly little work on how to implement a tabled logic programming interpreter efficiently and describing its essential features.

The most successful implementation of tabling is found in XSB. It is also the only general Prolog system that offers tabling implemented at the level of the underlying abstract machine. XSB is based on the SLG-WAM, an extension of the WAM (Warren Abstract Machine) with special tabling support [62]. Techniques like suspending and resuming computation, require a fair amount of modifications to some of the WAM's instructions. Although this approach seems to work efficiently, it is difficult to understand and to re-implement. The WAM itself already "resembles an intricate puzzle, whose many pieces fit tightly together in a miraculous way" [5]. The SLG-Wam is a carefully engineered extension where special support for tabling is woven into the WAM instruction set. This results in a small run-time overhead (roughly 10%), even if no tabled predicate is executed. To remedy this situation and provide a simpler explanation to implementing tabling, Demoen and Sagonas [20, 21] propose a different approach. Execution environments of the WAM are preserved by copying the state to a separate memory area. In theory, this approach can be arbitrarily worse, but it is competitive and often faster than SLG-WAM in practice. Although this approach does not require modifications to the WAM instructions, it is still very closely tied into the WAM.

In this chapter, we describe the implementation of a tabled higher-order logic programming interpreter in a functional language like SML. The implementation follows closely the previous semi-functional implementation of the higher-order logic programming interpreter [23]. We describe a true tabled interpreter, rather than how to implement tabling within logic programming or within the WAM. We give an operational interpretation of the uniform proof system presented in Chapter 4. In particular, we will focus on some of the implementation issues such as suspending and resuming computation, retrieving answers, and trailing. It is intended as a high-level explanation of adding tabling to an existing logic programming interpreter. This hopefully enables rapid prototype implementations of tabled logic programming interpreters, even for linear logic programming and other higher-order logic programming systems. In addition, it facilitates the comparisons and experiments with different implementations.

6.1 Background

We first describe on a high-level how the higher-order logic programming interpreter works operationally. We omit a discussion on compilation, but rather focus on the uniform proof system given in Chapter 4. The interested reader can find a first step on compiling uniform proofs into an intermediate language in [9]. Here we give an operational view of the overall search and discuss some of the implementation issues.

Computation starts by decomposing the goal A, which we are trying to prove, until it is atomic.

1. Given an atomic goal P, look through the program Γ to establish a proof for P.

2. Given goal $A_1 \rightarrow A_2$, add the dynamic assumption A_1 to the programs in Γ and attempt to solve the goal A_2 from the extended program $\Gamma, c{:}A_1$.

3. Given a universally quantified goal $\Pi x{:}A_1.A_2$, we generate a new parameter c, and attempt to solve $[c/x]A_2$ in the extended program context $\Gamma, c{:}A_1$.

Once the goal is atomic, we need to select a clause from the program context Γ to establish a proof for P. In a logic program interpreter, we consider all the clauses in Γ in order. First, we will consider the dynamic assumptions, and then we will try the static program clauses one after the other. Let us assume, we picked a clause A from the program context Γ and we now need to establish a proof for P.

1. Given the atomic clause P', we establish a proof for the atomic goal P by checking if P' unifies with P. If yes then succeed. Otherwise fail and backtrack.

2. Given the clause $A_2 \rightarrow A_1$, we attempt to establish a proof of the atomic goal P, by trying to use the clause A_1 to establish a proof for P. If it succeeds attempt to solve the goal A_2. If it fails, backtrack.

3. Given the clause $\Pi x{:}A_1.A_2$, we establish a proof for the atomic goal P by generating a new logic (existential) variable u, and use the clause $[\![u[\mathsf{id}]/x]\!]A_2$ to establish a proof for the atomic goal P.

There are several issues which come up when implementing this operational view. The first question is how to implement backtracking. The fundamental idea underlying our implementation is the concept of continuations. When solving a goal $A_2 \rightarrow A_1$, we continue decomposing the clause A_1 to establish a proof for a goal P and delay the task of solving the sub-goal A_2 into the success continuation. So we can think of the success continuation as a stack of open proof obligations. Once we succeed in using the clause A_1 to establish a proof for the goal P, we call the success continuation. Backtracking is achieved by using the internal failure continuation of the ML interpreter. The failure continuation is triggered by simply returning from solving the goal A_1 with an uninteresting value. This is a technique which goes back to Carlson [7] and is described in more detail in [23].

The second implementation choice concerns the representation of variables. We use de Bruijn indices and explicit substitutions to implement bound variables. De Bruijn indices allow a standard representation without introducing names for bound variables. As a consequence, we need not re-name bound variables. Explicit substitutions [1] allow us to delay applying substitutions. In other words, substitutions are handled lazily in the implementation and only applied when needed. A similar idea has been employed in the Teyjus compiler [40] for λProlog seems crucial to efficient implementations of higher-order systems. The third choice concerns the handling of existential (logic) variables. We implement existential variables via references and destructive updates. This has several consequences. First, the existential variables occur free in objects. This is unlike the modal variables used in the theoretical development which are declared in a modal context Δ. We can translate an object U with free existential variables into a closed object U' where existential variables are interpreted as modal variables and declared in a modal context Δ by abstracting over the free existential variables in U. The second consequence is that whenever we instantiate an existential variable to a term, any occurrence of this existential variable is immediately updated. Since these updates continue to exist until we undo them, we need to make provisions to roll back the instantiation of existential variables upon backtracking. We use a global trail to keep track of instantiations of existential variables. The trail is implemented as a stack of references. At a choice point, we set a marker on the trail. Upon backtracking to this choice point, we uninstantiate all variables, which have been pushed onto the trail and simultaneously unwind the trail (that is pop the stack). Therefore, we proceed as follows: when selecting a clause A from the program context Γ to solve an atomic goal P, we set a choice-point on the trail. If we cannot establish a proof for the goal P using clause A, we come back to this choice-point, unwind the trail, and try the next clause from Γ.

Finally, we are not only interested in establishing that there exists a proof for a goal P, but in addition to the answer substitution, we return the actual proof as a proof term. During proof search, we simultaneously construct a proof term M. This is done in the success continuation.

A more detailed description of a semi-functional implementation of a higher-order logic programming interpreter together with ML-code can be found in [23].

6.2 Support for tabling

Next, we consider the implementation of a tabled higher-order logic interpreter. In Chapter 4, we described the underlying idea and presented an abstract view of a tabled higher-order logic programming interpreter. Here we discuss the implementation of the tabled higher-order logic programming interpreter in a semi-functional language.

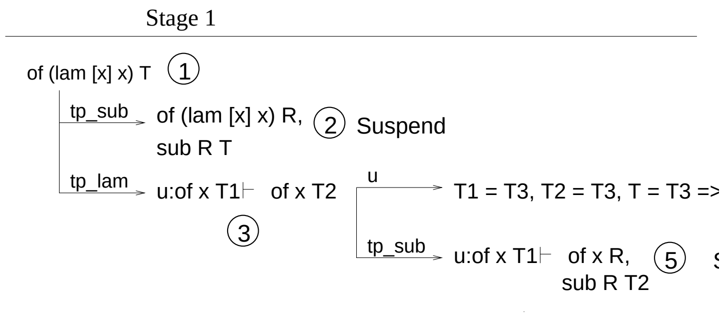
Tabling eliminates redundant and infinite computation by memoizing subcomputations and re-using their results later. When attempting to solve an atomic goal P in the program context Γ, we first check if the type family that P belongs to is marked tabled. If it is not, then we proceed as previously. If it is marked as tabled, then we proceed as follows: We check if we have seen the atomic goal P in context Γ before. If the current atomic goal P in context Γ is not in the table, we add it and proceed. Once we have established a proof for it, we will add the answer to it in the table.

If the atomic goal P in context Γ is already in the table, then there are two choices: If there are answers available, we would like to re-use them and continue. If no answers are available, then we need to suspend the execution and resume computation later on. In this case, we copy the trail together with the atomic goal P and the program context Γ and execution just fails. The overall search backtracks to the last choice point.

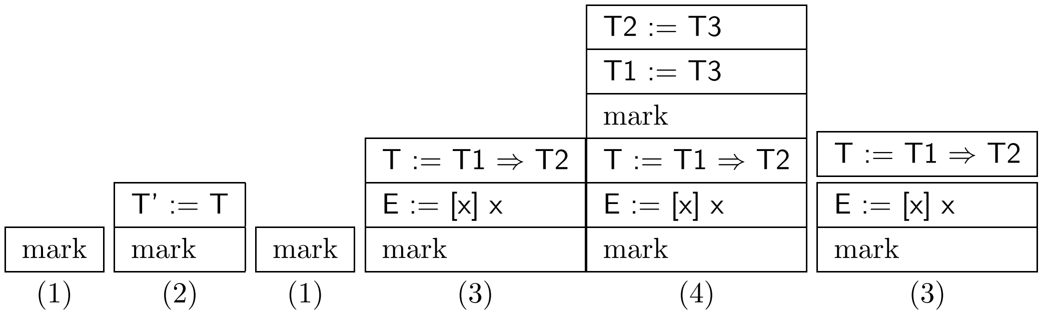
To illustrate the search process, recall the example given in Chapter 4. We first repeat the two clauses of the implementation of the type system including subtyping.

```
tp_sub  :  of E T'          tp_lam  :  of (lam ([x] E x)) (T1 => T2)
           <- of E R                   <- ({x:exp} of x T1 -> of (E x) T2).
           <- sub R T'.
```

Next, recall the proof tree for computing the types of the identity function.



We start with the goal of (lam [x] x) T marked with number (1). Before we apply the program clause tp_sub, we put a choice-point marker on the trail. Once the head of E T' of the clause tp_sub unifies with the current goal, we add to the trail T' := T. Since the goal of (lam [x] x) R is a variant of the original goal, computation is suspended, and the overall search fails and backtracks to the last choice point. Then we try the next clause tp_lam. When we unify the head of (lam [x] E x) (T1 => T2) with the current goal, we add two more entries to the trail. The evolution of the trail up to subgoal (4) is depicted below. Each column is labelled with the subgoal it belongs to.



Next, we will discuss the table, answers and suspended goals.

6.2.1 Table

The table plays a central role in the overall design of the tabled logic programming interpreter. Recall the definition of a table given earlier in Chapter 4. A table $\mathcal{T}$ is a collection of table entries. A table entry consists of two parts: a goal $\Delta; \Gamma \xrightarrow{u} a$ and a list $\mathcal{A}$ of pairs, answer substitutions $\Delta' \vdash \theta : \Delta$ and proof terms M, such that $\Delta'; [\![\theta]\!]\Gamma \xrightarrow{u} [\![\theta]\!]M : [\![\theta]\!]a$ is a solution.

A central question is then how to implement the table, store the table entries and answers, and define the right interface for accessing the table. We start with some general remark about the table and its entries. First of all, we implement for each tabled predicate a separate table and each table will be stored in a reference cell. So we have an array where the element `i` of the array contains the pointer (or reference) to the table of predicate `i`.

Second, we need to ensure quick table access. As we have shown in Chapter 3, higher-order pattern unification may not be efficient in general, but we can obtain an efficient algorithm for linear higher-order patterns. To ensure quick table access we therefore linearize the atomic goal P and the context Γ. As a result, we obtain a linear atomic goal a_l and a linear context Γ_l together with some variable definitions e.

Third, table entries must be closed in the sense that they are not allowed to contain any references to existential variables. References to existential variables may be instantiated during proof search which pollutes the table and would change the table entries destructively. To avoid this problem, we de-reference and abstract over all the existential variables in a term before inserting it into the table. We translate a term where existential variables may occur free into a term which is closed and existential variables are translated into modal variables which are bound in the modal context Δ.

Abstraction of the atomic goal P in program context Γ returns four parts:

- Δ: is the context for the modal variables
- Γ': is the abstraction of the context Γ
- P': is the abstraction of the atomic goal P.
- e': is the abstraction of the variable definitions e.

By invariant the abstracted atomic goal P' is well-typed in the ordinary context Γ' and the modal context Δ modulo the variable definitions e'. A table entry consists of this abstraction together with a reference to the corresponding answer list.

modal context	dynamic context	atomic goal	variable definitions	Answer Ptr
Δ	Γ'	P'	e	*answRef*

Table Entry

Now we can check if the abstracted linear atomic goal P' together with the context Γ, the modal context Δ and the variable definitions e is already in the table. To be more precise, if there is a table entry which is a variant of the abstracted goal. Since we use de Bruijn indices for representing bound variables, and all entries are in normal form, this can be done by checking syntactic equality. If the result is yes, we will return a pointer to the answer list. If the result is no, we add it to the table together with a pointer to an empty answer list and also return a pointer *answRef* to the still empty answer list. The intention is that once the goal P in context Γ has been solved, we add the answer substitution to the answer list *answRef* is pointing to. This is done in the success continuation.

This task will be just added to the success continuation. In other words the success continuation is not only a stack of open proof obligations (subgoals), but also of other obligations such as adding answers to the table. We illustrate this idea again by considering the previous example. At node (5), the success continuation looks as follows.

solve (sub R T2) using the program clauses and the dynamic assumption of x T
add answers for T1 and T2 to the *answRef* corresponding to of x T1 $\vdash$ of x T2
add answer for T to *answRef* corresponding to of (lam [x] x) T
initial success continuation

It not only contains the open subgoal sub R T2, but also the tasks of adding answers. It is important to note that adding answers to a table entry does not require another table lookup, since we have direct access to the reference of the corresponding answer list.

To facilitate adding answer substitutions, we extend the abstraction process slightly. Given a linear atomic goal a_l in a context Γ_l and the variable definitions e let $X_1, \dots, X_n$ be the free existential variables occurring in it. We not only compute the abstraction $\Delta; \Gamma' \vdash a'$ together with the variable definitions e', but also return a substitution θ from the context Δ to the existential variables $X_1, \dots, X_n$. In other words, the substitution θ, maps every abstracted existential variable in Δ to its reference such that $[\![\theta]\!]\Gamma' \vdash [\![\theta]\!]a'$ modulo the variable definitions $[\![\theta]\!]e'$ is a variant of the original the original goal $\Gamma \vdash a$.

Creating such a substitution θ for the existential variables is important for mainly two reasons: (1) As we will establish a proof for the goal $\Gamma \vdash a$, its existential variables will be instantiated. As a side-effect, θ will exactly contain these instantiations. Once we have found a proof for $\Gamma \vdash a$, θ will contain the answer substitution, and we can directly add θ to the answer list, without another additional table lookup. (2) Any retrieval operations will require no additional table lookup up. Since the substitution θ captures the free variables in the goal, and we have direct access to the answer list, subsequent retrieval operations do not again traverse the index together with the term, but can work solely on the substitutions. Therefore, we do exactly one lookup in the table for each tabled goal.

6.2.2 Indexing for table entries

The table itself is implemented via substitution tree indexing. We index on the abstracted linear atomic goal. At the leafs of the substitution tree we not only store the variable definitions e', but also the context Γ and the modal context Δ. In addition, we keep a pointer to the corresponding answer list at the leaf (see Figure 6.2.2).

There have been several reasons for this design, which we briefly discuss. First, indexing on the goal, rather than on the context Δ and Γ, allows for more structure sharing, since often the goals are similar, but the context Γ and Δ may differ. In addition it allows direct access to the goal of the table entry and allows to implement a table for each type family. Finally, the design is suitable for other optimization such as strengthening or context subsumption.

6.2.3 Answers

Next, we will describe adding answers and the answer list itself in more detail. Once we have solved a tabled goal, we need to add the answer substitution θ to the answer list. Recall that checking whether the table entry $(\Delta; \Gamma'; a'; e)$ is already in the table, returns a reference *answRef* to an answer list. Once we have established a proof for this entry, we need to add the corresponding answer substitutions θ to *answRef*. When adding the answer substitution to the answer list, there are two important observations to take into account. First, answer substitutions θ may contain free existential variables, as the previous example illustrates. Again we need to ensure that any answer substitutions in the table are ”closed” to avoid pollution of the table. Therefore we abstract over the answer substitution. Let θ' together with the modal context Δ be the abstraction of the answer substitution θ.

The second observation is that we not only generate answer substitutions, but also proof terms. One way to handle this issue is to store not only the abstracted answer substitution θ' together with the modal context Δ, but also the proof term. In the previous example, the corresponding proof term to node (3) is $(\mathsf{tp\_lam}\ [u{:}\mathsf{of}\ \ x\ P]\ u)$[1]. In practice however the corresponding proof terms may be large and may lead to severe performance penalties. Since the size of the actual proof term may be large, it would not be very space efficient to store them. Second, we also would need to compute proof terms simultaneously during proof search. This also may be expensive. In the implementation, we therefore choose to only carry around a skeleton of the proof term, from which the actual proof can be reconstructed. The skeleton keeps track of clauses have been applied to establish a proof, without keeping track of typing dependencies. For static clauses we take the identifier of the clause name and for dynamic clauses we

[1]We omitted here the implicit argument x. The explicit form is $(\mathsf{tp\_lam}\ [x{:}\mathsf{exp}]\ [u{:}\mathsf{of}\ \ x\ P]\ u)$.

take their position in the dynamic context Γ. So in the previous example, we store the identifier of tp_lam and a 1, denoting we used the first dynamic assumption. This is a simplified version of Necula and Rahul's proposal of oracles [42]. Necula and Rahul are even more minimalistic since they only encode the non-deterministic choices in the proof. In our setting, the answer list is set up as follows:

modal context	answer substitution	skeleton
Δ'	θ'	sk

Answer List

Finally, we need to have some more control on what answer substitutions are allowed to be re-used. As described previously, we enforce a multi-stage strategy, where we are only allowed to use answers from previous stages. Hence we also add a lookup counter n, which indicates any element greater then n is not allowed to be used.

The access cost for answers is proportional to the size of the answer substitution, rather than to the size of the answer itself. This idea is also known as substitution factoring [59]. As we see, this idea can be adopted to the higher-order setting in a straightforward way and modal substitutions inherently support substitution factoring. Every answer is essentially a reference to the actual answer record which allows direct access to answer substitutions.

To achieve a more efficient representation of the solutions, it is possible to replace the list of pairs, answer substitutions and proof skeletons with a trie. This would allow sharing of prefixes of substitutions. However, since in most example we did not encounter large lists of answer substitutions, we have not pursued this optimization.

6.2.4 Suspended goals

If we detect a a goal which is a variant of a table entry, we need to suspend the computation. We only need to store enough information to recover and retrieve answers once they become available. In fact, we only need to store a substitution θ, which contains the free existential variables, such that $\cdot \vdash \theta : \Delta$. In addition, we store the success continuation *sc* and the pointer *answRef* to the answer list. The idea is that once answers become available, we instantiate the substitution θ with the answer substitution from *answRef*. As we have direct access to *answRef* and we need no additional table lookup, we do not need to store the complete goal. We also do not store the stack of failure continuations. When resuming the suspended goals, we will not backtrack past them, since any earlier choice points have already been explored. A similar observation has been made by Demoen and Sagonas [19]. This also justifies our choice to re-use the ML internal continuation as a failure continuation, even if we add tabling to the interpreter.

This is however still not quite sufficient. The problem stems again from the existential variables which are implemented via references. Once we come back and resume the computation, the free existential variables in the substitution θ and in the success continuation may not be in the same state anymore. Therefore, we need to be able to capture the state of the existential variables and reconstruct it. We do this by copying the trail and storing a copy of the trail together with the substitution θ, the success continuation and the pointer to the answer list.

When copying the trail, we can remove all the marks from the trail, since we do not need to keep the choice points. In other words, we only need to store pairs of references and terms.

T' := T1
mark
T := T1 ⇒ T2
mark

ref

$\Rightarrow$

T'	T1
T	T1 ⇒ T2
E	[x] x

ref. inst.

Trail

Copied Trail

Copying the trail may be quite expensive and it may be possible to share common prefixes of the trails in suspended goals or even with the current trail. However, we have not investigated this idea further.

Finally, we need some more control on which answers from the answer list have been already used. Therefore, we associate with every suspended goal a lookup pointer k to which answers have already been used. So the suspended goals consist of the following 5 parts.

substitution	success cont.	copied trail	lookup ptr	reference to answer
θ	sc	ctrail	k	answRef

Suspended Goal

All the suspended goals are stored in a global queue, and resumed in the order they were suspended.

6.2.5 Resuming computation

To resume suspended goals, we set a choice-point and reset the existential variables on the copied trail to their previous state when we suspended the computation. Then we resume the computation by retrieving answers from the answer list `answRef`.

Recall that we use a multi-stage strategy, which restricts the the re-use of answers to the ones which have been derived in previous stages. Consider the following answer list to illustrate.

(Δ_1, θ_1), sk$_1$	(Δ_2, θ_2), sk$_2$	(Δ_3, θ_3), sk$_3$	(Δ_4, θ_4), sk$_4$	...
		k		n

n marks the answers generated in previous stages and any answers smaller than n are allowed to be re-used. The index k associated with every suspended goal, indicates that all answers smaller than k have already been re-used. Hence, we only retrieve answers between k and n.

Retrieval of an answer substitution is done in two steps: First, we create new existential variables for each element in Δ_i, s.t. ρ is a substitution with these fresh existential variables from modal context Δ_i to the empty context. The composition $[\![\rho]\!]\theta_i$ then re-introduces fresh existential variables to the closed answer substitution θ_i. Second, we unify $[\![\rho]\!]\theta_i$ with the substitution θ. If this succeeds, we return the corresponding proof skeleton sk_i to the success continuation and proceed to solve the success continuation.

If re-using the answer substitution θ_i failed, we backtrack the the last choice-point, and reuse the next answer substitution θ_{i+1}. If all answer substitutions have been re-used, we backtrack and try another suspended goal.

6.2.6 Multi-staged tabled search

Tabled search proceeds in stages. A stage terminates if all the nodes in the proof search tree are either success nodes or failure nodes. Once a stage terminates, we check if the table has changed, i.e. new answers have been added. If this is the case, then we revive the suspended goals and resume computation with the newly derived answers. If no new answers and no new table entries have been added, i.e. the table did not change, we are done and the search space has been saturated.

At the end of each stage, we update the answer pointer n, which indicates which answers were generated in previous stages and are therefore allowed to be reused.

6.3 Variant and subsumption checking

So far we have described tabled higher-order logic programming based on variant checking. The implementation is carefully designed and makes use of some important invariants to achieve efficient execution. For example, for each tabled goal we do exactly one lookup in the table. By returning the pointer to the answer list, subsequent retrieval operations only access the answer list, but do not again traverse the index together with the term. Moreover, no overhead is imposed for non-tabled evaluation, except one check whether the predicate is declared tabled or not. Any changes needed for tabling do not affect the execution of non-tabled predicates.

Subsumption checking allows us to check whether there is a more general entry already in the table such that the current goal is an instance of it. A clear advantage of subsumption checking is that it can potentially detect more loops and leads to smaller table sizes. Unfortunately, we cannot maintain all the invariants we exploit for variant-based lookups. In particular, there may be more than one entry which is more general than the current goal. Therefore a complete subsumption-based strategy needs to return all answer lists of these entries. This means that for each tabled goal, we may need more than one table lookup. When we encounter a tabled goal, we need to check whether it is already in the table. Later, any time when we retrieve answers for it, we need to perform another table-look up.

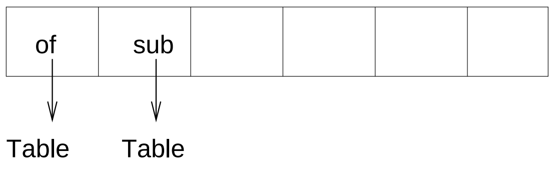
Experience in XSB has shown that although the table size may be smaller, the additional overhead required for subsumption often does not lead to performance improvements in tabled logic programming.

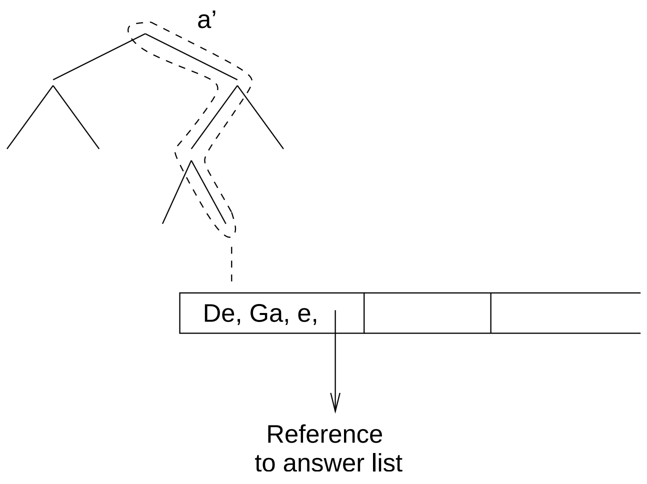
In the current implementation of the tabled higher-order logic programming interpreter, we have tried to minimize the changes and preserve the design for variant-checking. In particular, when a goal is added to the table, we still return a pointer to its answer list, and we will retrieve answers only from this list. This may possibly delay the reuse of answers and lead to incompleteness, but it avoids the effort of maintaining two different implementations.

The design previously discussed is then extended to allow subsumption checking in the following way: Look-up in the table is subsumption based, i.e. we check if the current abstracted linear atomic goal P' in context Γ' and modal context Δ' is an instance of a table entry $\Delta; \Gamma \vdash P$. Recall that all table entries are in canonical form. Therefore, $\Delta'; \Gamma' \vdash P'$ is an instance of $\Delta; \Gamma \vdash P$, if there exists a substitution ρ such that $\Delta' \vdash \rho : \Delta$ and $\Delta; [\![\rho]\!]\Gamma \vdash [\![\rho]\!]P \overset{c}{\Longleftrightarrow} P'$. In this case, we return the substitution ρ together with a reference *answRef* to the associated answer list. If $\Delta'; \Gamma; \vdash P'$ is a variant of the table entry $\Delta; \Gamma \vdash P$ then ρ will be a renaming substitution.

For retrieval, we then unify the composition of ρ and σ with the re-instantiated answer substitution θ_i. This is a conservative approach, which ties each goal to a unique table entry and avoids another table lookup, once we retrieve answers. Alternatively, we could do another table lookup when retrieving answers. We retrieve *all* answer lists of table entries such that the current goal is an instance of the entry. Note that in general there may be more than one such table entry. This may allow us to retrieve answers earlier, because we are not tied to one table entry. However, it requires an additional lookup every time we retrieve answers.

Although subsumption-checking allows many more optimizations and may lead to smaller tables, we have not found a substantial performance improvement. This is in fact consistent with observations made in the XSB system with subsumption-based tabling. On the other hand, theorem provers always rely on subsumption rather than variant checking to eliminate redundancy. As we will discuss in the next Chapter, subsumption-based checking may be more useful in higher-order theorem proving.


of
sub
Table
Table


a'
De, Ga, e,
Reference
to answer list

Chapter 7

Higher-order theorem proving with tabling

In this chapter, we concentrate on automating proof search in Twelf. Twelf not only provides a higher-order logic programming interpreter, but also a meta-theorem prover for reasoning *with* the inference rules of a deductive system and also about deductive systems using induction. The meta-theorem prover supports three key operations, splitting a proof into different cases, generating instances of the induction hypothesis and deriving a conjecture from a given set of assumptions and rules of inference. Using these three key operations, Twelf can prove many theorems and lemmas about specifications automatically.

Currently, the general proof search strategy used in Twelf is based on iterative deepening proof search. Although this works in many examples, it also has clear limitations. First, as the user has to specify a bound, failure of proof search becomes meaningless. The user does not know whether the bound was too low or whether there exists no proof for the conjecture. Moreover, often the performance of the search procedure highly depends on choosing the minimal bound. Although in theory it should not matter, if we give the minimal bound, in practice this may often be crucial for finding the proof.

Second, redundant computation may severely hamper the performance of proof search in Twelf in larger and more complex specifications. To prove properties about deductive systems often requires several lemmas and their application may lead to an explosion in the search space. Iterative deepening search does not scale well to larger examples and often fails to prove key lemmas and theorems due the large search space. The experience with memoization in logic programming in the other hand suggests that memoization in theorem proving may be useful to eliminate redundancy and artificial depth bounds on the proof search in many cases. In this chapter, we present a generic meta-theorem prover based on memoization and discuss several examples including type preservation proofs for type-system with subtyping, decomposition lemmas for a reduction semantics using evaluation context, several inversion lemmas about refinement types, and reasoning in classical natural deduction calculus. These examples include several lemmas and theorems which were not provable with iterative deepening, but can be proven with memoization-based search. Moreover, we show that in some cases no bound is needed on memoization-based search. This allows us to disprove some sub-cases in inductive proofs and show that no proof exists to the theorem. This is an important step towards a more robust and more efficient meta-theorem prover.

7.1 Example: type system with subtyping

As a motivating example, we discuss a type system with subtyping which we is a variant of the system already introduced in Chapter 4. In particular, we will present some proofs relating the declarative subtyping relation to a deterministic subtyping algorithm and show that the deterministic subtyping algorithm is sound and complete. Finally we show that type preservation holds.

To make the chapter self-contained, we start with defining the types. Included are the base types nat for natural numbers which includes the positive numbers and zero. zero denotes the number zero and pos contains all positive numbers. Numbers are constructed by z and the successor function s .

$$\begin{array}{lll} \text{Expressions} \quad M & ::= & \mathsf{z} \mid \mathsf{s}\ M \mid \mathsf{lam}\ x.M \mid \mathsf{app}\ M_1\ M_2 \mid \mathsf{pair}\ M_1\ M_2 \mid \\ & & \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \mathsf{case}\ M\ \ \mathsf{of}\ \ \mathsf{z} \Rightarrow M_0 \mid (\mathsf{s}\ x) \Rightarrow M_1 \\ \text{Types} \quad A & ::= & \mathsf{nat} \mid \mathsf{zero} \mid \mathsf{pos} \mid A_1 \rightarrow A_2 \end{array}$$

The subtyping judgment for this language has the form :

$$A \leq B \quad \text{Type } A \text{ is a subtype of } B$$

The subtyping relation is defined straightforwardly using reflexivity and transitivity by the following rules

$$\frac{}{A \ \leq A} \qquad \frac{A_1 \leq A_2 \quad A_2 \leq A_3}{A_1 \leq A_3}$$

$$\frac{}{\mathsf{pos} \leq \mathsf{nat}} \qquad \frac{}{\mathsf{zero} \leq \mathsf{nat}}$$

$$\frac{A_1 \leq B_1 \quad A_2 \leq B_2}{A_1 * A_2 \leq B_1 * B_2} \qquad \frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2}$$

Next, we give the typing judgment and the evaluation judgment:

$$\begin{array}{ll} \Gamma \vdash M : A & \text{Term } M \text{ has type } A \text{ in context } \Gamma \\ M \hookrightarrow V & \text{Term } M \text{ evaluates to value } V \end{array}$$

Typing rules

$$\frac{\Gamma \vdash M : A' \quad A' \leq A}{\Gamma \vdash M : A}$$

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathsf{nat}} \qquad \frac{}{\Gamma \vdash \mathsf{z} : \mathsf{zero}} \qquad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{s}\ M : \mathsf{nat}} \qquad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{s}\ M : \mathsf{pos}}$$

$$\frac{\Gamma \vdash M : \mathsf{nat} \quad \Gamma \vdash M_1 : A \quad \Gamma, x{:}\mathsf{nat} \vdash M_2 : A}{\Gamma \vdash \mathsf{case}\ M\ \ \mathsf{of}\ \ \mathsf{z} \Rightarrow M_1 \mid (\mathsf{s}\ x) \Rightarrow M_2 : A}$$

$$\frac{\Gamma \vdash M : \mathsf{zero} \quad \Gamma \vdash M_1 : A}{\Gamma \vdash \mathsf{case}\ M\ \ \mathsf{of}\ \ \mathsf{z} \Rightarrow M_1 \mid (\mathsf{s}\ x) \Rightarrow M_2 : A} \qquad \frac{\Gamma \vdash M : \mathsf{pos} \quad \Gamma, x{:}\mathsf{nat} \vdash M_2 : A}{\Gamma \vdash \mathsf{case}\ M\ \ \mathsf{of}\ \ \mathsf{z} \Rightarrow M_1 \mid (\mathsf{s}\ x) \Rightarrow M_2 : A}$$

$$\frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \mathsf{pair}\ M_1\ M_2 : (A_1 * A_2)} \qquad \frac{\Gamma \vdash M : (A_1 * A_2)}{\Gamma \vdash \mathsf{fst}\ M : A_1} \qquad \frac{\Gamma \vdash M : (A_1 * A_2)}{\Gamma \vdash \mathsf{snd}\ M : A_2}$$

$$\frac{\Gamma, x{:}A_1 \vdash M : A_2}{\Gamma \vdash \mathsf{lam}\ x.M : (A_1 \rightarrow A_2)} \qquad \frac{\Gamma \vdash M_1 : (A_2 \rightarrow A) \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \mathsf{app}\ M_1\ M_2 : A}$$

$$\frac{}{\mathsf{z} \hookrightarrow \mathsf{z}} \qquad \frac{M \hookrightarrow V}{\mathsf{s}\ M \hookrightarrow \mathsf{s}\ V}$$

Evaluation rules

$$\frac{M \hookrightarrow M' \quad N \hookrightarrow N'}{\mathsf{pair}\ M\ N \hookrightarrow \mathsf{pair}\ M'\ N'} \qquad \frac{M \hookrightarrow \mathsf{pair}\ V_1 V_2}{\mathsf{fst}\ M \hookrightarrow V_1} \qquad \frac{M \hookrightarrow \mathsf{pair}\ V_1 V_2}{\mathsf{snd}\ M \hookrightarrow V_2}$$

$$\frac{M \hookrightarrow \mathsf{z} \quad M_1 \hookrightarrow V_1}{\mathsf{case}\ M\ \ \mathsf{of}\ \ \mathsf{z} \Rightarrow M_1 \mid (\mathsf{s}\ x) \Rightarrow M_2 \hookrightarrow V_1} \qquad \frac{M \hookrightarrow (\mathsf{s}\ V_2) \quad [V_2/x]M_2 \hookrightarrow V_2}{\mathsf{case}\ M\ \ \mathsf{of}\ \ \mathsf{z} \Rightarrow M_1 \mid (\mathsf{s}\ x) \Rightarrow M_2 \hookrightarrow V_2}$$

$$\frac{}{\mathsf{lam}\ x.M \hookrightarrow \mathsf{lam}\ x.M} \qquad \frac{M_1 \hookrightarrow \mathsf{lam}\ x.M' \quad M_2 \hookrightarrow V_2 \quad [V_2/x]M'}{\mathsf{app}\ M_1\ M_2 \hookrightarrow V}$$

The following substitution principle holds:

Lemma 87 (Substitution principle)
If $\Gamma, x{:}A \vdash M : B$ and $\Gamma \vdash N : A$ then $\Gamma \vdash [N/x]M : B$.

Proof: Induction on the first derivation. □

The subtyping rules above do not immediately yield an algorithm for deciding subtyping. Thus we will present an algorithmic subtyping judgment and show that it is equivalent to the one above.

$$A \trianglelefteq B \quad \text{Type } A \text{ is a subtype of } B$$

$$\frac{}{\mathsf{nat} \trianglelefteq \mathsf{nat}} \qquad \frac{}{\mathsf{pos} \trianglelefteq \mathsf{pos}} \qquad \frac{}{\mathsf{zero} \trianglelefteq \mathsf{zero}}$$

$$\frac{}{\mathsf{zero} \trianglelefteq \mathsf{nat}} \qquad \frac{}{\mathsf{pos} \trianglelefteq \mathsf{nat}}$$

$$\frac{A_1 \trianglelefteq B_1 \quad A_2 \trianglelefteq B_2}{(A_1 * A_2) \trianglelefteq (B_1 * B_2)} \qquad \frac{B_1 \trianglelefteq A_1 \quad A_2 \trianglelefteq B_2}{(A_1 \rightarrow A_2) \trianglelefteq (B_1 \rightarrow B_2)}$$

Theorem 88 (Admissibility of reflexivity)
For any type A, there exists a derivation $A \trianglelefteq A$.

Proof: Structural induction on A. □

Theorem 89 (Admissibility of transitivity)
If $\mathcal{D}_1 : A \trianglelefteq B$ and $\mathcal{D}_2 : B \trianglelefteq C$ then $\mathcal{E} : A \trianglelefteq C$.

Proof: The size of a judgment $A \trianglelefteq B$ is determined by the size of A plus the size of B. Induction on the size of the judgment[1]. $\Box$

Theorem 90 (Soundness and completeness of algorithmic subtyping)

1. *If $\mathcal{E} : A_1 \trianglelefteq\ A_2$ then $\mathcal{D} : A_1 \leq A_2$*

2. *If $\mathcal{D} : A_1 \leq A_2$ then $\mathcal{E} : A_1 \trianglelefteq\ A_2$*

Proof: Soundness: Proof by structural induction on $\mathcal{E}$.
Completeness: Proof by structural induction on $\mathcal{D}$ referring to reflexivity and transitivity lemma. $\Box$

To prove type preservation, we need a couple of inversion lemmas.

Theorem 91 (Inversion)

1. *If $\Gamma \vdash \mathsf{z} : A$ and $A \trianglelefteq \mathsf{pos}$ then this is a contradiction.*

2. *If $\Gamma \vdash \mathsf{s}\ M : A$ and $A \trianglelefteq \mathsf{pos}$ then $\Gamma \vdash M : \mathsf{nat}$.*

3. *If $\Gamma \vdash \mathsf{s}\ M : A$ and $A \trianglelefteq \mathsf{nat}$ then $\Gamma \vdash M : \mathsf{nat}$.*

4. *If $\Gamma \vdash \mathsf{s}\ M : A$ and $A \trianglelefteq \mathsf{zero}$ then this is a contradiction.*

5. *If $\Gamma \vdash \mathsf{lam}\ x.M : A$ and $A \trianglelefteq (A_1 \to A_2)$ then $\Gamma, x{:}A_1 \vdash M : A_2$.*

6. *If $\Gamma \vdash \mathsf{pair}\ M_1\ M_2 : A$ and $A \trianglelefteq (A_1 * A_2)$ then $\Gamma \vdash M_1 : A_2$ and $\Gamma \vdash M_2 : A_2$.*

Proof: All the proofs follow by structural induction on the first derivation. $\Box$

Theorem 92 (Type preservation)
If $\mathcal{D} : \Gamma \vdash M : A$ and $\mathcal{E} : M \hookrightarrow V$ then $\Gamma \vdash V : A$.

[1] *Because of the contra-variance of types in the subtyping rule for function types, the induction ordering is a little tricky. In Twelf, we use a trick to handle this induction. We use simultaneous induction on D_1 and D_2 and define a second copy of the transitivity relation to handle the contra-variance case, by a structural nested induction on $\mathcal{D}_1$ and $\mathcal{D}_2$.*

Proof: Structural lexicographic induction on $\mathcal{D}$ and $\mathcal{E}$ using the previously proven lemmas. We only show a few cases here to illustrate the overall structure of the proof.

Case $\mathcal{D} = \dfrac{\Gamma \vdash \overset{\mathcal{D}_1}{M} : B \qquad \Gamma \vdash \overset{\mathcal{D}_2}{B} \leq A}{\Gamma \vdash M : A}$

$M \hookrightarrow V$	by assumption
$\Gamma \vdash V : B$	by induction hypothesis
$\Gamma \vdash V : A$	by tp_sub

Case $\mathcal{D} = \dfrac{\Gamma \vdash M_1 \overset{\mathcal{D}_1}{:} (A \to B) \qquad \Gamma \vdash \overset{\mathcal{D}_2}{M_2} : A}{\Gamma \vdash (\mathsf{app}\ M_1\ M_2) : B}$

$\mathsf{app}\ M_1\ M_2 \hookrightarrow V$	by assumption
$M_1 \hookrightarrow (\lambda x.M')$ and $M_2 \hookrightarrow V_2$ and $[V_2/x]M' \hookrightarrow V$	by inversion
$\Gamma \vdash V_2 : A$	by induction hypothesis
$\Gamma \vdash \lambda x.M' : (A \to B)$	by induction hypothesis
$\Gamma, x{:}A \vdash M' : B$	by inversion lemma
$\Gamma \vdash [V_2/x]M' : B$	by substitution lemma
$\Gamma \vdash V : B$	by induction hypothesis

Case $\mathcal{D} = \dfrac{\Gamma \vdash M \overset{\mathcal{D}_1}{:} (A * B)}{\Gamma \vdash (\mathsf{fst}\ M) : A}$

$\mathsf{fst}\ M \hookrightarrow V_1$	by assumption
$M \hookrightarrow (\mathsf{pair}\ V_1\ V_2)$	by inversion
$\Gamma \vdash (\mathsf{pair}\ V_1\ V_2) : (A * B)$	by induction hypothesis
$\Gamma \vdash V_1 : A$	by inversion lemma

$\square$

The proofs are straightforward by hand. We distinguish cases on the derivations (or on the type), reason by induction hypothesis, and apply lemmas or rules of inference to deduce the correct conclusion. On the other hand, these proofs, in particular the final type preservation theorem, are quite challenging for an automated meta-theorem prover.

7.2 Twelf: meta-theorem prover

In previous chapters, we outlined how to implement the previous type-system in Twelf as a higher-order logic program and discussed how to execute. In this chapter, we are interested in automating the reasoning *with* and *about* deductive systems. Examples of such meta-reasoning are the admissibility lemmas and the equivalence theorem about the refinement type systems in the previous section. We also refer to these properties as meta-theorems.

Schürmann showed in his Ph.D. thesis [63] that it is possible to reconcile higher-order abstract syntax and inductive reasoning in theory and even automate the inductive reasoning in practice. The theoretical development is complex, and we refer the interested reader to [63, 64]. In this section, we review the design and implementation of the inductive meta-theorem prover. Closely following the informal proof above, the meta-theorem prover has three main operations it iterates over. Case analysis (reasoning by inversion), computing instances of the induction hypothesis, proof search where we directly prove the current conclusion from the current proof assumptions and the rules of inference. Proof assumptions are either instances of the induction hypothesis or other assumptions generated applying case analysis, and assumptions from the statement of the theorem. Note that generating instances of the induction hypothesis is separated from proof search itself and added to the set of proof assumptions. During proof search, induction hypotheses are treated as any other proof assumptions.

The design reflects the three key operations in the informal proof: analyzing cases, generating instances of the induction hypothesis, and deriving a proof for using inference rules and proof assumptions. The overall meta-theorem prover simply iterates over all the three operations, until a proof for all open cases has been found. It is important to point out, that the meta-theorem prover can only make progress in the proof, once proof search has failed to show that the conclusion is derivable from the current proof assumptions. This is the pre-requisite to splitting a conjecture into different cases. Hence quick failure is essential to the overall performance of the meta-theorem prover.

Consider the proof for type preservation given earlier. First the meta-theorem prover will try to show that $\mathcal{F} : \Gamma \vdash v : a$ is derivable from the rules of inference, the lemmas and the assumptions $\mathcal{D} : \Gamma \vdash m : a$ and $\mathcal{E} : m \hookrightarrow v$. We write small letters v, m,

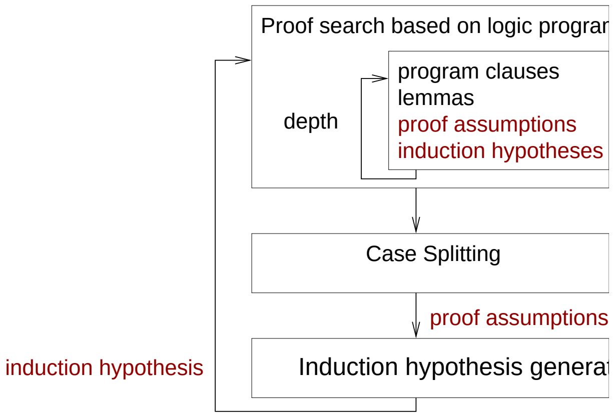


Figure 7.1: Design of the induction meta-theorem prover

a to indicate these variables are universal and are treated like constants. If direct proof search fails, then we split the conjecture into different cases. There are two possibilities: case analysis on $\mathcal{E} : m \hookrightarrow v$ and $\mathcal{D} : \Gamma \vdash m : a$. Using heuristics, the meta-theorem prover chooses to split on the typing derivation $\mathcal{D} : \Gamma \vdash m : a$. Applying inversion on the subtyping rule, this leads to the new proof assumptions $\Gamma \vdash m : a'$ and $a' \leq a$. Next, we generate $\Gamma \vdash v : a'$ which is an instance of the induction hypothesis. Finally, using all these assumptions, we search for a proof of $\Gamma \vdash v : a$. We can establish a proof for it, but using the proof assumption $a' \leq a$, the induction hypothesis $\Gamma \vdash v : a'$ and the subtyping rule.

To complete the proof, we need to consider the remaining 12 cases for typing rules.

As the example illustrate, generic proof search plays an important role in establishing a proof for type preservation automatically. It is first used to show that no proof exists for the original statement $\mathcal{F} : \Gamma \vdash v : a$, which is a prerequisite for applying case-analysis. Second, it is used to close the proof in each of the cases after induction hypotheses are generated.

Schürmann adopted in his Ph.D. thesis the simplest solution possible for searching for a proof given a set of assumptions. In principle, we can just use the proof search procedure of the logic programming interpreter to search for a proof. The main drawback of the depth-first search strategy used in the logic programming is that it is incomplete and may get stuck in a branch, and never consider other more fruitful branches in the proof tree. Therefore, the approach taken in [63] is to replace the incomplete and unfair depth-first proof search strategy of the logic programming interpreter with the fair bounded iterative deepening proof search. This means proof search will still be incomplete, but at least we can switch to other potentially more fruitful branches in the proof tree, once we have reached the bound. Although this seems to work well in many examples, it has several shortcomings: First, the user has to specify a depth bound. This has two consequences. If a proof does not exist within the given bound, then the user does not know whether the stated conjecture is unprovable or the bound was set to low. As a result, failure is not meaningful in this setting making the task of developing formal systems and proofs about them frustrating. Second, there may be redundancy in the proof search hampering the overall performance. Note that quick failure is essential in the overall design of the meta-theorem prover, to make progress and split the conjecture in different cases. Let us briefly consider the previous example for type preservation proof again. Before we can split the conjecture into different cases, we need to show that $\mathcal{F} : \Gamma \vdash v : a$ is not derivable from the current proof assumptions $\mathcal{D} : \Gamma \vdash m : a$ and $\mathcal{E} : m \hookrightarrow v$, inference rules and lemmas. One of the inversion lemmas and the subtyping rule are applicable and can be used in the first step. This can quickly lead to an explosion in the search space. In practice this means iterative deepening gets bogged down in the very first step and cannot make progress, thereby failing to establish an inductive proof for type preservation.

Third, iterative deepening search can be quite fragile. The order in which the inference rules are implemented matters a lot and proving lemmas and adding them to the database of known facts may have drastic effects on how well proof search performs. While it is reasonable to ask the user to carefully implement the typing rules, the user has little control over the use of lemmas and other proof assumptions.

In this chapter we propose memoization-based search as an alternative to iterative deepening. Memoization-based search terminates for programs over a finite domain. For those programs where memoization-based search may not terminate, the user still has to give a bound, within which the memoization-based search procedure will try to find a proof or establish that no proof exists. Experiments show that handling redundancy is critical to prove meta-theorems like the ones above leading to a more efficient and more powerful meta-theorem prover.

7.3 Generic proof search based on memoization

Memoization-based proof search is based on the tabled uniform proof system given in Chapter 4. Although the underlying ideas and large parts of the implementation are shared between tabled higher-order logic interpreter and the memoization-based higher-order proof search, there are a few key difference: 1) In logic programming, it is important to maintain an easy to understand and clearly defined operational semantics. The programmer can then exploit this semantics when writing programs. For this reason, the programmer may find an unfair but easy to understand strategy preferable in logic programming. In theorem proving, we want to know if a proof exists and an unfair search strategy is unacceptable, since it may prevent us from finding a proof. Since memoization-based proof search does not terminate in all cases, we cannot eliminate bounds completely. The search procedure will let the user know, if the search was not completed within this bound. However, as we will show in many cases the search is complete thereby providing valuable feedback for the user and quick failure. 2) In logic programming we have a fixed set of program clauses, over which the programmer has control. On the other hand in meta-theorem proving, we have additional proof assumptions such as induction hypotheses and lemmas, which are generated during proof search, and over which the programmer has no control. This makes the problem of search for a proof harder.

Overall, we have taken a very conservative approach of adding memoization to the meta-theorem prover. We only replace iterative deepening search procedure with the new memoization-based search. However after the memoization-based search fails or succeeds, we clear the memo-table. A more aggressive approach could be to have the memo-table survive across iterations. We mainly focus on using variance checking in the memoization-based search. The main reason has been to minimize the overhead of maintaining two very different implementations, one for the interpreter and one for the meta-theorem prover. However, we will show using proofs about refinement types that unlike higher-order logic programming the use of subsumption can be very valuable in meta-theorem proving.

We also discuss some of differences of iterative deepening search and memoization-based search. (1) Iterative deepening means, we first try to find a proof for depth 1 with depth-first search, then we try to find a proof of depth 2 with depth-first search etc. If the branch is deeper than k, then we fail and backtrack. This means if there exists a proof with depth k, we may have explored the left-most branches in the search tree up to depth k. Obviously, if proof search fails, then we have to explore the whole search space. (2) Memoization-based search bounded by n: we traverse the search space using depth-first search and memoization. if we explore a branch with depth greater than n we fail and backtrack. This means although we only needed depth k, where k is smaller than n, we explored fruitless branches up to depth n. As a result, it potentially may take longer to find solutions to some problems.

7.4 Experiments

We have carried out several experiments so far, including a refinement type system (soundness and completeness of algorithmic subtyping algorithm), type-system with subtyping (soundness and completeness of algorithmic subtyping and type preservation) and theorem proving in classical natural deduction. The meta-theorems were not previously provable with the iterative deepening proof search, thereby demonstrating that memoization-based search leads to a more powerful meta-theorem prover. In addition, we give some examples which were provable with iterative deepening with a bound, and can be proven without a bound using memoization-based search.

7.4.1 Type preservation for type system with subtyping

First, we consider the proofs about the type-system with subtyping given earlier. The proofs themselves are not hard, but the number of applicable lemmas increases the search space as we continue. Lemmas are turned into logic programming clauses which can be used during proof search. For example, the lemma

$$\text{If } \Gamma \vdash \mathsf{s}\ M : A \text{ and } A \trianglelefteq \mathsf{pos} \text{ then } \Gamma \vdash M : \mathsf{nat}.$$

will be turned into the clause

```
 invNat :    of M nat
             <- of (s M) A
             <- A <| pos
```

From a logic programming view, this clause may lead to an explosion in search space. Therefore we see that iterative deepening proof search deteriorates if we continue and prove more lemmas about the type-system. Figure 7.2 gives the run-time for proving all the lemmas and theorems needed in the type preservation proof. We did not include any speed-ups in this table since many numbers are very small and for those the speed-up is not very meaningful.

theorem	tabled	iterative deep
refl	0.18	0.00
trans	0.29	0.02
sound	0.19	0.01
comp	0.21	0.02
imposAll	0.04	0.00
inv+z+pos	0.08	0.00
inv+suc+pos	0.08	0.00
inv+suc+nat	0.09	0.23
inv+suc+zero	0.10	0.93
inv+lam	0.21	7.27
inv+pair	0.18	∞
tps	2.89	–

Figure 7.2: Type preservation for system with subtyping

The memoization-based meta-theorem prover has some overhead in managing the table which leads to slight deterioration in the first few examples. However, in the inversion lemma for functions memoization starts paying off and the proof for the inversion lemma for pairs cannot be found by iterative deepening search within a reasonable amount of time. The main benefits of memoization are not in quickly finding a proof for a sub-case in the proof, but rather in quickly failing, i.e. in showing no proof exists from the current proof assumptions, and therefore triggering a case split. The proof of the inversion lemma for functions for example has between 37 and 67 suspended goals. The proof of type preservation has between 31 and over 70 suspended goals. All these loops were detected when proving that it is not possible to derive the conclusion from the current set of assumptions. Hence, the overall performance improvement is mainly due to quick failure which allows the theorem prover to make progress. As a result, we succeed in proving all the lemmas including the final type preservation theorem using the memoization-based prover. The iterative deepening prover gets stuck during the proof for the inversion lemma for pairs. Even if we add the inversion lemma for pairs without a proof to the set of lemmas, the iterative deepening prover does not succeed in proving type preservation.

7.4.2 Derived rules of inference in classical natural deduction

Next, we show some experiments using classical natural deduction. Proof search in natural deduction calculi is considered hard. Memoization can lead to quicker proofs in some examples, while imposing a some performance penalty in others. In Table 7.3, we give the run-times for proving several derived rules. These examples do not require induction.

theorem	tabled search	depth-first-bounded	iterative deepening
split'	9.20	0.19	0.29
join'	9.18	1.68	1.69
assoc	20.14	450.53	451.49
uncurry	33.45	450.97	452.42

Figure 7.3: Derived rules of inference in NK (classical natural deduction with not, imp, and, double negation rule)

7.4.3 Refinement types

In this sections we used the memoization-based prover to proof all the lemmas leading up to the type preservation theorem. We use an implementation of the type system for refinement types by Pfenning [17]. For proving these statements, we used subsumption-based memoization. It is important to note, that this example clearly demonstrates the use of subsumption, since with variant-based memoization we still cannot find some proofs. Subsumption is critical to detect quickly that no proof exists and we need to split the conjecture into different cases. It also is critical to complete some of the sub-proofs[2].

Iterative deepening works fine for the first few proofs. In fact, its performance is better than memoization-based proof search. However, it cannot prove the inversion lemmas. Attempting the proof for the inversion lemma for functions, iterative deepening cannot find a proof within a reasonable amount of time. When skipping the inversion lemma for functions, we are able to make some more progress by proving the two impossibility lemma, but we get again stuck when proving the lemma imp+1+zero. Some of the lemmas, we are able to prove individually. However, this is unsatisfactory for developing the meta-proofs about the refinement types.

Using subsumption-based memoization, we detected over 100 loops when proving inversion lemmas. Similar to the subtyping example, memoization was most valuable for detecting that the current proof assumptions do not suffice to derive the conclusion.

7.4.4 Rewriting semantics

Next, we give two examples which were provable before but did require a bound. In fact, the user has to choose the minimal bound in order to get the meta-theorem prover to prove these theorems. With memoization, no bounds are required. This is especially important if we want to disprove conjectures and debug implementations and proofs about them. We first consider a rewriting semantic with weak congruence closure and show that any big-step evaluation has a sequence of rewriting steps in the reduction semantic (evalRed theorem). Next, we try to prove that any rewriting sequence has a corresponding big-step evaluation (RedEval theorem). However, this

[2] *run-times for variant-based memoization to be added*

theorem	tabled search (subsumption)	iterative deepening
lemma1	3.72	0.08
refl	0.21	0.00
trans	1.63	0.02
comp¡—	1.81	0.01
sound¡—	0.41	0.01
inv+lam	0.36	–
impossAll	0.020	–
impRV	0.140	–
imp+e+pos	0.220	–
imp+1+zero	0.260	–
imp+0+zero	0.28	–
inv+e+bits	0.080	–
inv+e+nat	0.08	–
inv+1+bits	0.450	–
inv+0+bits	0.500	–
inv+0+nat	0.520	–
inv+1+nat	0.510	–
inv+0+pos	0.510	–
inv+1+pos	0.490	–
conjLemma	0.110	–

Figure 7.4: Refinement type checking [17]

is in fact not true the way the rewrite rules are stated, since we not always will get back a value. Memoization-based search will return and say that no proof is possible. This in turn is valuable to analyze the specification and potentially change it. Finally, we include a type preservation theorem about the reduction semantics. Run-times for the memoization-based prover are slightly worse than the iterative deepening prover. However, we get a more expressive prover which doesn't require bounds.

theorem	tabled search	iterative deepening
evalRed (sound)	0.37sec	0.01sec
RedEval (complete)	9.88sec	–
typeRed	0.93sec	0.01sec

Figure 7.5: Proofs about the rewrite semantics vs big-step evaluation semantics

Similarly, for the proofs about Mini-ml we observe that the theorems fall into the complete fragment, and the theorem prover does not require any bounds. Again we include the proofs about the original specification and contrast it with slightly modified specification where the proofs about type preservation and value soundness fail.

theorem	tabled search	iterative deepening
val-sound	0.33sec	0.32sec
tp-preseve	0.53sec	0.01sec
tp-preserve (mod)	1.18sec	–

Figure 7.6: Proofs about big-step evaluation semantics, type preservation, value soundness, mini-ml

7.5 Conclusion

Redundancy elimination is critical to reason with and about non-trivial deductive systems. We have shown that memoization-based search can be smoothly integrated into the meta-theorem prover. This allows us to prove more complicated theorems which rely on several lemmas. From our experience, the better failure behavior is particularly useful in developing and debugging specifications and theorems. In addition, it leads to a substantial improvement of the overall performance of the meta-theorem prover. As we are able to fail quicker, we are able to make progress faster.

However, the conservative memoization-based search also has some limitations, which we plan to address in the future. In particular the proofs about refinement types illustrate that subsumption based memoization is useful in theorem proving. As we mentioned earlier, we have only an incomplete subsumption-based memoization strategy in the current implementation, to minimize the effort required to maintain different implementations for variant- and subsumption-based memoization. In the long run, it seems interesting to develop a complete subsumption-based memoization proof search procedure for the meta-theorem prover.

Chapter 8

Conclusion

In this thesis, we develop different techniques to efficiently execute and reason with and about deductive systems. We have shown that memoization as a form of redundancy elimination is a useful extension to the higher-order logic programing interpreter. It allows us to execute more examples and have better and more meaningful failure. It also plays an important role in the meta-theorem prover. To achieve good performance of memoization-based search, we have developed higher-order term indexing techniques and an optimized higher-order pattern unification algorithm, which eliminates many unnecessary occurs checks. Taken together the presented techniques constitute a significant step toward exploring the full potential of logical frameworks in real-world applications where deductive systems and proofs about them may be more complex.

Future Work

Applications: Typed Assembly Language

Recently, Twelf has been used in developing a foundational approach for typed assembly language [14, 15]. This approach is similar to the foundational proof-carrying code project [2] in the sense that we try to minimize the trusted computing base. In particular, we do not need to trust the safety policy, but the safety policy is proven sound. While in the foundational proof-carrying code approach higher-order logic together with some axioms about arithmetic is used as a foundation to define safety policies, the logical framework LF itself serves as the foundation for implementing different safety policies. In addition to the safety policy, described by a type system, all the meta-proofs required to prove soundness are implemented in Twelf. However, so far Twelf has been mainly used as a specification framework and not fully utilized the expressiveness and power of execution and reasoning strategies. There are many applications of the described work to the foundational typed assembly language project. First, it seems interesting to experiment and generate safety certificates for sample programs using Twelf's logic programming interpreter. Second, it would be interesting to investigate if some of the meta-theorems about the safety policy can be proven automatically using the memoization-based meta-theorem prover. Third, we can instrument Twelf's higher-order logic programming interpreter to produce a bit-string that encodes the non-deterministic choices in a proof and use memoization to factor out common subproofs. This extends prior work by Necula and Rahul [42] to full LF. Moreover, preliminary experiments demonstrate that memoization can lead to bit-string which are up to 40% smaller in size than their counterparts produced without memoization.

Eliminating further redundant typing information

There is still some redundancy in proof search which is due to dependent types. Dependent type information needs to be carried around during proof search imposing a substantial overhead [33]. Necula and Lee [43] have investigated the problem of redundant type information in the context of type-checking. They propose a more efficient term representation for a fragment of LF which eliminates a lot of the redundant typing information thereby minimizing the amount of information we need to keep for type-checking. One direction is to extend Necula and Lee's work to full LF type theory, thereby eliminating further redundancy. Moreover, it may be used to not only improve the performance of type-checking in Twelf, but also of proof search itself.

Completeness of memoization-based proof search

In this thesis, we have mainly focused on proving soundness of memoization-based proof search. Completeness of memoization-based proof search is harder to establish. The main reason is that we will find fewer proofs than with uniform proofs. For example in the subtyping example given in Sec. 4.1 the query sub zero zero has infinitely many proofs under the traditional logic programming interpretation while we find only one proof under the tabled logic programming interpretation. However, we often do not want or need to distinguish between different proofs for a formula A, but only care about the existence of a proof for A together with a proof term. In [50] Pfenning develops a dependent type theory for proof irrelevance and discusses potential applications in the logical framework. This could allow us to treat all proofs for A as equal if they produce the same answer substitution. In this setting, it seems possible to show that search based on tabled uniform proofs is complete, i.e. if there is a (canonical) proof $M : A$ then we can find A.

Optimizations for memoization-based search

Re-use of answers

Memoization-based search is critically influenced by when we retrieve answers and when we suspend goals and when and in what order we awaken suspended goals. Currently, we have implemented multi-stage depth-first strategy. As we have discussed in Chapter 4, this strategy might delay the re-use of answers, since we are not allowed to retrieve answers from the current stage. The XSB system therefore uses a strategy based on strongly connected components, which allows us to consume answers as soon as they are available. We build a dependency graph of the predicates and identify different independent sub-graphs, called strongly connected components. This dependency analysis separates subgoals that can and cannot influence each other. If computation for all the subgoals within a sub-graph is saturated, then we can dispose all the suspended sub-goal belonging to this component. As we are able to delete suspended nodes that cannot contribute to new solutions anymore, this leads to fewer suspended nodes and has the potential to be more space and time efficient. This might be advantageous especially in theorem proving, where we only care about one answer to the query and are not interested in mimicking Prolog execution. In LF, subordination analysis provides information about dependencies of predicates and constructs a subordination graph. Analyzing and exploiting the subordination information, it seems possible to design similar strategies as in XSB which allow the re-use of answers as soon as they are available and detect when all answers have been generated.

Subsumption

As some of the experiments in Chapter 7 suggest, the use of subsumption-based memoization in proof search is useful in meta-theorem proving. As we have also mentioned in Chapter 7, so far the implementation of subsumption is a conservative extension of the variant-based memoization proof search and may be incomplete. In the future, it seems fruitful to develop a complete subsumption-based proof search procedure for the meta-theorem prover.

High-level optimizations to proof search

Although memoization-based computation aims to make reasoning *within* logical specification efficient, it cannot rival the performance of a theorem prover that is specifically built for a given logic. One reason is that specialized state-of-the-art theorem provers exploit properties of the theory they are built for. For example, inverse method theorem provers for first-order logic like Gandalf [66] exploit the subformula property. Other theorem provers like Spass [69], which are very successful in equational reasoning, rely on orderings to restrict the search. These meta-level optimizations can improve performance of higher-order logic programming dramatically. Therefore, one interesting path to explore is to verify such properties about logical specifications in advance and exploit them during search.

Modal dependent type theory

In Chapter 2, we have conservatively extended the LF type theory with modal variables and shown that canonical forms exist and type-checking remains decidable. As we have briefly discussed in Chapter 2, there are two simple and clean ways to incorporate such variables. One is via a general modal operator $\Box_\Psi$, and the other is via a new quantifier $\Pi^\Box u{::}(\Psi{\vdash}A_1).\ A_2$. The main complication with the general modal operator is that canonical forms do not exist. In this thesis, we suggested the use of $\Pi^\Box$ to quantify over modal variables explicitely, but we have not fully extended the LF type theory. In the future, it seems interesting to develop the theory further towards a modal dependent type theory. The $\Pi^{\Box}$-abstraction and the $\Box$-application is less powerful than the general box-operator. We conjecture that canonical forms exist for this fragment and type-checking remains decidable.

In Chapter 3, we have briefly mentioned an extension which allows us to quantify directly over variable definitions using $\Pi^{\Box}u = M{::}(\Psi{\vdash}A)$. This would have two main advantages: First, we would be able to type-check linearized objects L directly. This avoids translating variable definitions into a modal substitution θ and the type-checking $[\![\theta]\!]L$. Second, it would allow us to describe proof terms as directed-acyclic graph and type-check them directly.

Linear logic programming

Linear logic programming [30, 8] has been proposed as an extension of higher-order logic programming to model imperative state changes in a declarative (logical) way. We believe that the some of the techniques presented in this thesis can be extended to linear logic programming, but it requires some new considerations. In particular, we plan to investigate the use of memoization in linear higher-order logic programming. This requires new consdiderations concerning the interaction between resource management strategies [10] or constraints [27] with tabling. However, the approach presented seems general and amenable also to the linear case.

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[2] Andrew Appel. Foundational proof-carrying code. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 247–256. IEEE Computer Society Press, June 2001. Invited Talk.

[3] W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, Jan. 2000.

[4] Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. In *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Denmark, July 2002.

[5] Egon Börger and Dean Rosenzweig. The WAM – definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Proactical Applications*, pages 1–71. Elsevier Science Publishers B.V., 1994.

[6] Pascal Brisset and Olivier Ridoux. Naive reverse can be linear. In Koichi Furukawa, editor, *International Conference on Logic Programming*, pages 857–870, Paris, France, June 1991. MIT Press.

[7] M. Carlson. On implementing Prolog in functional programming. *New Generation Computin*, 2(4):347–357, 1984.

[8] Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996.

[9] Iliano Cervesato. Proof-theoretic foundation of compilation in logic programming languages. In J. Jaffar, editor, *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming — JICSLP'98*, pages 115–129, Manchester, UK, 16–19 June 1998. MIT Press.

[10] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000.

[11] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Department of Computer Science, Carnegie Mellon University, April 1997.

[12] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[13] Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[14] Karl Crary. Toward a foundational typed assembly language. In *30th ACM Symposiumn on Principles of Programming Languages (POPL)*, pages 198–212, New Orleans, Louisisana, January 2003. ACM-Press.

[15] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *19th International Conference on Automated Deduction*, Miami, Florida, USA, 2003. Extended version published as CMU technical report CMU-CS-03-108.

[16] B. Cui, Y. Dong, X. Du, K. N. Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roychoudhury, S.A. Smolka, and D.S. Warren. Logic programming and model checking. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *International Symposium on Programming Language Implementation and Logic Programming (PLILP'98)*, volume 1490 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1998.

[17] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the International Conference on Functional Programming (ICFP 2000), Montreal, Canada*, pages 198–208. ACM Press, 2000.

[18] Steve Dawson, C. R. Ramakrishnan, Steve Skiena, and Terrance Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, 18(6):528–563, 1995.

[19] Bart Demoen and Konstantinos Sagonas. CAT: The copying approach to tabling. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *International Symposium on Programming Language Implementation and Logic Programming (PLILP'98)*, Lecture Notes in Computer Science (LNCS), vol. 1490, pages 21–36, 1998.

[20] Bart Demoen and Konstantinos Sagonas. CAT: The copying approach to tabling. *Journal of Functional and Logic Programming*, 2:1–38, 1999.

[21] Bart Demoen and Konstantinos Sagonas. CHAT: The copy-hybrid approach to tabling. In *The First International Workshop on Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science (LNCS), vol. 1551, pages 106–121, San Antonio, January 1999. Springer.

[22] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

[23] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[24] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[25] Peter Graf. Substitution tree indexing. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications, Kaiserslautern, Germany,*

Lecture Notes in Computer Science (LNCS) 914, pages 117–131. Springer-Verlag, 1995.

[26] Michael Hanus and Christian Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[27] James Harland and David Pym. Resource-distribution via boolean constraints. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, July 1997. Springer-Verlag LNAI 1249.

[28] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[29] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 2003. To appear. Preliminary version available as Technical Report CMU-CS-00-148.

[30] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[31] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[32] Lars Klein. Indexing für Terme höherer Stufe. Diplomarbeit, FB 14, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[33] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[34] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.

[35] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[36] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[37] Gopalan Nadathur. A treatment of higher-order features in logic programming. Technical Report draft, available upon request, Department of Computer Science and Engineering, University of Minnesota, January 2003.

[38] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.

[39] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[40] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus – a compiler and abstract machine based implementation of Lambda Prolog. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 287–291, Trento, Italy, July 1999. Springer-Verlag LNCS.

[41] Aleksander Nanevski, Brigitte Pientka, and Frank Pfenning. A modal foundation for meta-variables. In *2nd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with variable binding (Merlin), Uppsala, Sweden*, Electronic Notes in Theoretical Computer Science (ENTCS), 2003.

[42] G. Necula and S. Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 142–154, 2001.

[43] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In Vaughan Pratt, editor, *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, June 1998. IEEE Computer Society Press.

[44] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, pages 61–91. Springer-Verlag LNCS 1419, August 1998.

[45] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[46] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, pages 772–773, Argonne, Illinois, 1988. Springer Verlag Lecture Notes in Computer Science (LNCS) 310. System abstract.

[47] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[48] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[49] Frank Pfenning. Computation and deduction, 1997.

[50] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[51] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[52] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[53] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI) 1632.

[54] Brigitte Pientka. Memoization-based proof search in LF: an experimental evaluation of a prototype. In *Third International Workshop on Logical Frameworks and Meta-Languages (LFM'02), Copenhagen, Denmark*, Electronic Notes in Theoretical Computer Science (ENTCS), 2002.

[55] Brigitte Pientka. A proof-theoretic foundation for tabled higher-order logic programming. In P. Stuckey, editor, *18th International Conference on Logic Programming, Copenhagen, Denmark*, Lecture Notes in Computer Science (LNCS), 2401, pages 271 –286. Springer-Verlag, 2002.

[56] Brigitte Pientka. Higher-order substitution tree indexing. In C. Palamidessi, editor, *19th International Conference on Logic Programming, Mumbai, India*, Lecture Notes in Computer Science (LNCS), to appear. Springer-Verlag, 2003.

[57] Brigitte Pientka and Frank Pfennning. Optimizing higher-order pattern unification. In F. Baader, editor, *19th International Conference on Automated Deduction, Miami, USA*, Lecture Notes in Artificial Intelligence (LNAI) 2741, pages 473–487. Springer-Verlag, July 2003.

[58] Jeff Polakow and Frank Pfenning. Ordered linear logic programming. Technical Report CMU-CS-98-183, Department of Computer Science, Carnegie Mellon University, December 1998.

[59] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, 38(1):31–54, Jan 1999.

[60] I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, pages 1853–1962. Elsevier Science Publishers B.V., 2001.

[61] Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *International Conference on Principles and Practice of Declarative Programming(PPDP'00)*, pages 178–189, 2000.

[62] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[63] Carsten Schürmann. *Automating the meta theory of deductive systems.* PhD thesis, Department of Computer Sciences, Carnegie Mellon University, Available as Technical Report CMU-CS-00-146, 2000.

[64] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag Lecture Notes in Computer Science (LNCS) 1421.

[65] H. Tamaki and T. Sato. OLD resolution with tabulation. In E. .Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 1986.

[66] T. Tammet. A resolution theorem prover for intuitionistic logic. In *Proceedings of the 13th International Conference on Automated Deduction, New Brunswick, NJ, July 1996*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 2–16, 1996.

[67] Roberto Virga. *Higher-Order Rewriting with Dependent Types.* PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, Available as Technical Report CMU-CS-99-167, Sep 1999.

[68] David S. Warren. *Programming in tabled logic programming.* draft available from http://www.cs.sunysb.edu/w̃arren/xsbbook/book.html, 1999.

[69] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001.