Automating the Meta Theory of Deductive Systems

Carsten Schürmann
October 16, 2000
CMU-CS-00-146

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Thesis Committee:
Frank Pfenning, Chair
Robert Harper
Peter Lee
Dana Scott
Natarajan Shankar, SRI International

This research was sponsored by the Defense Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", DARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. This research was also sponsored by the National Science Foundation under grant CCR-9619584.

Abstract

This thesis describes the design of a meta-logical framework that supports the representation and verification of deductive systems, its implementation as an automated theorem prover, and experimental results related to the areas of programming languages, type theory, and logics.
Design: The meta-logical framework extends the logical framework LF [HHP93] by a meta-logic $\mathcal{M}_2^+$. This design is novel and unique since it allows higher-order encodings of deductive systems and induction principles to coexist. On the one hand, higher-order representation techniques lead to concise and direct encodings of programming languages and logic calculi. Inductive definitions on the other hand allow the formalization of properties about deductive systems, such as the proof that an operational semantics preserves types or the proof that a logic is consistent. $\mathcal{M}_2^+$ is a proof calculus whose proof terms are recursive functions that may be defined by cases and range over dependent higher-order types. The soundness of $\mathcal{M}_2^+$ follows from a realizability interpretation of proof terms as total recursive functions.
Implementation: A proof search algorithm for proof terms in $\mathcal{M}_2^+$ is implemented in the meta-theorem prover that is part of the Twelf system [PS99b]. Its takes full advantage of higher-order encodings while using inductive reasoning.
Experiments: Twelf has been used for many experiments. Among others, it proved automatically the Church-Rosser theorem for the simply-typed λ-calculus and the cut-elimination theorem for intuitionistic first-order logic. In programming languages, it proved various type preservation theorems for different operational semantics and compiler correctness theorems. In logics, it was able to derive the equivalence of various logic calculi, such as the natural deduction calculus, the sequent calculus, and the Hilbert calculus. Twelf also proved that Cartesian closed categories can be embedded into the simply-typed λ-calculus. In the special domains of programming languages, type theory, and logics, Twelf's reasoning power far exceeds that of any other theorem prover.

Keywords: meta logic, LF, induction, regular world assumption, realizability, higher-order abstract syntax, Twelf

Contents

List of Figures

Acknowledgments

Many people have contributed to the success of this thesis, and I am very grateful to all them. First and foremost, I would like to thank my advisor Frank Pfenning who has introduced me to the wonderful and elegant world of logical frameworks in his *Computation and Deduction* class in Spring 1994.

It was this class, that sparked my interest to work on meta-logical frameworks for LF. From a student's perspective, many of the theorems presented in the class looked rather complicated, yet their proofs were so elegant and used only very few proof techniques, that it seemed plausible to try to automate their derivations. Looking back on the class now, we are very happy to report that all but a few theorems can be automatically derived using Twelf.

Thanks, Frank, for your guidance, for your vision and insight, for your advice, and for the many discussions we have had over the previous years. I learned a lot from you. Without your experience on prior implementations of the Elf and the Ergo system, Twelf's core would not be as elegant as it is today.

Second, I would like to thank Peter Lee, Robert Harper, Dana Scott, and Natarajan Shankar for being on my thesis committee and for providing me with lots of suggestions, advice, and ideas. In particular, I want to express my thanks to Peter Lee and George Necula, who succeeded with their work on proof carrying code to export logical framework technology to other communities, Robert Harper who used the Twelf system to verify properties about abstract machines, Dana Scott, who was happy to discuss different aspects of the system with me, and Natarajan Shankar who demonstrated that automated theorem proving technology is useful and important for the real-world.

Third, I would like to thank Wilfried Sieg and Teddy Seinfeld for admitting me the Pure and Applied Logic Program at Carnegie Mellon University. Thanks also to Steve Brookes, John Reynolds, and Edmund Clarke for many great ideas.

Special thanks go to all my friends and fellow students for all the countless discussions, for your support and for the great time: Doug Baker, Andrej Bauer, Christoph Benzmüller, Lars Birkedal, Matthew Bishop, Michael Bowling, Iliano Cervesato, Perry Cheng, Karl Crary, Rowan Davis, Herb Derby, Jürgen Dingel, Armin Fiedler, Jesse Hughes, Somesh Jha, Will Marrero, Dominic Mazzoni, Raymond McDowell, Alberto Momigliano, Ljubomir Perkovic, Jeff Polakow, Mark Plesko, Brigitte Pientka, Ekkehard Rohwedder, Dario Salvucci, Dirk Schlimm, Aaron Stump, Peter Venable, Roberto Virga, Kevin Watkins, Hao-Chi Wong, and Hongwei Xi.

My time in Pittsburgh would have not been what it was without my dear friend Molly Bigelow, who believed in me throughout the years, and who waited patiently for me to finish. I want to thank her for all her love, patience, guidance, and simply for being her. I also feel very much in debt to my parents and my brother who supported me over all these years, and for their strength to stand me being so far away for so long.

Chapter 1

Introduction

> *We can look at the current field of problem solving by computers as a series of ideas about how to present a problem. If a problem can be cast into one of these representations in a natural way, then it is possible to manipulate it and stand some chance of solving it.* [Allen Newell]

It is common knowledge that it is very difficult to design software systems that work flawlessly and reliably. Most software products contain defects, some of them are harmless others might be potentially harmful. From experiences in programming language research we have learned that many software defects can be avoided by using appropriate programming languages. For example, strongly typed languages like Standard ML of New Jersey [MTHM97], or Haskell [Tho99] guarantee by design that a program can never cause a segmentation fault and crash.

Also Java [LY96] is designed with a strong type system. Following from properties of the Java language, the execution of a Java program theoretically never crashes. In fact, the Java bytecode verifier that is part of the Java distribution statically examines byte code for memory and type violations and rejects suspicious bytecode. But can we trust the byte code verifier? Certainly not, since its semantics is specified only informally and in plain English, which renders convincing formal proofs of any safety guarantees impossible.

Consequently, a rigorous formalization of the programming language and its semantics is necessary in order to reason about it and to convince others about the soundness of a design. A sound design of ML for example guarantees that the execution of a program of given type never returns a result that is of another type. It should also guarantee that the algorithm that computes the type of a program — the type-inference algorithm — always terminates and always return the correct results: the principle type if it exists or failure otherwise.

Therefore, in order to reason about programming languages we must rely on rigorous formalizations of their syntax operational semantics. Formulations of this kind have been developed for example for ML [MTHM97, HS97], and for subsets of Java [SA98, NvO98] but rigorous arguments about these formalizations are very difficult to do. Answers to questions such as "Is Java type safe?" or "Is ML type-checking decidable?" are tedious arguments, and they must consider typically so many cases that they are not easily verifiable by humans. This thesis is about tools to represent and reason about programming languages.

Another motivating example comes from the area of authentication protocol design. Using the Needham-Schroeder protocol [NS78] two corresponding parties can authenticate, but unfortunately the protocol is flawed. Lowe [Low96] has shown that it can be attacked by an intruder making his victim believe that he is somebody else. Is it possible to catch design flaws like this during the design process? It is, by using techniques such as model checking [MCJ97] or inductive theorem proving [Pau98].

A few decades ago the importance of automated reasoning has led researchers to develop systems that provide formal support for everyday tasks of mathematicians programming language designers. The first major breakthrough, for example, was possibly a computer assistant proof of the four color theorem [AH77a, AH77b]: Every planar graph is colorable by four colors in such a way that regions sharing a common boundary do not share the same color.

Historically speaking, one of the first general purpose theorem provers including induction is Nqthm system [BM79, BM88] that has been used to prove a tremendous variety of different results many directly relevant to programming language research. Shankar [Sha94], for example, has used Nqthm to check a proof of the Church-Rosser theorem for the untyped λ-calculus holds, and he has also verified Gödel's incompleteness theorem. Another example goes back to Kunen who formalized the proof Ramsey's theorem [Kun95] in Nqthm.

Following the lead of Nqthm, many other theorem provers have evolved based on different logics and different automated deduction algorithms with different strengths and weaknesses. Otter [McC94] for example has been used to show that all Robbins algebras are Boolean [McC97] as conjectured in 1933.

First-order automated theorem provers could be applicable to our domain of reasoning about programming languages. However, they are not appropriate for representing programming languages such as ML or Java since they do not provide inductive definitions. But there are others: INKA [HS96] for example is a theorem prover that can handle induction and so are many proof assistants that are based on type theory, such as for example Isabelle [Pau94], Coq [DFH$^+$93], NuPRL [C$^+$86], and Lego [LP92].

Isabelle is a very popular proof assistant and has been used, for example, to reason about programming languages such as Milner's type inference algorithm [NN99] and the operational semantics of a simple programming language [AC99]. It has also been used to reason about program refinement languages bases on an embedding of weakest preconditions [Sta99].

Similar experiments in the area of programming languages have been conducted with the Coq system. In functional programming the type inference algorithm of ML has been verified in Coq [CD99], and in logic programming the algorithm of SLD resolution [Jau99]. These experiments are not small, on the contrary in the case of the formalization of SLD resolution, approximately 600 technical lemmas were necessary in the entire development.

The Ensemble project [KHH98] is concerned with the development of reliable and efficient group communication systems. In order to execute and verify program transformations they have linked it to the NuPRL system.

For the purpose of machine developed and machine checked domain theory and program verification, Reus has implemented synthetic domain theory [Reu99] a constructive variant of domain theory in Lego. Other examples conducted with Lego include the formalization of type theories and λ-calculi [MP99], and a formalization of the strong normalization proof for system F [Alt93].

The recurring pattern in all these experiments is the following. The programming language that should be proven sound must be encoded into the language the theorem prover or the prover assistant provides. For the theorem provers mentioned above this language is either a quantifier free, a first-order, or a higher-order logic. Generally, the weaker the language, the more indirect the encoding. On the one hand, inductive definitions and higher-order features allow very direct encodings of programming languages. Constructs such as expressions, types, and inference rules are typically inductively defined and higher-order representation techniques [HHP87, PE88] allow direct encodings of variables and substitution concepts that are part of any programming languages.

Thus in general, the expressiveness of a representation language is crucial for the attempt to reason formally about programming languages. Reasoning about programming languages can only be as effective as the encoding is — or to put the other way around: the more direct the encoding is, the easier it is to reason about them.

Unfortunately, higher-order encodings and inductive definitions are incompatible since higher-order encodings violate the positivity condition associated with inductive definitions [DPS97]. Various attempts have been made to preserve the advantages of higher-order representation techniques in a setting with strong induction principles [DH94, DFH95], but none of these is entirely satisfactory from a practical or theoretical point of view. A first clean approach towards a solution of this problem was the modal λ-calculus [DPS97] that has been extended to dependent types [DL98]. A more recent proposal is due to Gabbay and Pitts [GP99], and Hofmann has given a categorical semantics for relating higher-order abstract syntax and induction principles [Hof99].

In this thesis, we describe a tool that provides higher-order representation techniques and inductive definitions. It is a meta-logical framework and it is implemented in the Twelf system [PS99b] and based on the logical framework LF [HHP93]. We discuss its design, its implementation, and demonstrate how to apply it to problems from the field of programming languages and logics. The Twelf system provides a special purpose theorem prover that draws its deductive power from the elegance of encoding.

Design of the Meta-Logical Framework

The design of a meta-logical framework can be best motivated by an informal example. Consider a developer who engineers safety architectures for mobile code such as proof carrying code [Nec97] or typed assembly language [MWCG99]. The basic idea underlying safety architectures is that a "code producer" augments mobile code with explicit safety proof objects that adhere to an a-priori specified safety policy. The code and the safety proof are then transmitted together through the network to a "code consumer". Once received, the code consumer examines the code and extracts independently verification conditions which it then verifies using the safety proof. If the proof checker signals success, the code can be trusted with respect to the safety policy, and the code consumer can execute it safely. Among the many challenges in devising a safety architecture is the design of a sound safety proof languages such as for example a logic or a type system.

Without any machine support the developer has to engineer the safety proof language by hand and verify its soundness using only pencil and paper. In general, this is a tedious, difficult, intricate, and error prone process. Slight changes in the design of the safety proof language can render months of hard work useless, leaving the developer without any other option but to revisit all the proofs again. With the technology presented in this thesis, the developer can formalize the safety proof languages such as logics and type systems, and reason about them automatically and effectively. In many of the examples discussed in this thesis, the system was able to check quickly if changes or extensions to a logic invalidate any of the desired properties.

Our meta-logical framework uses as representation language the logical framework LF [HHP93]. It is a higher-order type theory which provides dependent types and higher-order representation techniques. Judgments are formally represented as types and derivations as objects. Logics such as the sequent calculi and the natural deduction calculi [Gen35] can be directly encoded in the LF, taking full advantage of higher-order constructions. They directly support common concepts such as variable binding, capture-avoiding substitution, weakening, contraction, and exchange. For classical and intuitionistic logic, the representations are adequate which means that objects in the type theory are in one-to-one correspondence with derivations in a logic.

There are other logical frameworks, which are based on inductive definitions. To a large extent they are implemented in the aforementioned proof assistants such as Coq, Isabelle, Lego, or Nuprl. Inductive definitions rely on the *positivity* condition that guarantees the set of constructors for each datatype to be fixed. From a modal theoretic point of view, we say that the world in which a datatype is defined is *closed*, because datatypes must not be extended by new constructors. Synonymously, we say that a *closed world assumption* is precondition for standard inductive definitions.

In general, higher-order representation techniques violate the positivity condition, in particular deductive systems, which are of particular interest to this work: encodings of programming languages and logics, for example, possess very elegant higher-order encodings that cannot be expressed inductively. On the other hand, without higher-order representations, the developer is obliged to declare the variables, substitutions, and contexts and to reason about their respective properties, such as, for example, weakening, contraction, exchange, and substitution lemmas.

Nevertheless, one can reason about any object in LF (if functional or not functional) by induction. The proof of adequacy of any representation, for example, is based on an inductive argument over the structure of objects in LF. It is sound, because any object — including functional objects — possesses a canonical form, and canonical forms in LF are inductively defined [HP99]. Intuitively, the conversion of an object to a canonical form simply corresponds to the execution of substitution operations.

Intrinsically, inductive definitions are closely related to function definition by cases. Any proof of a property using standard induction principles can be realized as a total function that expects input arguments in place of universal quantifiers and that computes witness objects in place of existential quantifiers. These functions are defined by cases, and totality is established as an external property of the function. Termination follows from comparing argument vectors of recursive recalls to the argument vector the function was originally called with; they must decrease according to a well-founded (terminating) ordering. And coverage relies on the closed world assumptions; in every situation there are only finitely many cases to consider. Functions defined by cases should not be confused with the notion of function provided by the logical framework LF, which by construction cannot be defined by cases since they typically do not possess canonical forms in LF [DPS97].

Therefore in this thesis, we propose to use *two* inherently different function spaces. The first function space is *parametric* and it serves the purpose of adequate higher-order representation of deductive systems with implicit treatment of variables and capture avoiding substitutions. For the purpose of this work we have chosen the function space provided by LF since it satisfies all requirements and supports adequate encodings. But in general, it is conceivable to extend this work to other parametric function spaces defined in other logical frameworks, such as for example the linear logical framework [CP96], or the calculus of constructions [CH88].

Second, we propose a *recursive* function space that encodes proofs or properties about deductive systems, such as the soundness of a logic, or the Church-Rosser property of the simply-typed λ-calculus. These functions range over LF objects of arbitrary (possibly functional) type and can be defined by cases and recursion. The corresponding type theory, which is developed and presented in this thesis, is called $\mathcal{M}_2^+$. When restricted to total functions, the type theory $\mathcal{M}_2^+$ can be viewed as a meta-logic. Theorems are encoded as types in $\mathcal{M}_2^+$, and proofs as total functions called *realizers*.

The argument that a natural deduction representation of first-order intuitionistic logic is equivalent to a sequent formulation makes use of both function spaces. The parametric function space is used to represent the either of the two calculi whereas the recursive function space is used to express that any derivation in one calculus can be converted into a derivation in the other. Thus, from a programming point of view, $\mathcal{M}_2^+$ can be seen as the type system of a functional programming language that uses LF as language to express datatypes.

If deductive systems are encoded via higher-order functions, $\mathcal{M}_2^+$-proof terms may need to traverse λ-binders in order to make a recursive call and each traversal of a λ-binder corresponds to the introduction of a new parameter. Intuitively, these parameters can be viewed as dynamic extensions of the set of constructors of its type. Consequently, during runtime the set of constructors of any type is not fixed any more, which invalidates the closed world assumption. In contrast, in our setting, inductive definitions are open-ended because recursive functions may dynamically introduce new parameters as constructors. Therefore, inductive definitions are not adequate for higher-order encodings.

On the other hand, the *open world assumption* that allows open-ended definitions of datatypes does not present an appropriate foundation for the calculus of total functions we aim to design in this thesis. On the contrary! Under the open world assumption it is impossible to predict the canonical form of any LF-object. Therefore, the open world assumption cannot give any guarantees if a recursive function covers all cases! From a modal point of view, it is possible to argue that a recursive function covers all cases in *some* given world — but it is impossible to argue that a recursive function covers all cases in *any* given world.

$\mathcal{M}_2^+$'s design is based on the following observation: In general, during runtime, recursive functions follow always a few, but finitely many different patterns when traversing λ-binders before executing a recursive call. Therefore, datatypes are always extended in a regular and predictable fashion, in contrast to arbitrary extensions associated with the open world assumption. It is this regularity condition that allows us to judge if a recursive functions over open terms covers all cases. In this thesis we generalize the closed world assumption and simultaneously restricted the open world assumption.

The result is the *regular world assumption* which allows datatypes to be open ended but requires its extensions to be regular in structure. It enables us to reason about $\mathcal{M}_2^+$ proof terms and to determine if they cover all cases. Each proof term is augmented with a description of the world it is defined in, which ensures that only recursive functions defined in compatible regular worlds can call each other.

Returning to the example, an $\mathcal{M}_2^+$-proof term that maps first-order natural deduction derivations to first-order sequent derivations has to recurse on open subformulas of universal formulas. In the case of a higher-order encoding of terms, each traversal of the λ-binder that represents a bound variable extends the set of constructors. Clearly, those extensions are regular.

Under the regular world assumption, $\mathcal{M}_2^+$ is a type theory of partial functions that ranges over higher-order and dependently-typed LF objects. That $\mathcal{M}_2^+$ is also a sound logic to reason about deductive systems is one of the main contributions of this thesis. Proof terms of $\mathcal{M}_2^+$ are recursive functions witnessing the provability of (meta-)theorems about deductive systems. For this interpretation to hold, proof terms must be realizers, i.e. they are total recursive functions, that make always progress and terminate eventually on every input.

More precisely, progress is given if case analysis covers all cases, a property that follows from techniques similar to definitional reflection [SH93a]. Termination on the other hand follows if a measure associated with each on recursive calls decreases every time a call is executed [RP96]. Under these restrictions all proof terms of $\mathcal{M}_2^+$ are realizers and therefore $\mathcal{M}_2^+$ as a meta-logic is sound.

As consequence for the logic example, any total function in $\mathcal{M}_2^+$ that maps *any* natural deduction derivation to *some* sequent derivation is a proof of the soundness of the embedding, and vice versa, any total function in $\mathcal{M}_2^+$ that maps *any* sequent derivation to *some* natural deduction derivation realizes the completeness proof.

A similar approach toward the design of a meta-logic has been taken by Miller and McDowell [MM97] with their system $FO\lambda^{\Delta I\!N}$. $FO\lambda^{\Delta I\!N}$ is a meta-logic based on an intuitionistic first order logic extended by natural number induction and inductive definitions [SH93b]. It supports the representation of various logical frameworks, for example the intuitionistic and linear framework of hereditary Harrop formulas [McD97]. The embedded logical frameworks are used to represent deductive systems. Different from the soundness argument presented in this thesis, the soundness of $FO\lambda^{\Delta I\!N}$ follows by a cut-elimination argument [MM00].

From a purely logical perspective, $\mathcal{M}_2^+$ is weak, since the only connectives defined for it are universal, existential quantifiers, conjunction and truth. In addition it is restricted to conjunctions of Π_2-formulas, i.e. formulas that consist of a block of universal followed by a block of existential quantifiers. There are no propositional constants and it does neither provide implication nor disjunction nor negation nor equality. Nevertheless $\mathcal{M}_2^+$ draws its representational power from the underlying logical framework LF.

Because of the expressive strength of higher-order representation principles proofs in $\mathcal{M}_2^+$ are very efficient. For example substitution, weakening, strengthening, and exchange lemmas are implicitly provided by LF, and therefore they do not have to be proven explicitly. This is a tremendous win compared to systems that cannot use higher-order encodings due to the positivity condition. Therefore, proofs in $\mathcal{M}_2^+$ are in general shorter, more concise and more elegant.

Implementation of the Meta-Logical Framework

The logical framework LF and the meta-logic $\mathcal{M}_2^+$ are implemented in the Twelf system [PS99b]. In addition, we have implemented two proof search algorithms: one algorithm searches for LF objects of given LF type, and the other search for proof terms in $\mathcal{M}_2^+$. Because of the judgments-as-types and the derivations-as-objects paradigm, the LF-theorem prover is logic independent.

As opposed to traditional general purpose theorem provers which are designed to search for derivations in a particular deductive system, such as for example first-order logic with or without equality, Twelf's $\mathcal{M}_2^+$-theorem prover is considered to be special purpose theorem prover. It is designed to reason about deductive systems in general, and logics and programming languages in particular. In its current version, Twelf is designed to be mostly automatic. In particular, it does not provide any mechanisms for user-specified tactics or tacticals. Neither does it employ any form of rewriting. For each theorem we only specify a sequence of lemmas, the induction variables, and an upper bound for search. The proof is completely automatic in every other respect.

The Twelf system is entirely written in Standard ML. The latest version is available through the Twelf homepage `http://www.twelf.org`.

Application of the Meta-Logical Framework

The technology presented in thesis can be used to reason about prototypes of new programming languages, compilers, abstract machines, operational semantics, natural deduction calculi, and sequent calculi. In particular, in this thesis we report on the deductive power of Twelf and many experiments: In the area of programming languages for example, Twelf has been used to derive several important properties about Mini-ML, that is a version of an ML-like language without references, module system and exceptions. Mini-ML's operational semantics is type preserving, and it is complete with respect to a reduction semantics. Furthermore, we have used Twelf to show the completeness of compiling and executing Mini-ML programs on a continuation passing machine, similar to the CPM machine [FSDF93].

The Church-Rosser theorem for the simply-typed λ-calculus is the running example used in this thesis. Using the standard decomposition of the development into a sequence of lemmas Twelf can prove all of them automatically. It constructs a proof that is very similar to the one given in [Pfe93].

Many of our experiments include meta-theorems about logics: We have used Twelf to show the equivalence of natural deduction and sequent formulation of first-order intuitionistic logic. Twelf has also shown that the Hilbert derivations can be transformed into natural deduction derivations. For logic programming in the fragment of hereditary Harrop formulas, we have used Twelf to show that the search for uniform derivations and resolution are equivalent.

It took Twelf less than seven minutes on a Pentium II/400Mhz to show that cut-elimination holds for full intuitionistic logic. Consequently first-order logic is sound [Gen35].

Further examples stem from the area of category theory: Twelf has been used to show the existence of an embedding from Cartesian closed categories into the simply typed λ-calculus. The experiments express that the theorem proving technology described in this thesis is powerful enough to prove theorems far outside the realm of traditional theorem provers.

Twelf is currently actively used in other research groups for example at Princeton to investigate logics for proof carrying code [Nec97]. Appel, Felten, and Felty for example are using Twelf to build a generic architecture, that is applied in research on proof carrying code [AF00], and proof carrying authentication [AF99]. At Stanford, Stump and Dill are applying Twelf to develop proof terms for decision procedures [SD99].

1.1 Contributions

The first contribution of this thesis is the design of the meta-logic $\mathcal{M}_2^+$. It is novel in that it combines higher-order representation techniques and dependent types provided by the logical framework LF types with inductive definitions, a combination that has never been attempted before. One of the main consequences of this approach is that the *closed world assumption* underlying standard inductive definitions is not general enough to accommodate arguments over higher-order encodings; this observation leads to the *regular world assumption* that allows for dynamic and regular extensions of inductive definitions. $\mathcal{M}_2^+$ is sound by a realizability interpretation of its proof terms as total functions.

The second contribution is the implementation of the meta-logic $\mathcal{M}_2^+$ in the Twelf system [PS99b]. Because of higher-order representation techniques, proofs of difficult theorems have still concise and elegant forms in $\mathcal{M}_2^+$. We have implemented two proof search algorithm, one for LF and the other for $\mathcal{M}_2^+$. The expressive strength of LF together with the deductive strength of $\mathcal{M}_2^+$ makes Twelf a powerful meta-logical framework.

The third contribution is the application of Twelf to many problems. It has been successfully employed to derive the meta-theory of a variety of examples from the areas of functional programming languages, type theories, operational semantics, abstract machines, compilers, and logics.

1.2 Outline

This thesis is organized in three parts. The first part is designed to give the reader an overview about the background of LF and motivate how to use it as a representation language for deductive system. Specifically, in Chapter 2, we use the example of the simply-typed λ-calculus and the standard reduction rules to motivate dependent types, higher-order representation techniques, canonical forms and the desired adequacy of encoding. The simply typed λ-calculus and its meta-theory are the running example throughout this thesis. In Chapter 3, for example, we prove a sequence of lemmas that eventually leads up to the proof of the Church-Rosser theorem. Among others, we present the proof the diamond lemma in detail. The proofs of all lemmas and theorems can be computationally interpreted as functions, and they demonstrate thus the design principles behind the type theory $\mathcal{M}_2^+$ which we present in the second part of this thesis. Specifically, first we motivate it in Chapter 4 and expose the necessity to dynamically extend the set of constructors for LF types under the regular world assumption. In Chapter 5 then, we make the informal constructions from Chapter 4 formal by defining appropriate judgments and rules for $\mathcal{M}_2^+$. Informal proofs are represented as proof terms in $\mathcal{M}_2^+$. Moreover establish two side conditions, coverage and termination, that informally enforce that all proof terms once evaluated always make progress and are guaranteed to terminate. The meaning of $\mathcal{M}_2^+$-proof terms is defined via a big-step operational semantics in Chapter 6; it is type-preserving, but insufficient to show that all proof terms of $\mathcal{M}_2^+$ are realizers. Therefore, we introduce a state-based abstract machine, its transition rules and syntactic criteria for coverage and termination in Chapter 7; the main result of this chapter is that any proof term in $\mathcal{M}_2^+$ satisfying those two criteria is a realizer, warranting that the interpretation of $\mathcal{M}_2^+$ as a meta-logic is sound. In the third and last part of this thesis, we sketch the implementation of a proof search algorithm for realizers in $\mathcal{M}_2^+$, we discuss its implementation in the Twelf system, and we demonstrate how to use Twelf to prove the Church-Rosser theorem automatically in Chapter 8. Additionally, we briefly report on other experiments already conducted with the Twelf system. Finally we assess the results of this thesis and discuss future work in Chapter 9.

Part I

Background

Chapter 2

Logical Frameworks

2.1 Introduction

The development of programming languages is a challenging endeavor, and much more widespread than one might expect at first glance. Besides standard programming languages, such as C, C++, LISP, ML, and many others, there are scripting languages such as HTML, XML, PERL, or TEX, and query languages such as SQL, or XQL which can be categorized as programming languages.

We can make a very similar observation about logics. Logics are very important "languages" to express properties about any kind of system. Specification logics, temporal logics, and modal logics, are used in software engineering and model checking to describe large systems. Logics are also used to describe properties of secure systems and they form the foundation for logic programming languages.

If a developer follows sound design principles when drafting a programming language or a logic, the user of the language will benefit from it; programs are easier to write, easier to compile and very often easier to maintain. For example, a sound design principle underlying functional programming languages is that the evaluation of programs preserves types. Similarly a sound design principle underlying a specification logic is consistency. Since results such as type preservation of an operational semantics and soundness of an inference system always express properties about the designed language or logic, we call these results *meta-logical* properties. It is very important to verify all desired meta-logical properties after each change in the design of a programming language, e.g. adding new constructors to the language could violate type preservation, and similarly, adding new connectives and new inference rules to a logic could render it unsound.

In this work we are not concerned with the design principles themselves, but rather with tools which support the design process. In this chapter we are primarily interested in the encoding of systems such as programming languages, operational semantics, and logic calculi whereas in the subsequent chapters we investigate and devise a system which allows the formalization and automatic derivation of their meta-logical properties. Concretely, we begin with the presentation of the simply-typed λ-calculus with an appropriate reduction semantics for which we then give its well-known encoding in the logical framework LF. It is the basis of the running example which is used throughout this thesis: in Chapter 3, for example, we derive its Church-Rosser property informally, in Chapter 4 formally, and in Chapter 8 automatically.

2.2 The Simply-Typed λ-Calculus

The λ-calculus has been introduced by Church [CR36] as a model for partial functions. Initially, it was only of theoretical interest and it served as a vehicle for the study of computable functions. In particular it has be shown that each Turing-complete function is also computable in the λ-calculus and vice versa [Rog92]. A few decades later, with the growth of the field of computer science, the λ-calculus has gained a strong foothold in the area of basic computer science and functional programming language. Specifically, with the programming language LISP, a functional programming languages based on the λ-calculus, it has gained a lot of influence, and helped to shape the area of artificial intelligence.

The definition of λ-terms (which we simply call terms below) is deceptively simple. A term can be of the form

1. $\lambda x.e$, where λ is a binding operator, x a variable and e is the body of the term,
2. $e_1\ e_2$, where e_1, e_2 are two subterms, or
3. x, simply a variable.

The term $\lambda x.e_1$ can be interpreted as a function, which may be applied to an argument e_2. Strictly speaking $(\lambda x.e_1)\ e_2$ *reduces* by substituting e_2 for x in e_1, an operation for which use the following notation $e_1[e_2/x]$. Any expression of the form $(\lambda x.e_1)\ e_2$ is called a *redex*.

How exactly reduction is executed is expressed by the operational semantics of the λ-calculus which is given by reduction rules. In general, reduction rule of the form $lhs \Rightarrow rhs$ can be applied to any subterm of a given term; applying a reduction rule means to replace the subterm which matches the shape of the left hand side lhs of a rule and replace it by the right hand side rhs, where the free schematic variables have been instantiated accordingly.

$$\begin{array}{rcl} \lambda x.e & \Rightarrow_\alpha & \lambda y.e[y/x] \\ (\lambda x.e_1)\ e_2 & \Rightarrow_\beta & e_1[e_2/x] \end{array}$$

Informally the first rule called the α-rule allows arbitrary renaming of bound variables. It requires that y does not occur freely in e already. The second rule is called β-rule and it simplifies redices. Therefore a redex is also known as *β-redex*. In the example the application of two identity functions to each other reduces to just one identity function:

$$(\lambda x.x)\ (\lambda y.y) \Rightarrow_\beta (\lambda y.y)$$

We will not consider the α-rule any further because we can assume that substitution application will avoid variable capturing. This is a quite common assumptions and easy to enforce. Replacing x in e_1 by e_2 requires to first rename all variables in e_2 away from variables in e_1. This implicit operation guarantees that the substitution can be safely executed [Chu40].

Types are an important vehicle in programming, because they can be used to capture invariants. In this sense, the untyped λ-calculus has only one type, because everything is a term, and one cannot distinguish between functions and non-functions which attaches a rather misleading meaning to the name "untyped" λ-calculus. The more refined the concept of types, the more invariants the type system can capture.

For the purpose of our example, we introduce now a simple type system which goes back to Church [Chu41] and differentiates between atomic and function types. The syntactic formation rules are expressed using standard extended Backus Naur form notation (EBNF):

$$\textit{Types:} \quad \tau \quad ::= \quad a \mid \tau_1 \to \tau_2$$

This refinement of the untyped λ-calculus has its effects on terms: For the typing rules to be sound which we will introduce below, we must endow bound variables with type information.

$$\textit{Terms:} \quad e \quad ::= \quad x \mid \lambda x : \tau.e \mid e_1\ e_2$$

We call a term *closed*, if all variable occur in the scope of a λ-binder. For example, the term $\lambda x : \tau.x$ is closed whereas $\lambda x : \tau.y$ is not. Terms which are not closed are called *open*.

Types allow us to separate valid terms from invalid terms via a deductive system. In general, deductive systems are defined by a set of *judgments* and a set of *inference rules*. A judgment is an informal statement, the inference rules help to establish its truth in the following way: A judgment is said to be evident, if it can be deduced from axioms by applying the inference rules. For simplicity we think of axioms as inference rules without any premisses. For a very enlightening presentation we refer the interested reader to the work of Martin-Löf [ML80].

We assert that a term e is valid by the judgment: "term e has type τ" and which we abbreviate with $e : \tau$. There are only two inference rules for this judgment which we give in natural deduction style.

$$\dfrac{\begin{array}{c}\overline{\vdash x : \tau_1}\,u \\ \vdots \\ \vdash e : \tau_2\end{array}}{\vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2}\,\mathsf{tplam}^u \qquad \dfrac{\vdash e_1 : \tau_2 \to \tau_1 \qquad \vdash e_2 : \tau_2}{\vdash e_1\ e_2 : \tau_1}\,\mathsf{tpapp}$$

The rule tpapp is an inference rule with two premisses which reads: if the judgment $e_1 : \tau_2 \to \tau_1$ holds, and $e_2 : \tau_2$ then the judgment $e_1\ e_2 : \tau_1$ holds, too. The rule tplam is slightly more complicated, because it introduces an additional assumption marked by the label u which is discharged when the rule is applied. Note that there are no axiom rules. Deductions can only be closed by introduced hypotheses.

Going back to the previous discussion, the introduction of types and the typing relation makes a distinction between valid and invalid terms possible: A term e is valid if there is a type τ and $e : \tau$ is derivable from the two rules above. If not, it is invalid. For any type τ the term $(\lambda x : \tau.x\ x)$ for example is invalid, because when considering the body of the term, if x has type τ, the rule tpapp is not applicable, and neither is tplam.

The reduction rules from the untyped λ-calculus endowed with types at the variable binders form the reduction rules for the simply-typed λ-calculus.

$$\begin{array}{rcl}\lambda x : \tau.e & \Rightarrow_\alpha & \lambda y : \tau.e[y/x] \\ (\lambda x : \tau.e_1)\ e_2 & \Rightarrow_\beta & e_1[e_2/x]\end{array}$$

On the more pragmatic side, there are terms in the untyped λ-calculus which allow infinitely many applications of the reduction rules, as for example:

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \Rightarrow_\beta (\lambda x.x\ x)\ (\lambda x.x\ x) \Rightarrow_\beta \cdots$$

This particular infinite rewrite sequence cannot be derived with the reduction rules in the simply-typed case if we stipulate that we are only working with valid terms. As we have seen, the term $(\lambda x : \tau.x\ x)$ cannot be assigned a type, and hence, all terms in this rewrite sequence are ill-typed. As a matter of fact, we can show that for each well-typed term, there is only a finite sequence of reduction step before no reduction step is applicable no more. The right-most term of such a sequence is called a *normal form* of the initial term, and as we will discuss now, it is always unique.

2.2.1 Reduction Relations

The reduction rules of the simply-typed λ-calculus are commonly used to assign meaning to a term. One way of doing this is to identify all terms that reduce to the same result as a class, and to pick one witness of the class as a semantic representative. Is this semantic well-defined? Is it sound? Needless to say, that in order to decide if two terms mean the same thing we have to check that they are in the same class. Is it possible to calculate the class representative for each term quickly and effectively? Is the meaning of each term unique? In this section we formally define an appropriate reduction relation for the simply typed λ-calculus for which we prove the unique existence of class representatives in Section 3.2. This class representative is commonly referred to as normal form.

Informally, we apply the β-reduction rule in the following way: for a given term, select a subterm, match it with the left hand side of a reduction rule and then replace it by the right hand side. In the following, we make this more precise. To assert that a term e reduces to a term e' in one step, we use the judgment $e \xrightarrow{1} e'$. The rules which define this judgment are as follows:

$$\frac{}{(\lambda x : \tau.e_1)\ e_2 \xrightarrow{1} e_1[e_2/x]}\ \mathsf{rbeta}$$

$$\frac{e \xrightarrow{1} e'}{\lambda x : \tau.e \xrightarrow{1} \lambda x : \tau.e'}\ \mathsf{rlam}$$

$$\frac{e_1 \xrightarrow{1} e_1'}{e_1\ e_2 \xrightarrow{1} e_1'\ e_2}\ \mathsf{rapp}_1 \qquad \frac{e_2 \xrightarrow{1} e_2'}{e_1\ e_2 \xrightarrow{1} e_1\ e_2'}\ \mathsf{rapp}_2$$

For any given term, there might be more than just one possibility to apply a reduction rule. Consider for example the well-typed term $\lambda x : \tau.(\lambda y : \tau.y)\ ((\lambda z : \tau.x)\ x)$ which can reduce in one step to two different terms: $\lambda x : \tau.(\lambda z : \tau.x)\ x$ and $\lambda x : \tau.(\lambda y : \tau.y)\ x$. First, the body of the entire expression is amenable for β-reduction as this derivations shows:

$$\frac{\dfrac{}{(\lambda y : \tau.y)\ ((\lambda z : \tau.x)\ x) \xrightarrow{1} (\lambda z : \tau.x)\ x}\ \mathsf{rbeta}}{\lambda x : \tau.(\lambda y : \tau.y)\ ((\lambda z : \tau.x)\ x) \xrightarrow{1} \lambda x : \tau.(\lambda z : \tau.x)\ x}\ \mathsf{rlam}$$

Second, the argument inside the body is also amenable for β-reduction.

$$\dfrac{\dfrac{\dfrac{}{(\lambda z:\tau.x)\ x \stackrel{1}{\longrightarrow} x}\ \textsf{rbeta}}{(\lambda y:\tau.y)\ ((\lambda z:\tau.x)\ x) \stackrel{1}{\longrightarrow} (\lambda y:\tau.y)\ x}\ \textsf{rapp}_2}{\lambda x:\tau.(\lambda y:\tau.y)\ ((\lambda z:\tau.x)\ x) \stackrel{1}{\longrightarrow} \lambda x:\tau.(\lambda y:\tau.y)\ x}\ \textsf{rlam}$$

Repeated applications of single-step reduction sequence are captured by the multi-step reduction relation: If, for example, $e_1 \stackrel{1}{\longrightarrow} e_2$ and $e_2 \stackrel{1}{\longrightarrow} e_3$ and $e_3 \stackrel{1}{\longrightarrow} e_4$, then we write $e_1 \stackrel{*}{\longrightarrow} e_4$. Clearly, $e \stackrel{*}{\longrightarrow} e'$ is again a judgment, which we define by two inference rules.

$$\dfrac{}{e \stackrel{*}{\longrightarrow} e}\ \textsf{rid} \qquad \dfrac{e \stackrel{1}{\longrightarrow} e' \quad e' \stackrel{*}{\longrightarrow} e''}{e \stackrel{*}{\longrightarrow} e''}\ \textsf{rstep}$$

Finally, we define the conversion relation as the reflexive, transitive, and symmetric closure of the multi-step reduction. e_1 and e_n are convertible if and only if the new judgment $e_1 \longleftrightarrow e_2$ is derivable using the following inference rules:

$$\dfrac{}{e \longleftrightarrow e}\ \textsf{rrefl} \qquad \dfrac{e \stackrel{*}{\longrightarrow} e'}{e \longleftrightarrow e'}\ \textsf{rred} \qquad \dfrac{e \longleftrightarrow e'}{e' \longleftrightarrow e}\ \textsf{rsymm} \qquad \dfrac{e \longleftrightarrow e' \quad e' \longleftrightarrow e''}{e \longleftrightarrow e''}\ \textsf{rtrans}$$

It is very easy to see, that there is a derivation of $\lambda x:\tau.(\lambda z:\tau.x)\ x \longleftrightarrow \lambda x:\tau.(\lambda y:\tau.y)\ x$.

To guarantee soundness of the reduction semantics, we need to show the well-known Church-Rosser property, that is that any two convertible terms reduce to the same unique normal form given that their reductions terminate. The informal development of this proof will be the main content of Chapter 4. But first, we investigate possible formalizations of the simply-typed λ-calculus in a logical framework, their advantages and their disadvantages.

2.3 Methodology of Representation

The first step, when using a computer to facilitate the design and the formal development of a programming language or a logic, is to choose an appropriate formalism to represent these abstract systems or *object languages* as we sometimes call them, in order to make them amenable for algorithmic manipulation and automated reasoning. As a matter of fact, as we show in this thesis, this point cannot be overemphasized. We will see, the more elegant and direct a programming language can be represented — in our example the simply typed λ-calculus — the easier it is to do the second step namely to specify meta-theoretic properties, such as the Church-Rosser theorem.

Even though the formalism to represent an abstract system is called a meta-language in the literature [HHP93, McD97] we will not adopt this name in order not to confuse the reader with the continuous overloading of the term "meta". Throughout this thesis, we use the word "meta" only to refer to the reasoning layer, the upper level above the representation layer in Figure 2.1. For us, the informal description and the formal representation of a programming language is very close and natural, and since the adequacy of representation is the most basic assumption, we can almost identify the informal and formal representation of an abstract system. Instead of

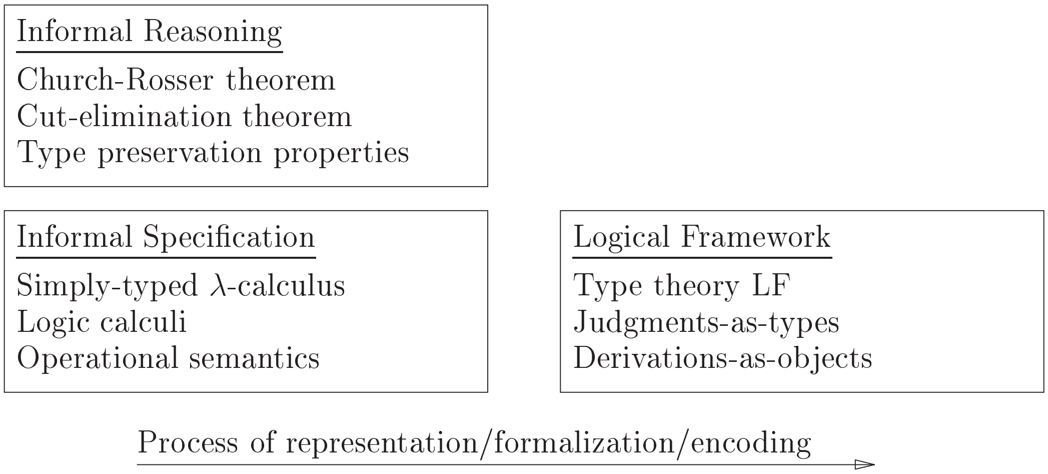

Figure 2.1: Methodology of representation

meta-language, we adopt the common name *logical framework* for the representation language, and we speak of the *encoding* of an abstract system, such as the simply-typed λ-calculus, as the image of the representation in the logical framework.

In this section, we motivate and describe the minimal requirements we stipulate for the representation language, which gradually leads to the definition of the logical framework LF [HHP93]. We also review other logical frameworks, such as the calculus of constructions [Coq86].

2.3.1 Type theory

The challenge in representing a programming language or a logic which is specified via a deductive system is to define suitable concepts to represent its components: the set of judgments and the set of inference rules. In the past few decades approaches based on type theory have prevailed. The underlying paradigms suggest to use types to represent judgments, and objects to represent derivations. To show that "a judgment is evident" reduces in type theory to the construction of an object, the so-called *witness* of a type corresponding to the judgment. If such a witness exists the type is called *inhabited*, otherwise *uninhabited*. Within this paradigm, judgments are hence represented as types and derivations as objects.

In order to validate formal arguments about derivations in a deductive systems, we must be sure that the objects in the logical framework that are being manipulated naturally correspond to derivations in the deductive system and vice versa. Therefore, it must be a priori enforced, that all derivations of a deductive system stand in one-to-one correspondence with their encodings. This requirement provides the central justification of formalization and formal reasoning in general, it must not be destroyed by any extensions to the logical framework.

2.3.2 Higher-order abstract syntax

The issues which arise when representing the simply-typed λ-calculus from the Section 2.2 in a logical framework are manifold. We hence tackle them, one by one, and we start with a technique called higher-order abstract syntax. Higher-order abstract syntax provides an extremely brief and elegant way of representing variables, and capture-avoiding substitutions. In our first example of the untyped λ-calculus, terms were defined by the following syntactic rules:

$$\textit{Untyped terms:} \quad e \quad ::= \quad x \mid \lambda x.e \mid e_1\ e_2$$

Implicitly, this syntactic description defines a judgment and a set of inference rules. It is very important to understand the elegant uniformity since it is a recurring scheme throughout this thesis, and only a deep understanding of this technique can explain the benefits of all the techniques which are developed and discussed in subsequent chapters. The judgment induced by the syntactic rules above is simply "is an untyped term" for which we simply write "term", and the inference rules are:

$$\dfrac{\dfrac{}{\text{term}}\,x \atop {\vdots \atop \text{term}}}{\text{term}}\,\mathsf{lam}^x \qquad \dfrac{\text{term} \qquad \text{term}}{\text{term}}\,\mathsf{app}$$

Note, that the treatment of variables is implicit in these rules. There is no need for a rule which states that x is a term, since this assumption is dynamically introduced by the lam rule and discharged thereafter. There is a crucial difference in presenting the syntax of terms in EBNF or as a deductive system. In the former case, one might first think of representing variables as strings, or integers, or some other auxiliary construct, which would lead to the representation of the two judgments as type "term" and type "var"

$$\begin{array}{rcl} \text{term} & : & \text{type} \\ \text{var} & : & \text{type} \end{array}$$

which, hypothetically speaking, would lead to the following representation of the object constants: "var" of type var $\to$ term which coerces variables to terms, "lam" of type var $\to$ term $\to$ term, and "app" of type term $\to$ term $\to$ term:

$$\begin{array}{rcl} \text{var} & : & \text{var} \to \text{term} \\ \text{lam} & : & \text{var} \to \text{term} \to \text{term} \\ \text{app} & : & \text{term} \to \text{term} \to \text{term} \end{array}$$

In the later case, on the other hand, one might be inspired to represent the variable of the untyped λ-calculus by a variable provided by the logical framework. This is the concept which we predominantly use in this thesis and it is called *higher-order abstract syntax* [PE88]. It leads to a much simplified representation of terms: we only need to represent one judgment, namely term. Formally, we write that $\ulcorner \text{term} \urcorner = \text{term}$, where the "term" on the left of the equality symbol is the judgment "term", and the "term" on the right is a type. The representation function maps judgments to types and derivations to objects and is written as $\ulcorner \cdot \urcorner$.

$$\text{term} \quad : \quad \text{type}$$

Using this technique, we can inductively define the encoding of the untyped terms by representing each of the inference rules. In the case of the λ-binder, we must dynamically introduce a new bound variable, x. Note that the λ-binder to the right of the equality sign is the λ-binder of the logical framework. The e to the left of the equality symbol represents the derivation of the premiss. Throughout the thesis, we will name the newly defined object constants in correspondence with the names of the rules they are representing. This greatly improves the presentation of this material. In addition, it is always be clear from the context what a name refers to.

$$\ulcorner \dfrac{\begin{array}{c}\overline{\text{term}}\,x \\ e \\ \text{term}\end{array}}{\text{term}}\,\mathsf{lam}^x \urcorner = \text{lam}\ (\lambda x.\ulcorner e \urcorner)$$

In a similar, but much easier way, the application rule is represented by an object constant "app". e_1 and e_2 are simply symbolic names of the derivations of the premisses.

$$\ulcorner \dfrac{\begin{array}{c} e_1 \\ \text{term}\end{array} \quad \begin{array}{c} e_2 \\ \text{term}\end{array}}{\text{term}}\,\mathsf{app} \urcorner = \text{app}\ \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$$

In summary, the representation of the lam and the app rule are two object constants, with corresponding names. Note that the type of "lam" expresses that it expects a function as argument.

$$\begin{array}{lcl} \text{lam} & : & (\text{term} \to \text{term}) \to \text{term} \\ \text{app} & : & \text{term} \to \text{term} \to \text{term} \end{array}$$

As a side remark we want to point out, that both possibilities are correct in the sense that it is possible to identify λ-terms with their images in the type theory. Such an encoding is called *adequate*. We discuss the problems related to adequacy in the the next subsection.

Why is the encoding using higher-order abstract syntax preferable? We make the following observation: Closely associated with the notion of a variable is the notion of substitution. If λ-terms were encoded as suggested in the first solution with "var" and "term", the reduction rules could not be represented directly, because the notion of substitution has to made explicit. As example, consider the left hand side $e_1[e_2/x]$ of the β-reduction rule from Section 2.2. In addition, the properties of substitutions must be analyzed and proven explicitly in order to take advantage of them.

Lemma 2.1 (Substitution) *If e_1 : term with zero or more occurrences of the variable x : var, and e_2 : term, then there exists a term e', where all occurrences of $(var\ x)$ have been replaced by e_2.*

Proof: The proof goes by induction over e_1. □

Even though it is easy in this particular example, substitution lemmas require in general very tedious and time consuming proofs in more complicated settings. In addition, experience has shown that lemmas of this form are quite common when experimenting with programming languages and logics. Most likely their mere existence will pollute the proof search of subsequent lemmas in the implementation which is being discussed in Chapter 8. For larger examples, such as the entire simply-typed λ-calculus (see in Section 2.2) including a typing relation and more (to be discussed in Section 3.2), proving these kind of lemmas is a necessary, time-consuming, and simultaneously not very rewarding activity. Therefore, it is of great benefit, if the treatment of variables and substitutions is implicit.

On the other hand, if we represent terms with higher-order abstract syntax, the substitution lemma comes for free by the means of the representation. $\ulcorner e_1[e_2/x] \urcorner$ for example is encoded by the β-rule of the logical framework. Since $\ulcorner \lambda x.e_1 \urcorner = \text{lam}\ (\lambda x : \text{term}. \ulcorner e_1 \urcorner)$ where $\ulcorner x \urcorner = x$, it follows that $(\lambda x : \text{term}. \ulcorner e_1 \urcorner)$ is a function of type term $\rightarrow$ term. Moreover, by construction, if we apply this function to any other term all variables x are being replaced by the argument term, hence force executing substitution in the λ-calculus. Consequently, the representation of the left hand side of the β-rule in our object language is simply

$$\ulcorner e_1[e_2/x] \urcorner = (\lambda x : \text{term}. \ulcorner e_1 \urcorner)\ \ulcorner e_2 \urcorner$$

where the juxtaposition to the right of the equality symbol is the application operation of a function to an argument provided by the logical framework.

The difference between first-order and higher-order representation techniques is that with first-order representations the concept of substitution and the substitution application mechanism must be explicitly defined and the associated properties explicitly proven. With higher-order representations on the other hand, we can use the variables and notion of substitution from the logical framework and inherit all associated properties for free. Naturally, when using higher-order representation techniques, the proof of adequacy is more complicated and less direct then in the first-order case. The adequacy of representation is essential in our approach and therefore discussed in the next subsection.

2.3.3 Adequacy

Deductive systems and their representations in a logical framework must correspond to each other. The reason is that any derivation in the deductive system should be representable as an object in the type theory and vice versa. In particular, after mechanically manipulating objects in the type theory, we must be certain that the results correspond to derivations in the deductive system. In addition, if higher-order abstract syntax is used, the representation must be compositional, i.e. β-reduction provided by the logical framework corresponds to substitution. This correspondence is called *adequacy*. The untyped λ-calculus can be represented in a very simple logical framework, as we have seen in the previous subsection namely the simply-typed λ-calculus (which would be the logical framework). On the other hand, representing the simply-typed λ-calculus from Section 2.2, requires a refined logical framework to guarantee the adequacy of encoding which we motivate in this subsection, and which discuss in detail in Section 2.4.

In Section 2.2 we have encountered well-typed and ill-typed terms. Since every simply-typed term e can be embedded into the untyped λ-calculus, clearly $\ulcorner e \urcorner$: term, but on the flip side, every ill-typed term e' can also be embedded: $\ulcorner e' \urcorner$: term. The encoding is hence not adequate. It is not because there are too many objects of type "term", many more then there are well-typed simply-typed terms.

This observation motivates the solution which has been widely accepted in the literature. In order to preserve the adequacy of the encoding, we must partition the type "term". This can be done by indexing it. But by what? The best solution is to index it by the type which all objects in this partition share! Intuitively, we partition the the set of objects of type "term", into subsets corresponding to the different types. We will see that these subsets are pairwise disjoint because typing is unique (by Lemma 2.7). A consequence is, that by construction, ill-typed terms do not belong to any of those partitions. Therefore, strictly speaking, the union of all index partitions yields the set of simply-typed terms we are interested in but there is an additional partition; the partition of all ill-typed terms. In order to distinguish non-indexed from indexed types we continue to call the former *type* and the latter *type family*.

In order to represent simply-typed terms, we combine the syntactic formation rules for well-typed terms and their typing rules, as discussed in Section 2.2. The resulting deductive system is described by a judgment "is a term of type τ", or short "term τ", and the two inference rules are given below.

$$\dfrac{\begin{array}{c}\overline{\text{term } \tau_1}^{\,x} \\ \vdots \\ \text{term } \tau_2\end{array}}{\text{term } (\tau_1 \to \tau_2)}\,\mathsf{lam}^x \qquad \dfrac{\text{term } (\tau_2 \to \tau_1) \qquad \text{term } \tau_2}{\text{term } \tau_1}\,\mathsf{app}$$

The representation of the judgment is defined by

$$\ulcorner \text{term } \tau \urcorner = \text{term } \ulcorner \tau \urcorner$$

where the juxtaposition to the right of the equality symbol is the type application operation provided by the logical framework, which we will discuss in Section 2.4. For the remainder of this section, it is sufficient to read the argument to the type family term as index.

Similarly to the representation of the untyped λ-calculus, we obtain two equations, one for the lam rule

$$\ulcorner \dfrac{\begin{array}{c}\overline{\text{term } \tau_1}^{\,x} \\ e \\ \text{term } \tau_2\end{array}}{\text{term } (\tau_1 \to \tau_2)}\,\mathsf{lam}^x \urcorner = \text{lam } (\lambda x : \text{term } \ulcorner \tau_1 \urcorner . \ulcorner e \urcorner) : \text{term } \ulcorner \tau_1 \to \tau_2 \urcorner$$

and another for the app rule

$$\ulcorner \dfrac{\begin{array}{c} e_1 \\ \text{term } (\tau_2 \to \tau_1)\end{array} \qquad \begin{array}{c} e_2 \\ \text{term } \tau_2\end{array}}{\text{term } \tau_1}\,\mathsf{app} \urcorner = \text{app } \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner : \text{term } \ulcorner \tau_1 \urcorner$$

which implicitly define the constants lam and app. Types of the simply-typed λ-calculus are represented by tp : type, and

$$\ulcorner \tau_1 \to \tau_2 \urcorner = \ulcorner \tau_1 \urcorner \text{ arrow } \ulcorner \tau_2 \urcorner$$

where "arrow" is a constant defined in LF. For better readability we use it as an infix operator. In summary, the representation of simple types, the judgment "is a term of type τ" and the

$$
\begin{array}{rcl}
\mathrm{tp} & : & \mathrm{type} \\
\mathrm{arrow} & : & \mathrm{tp} \to \mathrm{tp} \to \mathrm{tp} \\
\\
\mathrm{term} & : & \mathrm{tp} \to \mathrm{type} \\
\mathrm{lam} & : & (\mathrm{term}\ T_1 \to \mathrm{term}\ T_2) \to \mathrm{term}\ (T_1\ \mathrm{arrow}\ T_2) \\
\mathrm{app} & : & \mathrm{term}\ (T_2\ \mathrm{arrow}\ T_1) \to \mathrm{term}\ T_2 \to \mathrm{term}\ T_1
\end{array}
$$

Figure 2.2: Type and term constant declarations

inference rules lead to the constant declarations depicted in Figure 2.2. "tp" is a type, "term" is a type family, and both are alternatively called type constants. "arrow", "lam", "app" are object constants. In order not to confuse the type with the object level, we follow the standard definitions in the literature [HHP93], and call the type of a type constant *kind*, and continue to call the type of an object constant *type*. The uppercase variable names T_1 and T_2 are universally quantified place holders that can be instantiated with any type T_1 and T_2.

As a matter of fact, the distinction between objects, types, and kinds define already the syntactic hierarchy we require from a logical framework. A complete list of type and object constant declarations is called a *signature*, complete in a sense, that each type and each kind used in the signature does not contain any undeclared type or object constants.

We return to the question of adequacy. An encoding is adequate, if each derivation in the deductive system has exactly one counterpart in the type theory and vice versa. The adequacy result for the representation of types is in one direction a straightforward inductive argument. Let $a_1, \ldots, a_n$ be atomic types, which are directly represented in the logical framework as object constants $a_1 : \mathrm{tp} \ldots a_n : \mathrm{tp}$.

Lemma 2.2 (Adequacy of representation of types I) *If τ is a type, then $\ulcorner\tau\urcorner : tp$*

Proof: by induction on τ:

Case: $\tau = a_i$:

$a_i : \mathrm{tp}$	by assumption

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

$\ulcorner\tau_1\urcorner : \mathrm{tp}$	by i.h. on τ_1
$\ulcorner\tau_2\urcorner : \mathrm{tp}$	by i.h. on τ_2
$\ulcorner\tau_1\urcorner\ \mathrm{arrow}\ \ulcorner\tau_2\urcorner : \mathrm{tp}$	by application provided by the type theory
$\ulcorner\tau_1 \to \tau_2\urcorner : \mathrm{tp}$	by definition

□

The second direction is not more complicated, but it requires that the objects of the logical framework can be analyzed structurally, i.e. an object must have only finitely many shapes. This requirement is clearly not satisfied for types. Consider for example the following three objects:

$$(\lambda x : \mathrm{tp}.\, x)\ a_1 \tag{2.1}$$

$$a_1 \tag{2.2}$$

$$(\lambda x : \mathrm{tp}.\, a_1)\ a_2 \tag{2.3}$$

Obviously, all three have type tp. Moreover, if one stipulates the existence of an appropriate β-rule in the type theory (as one can), all three of them reduce to a_1. In other words, there are too many objects in the type theory corresponding to exactly one derivation in the deductive system, hence violating the desired and required one-to-one correspondence between derivations and objects, and hence clearly violating the adequacy of the representation.

What can be done? The answer comes naturally. We consider only those objects in the LF type theory, which are canonical, i.e. objects which cannot be reduced any further. In essence, the logical framework we are motivating here, guarantees the existence of these canonical forms for every well-typed object. The canonical form theorem is essential to the whole thesis, and is discussed in more detail in Section 2.4. But note, that it is implicitly already used here: a canonical object T : tp of the logical framework has always the shape of either of the two β-normal forms: $T = a_i$ or $T = T_1$ arrow T_2.

Lemma 2.3 (Adequacy of representation of types II)
If $T : tp$ is canonical then $T = \ulcorner\tau\urcorner$ and τ is a type.

Proof: by induction over the canonical forms of T:

$T = a_i$

$T = \ulcorner a_i \urcorner$	by assumption
a_i is a type	by assumption

$T = T_1$ arrow T_2

$T_1 = \ulcorner\tau_1\urcorner$ and τ_1 is a type	by i.h. on T_1
$T_2 = \ulcorner\tau_2\urcorner$ and τ_2 is a type	by i.h. on T_2
$\ulcorner\tau_1 \to \tau_2\urcorner = \ulcorner\tau_1\urcorner$ arrow $\ulcorner\tau_2\urcorner = T_1$ arrow T_2	by definition
$\tau_1 \to \tau_2$ is a type	by syntactic rule

□

In a very similar way, we can prove the adequacy of the representation of terms by structural induction. But in this example, β-normal forms do not describe uniquely the possible shapes of an object of type term: Consider for example the two objects:

$$\mathrm{lam}\ (\lambda x : \mathrm{term}\ (a_1\ \mathrm{arrow}\ a_2).\, \mathrm{lam}\ (\lambda y : \mathrm{term}\ a_1.\, \mathrm{app}\ x\ y)) \tag{2.4}$$

$$\mathrm{lam}\ (\lambda x : \mathrm{term}\ (a_1\ \mathrm{arrow}\ a_2).\, \mathrm{lam}\ (\mathrm{app}\ x)) \tag{2.5}$$

Both objects have type "term $(a_1 \to a_2) \to a_1 \to a_2$", and they correspond to the same derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\text{term } (a_1 \to a_2)}\,x \qquad \dfrac{}{\text{term } a_1}\,y}{\text{term } a_2}\,\mathsf{app}}{\text{term } a_1 \to a_2}\,\mathsf{lam}^y}{\text{term } (a_1 \to a_2) \to a_1 \to a_2}\,\mathsf{lam}^x$$

The difference between the two terms is one application of the so called η-reduction rule, which is also part of the logical framework:

$$\lambda x : A.\, M\; x \Rightarrow_\eta M \qquad\qquad \text{if } x \text{ does not occur in } M$$

For adequacy, besides being β-normal, the term must be in η-long form, i.e. the η-rule must be applied in reverse direction until the term cannot be expanded any further without introducing a β-redex. Canonical objects are always in β-normal and η-long form. We leave the details to Section 2.4. In our examples (2.2), (2.4) are canonical, and (2.1), (2.3), (2.5) are not.

Since they exist, canonical objects can be analyzed according to their structure. Note, that this observation holds for objects of atomic and of functional type. Any closed canonical object E of type "term T" has one of two possible shapes:

$$\begin{array}{lcllll} E & = & \text{lam } E' & \text{where} & E' : \text{term } T_1 \to \text{term } T_2 \\ & & & \text{and} & T = T_1 \text{ arrow } T_2 \\ E & = & \text{app } E_1\; E_2 & \text{where} & E_1 : \text{term } (T_1 \to T) \\ & & & \text{and} & E_2 : \text{term } T_1 \end{array}$$

Any closed canonical object E of type term $T_1 \to$ term T_2 has one of three possible shapes.

$$\begin{array}{lcllll} E & = & \lambda x : \text{term } T_1.\, x & \text{where} & T_1 = T_2 \\ E & = & \lambda x : \text{term } T_1.\, \text{lam } (E'\; x) & \text{where} & (E'\; x) : \text{term } T_3 \to \text{term } T_2 \\ E & = & \lambda x : \text{term } T_1.\, \text{app } (E_1\; x)\; (E_2\; x) & \text{where} & (E_1\; x) : \text{term } (T_3 \text{ arrow } T_2) \\ & & & \text{and} & (E_2\; x) : \text{term } T_3 \end{array}$$

The adequacy theorem follows by two simple structural inductions, the proofs of the individual cases proceed in a similar fashion as the ones for types.

Lemma 2.4 (Adequacy of representation of terms)

1. *If e :: term of type τ which may rely on assumptions of the form x_1 :: term $\tau_1, \ldots, x_n$ :: term τ_n then $\ulcorner e \urcorner$: term $\ulcorner \tau \urcorner$ which possibly contains variables of the form x_1 : term $\ulcorner \tau_1 \urcorner, \ldots, x_n$: term $\ulcorner \tau_n \urcorner$.*

2. *If E : term $\ulcorner \tau \urcorner$ is canonical, possibly containing variables of the form x_1 : term $\ulcorner \tau_1 \urcorner, \ldots, x_n$: term $\ulcorner \tau_n \urcorner$, then $E = \ulcorner e \urcorner$ where e :: term τ which may rely on assumptions of the form x_1 :: term $\tau_1, \ldots, x_n$ :: term τ_n*

Proof: by structural induction over e, and E. □

All that remains to be shown for the adequacy of encoding of terms is compositionality, i.e. that the β-rule of the logical framework can be used to represent substitution application. Compositionality is not important for the adequacy of the representation of types, since it does not employ higher-order abstract syntax, but is very important for the adequacy result for the representation of terms.

Consider a term e_1, with a free variable x. After unfolding the syntactic formation rules, e_1 is a derivation of the following form

$$\begin{array}{c} \overline{\text{term } \tau_2}\ x \\ e_1 \\ \text{term } \tau_1 \end{array}$$

and its representation in the logical framework is a function:

$$\ulcorner \begin{array}{c} \overline{\text{term } \tau_2}\ x \\ e_1 \\ \text{term } \tau_1 \end{array} \urcorner = \lambda x : \text{term} \ulcorner \tau_2 \urcorner . \ulcorner e_1 \urcorner : \text{term} \ulcorner \tau_2 \urcorner \to \text{term} \ulcorner \tau_1 \urcorner$$

Given another term e_2 of type τ_2, informally, the substitution means to replace all occurrences of x in e_1 by the new derivation of $e_2 :: \text{term } \tau_2$. The representation of e_2 yields

$$\ulcorner e_2 \urcorner : \text{term} \ulcorner \tau_2 \urcorner$$

Clearly, the term $(\lambda x : \text{term} \ulcorner \tau_2 \urcorner . \ulcorner e_1 \urcorner) \ulcorner e_2 \urcorner$ is well-typed, and it has a canonical form, but does it correspond to the $\ulcorner e_1[e_2/x] \urcorner$? The answer gives the compositionality lemma which is typically considered part of the adequacy property. It can be easily proven by structural induction given a precise definition of substitution, which we omit here.

Lemma 2.5 (Compositionality) *If e_1 is a well-typed term which is hypothetical in $x :: \mathit{term}\ \tau, x_1 :: \mathit{term}\ \tau_1, \ldots, x_n :: \mathit{term}\ \tau_n$ and e_2 is a well-typed term of type τ, then*

$$\ulcorner e_1[e_2/x] \urcorner = (\lambda x : \mathit{term} \ulcorner \tau_2 \urcorner . \ulcorner e_1 \urcorner) \ulcorner e_2 \urcorner$$

Proof: by structural induction over e_1. □

Consequently, the representation of the β-rule of the simply-typed λ-calculus as we introduced it above, is perfectly sound. The β-reduction rule of the logical framework can be used as a vehicle to represent substitutions.

2.3.4 Summary

Based on the principles we have introduced in this section, we can use logical frameworks to reason formally about deductive systems. Judgments are represented as types and derivations as objects. Consequently inference rules are encoded as constants. In this work, we consider only logical frameworks that provide a notion of objects, a notion of types, and a notion of kinds; in particular, in the next section we discuss the logical framework LF, that provides dependent types, and it satisfies the property that each object, each type, and each kind possesses a canonical form. It allows us to use higher-order representation techniques while preserving the adequacy of the encoding.

2.4 The Logical Framework LF

There are many logical frameworks suitable for the representation of deductive systems. The logical framework based on the simply-typed calculus, such as Isabelle [Pau94] requires extra infrastructure to guarantee adequacy theorems. For this work, however, we restrict our considerations to a logical framework that provides dependent types, such that LF [HHP93]. Indeed, dependent types facilitate adequate higher-order encodings. Thus, we have chosen LF as the framework of choice for this thesis. In future work we plan to extend this work to other logical frameworks, such as for example the calculus of constructions [CH88] or the linear logical framework [CP96].

In this section we give a detailed overview over the language, the judgments, the inference rules and the meta-theory of LF. Many, if not all of these results go back to the work of Harper, Honsell, and Plotkin [HHP93], and the interesting reader is referred to an excellent tutorial by Pfenning[Pfe00]. A detailed discussion about canonical forms in LF can be found in [HP99]. These are the three standard references for this section.

2.4.1 Syntax

Most of the syntactical constructions have been motivated in the previous section. All of them are present in the logical framework LF. LF's notion of dependent type provides enough expressive power to warrant adequate representations of judgments as types, which we denote with A. Kinds K are needed to classify well-formed type families. The formation rules for objects M admit constants c, variables x, application $M_1\ M_2$, λ-abstraction $\lambda x : A.\, M$. Types are formed from type constants (or type families) a, type application $A\ M$, and dependent types $\Pi x : A_1.\, A_2$. A dependent type binds an object variable x, and allows other types in its body to dependent on it. In other words, $\Pi x_1 : A_1.\, A_2$ is a generalized function type $A_1 \to A_2$, where the variable x is permitted to occur in the type A_2. As a matter of fact, we use the notation $A_1 \to A_2$ if the variable x does not occur in the type A_2. Consider for example our slight but not unreasonable simplification of the type of the "lam" constant

$$\mathrm{lam} \quad : \quad (\mathrm{term}\ T_1 \to \mathrm{term}\ T_2) \to \mathrm{term}\ (T_1\ \mathrm{arrow}\ T_2)$$

Strictly speaking $(\mathrm{term}\ T_1 \to \mathrm{term}\ T_2) \to \mathrm{term}\ (T_1\ \mathrm{arrow}\ T_2)$ is not a type but a family of types, since neither T_1 nor T_2 are declared anywhere. To transform it into a real LF type, we need to build the Π-closure and obtain

$$\mathrm{lam} \quad : \quad \Pi T_1 : \mathrm{tp}.\, \Pi T_2 : \mathrm{tp}.\, (\mathrm{term}\ T_1 \to \mathrm{term}\ T_2) \to \mathrm{term}\ (T_1\ \mathrm{arrow}\ T_2)$$

Note, that T_1 and T_2 are object level variables. There is a drawback to this complete notation; whenever the object constant lam is used, it must be first applied to its domain and its range type. Intuitively, this seems unnecessary since they can be easily be inferred from their positions and occurrences in the type itself. They must be types: tp! Indeed, it is safe to omit these implicit arguments if one uses the reconstruction algorithm proposed by Conal and Pfenning [PE88]. For better presentation, we hence omit inferable leading Π-abstractions throughout this thesis, without further mention. The reader should bear this in mind.

$$\begin{array}{llll}
\textit{Kinds:} & K & ::= & \mathrm{type} \mid \Pi x : A.\, K \\
\textit{Types:} & A & ::= & a \mid A\ M \mid \Pi x : A_1.\, A_2 \\
\textit{Objects:} & M & ::= & c \mid x \mid M_1\ M_2 \mid \lambda x : A.\, M
\end{array}$$

The representation of a deductive system is a set of constant declarations. Type constant declarations represent judgments, and object constant declarations represent inference rules. A collection of these declarations is called signature, which we denote by Σ. Similarly, we introduce the notion of a *context* as a collection of variable declarations $x_1 : A_1, \ldots, x_n : A_n$ which we denote by Γ. Contexts play an important rôle when we define the semantics and validity of object, types, and kinds.

$$\begin{array}{llll} \textit{Signatures:} & \Sigma & ::= & \cdot \mid \Sigma, c : A \mid \Sigma, a : K \\ \textit{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma, x : A \end{array}$$

The $\cdot$ stands for an empty signature and an empty context. We simply omit it (and the following ",") if the signature and the context are non-empty not to clutter the presentation unnecessarily.

2.4.2 Semantics

The semantics of LF type theory is defined by a set of of judgments and inference rules. Among the necessary judgments we must specify what are valid objects, types, kinds, signatures, and contexts. Note, that the following judgments are all indexed by the signature Σ, but we can consider it fixed for all our purposes, and therefore we take the liberty to omit it from the rules given below.

Judgments:

$$\begin{array}{ll} \textit{Valid kinds:} & \Gamma \vdash_\Sigma K \text{ kind} \\ \textit{Valid types:} & \Gamma \vdash_\Sigma A : K \\ \textit{Valid objects:} & \Gamma \vdash_\Sigma M : A \\ \textit{Valid signatures:} & \vdash \Sigma \text{ sig} \\ \textit{Valid contexts:} & \vdash_\Sigma \Gamma \text{ ctx} \end{array}$$

In Section 2.2 and Section 2.3, we have encountered two reduction rules, namely the β- and η-rule. As above, these rules also exist in the dependently typed setting, and they define a congruence relation on objects, kinds and terms, which allows us to identify all objects which do have the same unique canonical (i.e. β-normal, η-long) form. Canonical forms exist because of Theorem 2.6 below. Its proof depends on the congruence judgments to include typing information, but in this presentation omit it from the rules below in order to keep the presentation clean.

$$\begin{array}{ll} \textit{Congruence on kinds:} & K_1 \equiv K_2 \text{ kind} \\ \textit{Congruence on types:} & A_1 \equiv A_2 : K \\ \textit{Congruence on objects:} & M_1 \equiv M_2 : A \end{array}$$

Rules: Most of these judgments are mutually dependent, i.e. inference rules of one judgments are defined in terms of another. The rules defining these eight judgments are all standard. We start with the presentation of the typing rules of kinds.

$$\frac{}{\Gamma \vdash \text{type kind}}\,\mathsf{kndtyp} \qquad \frac{\Gamma \vdash A : \text{type} \qquad \Gamma, x : A \vdash K \text{ kind}}{\Gamma \vdash \Pi x : A.\, K \text{ kind}}\,\mathsf{kndpi}$$

$$\frac{\Gamma \vdash A : K \qquad K \equiv K' \qquad \Gamma \vdash K' : \text{ kind}}{\Gamma \vdash A : K'}\,\mathsf{kndcnv}$$

The typing rules for types and type families are defined as follows. They extend the simply-typed λ-calculus from the Section 2.2.

$$\frac{\Sigma(a) = K}{\Gamma \vdash a : K}\,\mathsf{famcon}$$

$$\frac{\Gamma \vdash A_1 : \Pi x : A_2.\, K \qquad \Gamma \vdash M : A_2}{\Gamma \vdash A_1\ M : K[M/x]}\,\mathsf{famapp} \qquad \frac{\Gamma \vdash A_1 : \text{type} \qquad \Gamma, x : A_1 \vdash A_2 : \text{type}}{\Gamma \vdash \Pi x : A_1.\, A_2 : \text{type}}\,\mathsf{fampi}$$

Note that in the rule famapp, the free occurrence of x in K must be replaced by the object M. A very similar replacement takes place in the rule objapp.

$$\frac{\Sigma(c) = A}{\Gamma \vdash c : A}\,\mathsf{objcon} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A}\,\mathsf{objvar}$$

$$\frac{\Gamma \vdash A_1 : \text{type} \qquad \Gamma, x : A_1 \vdash M : A_2}{\Gamma \vdash \lambda x : A_1.\, M : \Pi x : A_1.\, A_2}\,\mathsf{objlam} \qquad \frac{\Gamma \vdash M_1 : \Pi x : A_2.\, A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash M_1\ M_2 : A_1[M_2/x]}\,\mathsf{objapp}$$

$$\frac{\Gamma \vdash M : A_1 \qquad A_1 \equiv A_2 \qquad \Gamma \vdash A_2 : \text{type}}{\Gamma \vdash M : A_2}\,\mathsf{typcnv}$$

The rules for signatures are standard. Note, that the type A, and kind K in the rules sigobj and sigfam are well-defined in the signature to the right of the declaration.

$$\frac{}{\vdash \cdot \text{ sig}}\,\mathsf{sigemp} \qquad \frac{\vdash \Sigma \text{ sig} \qquad \cdot \vdash A : \text{type}}{\vdash \Sigma, c : A \text{ sig}}\,\mathsf{sigobj} \qquad \frac{\vdash \Sigma \text{ sig} \qquad \cdot \vdash K \text{ kind}}{\vdash \Sigma, a : K \text{ sig}}\,\mathsf{sigfam}$$

Similarly, the validity of Γ is established by the following rules.

$$\frac{}{\vdash \cdot \text{ ctx}}\,\mathsf{ctxemp} \qquad \frac{\vdash \Gamma \text{ ctx} \qquad \Gamma \vdash A : \text{type}}{\vdash \Gamma, x : A \text{ ctx}}\,\mathsf{ctxobj}$$

Throughout any typing derivations of object, types and kinds, Γ must always remain valid. Instead of enforcing this condition locally, we push this well-typedness condition all the way to the axioms. Read from the bottom up, contexts always increase. Hence, we must extend kndtyp, famcon, objcon, and objvar with this additional premiss. In order not to clutter the rules, we leave these premisses implicit, too.

The logical framework contains two rules for definitional equality: the β- and the η-rule. As we have discussed in Section 2.3, the β-rule is helpful in the representation of substitution lemmas. In Chapter 4 we will see further applications of this hard-wired substitution principle of the framework.

$$\frac{}{(\lambda x : A.\, M_1)\ M_2 \equiv M_1[M_2/x]}\,\beta$$

$$\frac{}{(\lambda x : A.\, M\ x) \equiv M}\,\eta \qquad \text{x not free in } M$$

Similar to the observation in Section 2.2.1 these two rules can be applied to any subterm of an object, or a type, or even a kind. In order to make this kind of application entirely precise, we define a conversion relation, naturally, one for each level. First, the conversion relation is turned into an equivalence relation by building the reflexive, transitive, and symmetric closure.

$$\frac{}{K \equiv K}\,\mathsf{kndrefl} \qquad \frac{K_2 \equiv K_1}{K_1 \equiv K_2}\,\mathsf{kndsym} \qquad \frac{K_1 \equiv K_2 \qquad K_2 \equiv K_3}{K_1 \equiv K_3}\,\mathsf{kndtrans}$$

$$\frac{}{A \equiv A}\,\mathsf{famrefl} \qquad \frac{A_2 \equiv A_1}{A_1 \equiv A_2}\,\mathsf{famsym} \qquad \frac{A_1 \equiv A_2 \qquad A_2 \equiv A_3}{A_1 \equiv A_3}\,\mathsf{famtrans}$$

$$\frac{}{M \equiv M}\,\mathsf{objrefl} \qquad \frac{M_2 \equiv M_1}{M_1 \equiv M_2}\,\mathsf{objsym} \qquad \frac{M_1 \equiv M_2 \qquad M_2 \equiv M_3}{M_1 \equiv M_3}\,\mathsf{objtrans}$$

And second it is turned into a congruence relation $\equiv$; conversion can be applied to subterms.

$$\frac{A \equiv A'}{\Pi x : A.\, K \equiv \Pi x : A'.\, K}\,\mathsf{cngkndpil} \qquad \frac{K \equiv K'}{\Pi x : A.\, K \equiv \Pi x : A.\, K'}\,\mathsf{cngkndpir}$$

$$\frac{A \equiv A'}{A\ M \equiv A'\ M}\,\mathsf{cngfamappl} \qquad \frac{M \equiv M'}{A\ M \equiv A\ M'}\,\mathsf{cngfamappr}$$

$$\frac{A_1 \equiv A_1'}{\Pi x : A_1.\, A_2 \equiv \Pi x : A_1'.\, A_2}\,\mathsf{cngfampil} \qquad \frac{A_2 \equiv A_2'}{\Pi x : A_1.\, A_2 \equiv \Pi x : A_1.\, A_2'}\,\mathsf{cngfampir}$$

$$\frac{A \equiv A'}{\lambda x : A.\, M \equiv \lambda x : A'.\, M}\,\mathsf{cngobjlaml} \qquad \frac{M \equiv M'}{\lambda x : A.\, M \equiv \lambda x : A.\, M'}\,\mathsf{cngobjlamr}$$

$$\frac{M_1 \equiv M_1'}{M_1\ M_2 \equiv M_1'\ M_2}\,\mathsf{cngobjappl} \qquad \frac{M_2 \equiv M_2'}{M_1\ M_2 \equiv M_1\ M_2'}\,\mathsf{cngobjappr}$$

This concludes the formal presentation of the rules for the logical framework LF. The signature of Section 2.3 is in fact a LF signature after appropriate reconstruction of the types. More examples can be found in Section 2.5, where we encode the rewrite relations from Section 2.2, and in Chapter 4, where we will represent the Church-Rosser theorem based on an argument of parallel reduction.

2.4.3 Canonical Forms

In Section 2.3, we have seen that canonical forms are indispensable for the adequacy Lemma 2.3 and Lemma 2.4. Canonical forms are β-normal, η-long forms. Formally, this property is reflected in two mutually dependent judgments: the judgment about canonical forms and the judgment about atomic forms. Informally again, a canonical form M^c has the form $\lambda x_1 : A_1. \ldots \lambda x_n : A_n. M^a$ where M^a is atomic, that is, its head h is either a variable or a constant, and it has generally the following form: $h\ M_1^c\ \ldots\ M_n^c$, where the M_i^c's are canonical. To guarantee η-long forms, M^a is required to be of atomic type. As auxiliary judgments, we also need to formalize canonical types, which enforce that all objects occurring as arguments to type families in λ-labels are also canonical.

Judgments

Canonical objects:	$\Gamma \vdash M \Uparrow A$
Atomic objects:	$\Gamma \vdash M \downarrow A$
Canonical types:	$\Gamma \vdash A \Uparrow \mathsf{type}$
Atomic types:	$\Gamma \vdash A \downarrow K$

Rules The following rules define canonical objects, atomic object, canonical types and atomic types.

$$\frac{\Gamma \vdash A_1 \Uparrow \mathsf{type} \qquad \Gamma, x : A_1 \vdash M \Uparrow A_2}{\Gamma \vdash \lambda x : A_1. M \Uparrow \Pi x : A_1. A_2}\,\mathsf{canpi} \qquad \frac{\Gamma \vdash A \downarrow \mathsf{type} \qquad \Gamma \vdash M \downarrow A}{\Gamma \vdash M \Uparrow A}\,\mathsf{canatm}$$

$$\frac{\Gamma \vdash M \Uparrow A_1 \qquad A_1 \equiv A_2 \qquad \Gamma \vdash A_2 : \mathsf{type}}{\Gamma \vdash M \Uparrow A_2}\,\mathsf{cancnv}$$

$$\frac{\Sigma(c) = A}{\Gamma \vdash c \downarrow A}\,\mathsf{atmcon} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x \downarrow A}\,\mathsf{atmvar}$$

$$\frac{\Gamma \vdash M_1 \downarrow \Pi x : A_2. A_1 \qquad \Gamma \vdash M_2 \Uparrow A_2}{\Gamma \vdash M_1\ M_2 \downarrow A_1[M_2/x]}\,\mathsf{atmapp} \qquad \frac{\Gamma \vdash M \downarrow A_1 \qquad A_1 \equiv A_2 \qquad \Gamma \vdash A_2 : \mathsf{type}}{\Gamma \vdash M \downarrow A_2}\,\mathsf{atmcnv}$$

$$\frac{\Gamma \vdash A_1 \Uparrow \mathsf{type} \qquad \Gamma, x : A_1 \vdash A_2 \Uparrow \mathsf{type}}{\Gamma \vdash \Pi x : A_1. A_2 \Uparrow \mathsf{type}}\,\mathsf{cntpi} \qquad \frac{\Gamma \vdash A \downarrow \mathsf{type}}{\Gamma \vdash A \Uparrow \mathsf{type}}\,\mathsf{cntatm}$$

$$\frac{\Sigma(a) = K}{\Gamma \vdash a \downarrow K}\,\mathsf{attcon}$$

$$\frac{\Gamma \vdash A \downarrow \Pi x : A'. K \qquad \Gamma \vdash M \Uparrow A'}{\Gamma \vdash A\ M \downarrow K[M/x]}\,\mathsf{attapp} \qquad \frac{\Gamma \vdash A \downarrow K \qquad K \equiv K' \qquad \Gamma \vdash K\ \mathsf{kind}}{\Gamma \vdash A \downarrow K}\,\mathsf{attcnv}$$

2.4.4 Meta-Theory

The adequacy results from Section 2.3 depend crucially on one property of LF: Every LF object has a canonical form. Otherwise one could not carry out an argument by structural induction over the form of LF objects, which is necessary to establish that there is a one-to-one correspondence between derivations and objects in the type theory. In Chapter 4 we will make a very similar observation and in fact the entire formalism we present in Chapter 5 is based on this property: Every object defined in the logical framework has a *unique* canonical form, i.e. it is β-normal and η-long. The interested reader may study the proof in [HP99].

Theorem 2.6 (Canonical form theorem)

1. *If* $\Gamma \vdash M \Uparrow A$ *then* $\Gamma \vdash M : A$.

2. *For each object* M *such that* $\Gamma \vdash M : A$*, there exists a unique object* M' *such that* $M \equiv M'$ *and* $\Gamma \vdash M' \Uparrow A$*. Moreover,* M' *can be effectively computed.*

3. *For each object* A *such that* $\Gamma \vdash A : type$*, there exists a unique object* A' *such that* $A \equiv A'$ *and* $\Gamma \vdash A \Uparrow type$*. Moreover,* A' *can be effectively computed.*

Proof: see [HP99]. □

A direct corollary of the canonical form theorem is that each object has a unique type.

Corollary 2.7 (Uniqueness of typing)
If $\Gamma \vdash M : A_1$
and $\Gamma \vdash M : A_2$
then there exists a unique type A *s.t.* $A \equiv A_1 \equiv A_2$
and $\Gamma \vdash A \Uparrow type$
and $\Gamma \vdash M : A$

Proof: see [HHP93]. □

That each object has a canonical form and a unique canonical type provides the theoretical foundation of the theory and the logic development in the subsequent chapters in this thesis. The necessity to have canonical forms is absolutely essential, and it cannot be emphasized enough: one can only extend this work to logical frameworks, which possess these properties.

2.5 More Examples

The simply-typed λ-calculus in Section 2.2 is defined by its terms and its reduction relation. In particular, in Section 2.3 we have already discussed an adequate representation of well-typed terms. In order to show some more examples of how to represent a deductive systems, specified by its judgments and its inference rules, we address now the representation of the reduction relation. The judgment $e_1 \stackrel{1}{\longrightarrow} e_2$ is represented by a type family $\stackrel{1}{\longrightarrow}$ which we use as an infix operator.

$$\ulcorner e_1 \stackrel{1}{\longrightarrow} e_2 \urcorner = \ulcorner e_1 \urcorner \stackrel{1}{\longrightarrow} \ulcorner e_2 \urcorner$$

Note, that here again we are overloading notation in order to simplify the presentation. We use the same arrow for the informal and formal representation of the reduction relation; the reduction arrow must not be confused with the function arrow of LF.

Because of the elegant representation of variables of the simply-typed λ-calculus using higher-order abstract syntax, we can easily represent the `rbeta`-rule:

$$\ulcorner \dfrac{}{(\lambda x:\tau.e_1)\ e_2 \xrightarrow{1} e_1[e_2/x]}\,\mathsf{rbeta} \urcorner = \mathrm{rbeta}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1\urcorner)\ \ulcorner e_2\urcorner$$

$$: \mathrm{app}\ (\mathrm{lam}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1\urcorner))\ \ulcorner e_2\urcorner \xrightarrow{1} (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1\urcorner))\ \ulcorner e_2\urcorner$$
$$\text{where } \ulcorner x\urcorner = x$$

Note that on the right hand side of the equation we need not represent τ as argument to "rbeta"; it is implicitly represented through the type of $\ulcorner e_1\urcorner$ as is the type of $\ulcorner e_2\urcorner$. The representation of the rlam-rule is very similar.

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ e \xrightarrow{1} e'\end{array}}{\lambda x:\tau.e \xrightarrow{1} \lambda x:\tau.e'}\,\mathsf{rlam} \urcorner$$

$$= \mathrm{rlam}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e\urcorner)\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e'\urcorner)\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner \mathcal{D}\urcorner)$$
$$: \ \mathrm{lam}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e\urcorner) \xrightarrow{1} \mathrm{lam}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e'\urcorner)$$
$$\text{where } \ulcorner x\urcorner = x$$

Differently from the informal representation, we make the fact that x might occur free in $\mathcal{D}$ unambiguously explicit. The representation of $\mathcal{D}$ is parametric in x! The third argument to "rlam" has therefore the following type: $\Pi x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e\urcorner \xrightarrow{1} \ulcorner e'\urcorner$ where $\ulcorner x\urcorner = x$.

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ e_1 \xrightarrow{1} e_1'\end{array}}{e_1\ e_2 \xrightarrow{1} e_1'\ e_2}\,\mathsf{rapp}_1 \urcorner = \mathrm{rapp}_1\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1\urcorner)\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1;\urcorner)\ \ulcorner e_2\urcorner\ \ulcorner\mathcal{D}\urcorner$$

$$: \mathrm{app}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1\urcorner)\ \ulcorner e_2\urcorner \xrightarrow{1} \mathrm{app}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1'\urcorner)\ \ulcorner e_2\urcorner$$
$$\text{where } \ulcorner x\urcorner = x$$

Very similar to the encoding of rapp_1 is the rule rapp_2:

$$\ulcorner \dfrac{\begin{array}{c}\mathcal{D}\\ e_2 \xrightarrow{1} e_2'\end{array}}{e_1\ e_2 \xrightarrow{1} e_1\ e_2'}\,\mathsf{rapp}_2 \urcorner = \mathrm{rapp}_2\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1\urcorner)\ \ulcorner e_2\urcorner\ \ulcorner e_2'\urcorner\ \ulcorner\mathcal{D}\urcorner$$

$$: \mathrm{app}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1\urcorner)\ \ulcorner e_2\urcorner \xrightarrow{1} \mathrm{app}\ (\lambda x : \mathrm{term}\ \ulcorner\tau\urcorner.\ulcorner e_1\urcorner)\ \ulcorner e_2'\urcorner$$
$$\text{where } \ulcorner x\urcorner = x$$

The encoding of the single step reduction relation for the simply-typed λ-calculus is adequate, as one can easily verify by induction.

Lemma 2.8 (Adequacy of the representation of $\xrightarrow{1}$) *1. If $\mathcal{D} :: e_1 \xrightarrow{1} e_2$ which may rely on assumptions of the form $x_1 :: term\ \tau_1, \ldots, x_n :: term\ \tau_n$ then $\ulcorner\mathcal{D}\urcorner : \ulcorner e_1\urcorner \xrightarrow{1} \ulcorner e_2\urcorner$ which possibly contains variables of the form $x_1 : term\ \ulcorner\tau_1\urcorner, \ldots, x_n : term\ \ulcorner\tau_n\urcorner$.*

2. If $D : \ulcorner e_1\urcorner \xrightarrow{1} \ulcorner e_2\urcorner$ is canonical, possibly containing variables of the form $x_1 : term\ \ulcorner\tau_1\urcorner, \ldots, x_n : term\ \ulcorner\tau_n\urcorner$, then $D = \ulcorner\mathcal{D}\urcorner$ where $\mathcal{D} :: e_1 \xrightarrow{1} e_2$ which may rely on assumptions of the form $x_1 :: term\ \tau_1, \ldots, x_n :: term\ \tau_n$

For all other encodings in remainder of this thesis we will not write out the adequacy theorems explicitly any more. They always follow the same scheme. Omitting inferable arguments, we obtain as extension of the LF-signature from Section 2.3 the adequate encoding of the $\xrightarrow{1}$-relation.

$$
\begin{array}{rcl}
\xrightarrow{1} & : & \text{term } T \to \text{term } T \to \text{type} \\
\text{rbeta} & : & (\text{app } (\text{lam } E_1)\ E_2) \xrightarrow{1} E_1\ E_2 \\
\text{rlam} & : & (\Pi x : \text{term } T_1 .\, E\ x \xrightarrow{1} E'\ x) \\
 & & \to (\text{lam } E) \xrightarrow{1} (\text{lam } E') \\
\text{rapp}_1 & : & E_1 \xrightarrow{1} E_1' \\
 & & \to (\text{app } E_1\ E_2) \xrightarrow{1} (\text{app } E_1'\ E_2) \\
\text{rapp}_2 & : & E_2 \xrightarrow{1} E_2' \\
 & & \to (\text{app } E_1\ E_2) \xrightarrow{1} (\text{app } E_1\ E_2')
\end{array}
$$

By applying the same representation techniques discussed in this section, we further extend the signature by an encoding of the multi-step relation $\xrightarrow{*}$ and the conversion relation $\longleftrightarrow$.

$$
\begin{array}{rcl}
\xrightarrow{*} & : & \text{term } T \to \text{term } T \to \text{type} \\
\text{rid} & : & E \xrightarrow{*} E \\
\text{rstep} & : & E \xrightarrow{1} E' \\
 & & \to E' \xrightarrow{*} E'' \\
 & & \to E \xrightarrow{*} E'' \\
\longleftrightarrow & : & \text{term } T \to \text{term } T \to \text{type} \\
\text{rrefl} & : & E \longleftrightarrow E \\
\text{rred} & : & E \xrightarrow{*} E' \\
 & & \to E \longleftrightarrow E' \\
\text{rsymm} & : & E \longleftrightarrow E' \\
 & & \to E' \longleftrightarrow E \\
\text{rtrans} & : & E \longleftrightarrow E' \\
 & & \to E' \longleftrightarrow E'' \\
 & & \to E \longleftrightarrow E''
\end{array}
$$

2.6 Function Spaces

The function spaces, definable in the logical framework LF are different from function spaces application programmers are used to. In general, programming relies on features such as function definition by cases or if then else constructions to code specific applications. Those features are not supported by the logical framework. In fact, the operational meaning of LF includes only two operations: β-reduction and η-reduction. Hence, LF is not expressive enough to represent functions that decide if a given term is a β-redex.

$$\text{Boolean values:} \quad B \quad ::= \quad \top \mid \bot$$

Informally, the decision procedure can be defined by pattern-matching

$$\begin{array}{ll} \lambda E : \text{term}\, T. & \text{case } E \\ & \text{of } (\text{app } (\text{lam } E_1)\ E_2) \mapsto \top \\ & \quad \mid (\text{app } (\text{app } E_1\ E_2)) \mapsto \bot \\ & \quad \mid (\text{lam } E') \mapsto \bot \end{array}$$

and clearly, this function is cannot further normalized since its argument E is only given at run-time. Therefore, this function does *not* possess a canonical form in LF, and thus functions of this kind violate the adequacy requirement of the encoding.

Therefore we must distinguish the two function spaces from each other. One function space is the LF function space $A_1 \to A_2$, which contains all LF objects that map objects of type A_1 to objects of type A_2. Because of the canonical form theorem, functional LF objects of this type are inductively defined, and therefore, we call it *parametric*. In Section 2.3, for example, we have examined all functions of the type "term $T_1 \to$ term T_2". The body of each function is either a constant from the signature Σ, or a local parameter, applied to arguments.

We call the other function space *recursive*, because it permits function definition by cases and recursion. The question of how to arrange it so that the parametric and the recursive function space can safely coexist is one of the main contributions of this thesis. In essence, the nature of the problem is that there are too many recursive functions destroying our requirement for the existence of canonical forms. It has been shown that in the setting of a non-dependently typed framework (the simply-typed λ-calculus) one can express the recursive function space in terms of the parametric using a modality, which satisfies the properties of the modal logic S_4. We refer the interested reader to [DPS97, Lel98].

2.7 Summary

A logical framework is a formal system which represents deductive systems using type theory. Elegant representations of deductive systems that include variable concepts and appropriate substitution principles are facilitated by higher-order representation techniques. In order to guarantee the adequacy of encoding, each object in the logical framework must possess a canonical form. The logical framework LF [HHP93], which is the logical framework of choice for this thesis, supports higher-order representation techniques and has proven to be appropriate for the representation of many deductive systems from logics, programming languages, operational semantics, and many others [Pfe99].

Chapter 3

Reasoning

3.1 Introduction

The quality of any design can be drastically improved by specifying and verifying associated characteristic properties during the design process. For example, we expect that a typed programming language satisfies the type preservation property, i.e. that the evaluation of any well-typed program preserves types. Similarly, a calculus of inference rules for any logic must be consistent; if falsehood is derivable, typically any other formula is also derivable, a circumstance that invalidates consistency. Following [Gen35], the consistency of the sequent calculus for first-order intuitionistic logic for example, follows from a purely syntactical argument. Gentzen has shown that any derivation *with* cuts can be transformed into a derivation *without* cuts while providing evidence for exactly the same judgment. By inspection of the other inference rules, the consistency of first-order intuitionistic logic follows easily.

Therefore, good designs of deductive systems requires designers to reason about their properties. In particular, the overall goal of this thesis is to provide the necessary technology and tools to support and automate these reasoning tasks. More specific in this chapter we extend the example presented in Section 2.2 and develop as case study the proof of the Church-Rosser property in Section 3.2. Then we review previously proposed techniques to formalize meta-theoretic arguments about deductive systems, and discuss briefly how far these techniques can be automated in Section 3.3.

3.2 Church-Rosser Theorem

The Church-Rosser theorem for the simply-typed λ-calculus states that two convertible terms e_1, e_2 have a common reduct e' and two reductions from e_1 to e' and from e_2 to e'. This is property is easily visualized by the following diagram.

$$\begin{tikzcd} e_1 \arrow[rr, leftrightarrow] \arrow[dr, dotted, "*"'] & & e_2 \arrow[dl, dotted, "*"] \\ & e' & \end{tikzcd}$$

In this presentation we use solid arrows to represent given reductions, and dotted arrows for reductions whose existence is still to be shown. The goal of this section is to develop the Church-Rosser theorem for the notion of reduction defined in Section 2.2. The way we proceed is to introduce a new notion of reduction which we call *parallel* reduction as opposed to the other notion of reduction which we call *ordinary* reduction in order to keep them apart. The technique of using *parallel* reduction and parallel conversion for the proof of the Church-Rosser property goes back to Martin-Löf and Tait (see [Bar80]). We proceed as follows: First, we take the ordinary reduction relation defined in Section 2.2 and prove some simple properties. Then, we introduce the notion of parallel reduction, show the Church-Rosser property and eventually finish with an equivalence proof between parallel and ordinary reduction. But the reader should be alert: The main goal of this section is not the theory itself, but rather the development of an example with which we can explain and test the automated reasoning engine we develop in this thesis. The argument itself is well-known, and we refer the interested reader to a further and more detailed explanation in [Pfe93].

3.2.1 Properties of Ordinary Reduction

We begin now with two easy proofs about ordinary reductions: First we show that the multi-step reduction is transitive, and second that all inference rules for the single-step reduction relation are still valid, even if we exchange the single-step reduction arrow $\xrightarrow{1}$ by the multi-step reduction arrow $\xrightarrow{*}$.

More precisely, the first lemma expresses that two multi-step reduction with a common term e'' at the end of the first and the beginning of the second can be merged. This is a very basic and easy meta-theorem. For example, it follows by induction over the reduction ending in e''. By careful analysis of the inference rules, we notice that the last applied inference rule is either the identity reduction rid or the step case rstep. In the latter case, one appeal of the induction hypothesis provides the right reduction derivation from which the necessary reduction can be constructed.

Lemma 3.1 (Transitivity of $\xrightarrow{*}$) *If $\mathcal{D}_1 :: e \xrightarrow{*} e'$ and $\mathcal{D}_2 :: e' \xrightarrow{*} e''$ then $e \xrightarrow{*} e''$.*

Proof: by induction over $\mathcal{D}_1$:

Case: $\mathcal{D}_1 = \dfrac{}{e \xrightarrow{*} e}\,\mathsf{rid}$

$\mathcal{D}_2 :: e \xrightarrow{*} e''$ by assumption

Case: $\mathcal{D}_1 = \dfrac{\begin{array}{c}\mathcal{D}_1' \\ e \xrightarrow{1} e'''\end{array} \qquad \begin{array}{c}\mathcal{D}_1'' \\ e''' \xrightarrow{*} e'\end{array}}{e \xrightarrow{*} e'}\,\mathsf{rstep}$

$\mathcal{P} :: e''' \xrightarrow{*} e''$ by i.h. on $\mathcal{D}_1''$ and $\mathcal{D}_2$
$\mathcal{Q} :: e \xrightarrow{*} e''$ by rstep on $\mathcal{D}_1', \mathcal{P}$

□

The proof of Lemma 3.1 visualizes the three most basic operations used when reasoning about deductive systems. The first technique is induction. It means, that the different proof cases may take advantage of the fact that the induction hypothesis holds for any smaller derivations according to some well-founded ordering. In particular, since the argument is by structural induction, the induction hypothesis holds for all subderivations of the given derivation, and hence the well-founded ordering is simply the subderivation ordering. The second technique is case analysis: Derivations can be distinguished by the last applied rule. The third and last technique is the use of other inference rules to reconstruct the desired result derivations (last step, in the rstep case).

The second lemma, generalizes the rules from Section 2.2.1. It states, that the multi-step reduction can be manipulated with the same rules that define the single step reduction when one exchanges the $\xrightarrow{1}$ symbol by the $\xrightarrow{*}$ symbol. The rules are admissible, because they require a reorganization of the premiss derivations in order to arrive at the conclusion.

Lemma 3.2 (Admissible rules)

1. *If* $\mathcal{D} :: e \xrightarrow{*} e'$ *then* $\lambda x : \tau_2 . e \xrightarrow{*} \lambda x : \tau_2 . e'$
2. *If* $\mathcal{D} :: e_1 \xrightarrow{*} e_1'$ *then* $e_1\ e_2 \xrightarrow{*} e_1'\ e_2$
3. *If* $\mathcal{D} :: e_2 \xrightarrow{*} e_2$ *then* $e_1\ e_2 \xrightarrow{*} e_1\ e_2'$

Proof: by structural induction over $\mathcal{D}$

1. **Case:** $\mathcal{D} = \dfrac{}{e \xrightarrow{*} e}\ \mathsf{rid}$

   $\lambda x : \tau_2 . e \xrightarrow{*} \lambda x : \tau_2 . e$ — by rid

   Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ e \xrightarrow{1} e''\end{array} \qquad \begin{array}{c}\mathcal{D}'' \\ e'' \xrightarrow{*} e'\end{array}}{e \xrightarrow{*} e'}\ \mathsf{rstep}$

   $\mathcal{P}_1 :: \lambda x : \tau_2 . e'' \xrightarrow{*} \lambda x : \tau_2 . e'$ — by i.h.(1) on $\mathcal{D}''$
   $\mathcal{P}_2 :: \lambda x : \tau_2 . e \xrightarrow{1} \lambda x : \tau_2 . e''$ — by rlam on $\mathcal{D}'$
   $\mathcal{Q} :: \lambda x : \tau_2 . e \xrightarrow{*} \lambda x : \tau_2 . e'$ — by rstep on $\mathcal{P}_2, \mathcal{P}_1$

2. **Case:** $\mathcal{D} = \dfrac{}{e_1 \xrightarrow{*} e_1}\ \mathsf{rid}$

   $e_1\ e_2 \xrightarrow{*} e_1\ e_2$ — by rid

   Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ e_1 \xrightarrow{1} e_1''\end{array} \qquad \begin{array}{c}\mathcal{D}'' \\ e_1'' \xrightarrow{*} e_1'\end{array}}{e_1 \xrightarrow{*} e_1'}\ \mathsf{rstep}$

$\mathcal{P}_1 :: e_1''\ e_2 \xrightarrow{*} e_1'\ e_2$ by i.h.(2) on $\mathcal{D}''$
$\mathcal{P}_2 :: e_1\ e_2 \xrightarrow{1} e_1''\ e_2$ by rapp_1 on $\mathcal{D}'$
$\mathcal{Q} :: e_1\ e_2 \xrightarrow{1} e_1'\ e_2$ by rstep on $\mathcal{P}_1, \mathcal{P}_2$

3. **Case:** $\mathcal{D} = \dfrac{}{e_2 \xrightarrow{*} e_2}\ \mathsf{rid}$

$e_1\ e_2 \xrightarrow{*} e_1\ e_2$ by rid

Case: $\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}' & \mathcal{D}'' \\ e_2 \xrightarrow{1} e_2'' & e_2'' \xrightarrow{*} e_2' \end{array}}{e_2 \xrightarrow{*} e_2'}\ \mathsf{rstep}$

$\mathcal{P}_1 :: e_1\ e_2'' \xrightarrow{*} e_1\ e_2'$ by i.h.(3) on $\mathcal{D}''$
$\mathcal{P}_2 :: e_1\ e_2 \xrightarrow{1} e_1\ e_2''$ by rapp_2 on $\mathcal{D}'$
$\mathcal{Q} :: e_1\ e_2 \xrightarrow{1} e_1\ e_2'$ by rstep on $\mathcal{P}_1, \mathcal{P}_2$

□

We observe, that we have used the same principles for the proof of Lemma 3.2 as we did to prove Lemma 3.1. In some sense, the third operation is slightly more general than before. In the proof of Lemma 3.1 only one rule of the inference system is used to construct the existential derivation, whereas here several are used. In summary, we have discovered three recurring proof principles:

1. Appeals to the induction hypothesis to smaller derivations according to some well-founded ordering on derivations

2. Case analysis over the last applied rule of a derivation.

3. Construction of desired derivations from other rules, assumed derivations, and result derivations of appeals to the induction hypothesis.

These three proof principles correspond directly to operations which are implemented in the automated theorem prover described in Chapter 8.

3.2.2 Parallel Reduction

In order to prove the Church-Rosser theorem for ordinary reduction from Section 2.2 we follow an idea of Martin-Löf and Tait (see [Bar80]) and use the method of parallel reduction. This method is based the following fundamental idea: Instead of reducing one β-redex after the other in sequence as with ordinary reduction, parallel reduction is defined in a way that several β-redices may be reduced simultaneously. The reduction relation is defined by the following three rules.

$$\dfrac{\begin{array}{c}\overline{x \stackrel{1}{\Longrightarrow} x}\,u \\ \vdots \\ e_1 \stackrel{1}{\Longrightarrow} e_1' \qquad e_2 \stackrel{1}{\Longrightarrow} e_2'\end{array}}{(\lambda x : \tau.e_1)\ e_2 \stackrel{1}{\Longrightarrow} e_1'[e_2'/x]}\,\mathsf{pbeta}^u \qquad \dfrac{\begin{array}{c}\overline{x \stackrel{1}{\Longrightarrow} x}\,u \\ \vdots \\ e \stackrel{1}{\Longrightarrow} e'\end{array}}{\lambda x : \tau.e \stackrel{1}{\Longrightarrow} \lambda x : \tau.e'}\,\mathsf{plam}^u$$

$$\dfrac{e_1 \stackrel{1}{\Longrightarrow} e_1' \qquad e_2 \stackrel{1}{\Longrightarrow} e_2'}{e_1\ e_2 \stackrel{1}{\Longrightarrow} e_1'\ e_2'}\,\mathsf{papp}$$

The rules pbeta and plam are hypothetical because they discharge the assumption labeled u. This is one of the crucial differences between this kind of reduction and ordinary reduction: With ordinary reductions, variables were never reduced whereas here they are. In fact they reduce to themselves. Reasoning with assumptions has consequences makes the formulation of lemmas and theorems more difficult; "$\mathcal{D}$ is a parallel reduction from e to e'" is a rather imprecise statement because nothing is said about the context in which this statement is supposed to be true. Since automated proof construction is the goal of this thesis, we have to be painstakingly precise. We say that "$\mathcal{D}$ is a *closed* parallel reduction from e to e'" if this statement does not rely on any other additional assumptions. On the other hand, we say that "$\mathcal{D}$ is a *open* parallel reduction from e to e'", if the context is not necessarily empty. In this situation, e, e' may contain variables $x_1, ..., x_n$ each of which reduces to itself: $x_i \stackrel{1}{\Longrightarrow} x_i$.

Following the example of ordinary reductions, we generalize the *single-step* parallel reduction relation (that may execute several β-reduction steps simultaneously) to a *multi-step* parallel reduction relation and for which we write $e \stackrel{*}{\Longrightarrow} e'$ if e parallel reduces in several steps to e'.

$$\dfrac{}{e \stackrel{*}{\Longrightarrow} e}\,\mathsf{pid} \qquad \dfrac{e \stackrel{1}{\Longrightarrow} e' \qquad e' \stackrel{*}{\Longrightarrow} e''}{e \stackrel{*}{\Longrightarrow} e''}\,\mathsf{pstep}$$

Next, we define the notion of parallel conversion between two terms e and e' . Intuitively, parallel conversion generalizes the multi-step parallel reduction relation in the same way as ordinary conversion generalizes the ordinary reduction relation (see Section 2.2). We write $e \Longleftrightarrow e'$, if there exists a sequence of intermediate terms $e_1, \ldots, e_n$, s.t.

$$e = e_1 \stackrel{*}{\Longrightarrow} e_2 \stackrel{*}{\Longleftarrow} e_3 \stackrel{*}{\Longrightarrow} \cdots \stackrel{*}{\Longrightarrow} e_{n-2} \stackrel{*}{\Longleftarrow} e_{n-1} \stackrel{*}{\Longrightarrow} e_n = e'$$

keeping in mind that $\stackrel{*}{\Longleftarrow}$ is not a new reduction relation but simply an alternative visual presentation of $\stackrel{*}{\Longrightarrow}$.

$$\dfrac{e \stackrel{*}{\Longrightarrow} e'}{e \Longleftrightarrow e'}\,\mathsf{pred} \qquad \dfrac{e' \stackrel{*}{\Longrightarrow} e}{e \Longleftrightarrow e'}\,\mathsf{pexp} \qquad \dfrac{e \Longleftrightarrow e' \qquad e' \Longleftrightarrow e''}{e \Longleftrightarrow e''}\,\mathsf{ptrans}$$

Applying the techniques presented in the previous chapter, we can now give an LF signature in Figure 3.1, which is an adequate encoding the three parallel reduction rules introduced in this section.

Lemma 3.3 (Adequacy of the presentation of parallel reduction)

$$
\begin{array}{rcl}
\overset{1}{\Longrightarrow} & : & \text{term } T \to \text{term } T \to \text{type.} \\
\text{pbeta} & : & (\Pi x : \text{term } T.\, x \overset{1}{\Longrightarrow} x \to e_1\; x \overset{1}{\Longrightarrow} e_1'\; x) \\
& & \to e_2 \overset{1}{\Longrightarrow} e_2' \\
& & \to (\text{app } (\text{lam } e_1)\; e_2) \overset{1}{\Longrightarrow} e_1'\; e_2' \\
\text{papp} & : & e_1 \overset{1}{\Longrightarrow} e_1' \\
& & \to e_2 \overset{1}{\Longrightarrow} e_2' \\
& & \to (\text{app } e_1\; e_2) \overset{1}{\Longrightarrow} (\text{app } e_1'\; e_2') \\
\text{plam} & : & (\Pi x : \text{term } T.\, x \overset{1}{\Longrightarrow} x \to e\; x \overset{1}{\Longrightarrow} e'\; x) \\
& & \to \text{lam } e \overset{1}{\Longrightarrow} \text{lam } e' \\
\overset{*}{\Longrightarrow} & : & \text{term } T \to \text{term } T \to \text{type} \\
\text{pid} & : & e \overset{*}{\Longrightarrow} e \\
\text{pstep} & : & e \overset{1}{\Longrightarrow} e' \\
& & \to e' \overset{*}{\Longrightarrow} e'' \\
& & \to e \overset{*}{\Longrightarrow} e'' \\
\Longleftrightarrow & : & \text{term } T \to \text{term } T \to \text{type} \\
\text{pred} & : & e \overset{*}{\Longrightarrow} e' \\
& & \to e \Longleftrightarrow e' \\
\text{pexp} & : & e \overset{*}{\Longrightarrow} e' \\
& & \to e' \Longleftrightarrow e \\
\text{ptrans} & : & e \Longleftrightarrow e' \\
& & \to e' \Longleftrightarrow e'' \\
& & \to e \Longleftrightarrow e''
\end{array}
$$

Figure 3.1: LF encoding of parallel reduction and parallel conversion (extends Figure 2.2)

1. *If* $\mathcal{D} :: e_1 \overset{1}{\Longrightarrow} e_2$ *which may rely on assumptions of the form* $x_1 :: \mathit{term}\ \tau_1, u_1 :: x_1 \overset{1}{\Longrightarrow} x_1, \ldots, x_n :: \mathit{term}\ \tau_n, u_n :: x_n \overset{1}{\Longrightarrow} x_n$ *then* $\ulcorner\mathcal{D}\urcorner : \ulcorner e_1 \urcorner \overset{1}{\Longrightarrow} \ulcorner e_2 \urcorner$ *which possibly contains variables of the form* $x_1 : \mathit{term}\ulcorner\tau_1\urcorner, u_1 : x_1 \overset{1}{\Longrightarrow} x_1, \ldots, x_n : \mathit{term}\ulcorner\tau_n\urcorner, u_n : x_n \overset{1}{\Longrightarrow} x_n$.

2. *If* $D : \ulcorner e_1 \urcorner \overset{1}{\longrightarrow} \ulcorner e_2 \urcorner$ *is canonical, possibly containing variables of the form* $x_1 : \mathit{term}\ulcorner\tau_1\urcorner, u_1 : x_1 \overset{1}{\Longrightarrow} x_1, \ldots, x_n : \mathit{term}\ulcorner\tau_n\urcorner, u_n : x_n \overset{1}{\Longrightarrow} x_n$, *then* $D = \ulcorner\mathcal{D}\urcorner$ *where* $\mathcal{D} :: e_1 \overset{1}{\longrightarrow} e_2$ *which may rely on assumptions of the form* $x_1 :: \mathit{term}\ \tau_1, u_1 :: x_1 \overset{1}{\Longrightarrow} x_1, \ldots, x_n :: \mathit{term}\ \tau_n, u_n :: x_n \overset{1}{\Longrightarrow} x_n$

3.2.3 Properties of Parallel Reduction

In this section we show the Church-Rosser theorem for parallel reduction. The theorems and proofs in this section are particularly important especially for the subsequent chapters, because they reveal the issues associated with reasoning about open derivations, that is, derivations which my be valid in terms of additional assumption. Recall, that we call a derivation closed, if no additional assumptions are used.

Used in the proof of one of the subsequent lemmas is the property, that the parallel reduction relation is reflexive. What we want to show is that for every term e, there exists a reduction $\mathcal{Q} :: e \stackrel{1}{\Longrightarrow} e$. In a first proof attempt one may assume that e is closed.

Lemma 3.4 (Reflexivity of $\stackrel{1}{\Longrightarrow}$, Version I) *For any closed term e, $e \stackrel{1}{\Longrightarrow} e$.*

This lemma is not directly provable in its current formulation by structural induction. To see why consider the case that e's outermost constructor is an abstraction and not an application. e has hence the form

$$\cfrac{\begin{array}{c}\cfrac{}{\text{term } \tau_1}\,x \\ e' \\ \text{term } \tau_2\end{array}}{\text{term } (\tau_1 \to \tau_2)}\,\mathsf{lam}^x$$

And indeed, the induction hypothesis is not general enough to conclude that $e' \stackrel{1}{\Longrightarrow} e'$. Obviously e' must be closed for the induction hypothesis to apply, but it is not. Therefore we must generalize the induction hypothesis in such a way, that it also applies to open terms e. In the second attempt we try the obvious: e can also depend on variables x_1 :: term $\tau_1, \ldots, x_n$ :: term τ_n:

.4

Lemma 3.4 (Reflexivity of $\stackrel{1}{\Longrightarrow}$, Version II) *For any term e, which is open in the sense that it may depend on assumptions x_1 :: term $\tau_1, \ldots, x_n$ :: term τ_n, there exists a derivation of $e \stackrel{1}{\Longrightarrow} e$.*

Strangely enough, this formulation of the lemma is still not general enough! To see why, consider $e = x_i$: The lemma should yield that $x_i \stackrel{1}{\Longrightarrow} x_i$, but how? There is no rule from the signature we could apply and there are no assumptions $x_i \stackrel{1}{\Longrightarrow} x_i$. The solution to the problem is to treat the reduction rules $x_i \stackrel{1}{\Longrightarrow} x_i$ in the same way as we treat assumptions. We must set the stage in such a way, that in addition to the parameter assumptions x_1 :: term $\tau_1, \ldots, x_n$ :: term τ_n also the following assumptions

$$\cfrac{}{x_1 \stackrel{1}{\Longrightarrow} x_1}\,u_1 \qquad \ldots \qquad \cfrac{}{x_n \stackrel{1}{\Longrightarrow} x_n}\,u_n$$

are available which we as usual abbreviate as list by $u_1 :: x_1 \stackrel{1}{\Longrightarrow} x_1, \ldots, u_n :: x_n \stackrel{1}{\Longrightarrow} x_n$. For a better conceptual understanding we pair the declaration of x_i and the correspond assumption u_i. Not too surprisingly any more, the reflexivity lemma is now provable in this generality.

.4

Lemma 3.4 (Reflexivity of $\stackrel{1}{\Longrightarrow}$, Version III) *Consider the situation where a list of the following assumptions is present*

$$x_1 :: \mathit{term}\ \tau_1, u_1 :: x_1 \stackrel{1}{\Longrightarrow} x_1, \ldots, x_n :: \mathit{term}\ \tau_n, u_n :: x_n \stackrel{1}{\Longrightarrow} x_n$$

Then for any well-typed term e, there exists a derivation of $e \stackrel{1}{\Longrightarrow} e$.

Proof: by structural induction on e:

Case: $e = \dfrac{}{\text{term } \tau_i}\, x_i$

$u_i :: x_i \stackrel{1}{\Longrightarrow} x_i$ by assumption

Case: $e = \dfrac{\begin{array}{c} \dfrac{}{\text{term } \tau_1}\, x_{n+1} \\ e' \\ \text{term } \tau_2 \end{array}}{\text{term } (\tau_1 \to \tau_2)}\, \mathsf{lam}^{x_{n+1}}$

Assume $x_{n+1} :: \text{term } \tau_1$
Assume $u_{n+1} :: x_{n+1} \stackrel{1}{\Longrightarrow} x_{n+1}$
$\mathcal{P} :: e' \stackrel{1}{\Longrightarrow} e'$ by i.h. on e'
$\mathcal{Q} :: \lambda x_{n+1} : \text{term } \tau_1 .\, e' \stackrel{1}{\Longrightarrow} \lambda x_{n+1} : \text{term } \tau_1 .\, e'$ by rule plam on $\mathcal{P}$

Case: $e = \dfrac{\begin{array}{c} e_1 \\ \text{term } (\tau_2 \to \tau_1) \end{array} \quad \begin{array}{c} e_2 \\ \text{term } \tau_2 \end{array}}{\text{term } \tau_1}\, \mathsf{app}$

$\mathcal{P}_1 :: e_1 \stackrel{1}{\Longrightarrow} e_1$ by i.h. on e_1
$\mathcal{P}_2 :: e_2 \stackrel{1}{\Longrightarrow} e_2$ by i.h. on e_2
$\mathcal{Q} :: \text{app } e_1\ e_2 \stackrel{1}{\Longrightarrow} \text{app } e_1\ e_2$ by rule papp on $\mathcal{P}_1, \mathcal{P}_2$

□

Note, that the proof works only in the situation where we have exactly the assumptions $x_1, u_1, ..., x_n, u_n$ if we ignore unrelated assumptions for now. If there are more, the proof is not a proof: some cases may not be covered. If there are less, the induction hypothesis might not be applicable. Without making it really precise, we want the reader to notice that the list of assumptions is very regular in structure. It is made out of basic building blocks of the form:

$$\rho ::= x :: \text{term } \tau, u :: x \stackrel{1}{\Longrightarrow} x \qquad \text{where } \tau \text{ is some type} \tag{3.1}$$

and the assumption lists can be inductively described by

$$\Phi \quad ::= \quad \cdot \mid \Phi, \rho$$

where variables x, u are α-converted to avoid duplicates. If we refer to the LF signature as a static description of the *world* that summarizes all inference rules, Φ is a dynamic extension of the world because it introduces new parameters. In addition, the proof of Lemma 3.4 also motivates a new meta-proving operation; in the case of lam we extend the current world by two new parameters x_{n+1} and u_{n+1}. All other proof principles used in this proof have already been discussed.

When reasoning informally about deductive systems, these assumptions stay in general hidden. Their regularity is tremendously important in this work, and it is thoroughly analyzed and formalized in Chapter 4. Lemma 3.4 is an explicit version of Lemma 3 in [Pfe93].

Following the sequence of lemmas presented in [Pfe93], we generalize each lemma to the appropriate level of detail in order to motivate the design of our system in Chapter 4. The transitivity lemma for parallel reductions for example is provable in a setting where $\mathcal{D} :: e \stackrel{*}{\Longrightarrow} e'$ are closed, which raises the question if this is general enough? In other words, the degree of generality of a lemma does not only depend on its proof, but it also depends on the generality of the lemma for where it is used. A transitivity property for closed derivations cannot be applied to derivations which are open. On the one hand, this sounds trivial, but on the other, there is a whole theory of which proof can appeal to what lemma, which we discuss in detail in Section 5.7.2. Nevertheless, we prove this lemma in more generality. For all the proofs following below, we let Φ describe dynamic extensions to the world, as defined above.

$$\Phi = x_1 :: \text{term } \tau_1, u_1 :: x_1 \stackrel{1}{\Longrightarrow} x_1, \ldots, x_n :: \text{term } \tau_n, u_n :: x_n \stackrel{1}{\Longrightarrow} x_n$$

Lemma 3.5 (Transitivity of $\stackrel{*}{\Longrightarrow}$) *Let Φ be the dynamic extension of the world. If $\mathcal{D}_1 :: e \stackrel{*}{\Longrightarrow} e'$ and $\mathcal{D}_2 :: e' \stackrel{*}{\Longrightarrow} e''$ are closed then $e \stackrel{*}{\Longrightarrow} e''$.*

Proof: by structural induction over $\mathcal{D}_1$:

Case: $\mathcal{D}_1 = \dfrac{}{x \stackrel{*}{\Longrightarrow} x}\,u$

$\mathcal{D}_2 :: x \stackrel{*}{\Longrightarrow} e''$ by assumption

Case: $\mathcal{D}_1 = \dfrac{}{e \stackrel{*}{\Longrightarrow} e}\,\mathsf{pid}$

$\mathcal{D}_2 :: e \stackrel{*}{\Longrightarrow} e''$ by assumption

Case: $\mathcal{D}_1 = \dfrac{\begin{array}{c}\mathcal{D}'_1 \\ e \stackrel{1}{\Longrightarrow} e'''\end{array} \qquad \begin{array}{c}\mathcal{D}''_1 \\ e''' \stackrel{*}{\Longrightarrow} e'\end{array}}{e \stackrel{*}{\Longrightarrow} e'}\,\mathsf{pstep}$

$\mathcal{P} :: e''' \stackrel{*}{\Longrightarrow} e''$ by i.h. on $\mathcal{D}''_1, \mathcal{D}_2$
$\mathcal{Q} :: e \stackrel{*}{\Longrightarrow} e''$ by pstep on $\mathcal{D}'_1, \mathcal{P}$

□

The following sequence of lemmas leads to the main result of this section: the parallel reduction relation possesses the Church-Rosser property. We present the lemmas in the same sequence as in [Pfe93], but enrich the formulation by information if the derivations are closed, or if they are open.

Lemma 3.6 (Substitution lemma) *Let Φ be the dynamic extension of the world. If*

$$\begin{array}{c} \overline{y \stackrel{1}{\Longrightarrow} y}\,v \\ \mathcal{D} \\ e_1 \stackrel{1}{\Longrightarrow} e_1' \end{array}$$

and $\mathcal{E} :: e_2 \stackrel{1}{\Longrightarrow} e_2'$ then exists a reduction $e_1[e_2/y] \stackrel{1}{\Longrightarrow} e_1'[e_2'/y]$.

Proof: by structural induction on $\mathcal{D}$.

Case: $\mathcal{D} = \dfrac{}{x \stackrel{1}{\Longrightarrow} x}\,u$ (where $x ::$ term $\tau, u :: x \stackrel{1}{\Longrightarrow} x \in \Phi$ and $x \neq y$)

$\mathcal{E} :: x \stackrel{1}{\Longrightarrow} x$ by assumption

Case: $\mathcal{D} = \dfrac{}{y \stackrel{1}{\Longrightarrow} y}\,v$

$\mathcal{E} :: e_2 \stackrel{1}{\Longrightarrow} e_2'$ by assumption

Case: $\mathcal{D} = \dfrac{\begin{array}{c} \overline{x \stackrel{1}{\Longrightarrow} x}\,u \\ \mathcal{D}_1 \\ e_3 \stackrel{1}{\Longrightarrow} e_3' \end{array} \qquad \begin{array}{c} \mathcal{D}_2 \\ e_4 \stackrel{1}{\Longrightarrow} e_4' \end{array}}{(\lambda x : \tau.e_3)\ e_4 \stackrel{1}{\Longrightarrow} e_3'[e_4'/x]}\,\mathsf{pbeta}^u$

Extend Φ by $x ::$ term $\tau, u :: x \stackrel{1}{\Longrightarrow} x$ to Φ'

$\mathcal{P}_1 :: e_3[e_2/y] \stackrel{1}{\Longrightarrow} e_3'[e_2'/y]$ by i.h. on $\mathcal{D}_1$ in Φ'

$\mathcal{P}_2 :: e_4[e_2/y] \stackrel{1}{\Longrightarrow} e_4'[e_2'/y]$ by i.h. on $\mathcal{D}_2$ in Φ

$\mathcal{Q} :: (\lambda x : \tau.e_3[e_2/y])\ e_4[e_2/y] \stackrel{1}{\Longrightarrow} e_3'[e_2'/y][e_4'[e_2'/y]/x]$ by rule pbeta^u on $\mathcal{P}_1, \mathcal{P}_2$

$\mathcal{Q} :: ((\lambda x : \tau.e_3)\ e_4)[e_2/y] \stackrel{1}{\Longrightarrow} (e_3'[e_4'/x])[e_2'/y]$ by Definition substitution

Case: $\mathcal{D} = \dfrac{\begin{array}{c} \overline{x \stackrel{1}{\Longrightarrow} x}\,u \\ \mathcal{D}_1 \\ e \stackrel{1}{\Longrightarrow} e' \end{array}}{\lambda x : \tau.e \stackrel{1}{\Longrightarrow} \lambda x : \tau.e'}\,\mathsf{plam}^u$

Extend Φ by $x :: \mathrm{term}\ \tau, u :: x \stackrel{1}{\Longrightarrow} x$ to Φ'

$\mathcal{P}_1 :: e[e_2/y] \stackrel{1}{\Longrightarrow} e'[e_2'/y]$ by i.h. on $\mathcal{D}_1$ in Φ'

$\mathcal{Q} :: \lambda x : \tau.e[e_2/y] \stackrel{1}{\Longrightarrow} \lambda x : \tau.e'[e_2'/y]$ by rule plam^u on $\mathcal{P}_1$

$\mathcal{Q} :: (\lambda x : \tau.e)[e_2/y] \stackrel{1}{\Longrightarrow} (\lambda x : \tau.e')[e_2'/y]$ by Definition substitution

Case: $\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ e_3 \stackrel{1}{\Longrightarrow} e_3' & e_4 \stackrel{1}{\Longrightarrow} e_4' \end{array}}{e_3\ e_4 \stackrel{1}{\Longrightarrow} e_3'\ e_4'}\ \mathsf{papp}$

$\mathcal{P}_1 :: e_3[e_2/y] \stackrel{1}{\Longrightarrow} e_3'[e_2'/y]$ by i.h. on $\mathcal{D}_1$ in Φ

$\mathcal{P}_2 :: e_4[e_2/y] \stackrel{1}{\Longrightarrow} e_4'[e_2'/y]$ by i.h. on $\mathcal{D}_2$ in Φ

$\mathcal{Q} :: (e_3[e_2/y])\ (e_4[e_2/y]) \stackrel{1}{\Longrightarrow} (e_3'[e_2'/y])\ (e_4'[e_2'/y])$ by rule papp on $\mathcal{P}_1, \mathcal{P}_2$

$\mathcal{Q} :: (e_3\ e_4)[e_2/y] \stackrel{1}{\Longrightarrow} (e_3'\ e_4')[e_2'/y]$ by definition substitution

$\square$

By careful inspection we can determine that the only four proof principles used in this proof are case analysis, appeals to the induction hypothesis, construction of witness objects from rules and assumptions, and dynamic extensions of the world. We continue this presentation with the proof of the diamond lemma for parallel reduction which shows clearly how difficult it is to argue that all cases are covered.

Lemma 3.7 (Diamond lemma) *Let Φ be the dynamic extension of the world. If $\mathcal{D}^l :: e \stackrel{1}{\Longrightarrow} e^l$ and $\mathcal{D}^r :: e \stackrel{1}{\Longrightarrow} e^r$ then there exists a common reduct e', such that $\mathcal{R}^l :: e^l \stackrel{1}{\Longrightarrow} e'$ and $\mathcal{R}^r :: e^r \stackrel{1}{\Longrightarrow} e'$.*

$$\begin{tikzcd} & e \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\ e^l \arrow[dr, dotted, "\mathcal{R}^l"'] & & e^r \arrow[dl, dotted, "\mathcal{R}^r"] \\ & e' & \end{tikzcd}$$

Proof: by simultaneous structural induction over $\mathcal{D}^l$ and $\mathcal{D}^r$.

Case: $\mathcal{D}^l = \dfrac{}{x \stackrel{1}{\Longrightarrow} x}\ u \qquad \mathcal{D}^r = \dfrac{}{x \stackrel{1}{\Longrightarrow} x}\ u \qquad$ (where $x :: \mathrm{term}\ \tau, u :: x \stackrel{1}{\Longrightarrow} x \in \Phi$)

$e' = x$ by assumption

$\mathcal{R}^l = \mathcal{R}^r = u$ by assumption

$$\begin{tikzcd}[column sep=small]
& x \arrow[dl, "u"'] \arrow[dr, "u"] & \\
x \arrow[dr, "u"'] & & x \arrow[dl, "u"] \\
& x &
\end{tikzcd}$$

Case: $$\mathcal{D}^l = \dfrac{\begin{array}{c}\dfrac{}{x \stackrel{1}{\Longrightarrow} x}\,u \\ \mathcal{D}_1^l \\ e_1 \stackrel{1}{\Longrightarrow} e_1^l\end{array} \qquad \begin{array}{c}\mathcal{D}_2^l \\ e_2 \stackrel{1}{\Longrightarrow} e_2^l\end{array}}{(\lambda x:\tau.e_1)\ e_2 \stackrel{1}{\Longrightarrow} e_1^l[e_2^l/x]}\,\mathsf{pbeta}^u \qquad \mathcal{D}^r = \dfrac{\begin{array}{c}\dfrac{}{x \stackrel{1}{\Longrightarrow} x}\,u \\ \mathcal{D}_1^r \\ e_1 \stackrel{1}{\Longrightarrow} e_1^r\end{array} \qquad \begin{array}{c}\mathcal{D}_2^r \\ e_2 \stackrel{1}{\Longrightarrow} e_2^r\end{array}}{(\lambda x:\tau.e_1)\ e_2 \stackrel{1}{\Longrightarrow} e_1^r[e_2^r/x]}\,\mathsf{pbeta}^u$$

Extend Φ by $x : \text{term } \tau, u :: x \stackrel{1}{\Longrightarrow} x$ to Φ'
There exists an e_1'
$\mathcal{P}_1 :: e_1^l \stackrel{1}{\Longrightarrow} e_1'$
$\mathcal{P}_2 :: e_1^r \stackrel{1}{\Longrightarrow} e_1'$ by i.h. on $\mathcal{D}_1^l, \mathcal{D}_1^r$ in Φ'
There exists an e_2'
$\mathcal{Q}_1 :: e_2^l \stackrel{1}{\Longrightarrow} e_2'$
$\mathcal{Q}_2 :: e_2^r \stackrel{1}{\Longrightarrow} e_2'$ by i.h. on $\mathcal{D}_2^l, \mathcal{D}_2^r$ in Φ

$$\begin{tikzcd}[column sep=small]
& e_1 \arrow[dl, "\mathcal{D}_1^l"'] \arrow[dr, "\mathcal{D}_1^r"] & \\
e_1^l \arrow[dr, "\mathcal{P}_1"'] & & e_1^r \arrow[dl, "\mathcal{P}_2"] \\
& e_1' &
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=small]
& e_2 \arrow[dl, "\mathcal{D}_2^l"'] \arrow[dr, "\mathcal{D}_2^r"] & \\
e_2^l \arrow[dr, "\mathcal{Q}_1"'] & & e_2^r \arrow[dl, "\mathcal{Q}_2"] \\
& e_2' &
\end{tikzcd}$$

$\mathcal{R}_1 :: e_1^l[e_2^l/x] \stackrel{1}{\Longrightarrow} e_1'[e_2'/x]$ by Lemma 3.6 on $\mathcal{P}_1, \mathcal{Q}_1$
$\mathcal{R}_2 :: e_1^r[e_2^r/x] \stackrel{1}{\Longrightarrow} e_1'[e_2'/x]$ by Lemma 3.6 on $\mathcal{P}_2, \mathcal{Q}_2$

$$\begin{tikzcd}[column sep=small]
& (\lambda x:\tau.e_1)\ e_2 \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\
e_1^l[e_2^l/x] \arrow[dr, "\mathcal{R}_1"'] & & e_1^r[e_2^r/x] \arrow[dl, "\mathcal{R}_2"] \\
& e_1'[e_2'/x] &
\end{tikzcd}$$

Case:

$$\mathcal{D}^l = \dfrac{\begin{array}{cc} \begin{array}{c}\dfrac{}{x \overset{1}{\Longrightarrow} x}\,u \\ \mathcal{D}_1^l \\ e_1 \overset{1}{\Longrightarrow} e_1^l\end{array} & \begin{array}{c}\mathcal{D}_2^l \\ e_2 \overset{1}{\Longrightarrow} e_2^l\end{array}\end{array}}{(\lambda x : \tau.e_1)\ e_2 \overset{1}{\Longrightarrow} e_1^l[e_2^l/x]}\,\mathsf{pbeta}^u \qquad \mathcal{D}^r = \dfrac{\begin{array}{cc} \begin{array}{c}\mathcal{D}_1^r \\ \lambda x : \tau.e_1 \overset{1}{\Longrightarrow} e_1^r\end{array} & \begin{array}{c}\mathcal{D}_2^r \\ e_2 \overset{1}{\Longrightarrow} e_2^r\end{array}\end{array}}{(\lambda x : \tau.e_1)\ e_2 \overset{1}{\Longrightarrow} e_1^r\ e_2^r}\,\mathsf{papp}$$

$$\mathcal{D}_1^r = \dfrac{\begin{array}{c}\dfrac{}{x \overset{1}{\Longrightarrow} x}\,u \\ \mathcal{D}_1'^r \\ e_1 \overset{1}{\Longrightarrow} e_1'^r\end{array}}{\lambda x : \tau.e_1 \overset{1}{\Longrightarrow} \lambda x : \tau.e_1'^r}\,\mathsf{plam}^u$$

$e_1^r = \lambda x : \tau.e_1'^r$ by inversion on $\mathcal{D}_1^r$

Extend Φ by $x : \text{term } \tau, u :: x \overset{1}{\Longrightarrow} x$ to Φ'

There exists a e_1'

$\mathcal{P}_1 :: e_1^l \overset{1}{\Longrightarrow} e_1'$

$\mathcal{P}_2 :: e_1'^r \overset{1}{\Longrightarrow} e_1'$ by i.h. on $\mathcal{D}_1^l, \mathcal{D}_1'^r$ in Φ'

There exists a e_2'

$\mathcal{Q}_1 :: e_2^l \overset{1}{\Longrightarrow} e_2'$

$\mathcal{Q}_2 :: e_2^r \overset{1}{\Longrightarrow} e_2'$ by i.h. on $\mathcal{D}_2^l, \mathcal{D}_2^r$ in Φ

$$\begin{tikzcd} & e_1 \arrow[dl, "\mathcal{D}_1^l"'] \arrow[dr, "\mathcal{D}_1'^r"] & \\ e_1^l \arrow[dr, "\mathcal{P}_1"'] & & e_1'^r \arrow[dl, "\mathcal{P}_2"] \\ & e_1' & \end{tikzcd} \qquad \begin{tikzcd} & e_2 \arrow[dl, "\mathcal{D}_2^l"'] \arrow[dr, "\mathcal{D}_2^r"] & \\ e_2^l \arrow[dr, "\mathcal{Q}_1"'] & & e_2^r \arrow[dl, "\mathcal{Q}_2"] \\ & e_2' & \end{tikzcd}$$

$\mathcal{R}_1 :: e_1^l[e_2^l/x] \overset{1}{\Longrightarrow} e_1'[e_2'/x]$ by Lemma 3.6 on $\mathcal{P}_1, \mathcal{Q}_1$

$\mathcal{R}_2 :: (\lambda x : \tau.e_1'^r)\ e_2^r \overset{1}{\Longrightarrow} e_1'[e_2'/x]$ by rule pbeta on $\mathcal{P}_2$ and $\mathcal{Q}_2$

$$\begin{tikzcd} & (\lambda x : \tau.e_1)\ e_2 \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\ e_1^l[e_2^l/x] \arrow[dr, "\mathcal{R}_1"'] & & (\lambda x : \tau.e_1'^r)\ e_2^r \arrow[dl, "\mathcal{R}_2"] \\ & e_1'[e_2'/x] & \end{tikzcd}$$

Case:

$$\mathcal{D}^l = \dfrac{\begin{array}{c} \overline{x \overset{1}{\Longrightarrow} x}\;u \\ \mathcal{D}_1^l \\ e \overset{1}{\Longrightarrow} e^l \end{array}}{\lambda x:\tau.e \overset{1}{\Longrightarrow} \lambda x:\tau.e^l}\,\mathsf{plam}^u \qquad \mathcal{D}^r = \dfrac{\begin{array}{c} \overline{x \overset{1}{\Longrightarrow} x}\;u \\ \mathcal{D}_1^r \\ e \overset{1}{\Longrightarrow} e^r \end{array}}{\lambda x:\tau.e \overset{1}{\Longrightarrow} \lambda x:\tau.e^r}\,\mathsf{plam}^u$$

Extend Φ by $x : \text{term } \tau, u :: x \overset{1}{\Longrightarrow} x$ to Φ'
There exists a e'
$\mathcal{P}_1 :: e^l \overset{1}{\Longrightarrow} e'$
$\mathcal{P}_2 :: e^r \overset{1}{\Longrightarrow} e'$ by i.h. on $\mathcal{D}_1^l, \mathcal{D}_1^r$ in Φ'

$$\begin{tikzcd} & e \arrow[dl, "\mathcal{D}_1^l"'] \arrow[dr, "\mathcal{D}_1^r"] & \\ e^l \arrow[dr, "\mathcal{P}_1"'] & & e^r \arrow[dl, "\mathcal{P}_2"] \\ & e' & \end{tikzcd}$$

$\mathcal{R}_1 :: \lambda x:\tau.e^l \overset{1}{\Longrightarrow} \lambda x:\tau.e'$ by plam^u on $\mathcal{P}_1$
$\mathcal{R}_2 :: \lambda x:\tau.e^r \overset{1}{\Longrightarrow} \lambda x:\tau.e'$ by plam^u on $\mathcal{P}_2$

$$\begin{tikzcd} & \lambda x:\tau.e \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\ \lambda x:\tau.e^l \arrow[dr, "\mathcal{R}_1"'] & & \lambda x:\tau.e^r \arrow[dl, "\mathcal{R}_2"] \\ & \lambda x:\tau.e' & \end{tikzcd}$$

Case:

$$\mathcal{D}^l = \dfrac{\begin{array}{c} \mathcal{D}_1^l \\ \lambda x:\tau.e_1 \overset{1}{\Longrightarrow} e_1^l \end{array} \qquad \begin{array}{c} \mathcal{D}_2^l \\ e_2 \overset{1}{\Longrightarrow} e_2^l \end{array}}{(\lambda x:\tau.e_1)\ e_2 \overset{1}{\Longrightarrow} e_1^l\ e_2^l}\,\mathsf{papp} \qquad \mathcal{D}^r = \dfrac{\begin{array}{c} \overline{x \overset{1}{\Longrightarrow} x}\;u \\ \mathcal{D}_1^r \\ e_1 \overset{1}{\Longrightarrow} e_1^r \end{array} \qquad \begin{array}{c} \mathcal{D}_2^r \\ e_2 \overset{1}{\Longrightarrow} e_2^r \end{array}}{(\lambda x:\tau.e_1)\ e_2 \overset{1}{\Longrightarrow} e_1^r[e_2^r/x]}\,\mathsf{pbeta}^u$$

$$\mathcal{D}_1^l = \dfrac{\begin{array}{c} \overline{x \overset{1}{\Longrightarrow} x}\;u \\ \mathcal{D}_1'^l \\ e_1 \overset{1}{\Longrightarrow} e_1'^l \end{array}}{\lambda x:\tau.e_1 \overset{1}{\Longrightarrow} \lambda x:\tau.e_1'^l}\,\mathsf{plam}^u$$

$e_1^l = \lambda x : \tau . e_1'^l$ by inversion on $\mathcal{D}_1^l$
Extend Φ by $x : \text{term } \tau, u :: x \overset{1}{\Longrightarrow} x$ to Φ'
There exists a e_1'
$\mathcal{P}_1 :: e_1'^l \overset{1}{\Longrightarrow} e_1'$
$\mathcal{P}_2 :: e_1^r \overset{1}{\Longrightarrow} e_1'$ by i.h. on $\mathcal{D}_1'^l, \mathcal{D}_1^r$ in Φ'
There exists a e_2'
$\mathcal{Q}_1 :: e_2^l \overset{1}{\Longrightarrow} e_2'$
$\mathcal{Q}_2 :: e_2^r \overset{1}{\Longrightarrow} e_2'$ by i.h. on $\mathcal{D}_2^l, \mathcal{D}_2^r$ in Φ

$$\begin{tikzcd}[column sep=small] & e_1 \arrow[dl, "\mathcal{D}_1'^l"'] \arrow[dr, "\mathcal{D}_1^r"] & \\ e_1'^l \arrow[dr, "\mathcal{P}_1"'] & & e_1^r \arrow[dl, "\mathcal{P}_2"] \\ & e_1' & \end{tikzcd} \qquad \begin{tikzcd}[column sep=small] & e_2 \arrow[dl, "\mathcal{D}_2^l"'] \arrow[dr, "\mathcal{D}_2^r"] & \\ e_2^l \arrow[dr, "\mathcal{Q}_1"'] & & e_2^r \arrow[dl, "\mathcal{Q}_2"] \\ & e_2' & \end{tikzcd}$$

$\mathcal{R}_1 :: (\lambda x : \tau . e_1'^l)\ e_2^l \overset{1}{\Longrightarrow} e_1'[e_2'/x]$ by rule pbeta on $\mathcal{P}_1$ and $\mathcal{Q}_1$
$\mathcal{R}_2 :: e_1^r[e_2^r/x] \overset{1}{\Longrightarrow} e_1'[e_2'/x]$ by Lemma 3.6 on $\mathcal{P}_2, \mathcal{Q}_2$

$$\begin{tikzcd}[column sep=small] & (\lambda x : \tau . e_1)\ e_2 \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\ (\lambda x : \tau . e_1'^l)\ e_2^l \arrow[dr, "\mathcal{R}_1"'] & & e_1^r[e_2^r/x] \arrow[dl, "\mathcal{R}_2"] \\ & e_1'[e_2'/x] & \end{tikzcd}$$

Case:

$$\mathcal{D}^l = \dfrac{\dfrac{\mathcal{D}_1^l}{e_1 \overset{1}{\Longrightarrow} e_1^l} \qquad \dfrac{\mathcal{D}_2^l}{e_2 \overset{1}{\Longrightarrow} e_2^l}}{e_1\ e_2 \overset{1}{\Longrightarrow} e_1^l\ e_2^l}\,\mathsf{papp} \qquad \mathcal{D}^r = \dfrac{\dfrac{\mathcal{D}_1^r}{e_1 \overset{1}{\Longrightarrow} e_1^r} \qquad \dfrac{\mathcal{D}_2^r}{e_2 \overset{1}{\Longrightarrow} e_2^r}}{e_1\ e_2 \overset{1}{\Longrightarrow} e_1^r\ e_2^r}\,\mathsf{papp}$$

There exists a e_1'
$\mathcal{P}_1 :: e_1^l \overset{1}{\Longrightarrow} e_1'$
$\mathcal{P}_2 :: e_1^r \overset{1}{\Longrightarrow} e_1'$ by i.h. on $\mathcal{D}_1^l, \mathcal{D}_1^r$ in Φ
There exists a e_2'
$\mathcal{Q}_1 :: e_2^l \overset{1}{\Longrightarrow} e_2'$
$\mathcal{Q}_2 :: e_2^r \overset{1}{\Longrightarrow} e_2'$ by i.h. on $\mathcal{D}_2^l, \mathcal{D}_2^r$ in Φ

$$
\begin{tikzcd}
 & e_1 \arrow[dl, "\mathcal{D}_1^l"'] \arrow[dr, "\mathcal{D}_1^r"] & \\
e_1^l \arrow[dr, "\mathcal{P}_1"'] & & e_1^r \arrow[dl, "\mathcal{P}_2"] \\
 & e_1' &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & e_2 \arrow[dl, "\mathcal{D}_2^l"'] \arrow[dr, "\mathcal{D}_2^r"] & \\
e_2^l \arrow[dr, "\mathcal{Q}_1"'] & & e_2^r \arrow[dl, "\mathcal{Q}_2"] \\
 & e_2' &
\end{tikzcd}
$$

$\mathcal{R}_1 :: e_1^l\ e_2^l \overset{1}{\Longrightarrow} e_1'\ e_2'$ by papp on $\mathcal{P}_1, \mathcal{Q}_1$
$\mathcal{R}_2 :: e_1^r\ e_2^r \overset{1}{\Longrightarrow} e_1'\ e_2'$ by papp on $\mathcal{P}_2, \mathcal{Q}_2$

$$
\begin{tikzcd}
 & e_1\ e_2 \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\
e_1^l\ e_2^l \arrow[dr, "\mathcal{R}_1"'] & & e_1^r\ e_2^r \arrow[dl, "\mathcal{R}_2"] \\
 & e_1'\ e_2' &
\end{tikzcd}
$$

□

The proof of the diamond lemma introduces two new proof principles. First we use inversion in the third and the sixth case of the proof, and second we repeatedly appeal to the substitution Lemma 3.6. Conceptually, inversion is a new operation, but technically, it is nothing else but a special form of case analysis. Given a derivation of some judgment cases can be analyzed according to the last applied rule, and if the last rule application is uniquely determined, case analysis is called commonly called inversion; in practice however inversion need not to be unique. One of these examples is the cut-elimination theorem for the sequent calculus of first-order intuitionistic logic [Pfe95].

The second proof principle is lemma application; it is very important since it allows the programming language and logic designers to stage their development into tasks of appropriate size.

Continuing in the development of the Church-Rosser theorem for parallel reduction, we present three more lemmas, which generalize the diamond Lemma 3.7 to parallel multi-step reduction and parallel conversion. We give the proofs explicitly, in order to have an extended set of examples necessary in the subsequent chapters of this thesis. Alternatively, the interested user may want to consult [Pfe93] for a more detailed presentation.

Lemma 3.8 (Strip lemma) *Let Φ be the dynamic extension of the world. If $\mathcal{D}^l :: e \overset{1}{\Longrightarrow} e^l$ and $\mathcal{D}^r :: e \overset{*}{\Longrightarrow} e^r$ then there exists a common reduct e', such that $\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e'$ and $\mathcal{R}_2 :: e^r \overset{1}{\Longrightarrow} e'$.*

$$
\begin{tikzcd}
 & e \arrow[dl] \arrow[dr, "*"] & \\
e^l \arrow[dr, dotted, "*"'] & & e^r \arrow[dl, dotted] \\
 & e' &
\end{tikzcd}
$$

Proof: by structural induction on $\mathcal{D}^r$

Case: $\mathcal{D}^r = \dfrac{}{e \overset{*}{\Longrightarrow} e}\,\mathsf{pid}$

$\mathcal{R}_1 :: e^l \overset{1}{\Longrightarrow} e^l$ by pid

$\mathcal{R}_2 = \mathcal{D}_l :: e \overset{1}{\Longrightarrow} e^l$ by assumption

$$
\begin{tikzcd}
& e \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\
e^l \arrow[dr, "\mathcal{R}_1"'] & & e \arrow[dl, "\mathcal{R}_2"] \\
& e^l &
\end{tikzcd}
$$

Case: $\mathcal{D}^r = \dfrac{\overset{\mathcal{D}_1^r}{e \overset{1}{\Longrightarrow} e_1^r} \qquad \overset{\mathcal{D}_2^r}{e_1^r \overset{*}{\Longrightarrow} e^r}}{e \overset{*}{\Longrightarrow} e_r}\,\mathsf{pstep}$

There exists a e_1'

$\mathcal{P}_1 :: e^l \overset{1}{\Longrightarrow} e_1'$

$\mathcal{P}_2 :: e_1^r \overset{1}{\Longrightarrow} e_1'$ by Lemma 3.7 on $\mathcal{D}^l, \mathcal{D}_1^r$ in Φ

There exists a e_2'

$\mathcal{P}_3 :: e_1' \overset{*}{\Longrightarrow} e_2'$

$\mathcal{R}_2 :: e^r \overset{1}{\Longrightarrow} e_2'$ by i.h. on $\mathcal{P}_2, \mathcal{D}_2^r$ in Φ

$$
\begin{tikzcd}
& e \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}_1^r"] & & \\
e^l \arrow[dr, "\mathcal{P}_1"'] & & e_1^r \arrow[dl, "\mathcal{P}_2"'] \arrow[dr, "\mathcal{D}_2^r"] & \\
& e_1' \arrow[dr, "\mathcal{P}_3"'] & & e_r \arrow[dl, "\mathcal{R}_2"] \\
& & e_2' &
\end{tikzcd}
$$

$\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e_2'$ by rule pstep on $\mathcal{P}_1$, $\mathcal{P}_3$

□

A further generalization yields the confluence lemma: The left reduction step is being generalized to a multi-step reduction.

Lemma 3.9 (Confluence lemma) *Let* Φ *be the dynamic extension of the world. If* $\mathcal{D}^l :: e \xRightarrow{*} e^l$ *and* $\mathcal{D}^r :: e \xRightarrow{*} e^r$ *then there exists a common reduct* e'*, such that* $\mathcal{R}_1 :: e^l \xRightarrow{*} e'$ *and* $\mathcal{R}_2 :: e^r \xRightarrow{*} e'$*.*

$$\begin{tikzcd}[column sep=small]
 & e \arrow[dl, "*"'] \arrow[dr, "*"] & \\
e^l \arrow[dr, dotted, "*"'] & & e^r \arrow[dl, dotted, "*"] \\
 & e' &
\end{tikzcd}$$

Proof: by structural induction on $\mathcal{D}^l$

Case: $\mathcal{D}^l = \dfrac{}{e \xRightarrow{*} e}\ \mathsf{pid}$

$\mathcal{R}_1 : e \xRightarrow{*} e^r$ by assumption $\mathcal{D}^r$

$\mathcal{R}_2 : e^r \xRightarrow{*} e^r$ by rule pid

$$\begin{tikzcd}[column sep=small]
 & e \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\
e \arrow[dr, "\mathcal{R}_1"'] & & e^r \arrow[dl, "\mathcal{R}_2"] \\
 & e^r &
\end{tikzcd}$$

Case: $\mathcal{D}^l = \dfrac{\genfrac{}{}{0pt}{}{\mathcal{D}^l_1}{e \xRightarrow{1} e^l_1} \qquad \genfrac{}{}{0pt}{}{\mathcal{D}^l_2}{e^l_1 \xRightarrow{*} e^l}}{e \xRightarrow{*} e^l}\ \mathsf{pstep}$

There exists a e'_1

$\mathcal{P}_1 :: e^l_1 \xRightarrow{1} e'_1$

$\mathcal{P}_2 :: e^r \xRightarrow{1} e'_1$ by Lemma 3.8 on $\mathcal{D}^l_1, \mathcal{D}^r$ in Φ

There exists a e'_2

$\mathcal{R}_1 :: e^l \xRightarrow{*} e'_2$

$\mathcal{P}_3 :: e'_1 \xRightarrow{*} e'_2$ by i.h. on $\mathcal{D}^l_2, \mathcal{P}_1$ in Φ

$$\begin{tikzcd}[column sep=small]
 & & e \arrow[dl, "\mathcal{D}^l_1"'] \arrow[dr, "\mathcal{D}^r"] & \\
 & e^l_1 \arrow[dl, "\mathcal{D}^l_2"'] \arrow[dr, "\mathcal{P}_1"] & & e^r \arrow[dl, "\mathcal{P}_2"] \\
e^l \arrow[dr, "\mathcal{R}_1"'] & & e'_1 \arrow[dl, "\mathcal{P}_3"] & \\
 & e'_2 & &
\end{tikzcd}$$

$$\mathcal{R}_2 :: e^r \overset{*}{\Longrightarrow} e'_2 \qquad \text{by rule } \mathsf{pstep} \text{ on } \mathcal{P}_2, \mathcal{P}_3$$

□

All is prepared to prove the Church-Rosser theorem for parallel reduction.

Theorem 3.10 (Church-Rosser) *Let Φ be the dynamic extension of the world. If $\mathcal{D} :: e^l \Longleftrightarrow e^r$ then there exists a common reduct e', such that $\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e'$ and $\mathcal{R}_2 :: e^r \overset{*}{\Longrightarrow} e'$.*

$$\begin{tikzcd}
e^l \arrow[rr, leftrightarrow] \arrow[dr, dotted, "*"'] & & e^r \arrow[dl, dotted, "*"] \\
& e' &
\end{tikzcd}$$

Proof: by structural induction on $\mathcal{D}$

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ e^l \overset{*}{\Longrightarrow} e^r\end{matrix}}{e^l \Longleftrightarrow e^r}\,\mathsf{pred}$

$$\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e^r \qquad \text{by assumption } \mathcal{D}_1$$
$$\mathcal{R}_2 :: e^r \overset{*}{\Longrightarrow} e^r \qquad \text{by rule } \mathsf{pid}$$

$$\begin{tikzcd}
e^l \arrow[rr, leftrightarrow, "\mathcal{D}"] \arrow[dr, "\mathcal{R}_1"'] & & e^r \arrow[dl, "\mathcal{R}_2"] \\
& e^r &
\end{tikzcd}$$

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ e^r \overset{*}{\Longrightarrow} e^l\end{matrix}}{e^l \Longleftrightarrow e^r}\,\mathsf{pexp}$

$$\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e^l \qquad \text{by rule } \mathsf{pid}$$
$$\mathcal{R}_2 :: e^r \overset{*}{\Longrightarrow} e^l \qquad \text{by assumption } \mathcal{D}_1$$

$$\begin{tikzcd}
e^l \arrow[rr, leftrightarrow, "\mathcal{D}"] \arrow[dr, "\mathcal{R}_1"'] & & e^r \arrow[dl, "\mathcal{R}_2"] \\
& e^l &
\end{tikzcd}$$

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ e^l \Longleftrightarrow e\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ e \Longleftrightarrow e^r\end{matrix}}{e^l \Longleftrightarrow e^r}\,\mathsf{ptrans}$

There exists a e'_1
$\mathcal{P}_1 :: e^l \stackrel{*}{\Longrightarrow} e'_1$
$\mathcal{P}_2 :: e \stackrel{*}{\Longrightarrow} e'_1$ by i.h. on $\mathcal{D}_1$ in Φ
There exists a e'_2
$\mathcal{P}_3 :: e \stackrel{*}{\Longrightarrow} e'_2$
$\mathcal{P}_4 :: e \stackrel{*}{\Longrightarrow} e'_2$ by i.h. on $\mathcal{D}_2$ in Φ
There exists a e'
$\mathcal{Q}_1 :: e'_1 \stackrel{*}{\Longrightarrow} e'$
$\mathcal{Q}_2 :: e'_2 \stackrel{*}{\Longrightarrow} e'$ by Lemma 3.9 on $\mathcal{P}_2, \mathcal{P}_3$ in Φ

$$\begin{tikzcd}
e^l \arrow[rr, leftrightarrow, "\mathcal{D}_1"] \arrow[dr, "\mathcal{P}_1"'] & & e \arrow[rr, leftrightarrow, "\mathcal{D}_2"] \arrow[dl, "\mathcal{P}_2"'] \arrow[dr, "\mathcal{P}_3"] & & e^r \arrow[dl, "\mathcal{P}_4"] \\
& e'_1 \arrow[dr, "\mathcal{Q}_1"'] & & e'_2 \arrow[dl, "\mathcal{Q}_2"] & \\
& & e' & &
\end{tikzcd}$$

$\mathcal{R}_1 :: e^l \stackrel{*}{\Longrightarrow} e'$ by Lemma 3.5 on $\mathcal{P}_1, \mathcal{Q}_1$
$\mathcal{R}_2 :: e^r \stackrel{*}{\Longrightarrow} e'$ by Lemma 3.5 on $\mathcal{P}_2, \mathcal{Q}_2$

□

This concludes the presentation of meta-theoretic results for parallel reduction. All proofs so far have exposed five basic and recurring proof principles. In order to prove a theorem by induction, different cases must be analyzed, and the formulation of the theorem can be used as induction hypothesis, as long as the argument derivations are smaller than the given ones. In the area of inductive theorem proving treated as one operation through the presence of induction principles, it is treated in our presentation as two different operations.

New derivations must be constructed from already known to exist derivations by the application of inference rules. This proof principle constructs witness derivations for existential quantified variables using assumptions (also from Φ), and inference rules.

If the induction hypothesis of a theorem is so general that it can be applied to open terms (which are open in a regular world extension Φ), new parameters can be dynamically introduced into the proof process. And last but not least, very often lemmas are needed to complete a proof. The possibility to appeal to lemmas is crucial in any interactive proof development system.

3.2.4 Equivalence of Parallel and Ordinary Reduction

The Church-Rosser property for parallel reduction is proven. But what about the Church-Rosser property of the ordinary reduction relation? We proceed by showing that it is also satisfied for ordinary reduction. The essential idea behind the proof is that any ordinary reduction can be transformed into a parallel reduction and vice versa.

Lemma 3.11 (Single-step correspondence)

1. *If* $\mathcal{D} :: e^l \stackrel{1}{\Longrightarrow} e^r$ *then* $e^l \stackrel{*}{\longrightarrow} e^r$.
2. *If* $\mathcal{D} :: e^l \stackrel{1}{\longrightarrow} e^r$ *then* $e^l \stackrel{1}{\Longrightarrow} e^r$.

Proof: by structural reduction on $\mathcal{D}(1), \mathcal{D}(2)$. For the detailed proof, see [Pfe93], Lemma 10, Lemma 11. □

This result can be generalized to an entire sequence of reduction steps. Each ordinary multi-step reduction can be expressed by a parallel multi-step reduction and vice versa.

Lemma 3.12 (Multi-step correspondence) $\mathcal{D} :: e^l \stackrel{*}{\longrightarrow} e^r$ *iff* $\mathcal{R} :: e^l \stackrel{*}{\Longrightarrow} e^r$

Proof: by structural induction on $\mathcal{D}$, $\mathcal{R}$, respectively. For the detailed proof, see [Pfe93], Lemma 12. □

The conversion rules for ordinary reduction do not correspond directly to the conversion rules for parallel reduction. For example, there is an explicit ordinary symmetry rule rsymm, but there is no such rule in the parallel case. But we can show that it is admissible.

Lemma 3.13 (Symmetry) *If* $\mathcal{D} :: e^l \Longleftrightarrow e^r$ *then* $\mathcal{R} :: e^r \Longleftrightarrow e^l$

Proof: by structural induction on $\mathcal{D}$. For the detailed proof, see [Pfe93], Lemma 14. □

Using this result, one can now show the equivalence of ordinary and parallel conversion.

Lemma 3.14 (Conversion correspondence)

1. *If* $\mathcal{D} :: e^l \longleftrightarrow e^r$ *then* $e^l \Longleftrightarrow e^r$
2. *If* $\mathcal{D} :: e^l \Longleftrightarrow e^r$ *then* $e^l \longleftrightarrow e^r$

Proof: by structural induction on $\mathcal{D}(1), \mathcal{D}(2)$. For the detailed proof, see [Pfe93], Lemma 13 and Lemma 15. □

Now it is obvious; also the ordinary reduction relation enjoys the Church-Rosser property. Given an ordinary conversion derivation between two terms e^l and e^r, Lemma 3.14 guarantees that there is a corresponding parallel conversion. By the Church-Rosser property for parallel reduction 3.10, one obtains a common reduct e', and two parallel reductions $\mathcal{D}_l$ and $\mathcal{D}_r$, which can easily be translated back into ordinary reductions using Lemma 3.14 twice.

Theorem 3.15 (Church-Rosser for ordinary reduction) *If* $e^l \longleftrightarrow e^r$ *then there exists a common reduct* e', *s.t.* $e^l \stackrel{*}{\longrightarrow} e'$ *and* $e^r \stackrel{*}{\longrightarrow} e'$.

Proof: Direct. For the detailed proof, see [Pfe93], Theorem 16. □

When studying the proofs in [Pfe93], the reader will notice that the only proof principles used are the ones discussed in this chapter.

3.3 Historical Overview

The formalization of formal theory of various kinds has been the focus of attention in the automated theorem proving and proof assistant community for at least four decades. First there were general-purpose theorem provers which were built to support mathematicians in their quest for the search of mathematical truth. Then other special-purpose automated theorem proving techniques have been invented, developed, and established; one of the most successful techniques is model-checking [CGP00] which has proven extremely successful not only in the academic environment but also for industrial applications. In order to classify the work presented in this thesis as a special purpose automated theorem proving system for the development of the meta-theory of deductive systems, we attempt to give a brief overview over previous developments and discuss the advantages and disadvantages of existing theorem proving techniques.

3.3.1 General-Purpose Theorem Provers

The work by Boyer and Moore [BM79] on the Nqthm theorem prover has triggered a whole research program concerned with the automated deduction of true statements. Even though mainly interested in reasoning about mathematical truth, this theorem prover has been applied to many different problems domains over the last two decades. In general, formal methods and automated deduction techniques have found numerous applications in hardware and in software design. Based on quantifier-free inductive definitions, Nqthm reads a list of theorems and proofs and tries to bridge the gaps in the proofs by automatically applying small reasoning steps. Only if a gap is too big, the theorem prover complains and asks the developer to introduce new lemmas. Many important theorems have been verified using Nqthm, among many others, the Church-Rosser theorem [Sha88], and Gödel's incompleteness theorem [Sha94], and the Ramsey theorem [Kun95].

When using a theorem prover like Nqthm for the development of the Church-Rosser theorem, the user is required to encode terms, the typing relation, and all reduction relations in form of quantifier-free inductive definitions. Variables for example must represented as strings or numbers, substitutions must be encoded explicitly, and naturally the soundness of substitution application must be proven explicitly. It is clearly possible to use Nqthm as a theorem prover to tackle this task (as Shankar has demonstrated [Sha88]), but the restriction to quantifier-free inductive definitions puts additional burden on the user's shoulders to implement the various variable concepts, capture avoiding substitutions, and to prove the corresponding substitution lemmas.

Over time, many techniques have been developed to perform efficient proof search in different logics, ranging from classical, over intuitionistic to linear logics with different degrees of expressiveness, ranging from propositional over first-order to higher-order logics. Techniques, such as resolution [Rob65], paramodulation [BGLS92], or the inverse method [DMTV99] have been devised to facilitate proof search in various calculi, such as natural deduction[Pra65], the sequent calculus [Gen35], the tableaux formulation [Häh99], or the intercalation calculus which is a specialized formulation of the natural deduction calculus [SB98] for proof search. These techniques are tuned to conduct efficient proof search *in* deductive systems.

Our endeavor however lies in reasoning *about* deductive systems. Early on, it has been noticed that the inductive formalization of natural numbers is directly reflected in proofs by mathematical induction [Göd90]. In computer science, where many constructs are inductively defined, induction has presented itself as a very valuable tool to express and reason about specifications in a formal way. Thus, many theorem provers are based on induction and inductive definitions in order to formalize deductive systems, such as programming languages and logics. In fact, induction is one of the core concepts present in almost every proof assistant, such as Isabelle [Pau94], Coq [DFH$^+$93], Lego [LP92], and PVS [ORS92], and many theorem provers, such as INKA [HS96], and Nqthm [BM79].

Unfortunately, inductive theorem provers are limited in their expressiveness. In fact, by definition, inductive definitions are restricted by the positivity condition: The type to be defined can only occur in positive positions in the constructor types. This means, however, that our preferred encoding of the simply-typed λ-calculus, which relies on a higher-order encoding, cannot be expressed using standard inductive definitions. The argument to "lam", for example, is a function of type "term $\ulcorner\tau_1\urcorner \to$ term $\ulcorner\tau_2\urcorner$" which clearly violates the positivity condition. Thus none of the presently available theorem provers supports our proposed way of encoding the Church-Rosser theorem (see Section 3.2). In this thesis we present a technique that allows inductive reasoning over deductive systems that are encoded using higher-order induction techniques.

Contrary to the proof strategy presented by Nqthm, almost all modern theorem provers have adopted a tactic based proof development style [Pau83]. The inference system of the logic intrinsic to the theorem prover consists in general of a set of rules. Given the current proof goal, it is the user's responsibility, to apply rules in the correct order. But in many cases, repeated application of the same rule, or the application of rules in a particular order becomes necessary, which has prompted the development of special purpose languages to express algorithms executing any kind of rule application. These algorithms are called *tactics* and they simplify the theorem proving effort tremendously. The application of a tactic can either succeed, leaving the user with a new (possible empty) set of subgoals, or fail in which case the proof goal remains unchanged.

The work that is presented in this thesis does not take advantage of recent advances in tactic theorem prover. But we recognize that this work can profit from techniques such as proof planning [BSvH$^+$93] and lemma generalization [FH94].

3.3.2 Special-Purpose Theorem Provers

Besides general-purpose theorem provers which are designed to to tackle any problem expressible in mathematics, there are theorem provers that are designed to serve a special purpose. In hardware verification, for example, many circuits can be described by finite state automata. Specifically, the technique of model checking allows to verify a piece of hardware (or better its model) against a given specification by means of a complete state space traversal. In general, the languages used to express those specifications are typically temporal logics. The interested reader might consult [CGP00] for a detailed discussion. If any of the states does not satisfy the specification, the model checking algorithm fails and may report a counter example that gives the hardware designer insight into the cause of failure. Model checking is tremendously successful because it serves a special and relevant purpose and it guarantees a high degree of automation.

Other special-purpose theorem proving techniques are based on rewriting [HO80] and geometry. In rewriting important decision procedures have been developed in order to decide if two terms are equal modulo a set of equalities. Clearly, this decision procedure is a special purpose theorem proving technique. Special purpose decision procedures have also been developed to reason quickly about geometry, for example by using Gröbner bases [Kap98].

It is very difficult (even though possible) to represent and reason about a model-checking problem in a general-purpose theorem prover. Almost certainly, since the overhead is enormous, this technology would probably not been as widely accepted as model-checking is today. Therefore we argue in favor of special purpose techniques to augment general purpose theorem provers. In particular, general purpose theorem provers only offer a restricted set of operators for specification and reasoning; therefore in order to use other operators, auxiliary constructions are mandatory. For example, in order to use a general purpose theorem prover to express a finite state traversal problem, one has to encode the reachability relation between states *explicitly* whereas it is *implicit* when using a model checker like SMV [CGP00].

In this sense, the meta-theorem prover which we develop in the next few chapters is a special purpose theorem prover. The technology presented in this thesis does not provide a new approach to general purpose theorem proving, on the contrary, it delivers special purpose theorem proving technology for the use of higher-order encodings. Proofs found by our meta-theorem prover can be transformed into proofs of a general purpose theorem prover. In fact, in Section 9.1.4, we discuss the possibility of translating our meta-proofs into proofs parsable and understandable by other theorem provers, such as Lego or Isabelle.

3.4 Summary

In this chapter we have presented a detailed proof of the Church-Rosser theorem for the simply-typed λ-calculus, and have characterized five basic and over and over recurring proof principles:

1. Case analysis of the last applied inference rule of a given derivation. The proof obligation is split into several new cases.

2. The construction of one or several witness derivations for one ore several existentially quantified judgments. This operation closes a proof obligation.

3. During a proof an appeal to the induction hypothesis may be invoked.

4. The development of a theory consists of a sequence of lemmas, where each lemma must be a derivable consequence from previous ones.

5. The proof may be hypothetical, that means that the derivations may be valid in a regular extension of the current world. The world may be dynamically extended during the course of a proof.

All proofs in this chapter are composed of a sequence of these basic operations, which should leave the reader with the following impression: The proofs themselves are not particularly difficult but they are tedious. The most difficult problem is to express the induction hypothesis in appropriate generality — that is the formulation of the theorem itself. In addition, we note that all theorems of this section can be expressed as Π_2-formulas.

As opposed to traditional theorem proving techniques, which are concerned with reasoning *in* a deductive system — a calculus for some logic — our goal is to reason *about* deductive systems. In the further development of this thesis, we will use some techniques from the former, but the overall emphasis of this thesis is the technology to accomplish the latter. In addition, we strongly believe that in the formal development of programming languages and logics, the contributions of this work are very important since they help to verify and automatically prove many of the properties of deductive systems. Furthermore, we strongly believe, that such a system should support the user with helpful hints of how to improve the formulation of a lemma or a theorem in the case of failure.

The theorem prover and its theory, which is presented in the subsequent chapters, is a special purpose theorem prover: it owes it success to the combination of elegant higher-order representation techniques, and proofs by cases and recursion. But in other respects, it is quite basic; it only takes advantage of few of traditional theorem proving techniques, and its implementation could largely profit from applying techniques, such as the inverse method [DMTV99], focusing [And92, How98], or rippling[BSvH$^+$93] — techniques that are well-known for traditional systems.

Part II

Design of a Meta-Logical Framework

Chapter 4

Meta-Logical Frameworks

4.1 Introduction

Logical frameworks are powerful (meta-)languages that support encodings of a large variety of deductive systems, including deductive systems which may contain side conditions, such as for example the Eigenvariable condition for first-order logic, or freshness conditions on parameters in programming languages. Object languages which contain a variable concept, logics which introduce hypotheses, and rewrite systems which dynamically extend the local rewrite relation by cases for newly introduced parameters can be very elegantly represented in these frameworks. For example in LF, the adequacy and soundness arguments of the encoding rely on the fact that canonical form exists for *any* LF object including those of functional type and that the framework provides dependent types (see Section 2.4.4).

Canonical forms are inductively defined by their very definition in LF. In particular, canonical forms of functional type always start with a leading prefix of λ-abstractions. We have argued in Section 2.6 that even though the notion of operational semantics associated with LF functions does not capture definition by cases, LF is an ideal candidate for adequately encoding deductive systems. Clearly, it is not expressive enough to formalize function manipulating derivations that need to be defined by recursion and case analysis. In this thesis, we use LF's function space only for the purpose of representation; for the purpose of defining functions by case analysis and recursion, we introduce in this chapter the notion of a *recursive* function space that is defined in terms of LF objects and LF types.

There is a very deep connection between the recursive function space and standard induction principles. First order encodings of natural numbers for example possess standard induction principles used to reason about natural numbers. More specific, the induction principle expresses how to derive property P for all natural numbers n.

$$\dfrac{\vdash P(0) \qquad \begin{array}{c} \overline{\vdash P(n)}^{\,u} \\ \vdots \\ \vdash P(n+1) \end{array}}{\vdash \forall n. P(n)}\,\mathsf{natind}^u$$

Using this induction principle for example, we can argue that the result of adding any number to itself is even, which is expressed by the predicate “even (n)”. Assuming that “even (n)” implies "even $(n+2)$", we can quickly prove that the formula "$\forall n.\text{even } (n+n)$".

$$\dfrac{\dfrac{}{\vdash \text{even } (0+0)}\,\mathsf{ev\_z} \qquad \dfrac{\dfrac{}{\vdash \text{even } (n+n)}\,u}{\vdash \text{even } (n+n+2))}\,\mathsf{ev\_ss}}{\vdash \forall n.\text{even } (n+n)}\,\mathsf{natind}^u$$

This proof contains some computational content that can be summarized by a recursive function defined by cases. Appeals to the induction hypothesis correspond to recursive calls — given that there is an appropriate formalization the two rules ev_z and ev_ss as "evz" and "evss", respectively.

```
fun double 0 = evz
  | double (n + 1) =
      let
        val D = double n
      in
        (evss D)
      end
```

It has been noticed, that the first-order case does not generalize well to the higher-order case. As our example shows, the main reason that induction principle exists is that we can *predict* the form of a natural number. It is either 0, or it is the successor of a some other natural number. These are the two only cases to be considered, there are no other constructors for natural numbers. The justification of the soundness of this induction principle relies on the general assumption of the world: It is assumed to be *closed*. Only if it is, it can be argued that the induction principle covers all cases. Correspondingly, it is easy to see that **double** covers all cases. In addition it is terminating, which makes it a realizer for the proof given above.

If the definition of natural number were open-ended, this particular induction principle is not sound. Thus, in order to make the closed world assumption explicit, we take the freedom and augment the induction schema with a "·" representing that the world is closed.

$$\dfrac{\cdot \vdash P(0) \qquad \begin{array}{c}\dfrac{}{\cdot \vdash P(n)}\,u \\ \vdots \\ \cdot \vdash P(n+1)\end{array}}{\cdot \vdash \forall n.P(n)}\,\mathsf{natind}^u$$

Are there standard induction principle for higher-order encodings? Not according to the standard literature. It is the goal of this chapter to motivate the design of a meta-logic that accommodates reasoning by cases in the presence of higher-order encodings. The fundamental problem is that induction over higher-order encodings violates the *closed world assumption*, since in order to appeal to the induction hypothesis, one has to traverse λ-binders, thereby extending the world. Clearly the *open world assumption* is too general: it is impossible to guarantee that an induction principle, or the related recursive function covers all cases because the world is always subject to change.

The solution suggested by Equation (3.2.3) is what we call the *regular world assumption* which characterizes the form of all possible worlds Φ (in a finitary way). Thus, one idea is to design an induction principle to reason about property P for *all* simply-typed λ-terms in a world with the regular extension:

$$\Phi \quad ::= \quad \cdot \mid \Phi, (x : \text{term } \tau, u : P(x))$$

Tentatively, one would expect an induction principle of the following form:

$$\dfrac{\begin{array}{cc} \dfrac{}{\Phi \vdash P(e_1)}u_1 \quad \dfrac{}{\Phi \vdash P(e_2)}u_2 & \dfrac{}{\Phi, x : \text{term } \tau_1, u : P(x) \vdash P(e\ x)}u_3 \\ \vdots & \vdots \\ \Phi \vdash P(\text{app } e_1\ e_2) & \Phi \vdash P(\text{lam } (\lambda x : \text{term } \tau_1.\, e\ x)) \end{array}}{\Phi \vdash \forall e : \text{term } \tau. P(e)}\,\mathsf{termind}^{u_1,u_2,u_3}$$

An induction principle of this form would be sufficient for our purposes. But on the other hand, we push its definition another step further. In this form, the appeals to the induction hypothesis are limited, since the worlds are fixed in the assumptions u_1, u_2 and u_3. For our experiments however, Φ describes a valid and regularly formed LF-context, and it must hence satisfy various requirements such as weakening, contraction, and exchange. Therefore we distance us ourselves from the standard notation for induction principles, but we develop instead a meta-logic based on a realizability interpretation of its proofs as total recursive functions. These functions range over arbitrary LF objects that are valid in some world Φ, which is regular in structure. Thus, the soundness of our technique relies crucially on termination and coverage properties of the recursive functions.

By basing inductive definitions on the regular world assumption, this thesis generalizes previous work on standard induction principles which requires the defined datatype not to occur in any negative position in any constructor type (see for example the inductive calculus of constructions [PM93]).

In this chapter we motivate the construction of our formal meta-logic that supports proof about higher-order encodings of deductive systems. In particular, we demonstrate how to define recursive functions over simply-typed λ-terms, and ordinary (Chapter 2) and parallel reduction relations (Chapter 3).

4.2 Methodology

A meta-logical framework is an extension of a logical framework. Besides the representation layer, it provides an explicit layer that supports formal arguments about representations. This section is designed to lead the reader into the area of formalizing the meta-theory of deductive systems. In particular, we start with the formalization of closed meta-theorems and their proofs in Section 4.2.1, i.e. meta-theorems where all participating derivations can be assumed to be closed that is Φ is guaranteed to be empty. In the Section 4.2.2 we generalize those techniques to open meta-theorems, and finally in Section 4.2.3 we extend these techniques to mutually dependent theorems.

Informal Reasoning	Meta-Logical Framework
Church-Rosser theorem Cut-elimination theorem Type preservation properties	Meta-logic $\mathcal{M}_2^+$ Propositions-as-formulas Proofs-as-realizers
Informal Specification	**Logical Framework**
Simply-typed λ-calculus Logic calculi Operational semantics	Type theory LF Judgments-as-types Derivations-as-objects

Process of representation/formalization/encoding $\longrightarrow$

Figure 4.1: The meta-logical layer

4.2.1 Closed Meta-Theorems

In Chapter 3 we have presented a list of theorems which led to the proof of the Church-Rosser theorem for the simply typed λ-calculus. Each proof followed very similar principles. We begin with the proof of the transitivity Lemma 3.1 for ordinary reductions. Two derivations $\mathcal{D}_1 :: e \stackrel{*}{\longrightarrow} e'$ and $\mathcal{D}_2 :: e' \stackrel{*}{\longrightarrow} e''$ are given, from which a third is to be constructed $\mathcal{P} :: e \stackrel{*}{\longrightarrow} e''$. The formulation of all theorems are very similar in structure. A theorem typically consists of a block of universal quantifiers followed by a block of existentials. In the literature, formulas of this kind are called Π_2-formulas [Rog92]. The index "2" expresses that only one quantifier alternation is admitted, and the "Π" specifies that the first quantifier block is universal. For the formalization of Lemma 3.1 we omit the leading universal quantifier for e, e', and e'':

$$\forall \mathcal{D}_1 :: e \stackrel{*}{\longrightarrow} e'.\, \forall \mathcal{D}_2 :: e' \stackrel{*}{\longrightarrow} e''.\, \exists \mathcal{P} :: e \stackrel{*}{\longrightarrow} e''.\, \top$$

Intuitively, representing this theorem in the meta-logical framework must yield a function which maps objects of type $\ulcorner e \stackrel{*}{\longrightarrow} e' \urcorner$ and objects of type $\ulcorner e' \stackrel{*}{\longrightarrow} e'' \urcorner$ to objects of type $\ulcorner e \stackrel{*}{\longrightarrow} e'' \urcorner$. Therefore, the universal quantifier can be read as a new function space constructor "$\supset$" for recursive functions:

$$(e \stackrel{*}{\longrightarrow} e') \supset (e' \stackrel{*}{\longrightarrow} e'') \supset (e \stackrel{*}{\longrightarrow} e'')$$

The recursive function space is different from the parametric, in that it allows function definition by cases, for the proof that goes by induction on $\mathcal{D}_1$. The recursive function space is part of a new conceptual layer above LF, the so-called *meta-logic* as shown in Figure 4.1. All quantifiers are first-order. In particular, the meta-logic we present in this thesis is the meta-logic $\mathcal{M}_2$ which extends previous work [SP98]. It is presented informally in this section and formally in Chapter 5. The soundness of the meta-logic is based on an argument very similar to the one used in constructing the Curry-Howard isomorphism. It is based on a realizability interpretation of meta-proofs as *total* recursive functions, which we call *realizers*. A realizer computes for *any* instantiation of the universal quantifiers *some* witness objects for the existentials. Back to the representation of the transitivity theorem.

$$\begin{aligned}&\ulcorner\forall\mathcal{D}_1 :: e \xrightarrow{*} e'.\,\forall\mathcal{D}_2 :: e' \xrightarrow{*} e''.\,\exists\mathcal{P} :: e \xrightarrow{*} e''.\,\top\urcorner =\\ &\qquad \forall D : \ulcorner e \xrightarrow{*} e'\urcorner.\,\forall E : \ulcorner e' \xrightarrow{*} e''\urcorner.\,\exists P : \ulcorner e \xrightarrow{*} e''\urcorner.\,\top\end{aligned}$$

The $\forall$ quantifier can be read as the recursive function space constructor (similar to a dependent Π type constructor), $\exists$ can be read as Σ-type constructor, and $\top$ as unit type, all on the meta-level. Strictly speaking, this version of the theorem is not complete since we must also universally quantify over all free variables:

$$\begin{aligned}&\forall\tau :: \mathrm{tp}.\,\forall e :: \mathrm{term}\ \tau.\,\forall e' :: \mathrm{term}\ \tau.\,\forall e'' :: \mathrm{term}\ \tau.\\ &\qquad \forall\mathcal{D}_1 :: e \xrightarrow{*} e'.\,\forall\mathcal{D}_2 :: e' \xrightarrow{*} e''.\,\exists\mathcal{P} :: e \xrightarrow{*} e''.\,\top\end{aligned}$$

It translates directly into a formula of the meta-logic. For better presentation, we frame the mathematical formulations of the theorems from Chapter 3.

Lemma 4.1 (Transitivity of $\xrightarrow{*}$, formalized)

$$\boxed{\textit{If } \mathcal{D}_1 :: e \xrightarrow{*} e' \textit{ and } \mathcal{D}_2 :: e' \xrightarrow{*} e'' \textit{ then } e \xrightarrow{*} e''.}$$

$$\begin{aligned}&= \forall T : \mathit{tp}.\,\forall E : \mathit{term}\ T.\,\forall E' : \mathit{term}\ T.\,\forall E'' : \mathit{term}\ T.\\ &\qquad \forall D_1 : E \xrightarrow{*} E'.\,\forall D_2 : E' \xrightarrow{*} E''.\,\exists P : E \xrightarrow{*} E''.\,\top\end{aligned}$$

Each variable that occurs in another type in the theorem is called an *index* variable. Different from the logical framework level, where we have a type reconstruction mechanism as described in Section 2.4.1, type reconstruction on the meta-level may lead to ambiguous results, because it cannot be uniquely determined if index assumptions are to be universally or existentially quantified. Consider the abbreviated version of the Church-Rosser theorem

$$\forall D : E_l \longleftrightarrow E_r.\,\exists R_1 : E_l \xrightarrow{*} E'.\,\exists R_2 : E_r \xrightarrow{*} E'.\,\top$$

where it is impossible to determine E''s status.

$$\begin{aligned}&\forall T : \mathrm{tp}.\,\forall E_l : \mathrm{term}\ T.\,\forall E_r : \mathrm{term}\ T.\\ &\qquad \forall D : E_l \longleftrightarrow E_r.\,\exists E' : \mathrm{term}\ T.\,\exists R_1 : E_l \xrightarrow{*} E'.\,\exists R_2 : E_r \xrightarrow{*} E'.\,\top\end{aligned}$$

Meta-theorems are encoded using recursive functions spaces, and therefore meta-proofs are represented by recursive functions. Throughout this section, those functions are written in an ML-like style with the important difference that the arguments do not range over ML-datatypes, but over LF objects well-formed according to a given signature. We repeat the signature encoding the $\xrightarrow{1}$-relation in LF from Section 2.5:

$$\begin{array}{rcl}\xrightarrow{1} & : & \mathrm{term}\ T \to \mathrm{term}\ T \to \mathrm{type}\\[1ex] \mathrm{rbeta} & : & (\mathrm{app}\ (\mathrm{lam}\ E_1)\ E_2) \xrightarrow{1} E_1\ E_2\\ \mathrm{rlam} & : & (\Pi x : \mathrm{term}\ T_1.\, E\ x \xrightarrow{1} E'\ x)\\ & & \to (\mathrm{lam}\ E) \xrightarrow{1} (\mathrm{lam}\ E')\\ \mathrm{rapp}_1 & : & E_1 \xrightarrow{1} E_1'\\ & & \to (\mathrm{app}\ E_1\ E_2) \xrightarrow{1} (\mathrm{app}\ E_1'\ E_2)\\ \mathrm{rapp}_2 & : & E_2 \xrightarrow{1} E_2'\\ & & \to (\mathrm{app}\ E_1\ E_2) \xrightarrow{1} (\mathrm{app}\ E_1\ E_2')\end{array}$$

Informally we have proven the transitivity Lemma 3.1 already in Section 3.2.1. For the sake of a clearer presentation we repeat it here.

Proof: (of Lemma 3.1) by induction over $\mathcal{D}_1$:

Case: $\mathcal{D}_1 = \dfrac{}{e \overset{*}{\longrightarrow} e}\ \mathsf{rid}$

$\mathcal{D}_2 :: e \overset{*}{\longrightarrow} e''$ by assumption

Case: $\mathcal{D}_1 = \dfrac{\begin{array}{c}\mathcal{D}_1' \\ e \overset{1}{\longrightarrow} e'''\end{array} \qquad \begin{array}{c}\mathcal{D}_1'' \\ e''' \overset{*}{\longrightarrow} e'\end{array}}{e \overset{*}{\longrightarrow} e'}\ \mathsf{rstep}$

$\mathcal{P} :: e''' \overset{*}{\longrightarrow} e''$ by i.h. on $\mathcal{D}_1''$ and $\mathcal{D}_2$
$\mathcal{Q} :: e \overset{*}{\longrightarrow} e''$ by rstep on $\mathcal{D}_1', \mathcal{P}$

$\square$

It is this proof which is encoded as the realizer **trans**. The informal way of stating "proof by structural induction on $\mathcal{D}_1$" from the proof of Lemma 3.1 is translated into "**trans** terminates because the argument D_1 decreases in size with every recursive call". When totally explicit, **trans** expects six arguments T, E, E', E'', D_1, and D_2, but for our purposes we will omit the first four (implicit) arguments in order not to clutter the presentation. This leaves **trans** with only two arguments D_1 and D_2

$$\textbf{fun trans } D_1\ D_2 = \ldots$$

which we gradually refine until it defines a total function. Keywords and function names are typeset in bold type face in order to make the difference between the meta-level and the language level more explicit. The proof of Lemma 3.1 proceeds by induction on $\mathcal{D}_1$. As we have seen, induction translates into a case analysis, generating two cases for D_1.

$$\begin{array}{l}\textbf{fun trans } \mathrm{rid}\ D_2 = \ldots \\ \quad \mid \textbf{trans } (\mathrm{rstep}\ D_1'\ D_1'')\ D_2 = \ldots\end{array}$$

Recall that the reason why we can use pattern matching here is that once instantiated, D_1 has a canonical form (by Theorem 2.6). D_1 will be bound to some (here closed) LF-object M, which matches either with the first or with the second case, but it must match — the case cover must be complete. "rid" and "rstep" are the only two constructors for type family $\overset{*}{\longrightarrow}$. The first case can be directly finished by returning object D_2.

$$\begin{array}{l}\textbf{fun trans } \mathrm{rid}\ D_2 = D_2 \\ \quad \mid \textbf{trans } (\mathrm{rstep}\ D_1'\ D_1'')\ D_2 = \ldots\end{array}$$

The second case is more difficult. The original proof proceeds with the application of the induction hypothesis, followed by the construction of the witness derivation. In this setting, we use termination orders [RP96] to express the well-foundedness of the induction scheme: the recursion will terminate, because with each recursive call, the first argument decreases in size, and since the subterm relation is well-founded the recursion will eventually come to a halt. Translated into formal jargon, we first execute a recursion operation on D_1'' and D_2 keeping in mind that we always have to justify why recursion does not invalidate the totality of the function. For this particular example, the case is clear. The induction hypothesis holds for $\mathcal{D}_1''$ because $\mathcal{D}_1''$ is smaller than $\mathcal{D}_1$:

$$\frac{\begin{array}{c}\mathcal{D}_1' \\ e \stackrel{1}{\longrightarrow} e'''\end{array} \qquad \begin{array}{c}\mathcal{D}_1'' \\ e''' \stackrel{*}{\longrightarrow} e'\end{array}}{e \stackrel{*}{\longrightarrow} e'}\ \mathsf{rstep}$$

In LF, D_1'' is smaller than D_1 because D_1'' is a subterm of D_1. Termination orders are presented in detail in Section 7.2.

$$\begin{array}{l}
\textbf{fun trans } \mathrm{rid}\ D_2 = D_2 \\
\quad |\ \textbf{trans}\ (\mathrm{rstep}\ D_1'\ D_1'')\ D_2 = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ P = \textbf{trans}\ D_1''\ D_2 \\
\qquad \textbf{in} \\
\qquad\quad \ldots \\
\qquad \textbf{end}
\end{array}$$

Finally, we return object "rstep D_1' P" and replace the last set of ... to arrive at the final version of the function.

$$\begin{array}{l}
\textbf{fun trans } \mathrm{rid}\ D_2 = D_2 \\
\quad |\ \textbf{trans}\ (\mathrm{rstep}\ D_1'\ D_1'')\ D_2 = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ P = \textbf{trans}\ D_1''\ D_2 \\
\qquad \textbf{in} \\
\qquad\quad \mathrm{rstep}\ D_1'\ P \\
\qquad \textbf{end}
\end{array}$$

We say, that **trans** is a realizer of transitivity theorem, and use the following shorthand:

$$\begin{array}{l}
\vdash \textbf{trans} \in \forall \tau :: \mathrm{tp}.\, \forall e :: \mathrm{term}\ \tau.\, \forall e' :: \mathrm{term}\ \tau.\, \forall e'' :: \mathrm{term}\ \tau. \\
\qquad\qquad\qquad \forall \mathcal{D}_1 :: e \stackrel{*}{\longrightarrow} e'.\, \forall \mathcal{D}_2 :: e' \stackrel{*}{\longrightarrow} e''.\, \exists \mathcal{P} :: e \stackrel{*}{\longrightarrow} e''.\, \top
\end{array}$$

The "$\in$" symbol is reserved for validity on the meta-level whereas ":" only expresses validity on the language-level as defined in Section 2.4. We postpone the formal presentation of the "$\in$" relation until Chapter 5.

Using the technique of successive refinements, we continue our quest for a formalized version of the Church-Rosser theorem with the encoding of Lemma 3.2. On first sight, all three cases of the lemma are very similar, but on the second, one recognizes, that the first is different from the second and the third: e and e' may contain the free variable x, whereas all terms in the other two cases are assumed to be closed. Without higher-order representation techniques, this lemma cannot be directly represented, but in our case it can: the representations of e :: term τ_1, e' :: term τ_1 and $\mathcal{D} :: e \stackrel{*}{\longrightarrow} e'$ are functions, parametrized in x :: term τ_2. More precisely:

$$\begin{array}{lcl}
\ulcorner e :: \text{term } \tau_1 \urcorner & = & \lambda x : \text{term } \ulcorner \tau_2 \urcorner . \ulcorner e \urcorner : \text{term } \ulcorner \tau_2 \urcorner \to \text{term } \ulcorner \tau_1 \urcorner \\
& & \quad \text{where } \ulcorner x \urcorner = x \\
\ulcorner e' :: \text{term } \tau_1 \urcorner & = & \lambda x : \text{term } \ulcorner \tau_2 \urcorner . \ulcorner e' \urcorner : \text{term } \ulcorner \tau_2 \urcorner \to \text{term } \ulcorner \tau_1 \urcorner \\
& & \quad \text{where } \ulcorner x \urcorner = x \\
\ulcorner \mathcal{D} :: e \xrightarrow{*} e' \urcorner & = & \lambda x : \text{term } \ulcorner \tau_2 \urcorner . \ulcorner e' \urcorner : \text{term } \ulcorner \tau_2 \urcorner \to (\ulcorner e \urcorner \; x \xrightarrow{*} \ulcorner e' \urcorner \; x) \\
& & \quad \text{where } \ulcorner x \urcorner = x
\end{array}$$

This encoding is adequate, and again, this result rests on the canonical form Theorem 2.6. The representation of the derivation $\mathcal{D} :: e \xrightarrow{*} e'$ is a function, and it can take exactly one of the two possible forms

$$\begin{array}{lcl}
\ulcorner \mathcal{D} \urcorner & = & \lambda x : \text{term } T_2 . \, \text{rid} \\
\ulcorner \mathcal{D} \urcorner & = & \lambda x : \text{term } T_2 . \, \text{rstep } (D_1 \; x) \; (D_2 \; x)
\end{array}$$

The representation of all three cases in Lemma 3.2 follows by successive refinement.

Lemma 4.2 (Admissible rules, formalized)

1. $\boxed{\textit{If } \mathcal{D} :: e \xrightarrow{*} e' \textit{ then } \lambda x : \tau_2 . \, e \xrightarrow{*} \lambda x : \tau_2 . \, e'}$

$$\begin{array}{l}
\forall T_1 : tp . \forall T_2 : tp . \forall E : term\, T_2 \to term\, T_1 . \forall E' : term\, T_2 \to term\, T_1 . \\
\qquad \forall D : \Pi x : term\, T_2 . \, E \; x \xrightarrow{*} E' \; x . \\
\qquad \exists P : lam \; (\lambda x : term\, T_2 . \, E \; x) \xrightarrow{*} lam \; (\lambda x : term\, T_2 . \, E' \; x) . \top
\end{array}$$

2. $\boxed{\textit{If } \mathcal{D} :: e_1 \xrightarrow{*} e_1' \textit{ then } e_1 \; e_2 \xrightarrow{*} e_1' \; e_2}$

$$\begin{array}{l}
\forall T_1 : tp . \forall T_2 : tp . \forall E_1 : term\, (T_2 \; arrow \; T_1) . \forall E_1' : term\, (T_2 \; arrow \; T_1) . \forall E_2 : term\, T_2 . \\
\qquad \forall D : E_1 \xrightarrow{*} E_1' . \\
\qquad \exists P : app \; E_1 \; E_2 \xrightarrow{*} app \; E_1' \; E_2 . \top
\end{array}$$

3. $\boxed{\textit{If } \mathcal{D} :: e_2 \xrightarrow{*} e_2 \textit{ then } e_1 \; e_2 \xrightarrow{*} e_1 \; e_2'}$

$$\begin{array}{l}
\forall T_1 : tp . \forall T_2 : tp . \forall E_1 : term\, (T_2 \; arrow \; T_1) . \forall E_2 : term\, T_2 . \forall E_2' : term\, T_2 . \\
\qquad \forall D : E_2 \xrightarrow{*} E_2' . \\
\qquad \exists P : app \; E_1 \; E_2 \xrightarrow{*} app \; E_1 \; E_2' . \top
\end{array}$$

Proof: The termination order is subterm order on D in all three cases.

1. **fun admissible**$_1$ $(\lambda x : \text{term } T_2 . \, \text{rid}) = \text{rid}$
   | **admissible**$_1$ $(\lambda x : \text{term } T_2 . \, \text{rstep } (D_1 \; x) \; (D_2 \; x)) =$
   let
   val $P =$ **admissible**$_1$ $(\lambda x : \text{term } T_2 . \, D_2 \; x)$
   in
   rstep (rlam $(\lambda x : \text{term } T_2 . \, D_1 \; x)$) P
   end

2. **fun** **admissible**$_2$ rid = rid
   | **admissible**$_2$ (rstep D_1 D_2) =
   let
   val P = **admissible**$_2$ D_2
   in
   rstep (rapp$_1$ D_1) P
   end

3. **fun** **admissible**$_3$ rid = rid
   | **admissible**$_3$ (rstep D_1 D_2) =
   let
   val P = **admissible**$_3$ D_2
   in
   rstep (rapp$_2$ D_1) P
   end

□

The intended way to read this formalized lemma, is that the proofs **admissible**$_1$, **admissible**$_2$, and **admissible**$_3$ are functions in the encoding of Theorem 3.2, i.e. more formally:

$$\begin{array}{lcl} \vdash \mathbf{admissible}_1 & \in & \ulcorner \text{If } \mathcal{D} :: e \stackrel{*}{\longrightarrow} e' \text{ then } \lambda x : \tau .\, e \stackrel{*}{\longrightarrow} \lambda x : \tau .\, e' \urcorner \\ \vdash \mathbf{admissible}_2 & \in & \ulcorner \text{If } \mathcal{D} :: e_1 \stackrel{*}{\longrightarrow} e'_1 \text{ then } e_1\, e_2 \stackrel{*}{\longrightarrow} e'_1\, e_2 \urcorner \\ \vdash \mathbf{admissible}_3 & \in & \ulcorner \text{If } \mathcal{D} :: e_2 \stackrel{*}{\longrightarrow} e_2 \text{ then } e_1\, e_2 \stackrel{*}{\longrightarrow} e_1\, e'_2 \urcorner \end{array}$$

The function **trans** and the family of **admissible** functions make use of only three of the proof operations, we have presented in Chapter 3: direct construction, case analysis, and application of the induction hypothesis.

4.2.2 Open Meta-Theorems

In the remainder of this section we will continue to formalize the meta-theorems and meta-proofs, with special emphasis on the parameter operation, which is used for example in the formulation of Lemma 3.4 and its proof. Lemma 3.4 guarantees that each term parallel reduces to itself: for every expression e there exists a derivation of $e \stackrel{1}{\Longrightarrow} e$.

The theorem is only provable if stated in appropriate generality; it must be so general, that it accounts for the term e to be well-formed in a regular extension of the world of the form

$$\Phi \quad ::= \quad \cdot \mid \Phi, x :: \text{term } \tau, u :: x \stackrel{1}{\Longrightarrow} x$$

and then, the resulting derivation $\mathcal{D} :: e \stackrel{1}{\Longrightarrow} e$ is valid in the same world Φ. Clearly, none of the techniques introduced so far, can be *directly* applied to encode this theorem; we must define an operator to allow quantification over those regularly formed world. The encoding of world extensions yields an LF context which we call *parameter context*. Similarly, each extension of the world is represented by a parameter context fragment called a *parameter block*. Parameter blocks must be regularly formed, i.e. they must be instantiations of some abstract description called a *block schema*. In our example, the block schema has the form:

$$\text{SOME}\ T : \text{tp}.\ \text{BLOCK}\ x : \text{term}\ T, u : x \stackrel{1}{\Longrightarrow} x$$

which reads as follows: for some object T of type tp, a parameter block must be an α-variant of $x : \text{term}\ T, u : x \stackrel{1}{\Longrightarrow} x$. Block schemas are partial descriptions of the form of a parameter contexts. Consequently, repeated instantiations of the block schema, yields a valid parameter context. Hence, a single block schema describes entire sets of parameter contexts, and therefore we refer to it as *context schema* for the remainder of this section. A motivation for more complex context schemas can be found in Section 4.2.3.

The well-formed world extension

$$x_1 :: \text{term}\ \tau_1, u_1 :: x_1 \stackrel{1}{\Longrightarrow} x_1, \ldots, x_n :: \text{term}\ \tau_n, u_n :: x_n \stackrel{1}{\Longrightarrow} x_n$$

is hence represented in the meta-logical framework as

$$x_1 : \text{term}\ \ulcorner\tau_1\urcorner, u_1 : x_1 \stackrel{1}{\Longrightarrow} x_1, \ldots, x_n : \text{term}\ \ulcorner\tau_n\urcorner, u_n : x_n \stackrel{1}{\Longrightarrow} x_n$$

and it is an instance of the context schema from above.

In order to express quantification over regularly formed contexts we extend the formal language of theorems provided by the meta-logical framework by a new operator $\Box$. With its help, we can finally formalize of the reflexivity Lemma 3.4:

Lemma 4.3 (Reflexivity theorem, formalized)

Let Φ the dynamic extension of the world. Then for any well-typed term e, there exists a derivation of $e \stackrel{1}{\Longrightarrow} e$.

$$= \Box\text{SOME}\ T : tp.\ \text{BLOCK}\ x : term\ T, u : x \stackrel{1}{\Longrightarrow} x.$$
$$\forall T : tp. \forall E : term\ T. \exists D : E \stackrel{1}{\Longrightarrow} E. \top$$

The next question we must address is how meta-proofs of meta-theorems using context quantification are represented. We begin with the definition of the proof representing function **refl** which we define by successive refinement keeping in mind that the context scheme "SOME T : tp. BLOCK x : term $T, u : x \stackrel{1}{\Longrightarrow} x$" is associated with **refl**.

$$\textbf{fun refl}\ E = \cdots$$

The informal proof proceeds by induction over e, which is formalized by the subterm order on E. Case analysis is not as straightforward as for the transitivity lemma for ordinary reduction above: in addition to the cases introduced by the signature it must also consider parameter cases from Φ. In our example, there can only be one: $E = x$. Since parameter contexts are regularly built, it follows by inspection of the context scheme that x must be declared in a parameter block of the form $x : \text{term}\ T, u : x \stackrel{1}{\Longrightarrow} x$. Obviously, the parameter context can be composed of many instantiations of the block schema, and in order to completely cover all possible forms of E, we would have to provide a case for each possibility. This is impossible, since we would have to consider infinitely many cases!

Fortunately, there is a feasible and more elegant solution to this problem. We can take advantage of the regularity of the parameter context Φ. As long as the proof of a parameter case does not take advantage of the relative position of parameter blocks among each other, but only of other assumptions declared in the same parameter block, we can arrange things so that all infinitely many cases are covered by one single case: Instead of distinguishing cases over all parameter contexts, we consider simultaneously all parameter contexts which contain a parameter block of the form $x : \text{term } T, u : x \overset{1}{\Longrightarrow} x$. Naturally x and u do not stand for a single parameter occurrences any more, but rather for a whole class, and in order to make this distinction explicit, we write $\underline{x}$ and $\underline{u}$ for variables ranging over parameter blocks.

Consequently, a case analysis of E yields three new cases, first a parameter case, second a app case, and the third a lam case:

$$\begin{array}{l} \textbf{fun refl } \underline{x} = \ldots \\ \quad \mid \textbf{refl } (\text{lam } (\lambda x : \text{term } T.\, E'\ x)) = \ldots \\ \quad \mid \textbf{refl } (\text{app } E_1\ E_2) = \ldots \end{array}$$

We incrementally construct this realizer by filling in the three holes ... top to bottom. First, we discuss the global parameter case for $E = \underline{x}$. The original proof case can be immediately closed with u_i. Note, that here $\ulcorner x_i \urcorner = \underline{x}$.

$$\boxed{\begin{array}{ll} \textbf{Case: } e = \dfrac{}{\text{term } \tau_i}\, x_i & \\[2em] \quad u_i :: x_i \overset{1}{\Longrightarrow} x_i & \text{by assumption} \end{array}}$$

On the formal side, $\ulcorner u_i \urcorner = \underline{u}$ can also be used to fill the first hole since it is the only object of desired type $\underline{x} \overset{1}{\Longrightarrow} \underline{x}$. Note, that this is the only information we extract from Φ and therefore we do not need to pass Φ along in the definition of **refl**. Instead, information about $\underline{x}$ and $\underline{u}$ can be directly extracted from the context schema.

$$\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \overset{1}{\Longrightarrow} x$$

Therefore, the LF signature Σ describes the static part of the world and the abstract specification of Φ its dynamic extensions. These two descriptions contain all information needed to complete and to formalize the proof.

$$\begin{array}{l} \textbf{fun refl } \underline{x} = \underline{u} \\ \quad \mid \textbf{refl } (\text{lam } (\lambda x : \text{term } T.\, E'\ x)) = \ldots \\ \quad \mid \textbf{refl } (\text{app } E_1\ E_2) = \ldots \end{array}$$

We continue the construction of the realizer by revisiting the lam-case of the proof.

$$\textbf{Case: } e = \dfrac{\begin{array}{c}\overline{\text{term } \tau_1}^{\,x_{n+1}} \\ e' \\ \text{term } \tau_2\end{array}}{\text{term } (\tau_1 \to \tau_2)} \,\mathsf{lam}^{x_{n+1}}$$

Assume $x_{n+1} :: \text{term } \tau_1$

Assume $u_{n+1} :: x_{n+1} \stackrel{1}{\Longrightarrow} x_{n+1}$

$\mathcal{P} :: e' \stackrel{1}{\Longrightarrow} e'$ by i.h. on e'

$\mathcal{Q} :: \lambda x_{n+1} : \text{term } \tau_1 .\, e' \stackrel{1}{\Longrightarrow} \lambda x_{n+1} : \text{term } \tau_1 .\, e'$ by rule plam on $\mathcal{P}$

In order to apply the induction hypothesis to term e', we appropriately extend the world in a way prescribed by the context schema. Only *new* instances of the block schema can be used, and in this case we refer to it as $\underline{x} : \text{term } \ulcorner \tau_1 \urcorner, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$. The parameter context remains regularly formed after adding these two new declarations.

$$\begin{array}{l}
\textbf{fun refl } \underline{x} = \underline{u} \\
\quad \mid \textbf{refl } (\text{lam } (\lambda x : \text{term } T.\, E' \; x)) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new } \underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \ldots \\
\qquad \textbf{in} \\
\qquad\quad \ldots \\
\qquad \textbf{end} \\
\quad \mid \textbf{refl } (\text{app } E_1 \; E_2) = \ldots
\end{array}$$

In this extended context, we apply the induction hypothesis to expression $\ulcorner e' \urcorner = E' \; \underline{x}$ and obtain an object P, which is still defined in the extended context. Note that P represents a derivation $\mathcal{P} :: e' \stackrel{1}{\Longrightarrow} e'$ by the adequacy result from Lemma 3.3. But $\mathcal{P}$ is hypothetical in $u :: x \stackrel{1}{\Longrightarrow} x$ (and naturally in $x :: \text{term } \tau_1$).

$$\left\ulcorner \begin{array}{c} \overline{x \stackrel{1}{\Longrightarrow} x}^{\,u} \\ \mathcal{P} \\ e' \stackrel{1}{\Longrightarrow} e' \end{array} \right\urcorner = \Pi x : \text{term } \ulcorner \tau_1 \urcorner .\, \Pi u : x \stackrel{1}{\Longrightarrow} x .\, (E' \; x) \stackrel{1}{\Longrightarrow} (E' \; x) \tag{4.1}$$

In order to make $P = \ulcorner \mathcal{P} \urcorner$ available to the subsequent operations of this proof case, we insert another declaration into the body function **refl**.

$$
\begin{array}{l}
\textbf{fun refl } \underline{x} = \underline{u} \\
\quad \mid \textbf{refl } (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T.\, E'\ x)) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new } \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val}\ \ P\ \underline{x}\ \underline{u} = \textbf{refl}\ (E'\ \underline{x}) \\
\qquad \textbf{in} \\
\qquad\quad \ldots \\
\qquad \textbf{end} \\
\quad \mid \textbf{refl } (\mathrm{app}\ E_1\ E_2) = \ldots
\end{array}
$$

The derivation $\mathcal{P}$ matches the premiss of the plam-rule and we return "plam $(\lambda x : \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, P\ x\ u)$" which closes this case in the proof.

$$
\begin{array}{l}
\textbf{fun refl } \underline{x} = \underline{u} \\
\quad \mid \textbf{refl } (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T.\, E'\ x)) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new } \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val}\ \ P\ \underline{x}\ \underline{u} = \textbf{refl}\ (E'\ \underline{x}) \\
\qquad \textbf{in} \\
\qquad\quad \mathrm{plam}\ (\lambda x : \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, P\ x\ u) \\
\qquad \textbf{end} \\
\quad \mid \textbf{refl } (\mathrm{app}\ E_1\ E_2) = \ldots
\end{array}
$$

The representation of the final case in the proof of the reflexivity theorem does not present any new concepts or difficulties.

Case: $e = \dfrac{\begin{array}{cc} e_1 & e_2 \\ \mathrm{term}\ (\tau_2 \to \tau_1) & \mathrm{term}\ \tau_2 \end{array}}{\mathrm{term}\ \tau_1}\,\mathsf{app}$

$\mathcal{P}_1 :: e_1 \overset{1}{\Longrightarrow} e_1$ by i.h. on e_1

$\mathcal{P}_2 :: e_2 \overset{1}{\Longrightarrow} e_2$ by i.h. on e_2

$\mathcal{Q} :: \mathrm{app}\ e_1\ e_2 \overset{1}{\Longrightarrow} \mathrm{app}\ e_1\ e_2$ by rule papp on $\mathcal{P}_1, \mathcal{P}_2$

Two applications of the induction hypothesis provide two new objects representing derivations, P_1 and P_2 which form as pair the return value of this case. In order to compare the informal formal proof and its representation as a realizer, we repeat the proof here.

Proof: (of Lemma 3.4) by structural induction on e:

Case: $e = \dfrac{}{\mathrm{term}\ \tau_i}\, x_i$

$u_i :: x_i \overset{1}{\Longrightarrow} x_i$ by assumption

Case: $e = \dfrac{\begin{array}{c}\overline{\text{term } \tau_1}\ x_{n+1} \\ e' \\ \text{term } \tau_2\end{array}}{\text{term } (\tau_1 \to \tau_2)}\ \mathsf{lam}^{x_{n+1}}$

Assume $x_{n+1} :: \text{term } \tau_1$
Assume $u_{n+1} :: x_{n+1} \stackrel{1}{\Longrightarrow} x_{n+1}$
$\mathcal{P} :: e' \stackrel{1}{\Longrightarrow} e'$ by i.h. on e'
$\mathcal{Q} :: \lambda x_{n+1} : \text{term } \tau_1 .\, e' \stackrel{1}{\Longrightarrow} \lambda x_{n+1} : \text{term } \tau_1 .\, e'$ by rule plam on $\mathcal{P}$

Case: $e = \dfrac{\begin{array}{cc} e_1 & e_2 \\ \text{term } (\tau_2 \to \tau_1) & \text{term } \tau_2 \end{array}}{\text{term } \tau_1}\ \mathsf{app}$

$\mathcal{P}_1 :: e_1 \stackrel{1}{\Longrightarrow} e_1$ by i.h. on e_1
$\mathcal{P}_2 :: e_2 \stackrel{1}{\Longrightarrow} e_2$ by i.h. on e_2
$\mathcal{Q} :: \text{app } e_1\ e_2 \stackrel{1}{\Longrightarrow} \text{app } e_1\ e_2$ by rule papp on $\mathcal{P}_1, \mathcal{P}_2$

□

Proof: (realizer of Lemma 4.3)

- termination order is a subterm order on E
- using context schema "SOME T : tp. BLOCK x : term $T, u : x \stackrel{1}{\Longrightarrow} x$"

fun refl $\underline{x} = \underline{u}$
  | **refl** (lam (λx : term $T.\, E'\ x$)) =
    let
      new $\underline{x}$: term $T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$
      val $P\ \underline{x}\ \underline{u}$ = **refl** ($E'\ \underline{x}$)
    in
      plam (λx : term $T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, P\ x\ u$)
    end
  | **refl** (app $E_1\ E_2$) =
    let
      val P_1 = **refl** E_1
      val P_2 = **refl** E_2
    in
      papp $P_1\ P_2$
    end

□

fun partrans $\underline{x}$ $D_2 = D_2$
  | **partrans** pid $D_2 = D_2$
  | **partrans** (pstep D_1' D_1'') $D_2 =$
    let
      val $P =$ **partrans** D_1'' D_2
    in
      pstep D_1' P
    end

Figure 4.2: Formal proof of the transitivity Theorem 4.4.

This concludes the presentation of the formalization of the proof of the reflexivity lemma for parallel reduction and we continue with the formalization of the transitivity Lemma 3.5 and the substitution Lemma 3.6 both for parallel reduction. The formalization of the proof itself is very similar, almost identical to the one of transitivity Lemma 4.1 for ordinary reduction.

Lemma 4.4 (Transitivity of $\stackrel{*}{\Longrightarrow}$, formalized)

Let Φ be the dynamic extension of the world. If $\mathcal{D}_1 :: e \stackrel{*}{\Longrightarrow} e'$ and $\mathcal{D}_2 :: e' \stackrel{*}{\Longrightarrow} e''$ are closed then $e \stackrel{*}{\Longrightarrow} e''$.

$$= \Box\cdot.\forall T : tp.\forall E : term\,T.\forall E' : term\,T.\forall E'' : term\,T.$$
$$\forall D_1 : E \stackrel{*}{\Longrightarrow} E'.\forall D_2 : E' \stackrel{*}{\Longrightarrow} E''.\exists P : E \stackrel{*}{\Longrightarrow} E''.\top$$

Proof:

- termination order is a subterm order on D_1
- with an empty parameter context

Figure 4.2 shows the formal proof. □

The proof of the substitution lemma does not provide us with any new fundamental insights into how to formalize meta-theorems and meta-proofs either.

Lemma 4.5 (Substitution lemma, formalized)

Consider the situation where a list of the following assumptions is present

$$x_1 :: \mathit{term}\ \tau_1, u_1 :: x_1 \stackrel{1}{\Longrightarrow} x_1, \ldots, x_n :: \mathit{term}\ \tau_n, u_n :: x_n \stackrel{1}{\Longrightarrow} x_n$$

If

$$\begin{array}{c} \overline{y \stackrel{1}{\Longrightarrow} y}\ v \\ \mathcal{D} \\ e_1 \stackrel{1}{\Longrightarrow} e_1' \end{array}$$

and $\mathcal{E} :: e_2 \stackrel{1}{\Longrightarrow} e_2'$ *then exists a reduction* $e_1[e_2/y] \stackrel{1}{\Longrightarrow} e_1'[e_2'/y]$.

$$\begin{array}{l} \Box\text{SOME}\ T : \mathit{tp}.\ \text{BLOCK}\ x : \mathit{term}\ T, u : x \stackrel{1}{\Longrightarrow} x. \\ \quad \forall T_1 : \mathit{tp}. \forall T_2 : \mathit{tp}. \forall E_1 : \mathit{term}\ T_2 \to \mathit{term}\ T_1. \forall E_1' : \mathit{term}\ T_2 \to \mathit{term}\ T_1. \\ \quad \forall E_2 : \mathit{term}\ T_2. \forall E_2' : \mathit{term}\ T_2. \\ \qquad \forall D_1 : (\Pi y : \mathit{term}\ T_2. y \stackrel{1}{\Longrightarrow} y \to E_1\ y \stackrel{1}{\Longrightarrow} E_1'\ y). \forall D_2 : E_2 \stackrel{1}{\Longrightarrow} E_2'. \\ \qquad \exists P : E_1\ E_2 \stackrel{1}{\Longrightarrow} E_1'\ E_2'. \top \end{array}$$

As in the proof of Lemma 4.2 we have to perform a case analysis on the hypothetical derivation $\mathcal{D}$. Because it is hypothetical, $\ulcorner\mathcal{D}\urcorner = \lambda y : \text{term}\ T_2. \lambda v : y \stackrel{1}{\Longrightarrow} y. D'$ five different cases of D' have to be considered: D' could be either a parameter $\underline{u}$ declared in the dynamic extension of the world Φ, simply v, or an object starting with any of the three constants "pbeta", "plam", or "papp".

Proof:

- termination order is a subterm order on D_1
- using context schema "SOME T : tp. BLOCK x : term $T, u : x \stackrel{1}{\Longrightarrow} x$"

Figure 4.3 shows the formal proof. □

With the diamond lemma, arguably the most difficult lemma presented in Chapter 3, we shed some more light on the formalization process of the meta-theory of object languages such as programming languages and logics, and also the meta-logic we are going to present in Chapter 5. So far we have demonstrated how to formalize "proofs by structural induction" using several operations, such as case analysis, direct construction of witness objects, appeals to the induction hypothesis, and regular extensions of the world. In addition, the formalization of the diamond lemma requires appeals to lemmas and extensions of termination orders.

$$
\begin{array}{l}
\textbf{fun subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \underline{u})\ E = \underline{u} \\
\quad \mid \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v)\ E = E \\
\quad \mid \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \text{pbeta } (\lambda x :\ \text{term } T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, D_1\ y\ v\ x\ u)\ (D_2\ y\ v))\ E = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new } \underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val}\ \ P_1\ \underline{x}\ \underline{u} = \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, D_1\ y\ v\ \underline{x}\ \underline{u})\ E \\
\qquad \textbf{in} \\
\qquad\quad \textbf{let} \\
\qquad\qquad \textbf{val } P_2 = \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, D_2\ y\ v)\ E \\
\qquad\quad \textbf{in} \\
\qquad\qquad \text{pbeta } P_1\ P_2 \\
\qquad\quad \textbf{end} \\
\qquad \textbf{end} \\
\quad \mid \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \text{plam } (\lambda x :\ \text{term } T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, D_1\ y\ v\ x\ u))\ E = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new } \underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val}\ \ P_1\ \underline{x}\ \underline{u} = \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, D_1\ y\ v\ \underline{x}\ \underline{u})\ E \\
\qquad \textbf{in} \\
\qquad\quad \text{plam } P_1 \\
\qquad \textbf{end} \\
\quad \mid \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \text{papp } (D_1\ y\ v)\ (D_2\ y\ v))\ E = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val } P_1 = \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, D_1\ y\ v)\ E \\
\qquad\quad \textbf{val } P_2 = \textbf{subst } (\lambda y : \text{term } T'.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, D_2\ y\ v)\ E \\
\qquad \textbf{in} \\
\qquad\quad \text{papp } P_1\ P_2 \\
\qquad \textbf{end}
\end{array}
$$

Figure 4.3: Formal proof of the substitution Lemma 4.5.

Lemma 4.6 (Diamond lemma, formalized)

Let Φ be the dynamic extension of the world. If $\mathcal{D}^l :: e \stackrel{1}{\Longrightarrow} e^l$ and $\mathcal{D}^r :: e \stackrel{1}{\Longrightarrow} e^r$ then there exists a common reduct e', such that $\mathcal{R}^l :: e^l \stackrel{1}{\Longrightarrow} e'$ and $\mathcal{R}^r :: e^r \stackrel{1}{\Longrightarrow} e'$.

$$
\begin{array}{l}
\Box \text{SOME } T : tp.\ \text{BLOCK } x : term\, T, u : x \stackrel{1}{\Longrightarrow} x. \\
\qquad \forall T : tp. \forall E : term\, T. \forall E^l : term\, T. \forall E^r : term\, T. \\
\qquad\quad \forall D^l : E \stackrel{1}{\Longrightarrow} E^l. \forall D^r : E \stackrel{1}{\Longrightarrow} E^r. \\
\qquad\quad \exists E' : term\, T. \exists R^l : E^l \stackrel{1}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{1}{\Longrightarrow} E'. \top
\end{array}
$$

As we have presented the proof of the diamond Lemma 3.7 in Chapter 3, it proceeds by simultaneous structural induction over the derivations $\mathcal{D}^l$ and $\mathcal{D}^r$. Specifically, an induction hypothesis is applicable to two parallel reductions $\mathcal{D}'^l$ and $\mathcal{D}'^r$ given that $\mathcal{D}'^l$ is a subderivation of $\mathcal{D}^l$ and $\mathcal{D}'^r$ is either equal to or also a subderivation of $\mathcal{D}^r$. Formally, the proof principle "proof by simultaneous structural induction" is represented by a new termination order, a simultaneous extension of the subterm ordering. We write $[D^l\ D^r]$ for this new termination order and it is defined as follows: A pair of objects $[D'^l\ D'^r]$ is smaller than $[D^l\ D^r]$, if either

> D'^l is structurally smaller than D^l and either $D'^r = D^r$ or D'^r is a structurally smaller than D^r

or

> either $D'^l = D^l$ or D'^l is a structurally smaller than D^l and D'^r is structurally smaller than D^r.

Another very common termination principle is "proof by lexicographical structural induction", which we will not demonstrate by example but merely state here. It is used for example in the proof of cut-elimination for various logics [Pfe95].

A proof by lexicographical induction on $\mathcal{D}^l$ and $\mathcal{D}^r$ provides induction hypothesis, which can be applied to terms $\mathcal{D}'^l$ and $\mathcal{D}'^r$ as long as $\mathcal{D}'^l$ is a subderivation of $\mathcal{D}^l$ and $\mathcal{D}'^r$ is arbitrary, or $\mathcal{D}'^l = \mathcal{D}^l$, and $\mathcal{D}'^r$ is a subderivation of $\mathcal{D}^r$. Formally, we write $\{D^l, D^r\}$ for the lexicographical termination ordering. We say that $\{D'^l, D'^r\}$ is below $\{D^l, D^r\}$, if either

> D'^l is structurally smaller than D^l, and D'^r might be arbitrary

or

> $D'^l = D^l$ and D'^r is structurally smaller than D^r.

Termination orderings based on simultaneous and lexicographical extensions of the subterm ordering have been studied in [RP96]. We reuse those results in order to prove that each recursive function formalizing a meta-proof is terminating. Recall that realizers must be total functions, specifically upon instantiation they must terminate and the execution can never get stuck. Termination is enforced by allowing only recursive calls on argument vector that are smaller according to some a priori specified well-founded termination order.

The first two cases of the proof of Lemma 4.6 deserve special attention. We start with the discussion of the base case:

Case: $\mathcal{D}^l = \dfrac{}{x \stackrel{1}{\Longrightarrow} x}\, u \qquad \mathcal{D}^r = \dfrac{}{x \stackrel{1}{\Longrightarrow} x}\, u$

$e' = x$	by assumption
$\mathcal{R}^l = \mathcal{R}^r = u$	by assumption

How did this case come about? First, we distinguish cases on the derivation $\mathcal{D}^l$ and consider the global case, where x :: term τ, and $u :: x \stackrel{1}{\Longrightarrow} x$. Second, we distinguish cases on $\mathcal{D}^r$, and because the parallel reduction $\mathcal{D}^r$ starts with the same term x, we conclude, that the only possible instantiation of $\mathcal{D}^r$ is u. There are no other cases to be considered for $\mathcal{D}^r$.

Formally, the proof of the diamond lemma is expressed by a function mapping two representations of parallel reductions D^l and D^r to two other parallel reductions R^l and R^l in order to form a diamond — graphically speaking.

$$\textbf{fun dia } D^l\ D^r = \ldots$$

First we distinguish cases of D^l. For brevity, we only show the global parameter and the β-rule case.

$$
\begin{array}{l}
\textbf{fun dia}\ \underline{u}\ D^r = \ldots \\
\quad \mid \textbf{dia}\ (\text{pbeta}\ (\lambda x : \ \text{term}\ T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l)\ D^r = \ldots \\
\quad \vdots
\end{array}
$$

Assume, that there is one parameter context containing several parameter blocks, and each parameter block is an instance of the given block schema. At this point D^r is instantiated with some $\underline{u}$ of one of the parameter blocks. It is not clear if it is the first, the second, or the last, all we know, that there is one it is instantiated to. Clearly, D^r is a derivation which reduces $\underline{x}$ (the other assumptions associated with the parameter block which contains u) to some term e^r.

Next we have to consider all cases for D^r. Again there are several cases to be considered. The first case to try is that $D^r = \underline{v}$ assuming that the regular world extension contains a parameter block of the form $\underline{y}, \underline{v}$. Hence $D^r : \underline{y} \stackrel{1}{\Longrightarrow} \underline{y}$. We notice, that this can only be the case if $\underline{x}$ and $\underline{y}$ refer to the same parameter in the same parameter block in Φ, since from the case analysis on D^l we can infer that $D^r : \underline{x} \stackrel{1}{\Longrightarrow} E^r$. Therefore $D^r = \underline{u} = \underline{v}$ and $\underline{x} = \underline{y}$. This is the first possible form of D^r. It is also the only possible form of D^r, because any other instantiation of D^r whose head constant is defined in the signature clashes with the fact that D^r stands for a reduction of $\underline{x}$.

$$
\begin{array}{l}
\textbf{fun dia}\ \underline{u}\ \underline{u} = \ldots \\
\quad \mid \textbf{dia}\ (\text{pbeta}\ (\lambda x : \ \text{term}\ T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l)\ D^r = \ldots \\
\quad \vdots
\end{array}
$$

Why is this kind of argument sound? It is sound, because we start with a minimal amount of information, namely that there exists a second parameter block in the parameter context, and it is only because of additional constraints that we can identify it with one whose existence we have already assumed. In order to close this proof branch, we simply return the pair $(\underline{u}, \underline{u})$.

$$
\begin{array}{l}
\textbf{fun dia}\ \underline{u}\ \underline{u} = (\underline{u}, \underline{u}) \\
\quad \mid \textbf{dia}\ (\text{pbeta}\ (\lambda x : \ \text{term}\ T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l)\ D^r = \ldots \\
\quad \vdots
\end{array}
$$

The second case of the proof demonstrates an appeal to the a lemma. It is the substitution Lemma 3.6 discussed above.

Case:

$$\mathcal{D}^l = \dfrac{\begin{matrix} \dfrac{}{x \overset{1}{\Longrightarrow} x}\,u \\ \mathcal{D}_1^l \\ e_1 \overset{1}{\Longrightarrow} e_1^l \end{matrix} \qquad \begin{matrix} \mathcal{D}_2^l \\ e_2 \overset{1}{\Longrightarrow} e_2^l \end{matrix}}{(\lambda x : \tau . e_1)\ e_2 \overset{1}{\Longrightarrow} e_1^l[e_2^l/x]}\ \mathsf{pbeta}^u$$

$$\mathcal{D}^r = \dfrac{\begin{matrix} \dfrac{}{x \overset{1}{\Longrightarrow} x}\,u \\ \mathcal{D}_1^r \\ e_1 \overset{1}{\Longrightarrow} e_1^r \end{matrix} \qquad \begin{matrix} \mathcal{D}_2^r \\ e_2 \overset{1}{\Longrightarrow} e_2^r \end{matrix}}{(\lambda x : \tau . e_1)\ e_2 \overset{1}{\Longrightarrow} e_1^r[e_2^r/x]}\ \mathsf{pbeta}^u$$

Extend Φ by $x : \text{term } \tau, u :: x \overset{1}{\Longrightarrow} x$ to Φ'
There exists an e_1'
$\mathcal{P}_1 :: e_1^l \overset{1}{\Longrightarrow} e_1'$
$\mathcal{P}_2 :: e_1^r \overset{1}{\Longrightarrow} e_1'$ by i.h. on $\mathcal{D}_1^l, \mathcal{D}_1^r$ in Φ'
There exists an e_2'
$\mathcal{Q}_1 :: e_2^l \overset{1}{\Longrightarrow} e_2'$
$\mathcal{Q}_2 :: e_2^r \overset{1}{\Longrightarrow} e_2'$ by i.h. on $\mathcal{D}_2^l, \mathcal{D}_2^r$ in Φ
$\mathcal{R}_1 :: e_1^l[e_2^l/x] \overset{1}{\Longrightarrow} e_1'[e_2'/x]$ by Lemma 3.6 on $\mathcal{P}_1, \mathcal{Q}_1$
$\mathcal{R}_2 :: e_1^r[e_2^r/x] \overset{1}{\Longrightarrow} e_1'[e_2'/x]$ by Lemma 3.6 on $\mathcal{P}_2, \mathcal{Q}_2$

And again as in the previous case, an analysis of the second derivations leaves only one case. After two more appeals to the induction hypothesis we obtain the following partially defined realizer **dia**.

$$\begin{array}{l}
\textbf{fun dia}\ \underline{u}\ \underline{u} = (\underline{u}, \underline{u}) \\
\quad \mid\ \textbf{dia}\ (\text{pbeta}\ (\lambda x :\ \text{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l) \\
\qquad\quad (\text{pbeta}\ (\lambda x :\ \text{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^r\ x\ u)\ D_2^r) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new}\ \underline{x} : \text{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \textbf{dia}\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1^r\ \underline{x}\ \underline{u}) \\
\qquad \textbf{in} \\
\qquad\quad \textbf{let} \\
\qquad\qquad \textbf{val}\ (Q_1, Q_2) = \textbf{dia}\ D_2^l\ D_2^r \\
\qquad\qquad \ldots \\
\qquad\quad \textbf{in} \\
\qquad\qquad \ldots \\
\qquad\quad \textbf{end} \\
\qquad \textbf{end} \\
\quad \vdots
\end{array}$$

According to the informal proof the only steps missing to close this branch of the proof are two appeals to the substitution Lemma 3.6. In order to apply a lemma, we first have to ensure that the context scheme of the lemma to be proven (i.e. the diamond lemma) and the lemma to be applied (i.e. Lemma 3.6) are compatible. In a nutshell, a lemma cannot be applied in a parameter context which is larger than the one in which the lemma is proven, in the sense, that the lemma must always guarantee coverage of all cases. These considerations establish a notion of subsumption on context schemas which we investigate in more detail in Section 5.7.2.

The context schema associated with the proof of substitution Lemma 4.5 and the context schema associated with the diamond lemma are equal, which informally implies that the substitution lemma covers all cases. More specifically, it is safe to appeal to the substitution in the proof of the diamond lemma.

Having checked the subsumption property of the context schemas, the application of lemma translates to function application on the meta-level. **subst** formalizes the proof of the substitution lemma in form of a recursive function; applying this function yields objects representing the derivations whose existence is guaranteed by the lemma. Specifically, this case of the proof requires two appeals to the substitution lemma which yield two objects E_1 and E_2.

$$
\begin{array}{l}
\textbf{fun dia}\ \underline{u}\ \underline{u} = (\underline{u}, \underline{u}) \\
\quad \mid \textbf{dia}\ (\text{pbeta}\ (\lambda x:\ \text{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l) \\
\qquad\quad (\text{pbeta}\ (\lambda x:\ \text{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^r\ x\ u)\ D_2^r) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new}\ \underline{x} : \text{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \textbf{dia}\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1^r\ \underline{x}\ \underline{u}) \\
\qquad \textbf{in} \\
\qquad\quad \textbf{let} \\
\qquad\qquad \textbf{val}\ (Q_1, Q_2) = \textbf{dia}\ D_2^l\ D_2^r \\
\qquad\qquad \textbf{val}\ R_1 = \textbf{subst}\ P_1\ Q_1 \\
\qquad\qquad \textbf{val}\ R_2 = \textbf{subst}\ P_2\ Q_2 \\
\qquad\quad \textbf{in} \\
\qquad\qquad \ldots \\
\qquad\quad \textbf{end} \\
\qquad \textbf{end} \\
\quad \vdots
\end{array}
$$

As a matter of fact, R_1 and R_2 are the required two derivations which the function formalizing this proof has to return. Therefore, filling the last hole in the body of the **let** clause with (R_1, R_2) closes the proof branch.

$$
\begin{array}{l}
\textbf{fun dia}\ \underline{u}\ \underline{u} = (\underline{u}, \underline{u}) \\
\quad \mid \textbf{dia}\ (\text{pbeta}\ (\lambda x : \text{ term } T.\ \lambda u : x \overset{1}{\Longrightarrow} x.\ D_1^l\ x\ u)\ D_2^l) \\
\qquad\quad (\text{pbeta}\ (\lambda x : \text{ term } T.\ \lambda u : x \overset{1}{\Longrightarrow} x.\ D_1^r\ x\ u)\ D_2^r) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new}\ \underline{x} : \text{term } T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \textbf{dia}\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1^r\ \underline{x}\ \underline{u}) \\
\qquad \textbf{in} \\
\qquad\quad \textbf{let} \\
\qquad\qquad \textbf{val}\ (Q_1, Q_2) = \textbf{dia}\ D_2^l\ D_2^r \\
\qquad\qquad \textbf{val}\ R_1 = \textbf{subst}\ P_1\ Q_1 \\
\qquad\qquad \textbf{val}\ R_2 = \textbf{subst}\ P_2\ Q_2 \\
\qquad\quad \textbf{in} \\
\qquad\qquad (R_1, R_2) \\
\qquad\quad \textbf{end} \\
\qquad \textbf{end} \\
\quad \vdots
\end{array}
$$

The remaining cases are easily represented using the same techniques presented in this chapter. The diamond lemma is therefore correct, and the function **dia** a formalization of its proof, keeping in mind the context schema which was used to determine all the cases.

Proof: of Lemma 4.6:

- termination order is a subterm order on $[D^l\ D^r]$
- using context schema "SOME T : tp. BLOCK x : term $T, u : x \overset{1}{\Longrightarrow} x$"

Figure 4.4 shows the formal proof. □

The diamond lemma is used in the proof of the strip lemma. It guarantees that a multi-step parallel reduction and a single-step parallel reduction have a common reduct. The theorem need not to be as general as the reflexivity Lemma 4.3, the substitution Lemma 4.5, or the diamond Lemma 4.6; we assume the parameter context to be empty. Naturally, since every empty parameter context is also a parameter context of the context schema "SOME T : tp. BLOCK x : term $T, u : x \overset{1}{\Longrightarrow} x$", the diamond lemma can be used for the proof of the strip lemma.

Lemma 4.7 (Strip lemma, formalized)

> *Let Φ be the dynamic extension of the world. If $\mathcal{D}^l :: e \overset{1}{\Longrightarrow} e^l$ and $\mathcal{D}^r :: e \overset{*}{\Longrightarrow} e^r$ then there exists a common reduct e', such that $\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e'$ and $\mathcal{R}_2 :: e^r \overset{1}{\Longrightarrow} e'$.*

$$
\begin{array}{l}
\Box\cdot.\ \forall T : tp.\ \forall E : term\ T.\ \forall E^l : term\ T.\ \forall E^r : term\ T. \\
\qquad\qquad \forall D^l : E \overset{1}{\Longrightarrow} E^l.\ \forall D^r : E \overset{*}{\Longrightarrow} E^r. \\
\qquad\qquad \exists E' : term\ T.\ \exists R^l : E^l \overset{*}{\Longrightarrow} E'.\ \exists R^r : E^r \overset{1}{\Longrightarrow} E'.\ \top
\end{array}
$$

$$
\begin{array}{l}
\mathbf{fun\ dia}\ \underline{u}\ \underline{u} = (\underline{u}, \underline{u}) \\
\quad \mid\ \mathbf{dia}\ (\mathrm{pbeta}\ (\lambda x :\ \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l)\ (\mathrm{pbeta}\ (\lambda x :\ \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^r\ x\ u)\ D_2^r) = \\
\qquad \mathbf{let} \\
\qquad\quad \mathbf{new}\ \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \mathbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \mathbf{dia}\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1^r\ \underline{x}\ \underline{u}) \\
\qquad \mathbf{in} \\
\qquad\quad \mathbf{let} \\
\qquad\qquad \mathbf{val}\ (Q_1, Q_2) = \mathbf{dia}\ D_2^l\ D_2^r \\
\qquad\qquad \mathbf{val}\ E_1 = \mathbf{subst}\ P_1\ Q_1 \\
\qquad\qquad \mathbf{val}\ E_2 = \mathbf{subst}\ P_2\ Q_2 \\
\qquad\quad \mathbf{in} \\
\qquad\qquad (E_1, E_2) \\
\qquad\quad \mathbf{end} \\
\qquad \mathbf{end} \\
\quad \mid\ \mathbf{dia}\ (\mathrm{pbeta}\ (\lambda x :\ \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l)\ (\mathrm{papp}\ (\mathrm{plam}\ (\lambda x :\ \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1'^r\ x\ u))\ D_2^r) = \\
\qquad \mathbf{let} \\
\qquad\quad \mathbf{new}\ \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \mathbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \mathbf{dia}\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1'^r\ \underline{x}\ \underline{u}) \\
\qquad \mathbf{in} \\
\qquad\quad \mathbf{let} \\
\qquad\qquad \mathbf{val}\ (Q_1, Q_2) = \mathbf{dia}\ D_2^l\ D_2^r \\
\qquad\qquad \mathbf{val}\ E_1 = \mathbf{subst}\ P_1\ Q_1 \\
\qquad\quad \mathbf{in} \\
\qquad\qquad (E_1, \mathrm{pbeta}\ P_2\ Q_2) \\
\qquad\quad \mathbf{end} \\
\qquad \mathbf{end} \\
\quad \mid\ \mathbf{dia}\ (\mathrm{plam}\ (\lambda x :\ \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^l\ x\ u))\ (\mathrm{plam}\ (\lambda x :\ \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^r\ x\ u)) = \\
\qquad \mathbf{let} \\
\qquad\quad \mathbf{new}\ \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \mathbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \mathbf{dia}\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1^r\ \underline{x}\ \underline{u}) \\
\qquad \mathbf{in} \\
\qquad\quad (\mathrm{plam}\ P_1, \mathrm{plam}\ P_2) \\
\qquad \mathbf{end} \\
\quad \mid\ \mathbf{dia}\ (\mathrm{papp}\ (\mathrm{plam}\ (\lambda x :\ \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1'^l\ x\ u))\ D_2^l)\ (\mathrm{pbeta}\ (\lambda x :\ \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^r\ x\ u)\ D_2^r) = \\
\qquad \mathbf{let} \\
\qquad\quad \mathbf{new}\ \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \mathbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \mathbf{dia}\ (D_1'^l\ \underline{x}\ \underline{u})\ (D_1^r\ \underline{x}\ \underline{u}) \\
\qquad \mathbf{in} \\
\qquad\quad \mathbf{let} \\
\qquad\qquad \mathbf{val}\ (Q_1, Q_2) = \mathbf{dia}\ D_2^l\ D_2^r \\
\qquad\qquad \mathbf{val}\ E_2 = \mathbf{subst}\ P_2\ Q_2 \\
\qquad\quad \mathbf{in} \\
\qquad\qquad (\mathrm{pbeta}\ P_1\ Q_1, E_2) \\
\qquad\quad \mathbf{end} \\
\qquad \mathbf{end} \\
\quad \mid\ \mathbf{dia}\ (\mathrm{papp}\ D_1^l\ D_2^l)\ (\mathrm{papp}\ D_1^r\ D_2^r) = \\
\qquad \mathbf{let} \\
\qquad\quad \mathbf{val}\ (P_1, P_2) = \mathbf{dia}\ D_1^l\ D_1^r \\
\qquad\quad \mathbf{val}\ (Q_1, Q_2) = \mathbf{dia}\ D_2^l\ D_2^r \\
\qquad \mathbf{in} \\
\qquad\quad (\mathrm{papp}\ P_1\ Q_1, \mathrm{papp}\ P_2\ Q_2) \\
\qquad \mathbf{end}
\end{array}
$$

Figure 4.4: Formal proof of the diamond Lemma 3.7

$$
\begin{array}{l}
\textbf{fun strip } D^l \text{ pid} = (\text{pid}, D^l) \\
\quad | \ \textbf{strip } D^l \ (\text{pstep } D_1^r \ D_2^r) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val } (P_1, P_2) = \textbf{dia } D^l \ D_1^r \\
\qquad\quad \textbf{val } (P_3, E_2) = \textbf{strip } P_2 \ D_2^r \\
\qquad \textbf{in} \\
\qquad\quad (\text{pstep } P_1 \ P_3, E_2) \\
\qquad \textbf{end}
\end{array}
$$

Figure 4.5: Formal proof of the strip Lemma 3.8

Proof:

- termination order is a subterm order on D^r
- with an empty parameter context

Figure 4.5 shows the formal proof. □

The confluence lemma, a generalization of the strip lemma, by allowing both given reductions to be multi-step reduction, relies on the strip lemma in its proof as the reader might recall from Section 3.2.3.

Lemma 4.8 (Confluence lemma, formalized)

> *Let Φ be the dynamic extension of the world. If $\mathcal{D}^l :: e \overset{*}{\Longrightarrow} e^l$ and $\mathcal{D}^r :: e \overset{*}{\Longrightarrow} e^r$ then there exists a common reduct e', such that $\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e'$ and $\mathcal{R}_2 :: e^r \overset{*}{\Longrightarrow} e'$.*

$$
\begin{array}{l}
\Box\cdot. \forall T : tp. \forall E : term\, T. \forall E^l : term\, T. \forall E^r : term\, T. \\
\qquad \forall D^l : E \overset{*}{\Longrightarrow} E^l. \forall D^r : E \overset{*}{\Longrightarrow} E^r. \\
\qquad \exists E' : term\, T. \exists R^l : E^l \overset{*}{\Longrightarrow} E'. \exists R^r : E^r \overset{*}{\Longrightarrow} E'. \top
\end{array}
$$

Proof:

- termination order is a subterm order on D^l
- with an empty parameter context

Figure 4.6 shows the formal proof. □

In the proof of the Church-Rosser theorem, all our results so far flow together. The interesting case is transitivity: Two appeals to the induction hypothesis, one application to the confluence lemma, and finally two appeals to the transitivity lemma for parallel reduction conclude that any two parallel convertible terms have a common reduct.

$$
\begin{array}{l}
\textbf{fun conf } \mathrm{pid}\ D^r = (D^r, \mathrm{pid}) \\
\quad |\ \textbf{conf } (\mathrm{pstep}\ D_1^l\ D_2^l)\ D^r = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val } (P_1, P_2) = \textbf{strip } D_1^l\ D^r \\
\qquad\quad \textbf{val } (E_1, P_3) = \textbf{conf } D_2^l\ P_1 \\
\qquad \textbf{in} \\
\qquad\quad (E_1, \mathrm{pstep}\ P_2\ P_3) \\
\qquad \textbf{end}
\end{array}
$$

Figure 4.6: Formal proof of the confluence Lemma 3.9

$$
\begin{array}{l}
\textbf{fun cr } (\mathrm{pred}\ D_1) = (D_1, \mathrm{pid}) \\
\quad |\ \textbf{cr } (\mathrm{pexp}\ D_1) = (\mathrm{pid}, D_1) \\
\quad |\ \textbf{cr } (\mathrm{ptrans}\ D_1\ D_2) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val } (P_1, P_2) = \textbf{cr } D_1 \\
\qquad\quad \textbf{val } (P_3, P_4) = \textbf{cr } D_2 \\
\qquad\quad \textbf{val } (Q_1, Q_2) = \textbf{conf } P_2\ P_3 \\
\qquad\quad \textbf{val } E_1 = \textbf{partrans } P_1, Q_1 \\
\qquad\quad \textbf{val } E_2 = \textbf{partrans } P_2, Q_2 \\
\qquad \textbf{in} \\
\qquad\quad (E_1, E_2) \\
\qquad \textbf{end}
\end{array}
$$

Figure 4.7: Formal proof of the Church-Rosser Theorem 3.10 for parallel reduction

Theorem 4.9 (Church-Rosser theorem for parallel reduction, formalized)

Let Φ be the dynamic extension of the world. If $\mathcal{D} :: e^l \Longleftrightarrow e^r$ then there exists a common reduct e', such that $\mathcal{R}_1 :: e^l \stackrel{*}{\Longrightarrow} e'$ and $\mathcal{R}_2 :: e^r \stackrel{*}{\Longrightarrow} e'$.

$$
\begin{array}{l}
\Box\cdot.\forall T : tp.\forall E^l : term\,T.\forall E^r : term\,T. \\
\qquad \forall D : E^l \Longleftrightarrow E^r. \\
\qquad \exists E' : term\,T.\exists R^l : E^l \stackrel{*}{\Longrightarrow} E'.\exists R^r : E^r \stackrel{*}{\Longrightarrow} E'.\top
\end{array}
$$

Proof:

- termination order is a subterm order on D
- with an empty parameter context

Figure 4.7 shows the formal proof. □

This concludes our presentation of the formalization of meta-theorems and meta-proofs related to ordinary and parallel reductions. One could continue with the presentation of the proofs of Lemma 3.11-3.14 from Section 3.2.4 and the interested reader is invited to do so, but we prefer to leave them to the automated theorem prover, which will be presented in Chapter 8.

4.2.3 More on Meta-Theorems

The formalization techniques motivated in the previous chapter are not complete. We have omitted two important techniques, which we discuss in this section.

First, note that all parameter contexts presented in the previous section were generated by at most one block schema. This is not always the case. In general, context schemas consist of many block schemas, which makes it necessary to label different parameter blocks in a parameter context in order to reconstruct which context block is an instance of which block schema. In particular, when we extend the simply typed λ-calculus by polymorphism we also have to generalize the induction hypothesis of the entire sequence the theorems accordingly.

Second, there are many theorems which must be proven by mutual induction. All theorems from the previous section were provable on their own without mutually relying on any other lemma. Consider for example the reflexivity result for a normalized version of the simply-typed λ-calculus, where we distinguish between atomic and canonical forms. The definition of canonical forms relies on the definition of atomic forms, and this circularity must be reflected in the meta-logic.

Context schemas

Context schemas inductively and abstractly describe all admissible parameter contexts. In the previous section we have encountered one form of a context schema which is described by one block schema: "SOME T : tp. BLOCK x : term $T, u : x \stackrel{1}{\Longrightarrow} x$". In general, one block schema is not enough, since parameters can be introduced anywhere into the proof, and they may not always look the same. In order to demonstrate this effect, we slightly extend our version of the simply-typed λ-calculus from Figure 2.2 by polymorphism. On the type level, we add type variables α and a type quantifier $\forall\alpha.\tau$ which binds all free occurrences of the type variable α in τ. The following extends the definition of types from Section 2.2.

$$\textit{Types:} \quad \tau \quad ::= \quad \ldots \mid \alpha \mid \forall\alpha.\tau$$

Those new types can be adequately represented using higher-order abstract syntax, which means in this context that type variables are represented by LF variables: $\ulcorner\alpha\urcorner = \alpha$.

$$\text{all} \quad : \quad (\text{tp} \to \text{tp}) \to \text{tp}$$

The changes in the type system reflect on the syntactic category of terms in a natural way. On the one hand, there are polymorphic terms which expect a type as argument in order to specialize the type of the body. And on the other hand, there is an application operator which applies polymorphic terms to types and hence executes the specialization.

$$\textit{Terms:} \quad e \quad ::= \quad \ldots \mid \Lambda\alpha.e \mid e \cdot \tau$$

The term $\Lambda\alpha.e$ is well-typed of type $\forall\alpha.\tau$, if e is well-typed, assuming α as a new type, and $e \cdot \tau'$ is well-typed of type $\tau[\tau'/\alpha]$ if e has type $\forall\alpha.\tau$ and τ' is a type. As one might already suspect, this extended notion of terms can be adequately represented in the logical framework.

$$\begin{array}{lcl} \text{tlam} & : & (\Pi\alpha : \text{tp}.\, \text{term}\ (T\ \alpha)) \to \text{term}\ (\text{all}\ (\lambda\alpha : \text{tp}.\, T\ \alpha)) \\ \text{tapp} & : & \text{term}\ (\text{all}\ (\lambda\alpha : \text{tp}.\, T_1\ \alpha)) \to \Pi T_2 : \text{tp}.\, \text{term}\ (\text{all}\ (\lambda\alpha : \text{tp}.\, T_1\ \alpha)\ T_2) \end{array}$$

Finally, we extend the parallel reduction relation from Section 3.2.2 with reduction rules for type abstraction and type application. The rules are entirely straightforward.

$$\frac{e \overset{1}{\Longrightarrow} e'}{\Lambda\alpha.e \overset{1}{\Longrightarrow} \Lambda\alpha.e'}\ \mathsf{ptlam} \qquad \frac{e \overset{1}{\Longrightarrow} e'}{e \cdot \tau \overset{1}{\Longrightarrow} e' \cdot \tau}\ \mathsf{ptapp}$$

In addition, they can be adequately represented in the logical framework.

$$\begin{array}{lcl} \text{ptlam} & : & (\Pi\alpha : \text{tp}.\, E\ \alpha \overset{1}{\Longrightarrow} E'\ \alpha) \\ & & \to \text{tlam}\ (\lambda\alpha : \text{tp}.\, E\ \alpha) \overset{1}{\Longrightarrow} \text{tlam}\ (\lambda\alpha : \text{tp}.\, E'\ \alpha) \\ \text{ptapp} & : & E \overset{1}{\Longrightarrow} E' \\ & & \to \text{tapp}\ E\ T \overset{1}{\Longrightarrow} \text{tapp}\ E'\ T \end{array}$$

This concludes the presentation of an polymorphic extension of the simply typed λ-calculus. After extending it, one has to verify that the series of lemmas leading to the Church-Rosser theorem still hold. They could be invalidated by extending the underlying deductive systems, and indeed they are. Already the first theorem, namely the reflexivity property of the parallel reduction relation (Lemma 3.4) does not hold anymore. Why not? In the original version of the lemma we assumed the context to be

$$x_1 :: \text{term}\ \tau_1, u_1 :: x_1 \overset{1}{\Longrightarrow} x_1, \ldots, x_n :: \text{term}\ \tau_n, u_n :: x_n \overset{1}{\Longrightarrow} x_n \tag{4.2}$$

But this is not enough in order to prove reflexivity for the polymorphic parallel reduction. In the ptlam case, we have to traverse a λ-binder that binds a type variable α! But this assumption does not fit into the overall structure of the assumption list (4.2). In general, we might assume the presence of several type variables:

$$\alpha_1 :: \text{tp}, \ldots, \alpha_m :: \text{tp} \tag{4.3}$$

Assumption lists (4.2) and (4.3) may be arbitrarily interspersed while still respecting parameter block boundaries. When formalizing the generalized version of the reflexivity lemma, we must provide for these additional assumptions by adding a new block schema, in this case BLOCK a : tp, to the context schema.

Lemma 4.10 (Reflexivity theorem for polymorphic parallel reduction, formalized)

> *Let*
>
> $$\Phi \quad ::= \quad \cdot \mid \Phi, x :: \mathit{term}\ \tau, u :: x \overset{1}{\Longrightarrow} x \mid \Phi, \alpha :: \mathit{tp}$$
>
> *a regularly formed extension of the world. Then for any well-typed term e, there exists a derivation of* $e \overset{1}{\Longrightarrow} e$.

$$\begin{array}{l} = \Box\text{SOME}\ T : \mathit{tp}.\ \text{BLOCK}\ x : \mathit{term}\ T, u : x \overset{1}{\Longrightarrow} x | \text{BLOCK}\ a : \mathit{tp}. \\ \qquad \forall T : \mathit{tp}. \forall E : \mathit{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top \end{array}$$

Therefore, context schemas are defined as a list of block schemas, and in order to identify different occurrences of parameter blocks as instances of the same block schema, we assign a necessarily unique label to each block schema. The first context block is labeled L_1, and the second is labeled L_2.

$$\Box(\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \overset{1}{\Longrightarrow} x)^{L_1} \mid (\text{BLOCK } a : \text{tp})^{L_2}.$$
$$\forall T : \text{tp}. \forall E : \text{term } T. \exists D : E \overset{1}{\Longrightarrow} E. \top$$

This concludes the discussion on more complex context schemas. We continue with a brief overview about mutually dependent meta-theorems.

Mutually dependent meta-theorems

We say that two or more meta-theorems are mutually dependent, if none of them can be proved without the others. Mutually dependent theorems occur frequently in the formal theory of programming languages and logics. Often they are needed if the argument proceeds by induction over the derivation of a judgment (or several, depending on the termination order) which mutually depends on another. Consider for example our definition of canonical forms from Section 2.4.3. Canonical forms are defined in terms of atomic forms, and atomic forms are defined in terms of canonical forms.

Below we define canonical forms for the simply-typed λ-calculus (without dependencies). In this setting proving some property P for canonical forms typically requires another property Q to be proven for atomic forms. Consider for example, the proof that canonical forms enjoy the reflexivity property; it is also necessary to show that this property holds for atomic forms.

We omit the informal presentation of canonical and atomic forms and instead simply describe their representation in LF. There are two type families can and atm which represent well-typed canonical and well-typed atomic forms.

$$\begin{array}{lcl} \text{can} & : & \text{tp} \to \text{type} \\ \text{atm} & : & \text{tp} \to \text{type} \end{array}$$

Using these two type families, application and λ-abstraction are easily represented, and for coercion purposes, there is a rule very similar to canatm.

$$\begin{array}{lcl} \text{eapp} & : & \text{can } (T_2 \text{ arrow } T_1) \to \text{atm } T_2 \to \text{atm } T_1 \\ \text{elam} & : & (\text{atm } T_1 \to \text{can } T_2) \to \text{can } (T_1 \text{ arrow } T_2) \\ \text{eca} & : & \text{atm } T \to \text{can } T \end{array}$$

Intuitively, each closed canonical term is well-typed. As expected this lemma cannot be proven directly. First, it must be generalized to account for closed atomic terms, which are clearly well-typed, too. But this is still not enough. When reasoning inductively about canonical forms, one notices quickly that terms may be open with respect to a set of atomic well-typed variables.

Lemma 4.11 (Embedding) *Consider the situation where a list of the following assumptions is present*

$$x_1 :: atm\ \tau_1, y_1 :: term\ \tau_1, \ldots, x_n :: atm\ \tau_n, y_n :: term\ \tau_n$$

- *Every canonical form e_c is well-typed*

- *Every atomic form e_a is well-typed*

Proof: by mutual induction over e_a and e_c. □

In our meta-logic, this theorem is formalized by using conjunction.

Lemma 4.12 (Embedding (formalized))

$$\Box\text{SOME } T : tp.\ \text{BLOCK } x : atm\,T, y : term\,T.$$
$$(\forall T : tp. \forall E^c : can\,T. \exists E' : term\,T. \top) \wedge (\forall T : tp. \forall E^a : atm\,T. \exists E' : term\,T. \top)$$

But how can we guarantee termination of the recursive function corresponding to the proof of this theorem? In order to answer this question, we have to generalize the notion of termination orders. From an abstract point of view, the realizer formalizing the first and the second part of the theorem call each other recursively. In order to ensure termination, we must guarantee that the argument to the functions always decreases in size according to some well-founded measure. Recall, that in this thesis the measure of choice is the subterm relation. Specifically, when the function representing the first part calls the other with some E^a, we always enforce E^a to be a subterm of the original argument term E^c. Similarly, when the second function calls the first with argument E^c, E^c must be smaller than or equal to the initial argument E^a. This termination order is expressed formally as $(E^c\ E^a)$. Note that there is an important difference between a termination order which expresses simultaneous induction $[D^l\ D^r]$ as in the proof of the diamond Lemma 4.6, for example, and the one for mutual induction.

Proof: of Lemma 4.12

- termination order is a subterm order on (E^c, E^a)
- using context schema "SOME T : tp. BLOCK x : atm T, u : term T"

Figure 4.8 shows the formal proof. □

We conclude this subsection with a final remark about applying the induction hypothesis under a local extension of the parameter context. In the formalization of the reflexivity Lemma 3.4 for parallel reduction, we extend the regular world (or formally the parameter context) by two new parameters before we apply the induction hypothesis. First, we assumed that $\underline{x}$ is a term of type $\ulcorner \tau_1 \urcorner$, and second that it reduces in parallel to itself: $\underline{x} \overset{1}{\Longrightarrow} \underline{x}$. After appealing to the induction hypothesis, or functionally speaking, after calling the function **refl** recursively we obtained a new derivation $P\ \underline{x}\ \underline{u}$, which had to be abstracted to the correct context. Equation (4.1) provided us with the correct insight, that a hypothetical judgment is being represented as function type in LF.

$$\ulcorner \begin{array}{c} \overline{x \overset{1}{\Longrightarrow} x}\,u \\ \mathcal{P} \\ e' \overset{1}{\Longrightarrow} e' \end{array} \urcorner = \Pi x : \text{term}\ \ulcorner \tau_1 \urcorner. \Pi u : x \overset{1}{\Longrightarrow} x. (E'\ x) \overset{1}{\Longrightarrow} (E'\ x)$$

In the formalization of the embedding Lemma 4.11, we only used one of the two parameters "$\underline{x}$: atm $T_1, \underline{y}$: term T_1" to conclude that "E' : term $T_1 \to$ term T_2". If we had not omitted $\underline{x}$,

```
fun embedding^a (eapp E^c E^a) =
      let
        val E_1 = embedding^c E^c
        val E_2 = embedding^a E^a
      in
        app E_1 E_2
      end
and embedding^c (elam (λx : atm T_1. E^c x)) =
      let
        new x : atm T_1, y : term T_1
        val  E' y = embedding^c ((λx : atm T_1. E^c x) x)
      in
        lam E'
      end
  | embedding^c (eca E^a) =
      let
        val E' = embedding^a E^a
      in
        E'
      end
```

Figure 4.8: Formal proof of the embedding Lemma 4.11 for parallel reduction

E' would have the type "atm T_1 → term T_1 → term T_2" and consequently it is impossible to apply lam to E' in order to close the proof branch. But note: By typing reasons we can infer from the signature that it is impossible that E' ever depends on $\underline{x}$. Therefore, we can strengthen the type of E' by omitting "atm T_1". On the other hand if E' contained an occurrence of $\underline{x}$, $\underline{x}$ would surely escape its scope, and destroy the adequacy of encoding for terms.

How can we mechanize the decision when to omit $\underline{x}$? The answer to this question requires a careful analysis of the signature: It follows by inspection that "term" and "atm" are defined entirely independent from each other, i.e. no object of type atm T for any arbitrary T can contain an object of type term T', and vice versa. We say that a type family a_2 depends on another type family a_1, if objects of a_1 can be subterms of objects in a_2, or — synonymously — a_1 is subordinate to a_2. This relation on type families is called dependency or subordination relation in the literature and has been introduced by Rohwedder [Roh96] and thoroughly studied by Virga [Vir99]. In order not to clutter the presentation of the meta-logic, we postpone the issue of subordination until Section 6.2.2.

4.3 Overview Of This Thesis

A meta-logical framework serves a number of important purposes: First, it allows system developers to formalize their designs and cast them into a machine interpretable language. Second, it provides a language to express properties about these designs, and third it implements the necessary technology to verify these properties.

In this work, we have committed to the logical framework LF [HHP93] as representation language. We believe that it is currently the best representation language for our work since we are mainly interested in formal systems, such as programming languages, logics, and type systems. What makes LF the framework of choice is, that it permits elegant and adequate encodings of deductive systems using higher-order representation techniques and dependent types. Judgments are represented as types and deductions as objects.

One of the main contributions of this thesis to extend LF to a meta-logical framework. We observe, that the majority of properties about programming languages and logics are proven by induction, in particular all the properties in the previous chapter. The goal of this work is the design of the meta-logic $\mathcal{M}_2^+$ that can formalize the meta-theory of deductive systems.

Finally, we develop tools for automated reasoning in this thesis. Designing, developing, implementing, enhancing, and verifying the design of formal systems is a very tedious and time intensive endeavor. In order for a meta-logical framework to be a useful tool, it must support and automate the user's task.

More concretely, in this thesis we develop a two-layer meta-logical framework. Based on LF we develop a meta-logic $\mathcal{M}_2^+$ in Chapter 5 that is expressive enough to formalize interesting properties about programming languages, logics and type system. It is an intuitionistic logic, that defines a language of formulas useful to formalize properties, and a language of proof terms, witnessing the derivability of a property. What distinguishes $\mathcal{M}_2^+$ from other logics is the ability over higher-order encodings of deductive system relying on the regular world assumption.

Unlike standard inductive theorem provers that rely on the closed world assumption, $\mathcal{M}_2^+$ allows dynamic but regular extensions of the world. Under the closed world assumption the set of constructors for a particular inductively defined datatype is statically fixed a priori. However under the regular closed world assumption it can be dynamically extended by new constructors during a proof.

The regular world assumption is sound, because from the property of LF that canonical form are inductively defined, we can infer that any recursive function that is valid in $\mathcal{M}_2^+$ is a realizer. For examples **refl**, **subst**, **dia**, **strip**, **conf**, and **cr** are all derivable in $\mathcal{M}_2^+$, and they are realizers. In order to make the soundness argument formal, we specify an operational semantics for $\mathcal{M}_2^+$ in Chapter 6, and in Chapter 7 we show that each function derivable in $\mathcal{M}_2^+$ is total.

We also present some automated deduction algorithms in Chapter 8 that have been implemented in the Twelf system. In fact Twelf contains a working meta-theorem prover (`http://www.twelf.org`) that can prove all the theorems we have shown in the previous sections and chapters. The theorem prover works mostly automatic; all that is required is the proper formulation of the induction hypothesis, a termination order, and a number which limits the search space when Twelf is constructing a witness object to close a proof subgoal.

Twelf has been used in many experiments. In logic for example, Twelf has been successfully applied to derive the cut-elimination results for full-first order intuitionistic and full first-order classical logic [Pfe95]. In logic programming, it has been used to show that the fragment of hereditary Harrop formulas implemented in λ-Prolog [NM88], proof search for uniform derivations and resolution are equivalent. It also derived the same property for the Horn fragment of predicate logic. In the area of functional programming, Twelf was used to show that the operational semantics of Mini-ML, an ML dialect without exceptions, references and modules, preserves types. In addition, it derived a completeness result for compiling Mini-ML programs into a continuation based transition machine CPM [FSDF93]. Most proofs could be found in a few seconds, for other some Twelf needed more time.

4.4 Related Work

In the last few decades it has been realized that type theory is an appropriate formalism for the representation of propositions and proofs. After the discovery of the Curry-Howard isomorphism [How80], it has become common practice to represent proposition as types, and express derivability by the existence of objects. In particular, it guarantees that propositional natural deduction derivations [Pra65] can be represented as λ-terms in the simply-typed λ-calculus.

Thereafter many type theories were developed, arguably the most influential being Martin-Löf's type theory [ML80]. Most importantly, it demonstrated how dependent types and an equality relation can be used to adequately represent judgments and derivations in a formal framework. Martin-Löf's type theory eventually led to the development of the NuPRL system [C^+86], and it is implemented in ALF [Mag95].

There has been a whole series of different systems, following this tradition, among others the Isabelle system [Pau94] based on the simply-typed λ-calculus, the Coq system [DFH^+93], which is based on the calculus of constructions [CH88], and the Lego system [LP92], which is based on a refined version of the calculus of constructions. A more detailed discussion about these these systems and logical frameworks in general can be found in [Pfe99].

All these systems are very similar in nature. One logical framework makes use of polymorphism, the other of type constructors. Many of these systems provide the facilities to reason by induction. But in all cases, the underlying assumption is that the world is closed. Consequently, higher-order encodings as we use them in this thesis are not directly expressible in any of these systems, and therefore, none of the systems can express proofs as elegantly as we have presented them in this chapter.

In order to rectify this inefficiency, many of the systems have introduced inductive datatypes to which induction principles are associated. In general, it has been accepted that the negativity condition associated with the inductively defined datatypes (as shown in Section 4.1) is unavoidable. Therefore higher-order representation techniques have hardly been used, and alternative first-order encodings have been chosen. A common way to represent variables for example is the use of de Bruijn indices or integers.

The main drawback of first-order representation techniques is that they are not very elegant. They do not exploit the type theory in order to define, represent, and execute substitutions, instead, everything that has to do with substitutions must be explicitly encoded and proven correct. One can think of higher-order representations as alive since they can change their shape due to internal $\beta\eta$-reductions, whereas first-order representations are dead, since every reduction operation must be defined outside the logical framework[1].

This way, the original calculus of construction [CH88] has been extended to the inductive calculus of construction [PM93] which is now used as the formal basis for Coq, and Isabelle, Lego, and ALF all allow inductive definition given that the positivity condition is satisfied.

On the other hand, the LF type theory does not contain a concept of inductive datatypes. As already discussed, the recursive functions space implicitly associated with with the elimination rule of inductive definitions is inherently incompatible with the parametric function space provided by LF (see Section 2.6), and the Elf project [Pfe89] has taken the stand for higher-order representation techniques and against inductive datatypes. LF is a very elegant tool to represent deductive systems, but it lacks a general theory to represent meta-theory adequately.

[1] This analogy is due to Henk Barendregt

Even though Elf does not provide a recursive function space, its operational semantics implicitly defines recursive relations. Specifically, recursive functions which lie in the Π_2-fragment can be encoded in Elf as relations [Pfe89]. Each relation relates the universally quantified assumptions (read as input arguments) to the existentially quantified assumptions (read as output arguments). The relation is representable as LF-signature, and executable via a logic programming interpretation. As example we present an encoding of Lemma 3.5 as a recursive function which maps two derivations $\mathcal{D} :: e \xrightarrow{*} e'$ and $\mathcal{E} :: e' \xrightarrow{*} e''$ to a derivation $\mathcal{P} :: e \xrightarrow{*} e''$. The function is being represented as relation

$$\text{trans}\ \ulcorner \mathcal{D}_1 \urcorner\ \ulcorner \mathcal{D}_2 \urcorner\ \ulcorner \mathcal{P} \urcorner$$

which is encoded as type family. The first two arguments must be interpreted as input arguments, and the last as output argument. We omit that E, E', and E'' are also treated as input arguments, since also the Elf type reconstruction algorithm infers this information itself.

$$\begin{array}{rcl}
\text{trans} & : & (E \xrightarrow{*} E') \to (E' \xrightarrow{*} E'') \to (E \xrightarrow{*} E'') \to \text{type} \\
\text{transrid} & : & \text{trans rid } D_2\ D_2 \\
\text{transrstep} & : & \text{trans (rstep } D_1'\ D_1'')\ D_2\ (\text{rstep } D_1'\ P) \\
 & & \leftarrow \text{trans } D_1''\ D_2\ P
\end{array}$$

Obviously, from the point of view of LF, this is not the encoding of a function, it is a sequence of constant declarations! The semantics of ordinary parametric functions, given by the β- and η-rule, is not enough to establish an operational semantics of a function represented this way. Therefore, LF-signatures have been equipped with a logic programming interpretation, which assigns an operational meaning to $\to$ and Π [Pfe89] that interprets each declaration in the signature as applicable if the head is unifiable. This way, a query of the form "trans rid rid P" can be executed, and the value being returned is the constant "rid" bound to the variable P. The reader is invited to consult [Pfe00] for a large collection of more examples.

Because of this external interpretation of a signature as a program, recursive functions can represented in LF. But do these declarations necessarily represent proofs? The answer is clearly no! To represent a proof the recursive functions must be total, i.e. their evaluation will always make progress and eventually terminate. But this property is not enforced, neither by the type system of LF nor by the definition of the operational semantics itself. As a matter of fact, it is very easy to write non-terminating functions. Adding

$$\begin{array}{rcl}
\text{infinite} & : & \text{trans } D_1\ D_2\ P \\
 & & \leftarrow \text{trans } D_1\ D_2\ P
\end{array}$$

as first object constant declaration to the LF signature, will cause the evaluation to loop. Similarly, omitting the rule rid from the signature will force the operational semantics to get stuck when executing "trans rid rid P", and the value of P cannot be determined.

In order to determine that a type family represents a proof one has to employ an external check for totality, a procedure to which we refer as *schema-checker*. Early attempts have been made to devise an efficient and reliable schema-checking algorithm by Rohwedder [Roh96]. The formal conditions for termination (see Section 7.2) and coverage (see Section 7.3) can be used to devise an appropriate schema-checking algorithm.

It is inherently difficult to extend logical frameworks *directly* with a parametric function space by a recursive function space in a way that both function spaces can coexist. We only know of one successful attempt which goes back to Schürmann, Despeyroux, and Pfenning [DPS97]: the $\Box$-calculus — a conservative extension of the simply-typed λ-calculus. This work introduces a new type $\Box A$ that reads as the type of all *closed* objects of type A. Using the modal operator and the parametric function arrow $\to$, the recursive function space $A_1 \Rightarrow A_2$ is defined in the following way.

$$A_1 \Rightarrow A_2 = \Box A_1 \to A_2$$

The $\Box$-calculus also provides iteration and case operators that provide function definition by case analysis (over any closed possibly functional object). Specifically, a recursive functions f mapping natural numbers to natural numbers has either type $\Box\mathrm{nat} \to \mathrm{nat}$ or type $\Box\mathrm{nat} \to \Box\mathrm{nat}$, depending if the result of an application of f should be used as argument to another recursive function or not.

The $\Box$-calculus is a very elegant solution to the problem of having a recursive and parametric function space coexist in one logical framework but it has two severe restrictions, which make it an unsuitable candidate for a meta-logical framework: First, it requires that arguments to recursive functions are always closed, which excludes the representation of the proof reflexivity Lemma 3.4 for parallel reduction as far as we know. Second, it is only defined for the simply-typed setting. Therefore, it is by far not general enough to be used as a meta-logical framework. The second restriction has been partially addressed in the thesis of Leleu [Lel98], where he develops an extension of the $\Box$-calculus to also include dependent types. But the first restriction remains, and it is not at all clear of how to extend it to also reflect parameter contexts and allow reasoning about *open* terms.

A more general approach has been taken by Miller and McDowell with their system $FO\lambda^{\Delta I\!N}$. $FO\lambda^{\Delta I\!N}$ is a meta-logic based on an intuitionistic first-order logic extended by natural number induction and definitional reflection [SH93b]. This meta-logic is very general, it is so general that it supports the representation of various logical frameworks, for example the intuitionistic and linear framework of hereditary Harrop formulas [McD97]. The embedded logical frameworks are used to represent deductive systems. In [MM97], McDowell discusses the formalization of the type preservation proof for Mini-ML.

$FO\lambda^{\Delta I\!N}$ is similar to $\mathcal{M}_2^+$ because it explicitly separates the meta-logic from the logical framework, but on the other hand, it is quite different: The only induction principle underlying $FO\lambda^{\Delta I\!N}$ is natural number induction. In particular, every structural inductive argument must be mapped onto natural numbers which puts additional strains on the formulation of meta-theorems. A second drawback of $FO\lambda^{\Delta I\!N}$ is the treatment of parameter contexts. The logic is not specific enough to treat parameter contexts as special entities. To the contrary, parameter contexts and hypothesis must be explicitly represented as lists or as functions as must the regularity condition.

In addition, $FO\lambda^{\Delta I\!N}$ is an intuitionistic logic, without proof terms. Contrary to our approach where we show soundness of our meta-logic by guaranteeing the proof terms are total functions, McDowell uses a purely logical argument. He shows that $FO\lambda^{\Delta I\!N}$ enjoys the cut-elimination property. Naturally, cut-elimination implies consistency. Considering how complicated the original cut-elimination proof already is [MM00], the soundness argument is the major impediment when generalizing $FO\lambda^{\Delta I\!N}$'s natural number induction principle to full structural induction.

$FO\lambda^{\Delta I\!N}$'s ability to represent other logical frameworks raises immediately two questions. First, which other logical frameworks are there, and are they interesting? And second, how well can $\mathcal{M}_2^+$ adopt to these new logical frameworks. The answer to the first question is yes, there are many important logical frameworks, and the answer to the second second question will be postponed until Section 9.1.2.

The interested reader might wonder if it is possible to develop the meta-logic in $\mathcal{M}_2^+$ using a proof assistant, such as NuPRL or Coq. The formal development of the meta-logic requires a sound formalization of LF including congruence rules, and much of its meta-theory; it will require proofs of many properties such as substitution lemmas, the canonical form theorem, and many others. In addition, one had to formalize unification and subordination, and derive their necessary properties. We predict, that the proof search engines of the proof assistants will not be efficient enough to perform the search for derivations inside the deductive systems since the LF substitution lemmas and canonical form lemmas will be explicitly and repeatedly applied. In our system, we can exploit the fact that terms are alive, they normalize to their canonical form by themselves. However, for traditional theorem provers terms are dead, which means that it is the provers responsibility to return a result in canonical form.

In summary, we believe the work carried out in this thesis cannot be developed in other proof assistant without spending a significant amount of time and energy. Even if it were possible, one cannot expect a working theorem prover for free as result of the formal development.

The theorem prover implemented in Twelf that we present in this thesis in Chapter 8 works by searching for realizers for a given formula in $\mathcal{M}_2^+$. These realizers are recursive functions, which can be executed, and they compute witness objects for existential quantifies from instantiations of universal ones. In this sense, Twelf is program synthesis tool [Kre98], that generates correct programs in a not yet well explored programming language whose datatype declarations are written as LF signatures.

4.5 Summary

In this chapter, we have demonstrated of how to formalize meta-proofs and meta-theorems in a meta-logical framework leading up to an informal description of the meta-logic $\mathcal{M}_2^+$. Conceptually, $\mathcal{M}_2^+$ lies on a different and separate level above the logical framework LF. In particular, it encompasses universal and existential quantification, and conjunction. This is sufficient because the meta logic does not provide any other atomic constants or propositions other than truth. The meta-logic provides a proof term calculus, where each proof term corresponds to a total recursive function. Totality is required in order to guarantee soundness, i.e. upon instantiation of its arguments, the function must terminate and return with an answer.

Chapter 5

The Meta-logic $\mathcal{M}_2^+$

5.1 Introduction

The design cycle of programming languages, compilers, and logics is long, tedious, and error-prone. In particular, when extending a programming language by new constructs, one has to be very careful not to render the entire system design unsound. Even worse, an unsoundness occurring in a programming language is sometimes very difficult to detect by testing, sometimes it takes years, and very often it is extremely difficult to rectify since it involves a change in the language design.

The earlier mistakes in the development of a programming language are caught, the better the final result is. During the early design stages, adjustments to a language need not to be local, they might and often will be global. In general, it is impossible to remove all flaws from a programming language already at the drawing board, but experience has shown, that many flaws could be avoided by checking the design against certain a-priori defined specifications, such as type soundness, progress, and others.

Consider for example the untyped λ-calculus, a very simple functional programming language, from Chapter 2. From [CR36] we learned that the diamond lemma and the Church-Rosser theorem holds for this language. What about extending it to the simply-typed case? All we had to do is to edit the sequence of theorems, by indexing all occurrences of "term" by a type. Next, we refined it to the polymorphic λ-calculus, and again we had to slightly generalize the formulation of the lemmas, this time by extending the context schemas (for example Lemma 4.10). This example shows of how we envision users working with our tool. It serves the incremental development of programming languages and their theory while offering sophisticated verification procedures.

In this chapter however, we begin with a formal presentation of the meta-logic which is at the very heart of this thesis. Its purpose is to express specifications about deductive systems. We develop an appropriate proof system based on the sequent calculus, for which we develop an automated proof search procedure in Chapter 8. The meta-logic is called $\mathcal{M}_2^+$ and it supersedes an earlier versions that were published for example in [Sch95, SP98]. Unlike $\mathcal{M}_2$ that relies on the closed world assumption, $\mathcal{M}_2^+$ relies on the regular world assumption.

This chapter is organized in the following way. In Section 5.2 we introduce a notion of substitution for LF (see Section 2.4) since we will use substitutions from early on, and they will occur in different shapes over and over in this chapter. Using the notion of substitution we start with the presentation of the logic, its syntax and semantics in Section 5.3, followed by a formal inference system, based on extensions to the sequent calculus in Section 5.4. In Section 5.5 we endow the inference rule system with proof terms, constructed in such a way that they can be used to represent (non-inductive) meta-proofs. In Section 5.6 we extend the proof term calculus by constructs for recursion, which allow the formalization of meta-proofs carried out via induction and we add lemmas in Section 5.7. In Section 5.8 we conclude this chapter, and assess the results.

5.2 Preliminaries

Variables and substitutions are two closely related concepts. In fact, in Chapter 2 we have used substitutions, for example, for the definition of the β-rule for the untyped, the simply-typed, and even the dependently typed λ-calculus. Be it in a formal development, or in a theorem prover implementation, or even in the design of a programming language or logic, the treatment of variables is very difficult to get right. Since the use of higher-order abstract syntax makes heavy use of the variable concept of the logical framework, variables and substitutions are the backbone of this development and hence deserve extremely careful attention. Specifically, LF substitutions are defined as a list of object/variable pairs M/x where x is the variable to be instantiated, and M an object which is well-defined in some context Γ.

$$\textit{Substitutions:} \quad \sigma \quad ::= \quad \cdot \mid \sigma, M/x$$

In this work we follow standard practice, and allow only valid substitutions to be applied to valid terms. Because contexts contain explicit type information, validity can be easily expressed as a static property of substitutions.

Judgment

$$\textit{Valid substitutions:} \quad \Gamma_2 \vdash \sigma : \Gamma_1$$

We say, that "a substitution σ goes from Γ_1 to Γ_2", which means that — when applied — it substitutes objects valid in Γ_2 for variables declared in Γ_1. We refer to Γ_1 as the *domain* of the substitution, and to Γ_2 as the *co-domain*.

Rules

$$\frac{}{\Gamma \vdash \cdot : \cdot}\,\mathsf{subempty} \qquad \frac{\Gamma_2 \vdash M : A[\sigma] \qquad \Gamma_2 \vdash \sigma : \Gamma_1}{\Gamma_2 \vdash \sigma, M/x : \Gamma_1, x : A}\,\mathsf{subcons}$$

Note that M has type $A[\sigma]$ in the first premiss of rule $\mathsf{subcons}$, where $A[\sigma]$ is the type one obtains from A by applying the substitution σ. Without going into detail of how substitution application is defined for LF, we always assume that substitutions can be applied to LF types, LF objects, or LF kinds if they are valid in the domain of the substitution. σ can be applied to A, because $\Gamma_2 \vdash \sigma : \Gamma_1$ and $\Gamma_1 \vdash A : \mathrm{type}$.

A similar comment holds for the composition of substitutions. A substitution σ_1 can only be composed with σ_2 if σ_1's co-domain and σ_2's domain coincide. Formally this is expressed by $\Gamma_3 \vdash \sigma_2 : \Gamma_2$ and $\Gamma_2 \vdash \sigma_1 : \Gamma_1$. Substitutions composition is written as $\Gamma_3 \vdash \sigma_1 \circ \sigma_2 : \Gamma_1$.

Definition 5.1 (Composition of substitutions)

$$\begin{array}{rcl} \cdot \circ \sigma_2 & = & \sigma_2 \\ (\sigma_1, M/x) \circ \sigma_2 & = & (\sigma_1 \circ \sigma_2), M[\sigma_2]/x \end{array}$$

It is an easy consequence from the substitution lemmas for LF [HHP93], that the composition of two valid substitutions is valid.

Lemma 5.2 (Composition of substitutions)

If $\mathcal{D}_1 :: \Gamma_2 \vdash \sigma_1 : \Gamma_1$
and $\mathcal{D}_2 :: \Gamma_3 \vdash \sigma_2 : \Gamma_2$
then $\Gamma_3 \vdash \sigma_1 \circ \sigma_2 : \Gamma_1$

Proof: by structural induction on $\mathcal{D}_1$. □

This concludes the section on preliminary concepts and we continue with the presentation of the logic $\mathcal{M}_2^+$ where substitutions are needed at many different occasions.

5.3 The Logic

We begin with the discussion of the logic $\mathcal{M}_2^+$, its syntax, and its semantics. The syntax of formulas is more complicated than in other logics, because formulas also describe partial extensions of the current world. At the end of this section we define a formal semantics for this logic.

5.3.1 Syntax

We introduce the syntax of $\mathcal{M}_2^+$ in three steps. First formally define what context schemas are. Second we motivate two different variable concepts. One kind of variables range over assumptions, i.e. LF types, and the other kind of variables ranges over parameter blocks. Third, we characterize formulas.

Context schemas

In the formulation of each theorem, we explicitly require that there is a context schema given, which describes the regular extensions of the world. In Section 4.2.3, we have encountered an example, where valid extensions to the world can only be described by more than one block schema. Context schemas are defined by a labeled list of block schemas, and each block schema has two components, a SOME-component, and a BLOCK-component, where the BLOCK-component defines the form of a parameter block, and the SOME-component quantifies over free variable occurrences in this block. Block schemas are always labeled. Context schemas are an integral part of any formula.

$$\begin{array}{llcl} \textit{Context form:} & C & ::= & \cdot \mid C, x : A \\ \textit{Block schema:} & B & ::= & \text{SOME } C_1.\ \text{BLOCK } C_2 \\ \textit{Context schemas:} & S & ::= & \cdot \mid S, B^L \end{array}$$

Context forms are LF contexts, they enjoy all substitution and α-conversion properties as regular LF contexts do. We have given them a different name and denote them with a different letter C in order to emphasize that they are blueprints for context blocks.

Variable concepts

In traditional intuitionistic or classical logic, when we write $\forall x.\exists y.P(x,y)$, we typically do not specify the domain of the two variables x and y. If this formula is true, then independently of what x is bound to, it is certain that there exists y which makes $P(x,y)$ true.

For our purposes on the other hand, we think of x and y as LF-objects, representing derivation of an encoded deductive system. Therefore, x and y range over objects of a certain type; we have demonstrated this already in Chapter 3, when we developed the different formalizations of theorems in the meta-logic. Consider for example the formalization of the reflexivity Lemma 4.3 for parallel reduction. The colon ":" indicates that T, E, and D range over LF-objects.

$$\Box(\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \overset{1}{\Longrightarrow} x)^L.$$
$$\forall T : \text{tp}. \forall E : \text{term } T. \exists D : E \overset{1}{\Longrightarrow} E. \top$$

Note, that in this formalization T and E are not standard LF variables declarations as described in Section 2.4. There is a new property attached to these variables which is not available in LF at all: In order to reason by induction, we can analyze the different forms of T and E.

Nevertheless, since we always keep the LF level and the $\mathcal{M}_2^+$ level entirely separate, we continue to write $x : A$ for the assumption that x is of type A, and we keep in mind that we can analyze x on a case by case basis. From a logical point of view, we call $x : A$ an *assumption*.

The regular world assumption introduces a new level of complexity. Recall that the regular world assumption allows dynamic but regular extensions of the LF signature. A recursive function, as we have seen in the previous chapter can extend the current world by new constructors. Extensions of the world must always match the abstract description of the world through context schemas.

In addition to the standard variable concept, we need a notion of variables that range over parameter blocks. We motivate these new variables using the reflexivity Lemma 4.3. After analyzing the cases on E, we had to consider the one case that E in fact refers to a parameter $\underline{x}$ in an extension of the current world Φ. $\underline{x}$ is a variable, it simply ranges over any parameter. Recall that the context schema states that

$$(\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \overset{1}{\Longrightarrow} x)^L$$

which means, that any $\underline{x}$ in Φ is always accompanied by $\underline{u}$. Thus, $\underline{u}$ ranges over the second parameter in a parameter block. Since we need to reason abstractly about parameter blocks, we refer to $\underline{x}$ and $\underline{u}$ collectively as *variable block*. In full generality, variable blocks consist of many *parameter variables*. We write $\rho = (\underline{x} : \text{term } T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})$ for variable blocks.

$$\textit{Variable blocks:} \quad \rho \quad ::= \quad \cdot \mid \rho, \underline{x} : A$$

Variable blocks are typically labeled as described in Section 4.2.3, and these labels are written in exponent notation. Consequently, variable blocks ρ ranging over parameter blocks labeled with L are written as ρ^L. In this setting, regular world extensions Φ can be defined as a list of labeled variable blocks. As example consider the following extension of the world that is clearly an instance of the context schema:

$$\Phi = (\underline{x_1} : \text{term } \ulcorner\tau\urcorner, \underline{u_1} : \underline{x_1} \overset{1}{\Longrightarrow} \underline{x_1})^L, \ldots, (\underline{x_n} : \text{term } \ulcorner\tau\urcorner, \underline{u_n} : \underline{x_n} \overset{1}{\Longrightarrow} \underline{x_n})^L$$

Variable blocks and regular world extensions enjoy the standard properties, such as substitution, weakening, contraction, and limited exchange [HHP93]. α-convertibility of variable blocks $\rho_1 \equiv_\alpha \rho_2$ is decidable and follows from a simple generalization of convertibility of types. Formally, variable blocks are simply lists of parameter binding variables together with their types.

Formulas

Given the two different variable concepts, the formula level must provide two quantifiers, binding each of the variables. In $\mathcal{M}_2^+$ we use the standard universal quantifier to quantify over LF objects as used in the formula for the reflexivity lemma.

$$\forall E : \text{term}\, T.\, \exists D : E \xrightarrow{1} E.\, \top$$

The other quantifier ranges over variable blocks ρ. We motivate this new quantifier by further examples. Consider again the standard extension of the current world as often assumed in the previous chapter.

$$\Phi = (\underline{x_1} : \text{term}\, \ulcorner \tau_1 \urcorner, \underline{u_1} : \underline{x_1} \xrightarrow{1} \underline{x_1})^L, \ldots, (\underline{x_n} : \text{term}\, \ulcorner \tau_n \urcorner, \underline{u_n} : \underline{x_n} \xrightarrow{1} \underline{x_n})^L$$

In the reflexivity Lemma 4.3 for parallel reduction we must analyze cases over $E : \text{term}\, T$. That means we have to consider a variable block ρ ranging over any parameter block of label L. In this case, we must show that forall types T, and for variable blocks ranging over parameter blocks in Φ there exists a D of appropriate type. We ues quantification over variable blocks of label L using the $\mathbf{\Pi}$-quantifier to express this formula.

$$\forall T : \text{tp}.\, \mathbf{\Pi}(\underline{x} : \text{term}\, T, \underline{u} : \underline{x} \xrightarrow{1} \underline{x})^L.\, \exists D : \underline{x} \xrightarrow{1} \underline{x}.\, \top$$

A formula in $\mathcal{M}_2^+$ is built from two parts. The first part describes the form of possible extensions of the world. It is expressed by the context schema. Informally, the reflexivity Lemma 3.4 states:

> Consider the situation where a list of the following assumptions is present
>
> $$x_1 :: \text{term}\, \tau_1, u_1 :: x_1 \xrightarrow{1} x_1, \ldots, x_n :: \text{term}\, \tau_n, u_n :: x_n \xrightarrow{1} x_n$$
>
> Then for any well-typed term e, there exists a derivation of $e \xrightarrow{1} e$.

The context schema formalizes the statement about the list of assumptions whereas a formula expresses the property to be shown.

Formulas are defined in terms of universal and existential quantifiers ranging assumptions and variable blocks, conjunction to represent mutual inductive theorems, and truth. For this work, we are particularly interested in formulas that lie in the Π_2-fragment, since it is this fragment for which we develop automated deduction algorithms that are described in Chapter 8. The well-formedness condition for formulas is discussed in Section 5.4.3.

General formulas:	G	$::=$	$\Box S.\, F$
Formulas:	F	$::=$	$\forall x : A.\, F \mid \mathbf{\Pi}\rho^L.\, F \mid \exists x : A.\, F \mid F_1 \wedge F_2 \mid \top$

It is possible to separate the two parts of a general formula, and to leave the definition of the context schema implicit, similarly as we leave the definition of the signature implicit. However for clarity we carry the context schema as part of the formula in this work.

As a reminder, here are some examples of meta-theorems expressible in this logic. The first example is the diamond lemma for parallel reduction.

Example 5.3 (Diamond lemma) (see Lemma 4.6)

$$\begin{array}{l} \Box(\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \overset{1}{\Longrightarrow} x.)^{L} \\ \qquad \forall T : \text{tp}. \forall E : \text{term } T. \forall E^{l} : \text{term } T. \forall E^{r} : \text{term } T. \\ \qquad\quad \forall D^{l} : E \overset{1}{\Longrightarrow} E^{l}. \forall D^{r} : E \overset{1}{\Longrightarrow} E^{r}. \\ \qquad\quad \exists E' : \text{term } T. \exists R^{l} : E^{l} \overset{1}{\Longrightarrow} E'. \exists R^{r} : E^{r} \overset{1}{\Longrightarrow} E'. \top \end{array}$$

The second example is the reflexivity lemma for the polymorphic λ-calculus. Note, that here the context schema contains two block schemas.

Example 5.4 (Reflexivity lemma for the polymorphic λ-calculus) (see Lemma 4.10)

$$\begin{array}{l} \Box(\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \overset{1}{\Longrightarrow} x)^{L_1} \mid (\text{BLOCK } a : \text{tp})^{L_2}. \\ \qquad \forall T : \text{tp}. \forall E : \text{term } T. \exists D : E \overset{1}{\Longrightarrow} E. \top \end{array}$$

The logic is very simple, and simultaneously very strong because it inherits the expressiveness from the underlying logical framework LF. In particular, there are no other constants defined besides truth. There is no equality. There is no falsehood. There is no disjunction. On the one hand this sounds like a severe restriction, on the other it may not be. For specific instances, it is possible to define disjunction in LF and to make it accessible to $\mathcal{M}_2^+$. A more concise investigation of other useful connectives for $\mathcal{M}_2^+$ is left to future work.

5.3.2 Semantics

In this subsection we extrapolate a suitable semantics for $\mathcal{M}_2^+$ from the examples presented in Chapter 3. The semantic is straightforward and intuitive. Before we present the meaning of a general formula G in $\mathcal{M}_2^+$ in detail, we first define an interpretation of the new $\Box$-operator, which prompts the definition of an interpretation of context schemas.

A closer look on context forms C reveals that C's are defined structurally in a way very similar to LF contexts, namely as a list of declarations. In order to judge if a given context satisfies a context schema, we must check that every block is an instantiation of a block schema — block by block.

Consider for example a regular extension of the world, that we denoted by $\Phi = \Phi', \rho^L$ where ρ is the most recent block introduced in the world. This block is labeled with L. Obviously, for Φ to be be valid, Φ' must be valid and ρ must be an instance of the block schema SOME C_1. BLOCK C_2. In other words, there must be an instantiation for the variables in C_1 from Φ', and ρ must match C_2' where C_2' is the result of instantiating all variables from C_1 in C_2. In the first case we speak of a SOME-instantiation, and in the second of a BLOCK-construction which includes an explicit α-conversion step to ensure that the naming of parameters is unique.

BLOCK-construction creates the new parameter context by traversing C from left to right instead of right to left as suggested by the syntactical definition of C. We write $[\sigma]C$ for the instantiation of context form C followed by an α-conversion step. σ is a substitution.

The interpretation of a block schema is defined in terms of SOME-instantiations and BLOCK-constructions. It is a set of all parameter blocks, which are the result of BLOCK-construction, after some appropriate SOME-instantiation.

Definition 5.5 (Interpretation of a block schema)
For all σ, s.t. $\Phi \vdash \sigma : C_1$, it holds that $\Phi \vdash [\sigma]C_2 \in [\![\text{SOME } C_1.\ \text{BLOCK } C_2]\!]$

We say that Φ lies in the interpretation of S, if any parameter block of Φ is in the interpretation of some block schema defined by S. Obviously, the empty parameter context is an element of the interpretation of any context schema.

Definition 5.6 (Interpretation of context schemas)

$$[\![S]\!] := \{\cdot\} \cup \{\Phi, \rho^L \mid \Phi \in [\![S]\!] \text{ and there exists a } B^L \in S,\ s.t.\ \Phi \vdash \rho \in [\![B]\!]\}$$

On the basis of the interpretation of context schemas we can now define the semantics of formulas. A general formula is semantically valid, if its body is valid in any parameter context compatible with the context schema. Universally quantified formulas are valid, if for all instantiations of the assumption variable, the body of the formula is valid. Similarly for variable block quantification: A $\mathbf{\Pi}$-formula is semantically valid if and only if its body is semantically valid after instantiating the variable block with a parameter block from the context (carrying the same label). An existentially quantified variable is semantically valid if there exists a term M which makes the body of the formula valid. The conjunctions of two formulas is valid if each of the conjuncts is, and last but not least, $\top$ is always semantically valid.

Definition 5.7 (Meaning of formulas)

$$\begin{array}{rlll}
\models \Box S.\, F & \textit{iff} & \Phi \models F & \text{for all } \Phi \in [\![S]\!] \\
\Phi \models \forall x : A.\, F & \textit{iff} & \Phi \models F[M/x] & \text{for all } M,\ s.t.\ \Phi \vdash M : A \\
\Phi \models \mathbf{\Pi}\rho^L.\, F & \textit{iff} & \Phi \models F[\rho'/\rho] & \text{for all } \rho'^L \in \Phi,\ s.t.\ \Phi \vdash \rho' \equiv_\alpha \rho \\
\Phi \models \exists x : A.\, F & \textit{iff} & \Phi \models F[M/x] & \text{for some } M,\ s.t.\ \Phi \vdash M : A \\
\Phi \models F_1 \wedge F_2 & \textit{iff} & \Phi \models F_1 \text{ and } \Phi \models F_2 & \\
\Phi \models \top & & &
\end{array}$$

The goal of this thesis is to develop an automated meta-theorem prover which can prove meta-theorems about deductive systems. Unfortunately, the semantics of the meta-logic does not provide enough structure for a construction of a theorem-prover for $\mathcal{M}_2^+$. Thus we develop a formal proof theory for $\mathcal{M}_2^+$ in the remainder of this chapter. The proof system is based on an extension of intuitionistic logic [Gal93], as the definition of the semantics of $\mathcal{M}_2^+$ already suggests. Given any instantiation of the universal quantifiers the proof determines witness objects for the existential quantifiers. That the derivability in this proof theory implies semantic validity is shown in Chapter 7. The attentive reader might recall that meta-proofs are formalized by recursive functions for which we have already given many examples in Chapter 4.

The development of the formal proof system is rather complex and quite challenging. In order to facilitate the presentation, we begin with the presentation of a set of inference rules, which extends the standard formalization of the sequent calculus for intuitionistic logic by variable blocks and the appropriate quantifier in Section 5.4. We then endow the calculus with proof terms in Section 5.5, add two operators to permit definition by cases and recursion in Section 5.6. Finally we add another operator to express lemma application in Section 5.7.

5.4 The Proof System

The proof system for $\mathcal{M}_2^+$ not only contains a set of inference rules in order to define provability, but it also contains a set of rules which characterize well-formed context schemes and well-formed formulas. All three inference rule systems rely on a proper treatment of assumptions. Recall that there are assumption variables, which correspond to LF objects, and there are variable blocks which range over entire parameter blocks. In this section we first discuss these assumption contexts in detail, in Section 5.4.1 we present then the inference system for context schemas in Section 5.4.2, the inference system for well-formed formulas in Section 5.4.3, and finally the inference system which defines provability in the sequent calculus in Section 5.4.4.

5.4.1 Generalized Contexts

Generalized contexts for $\mathcal{M}_2^+$ are inherently different from the standard LF-contexts Γ from Section 2.4, and they extend the notion of context used in previous version $\mathcal{M}_2$ [SP98]. There, contexts were defined as a list of assumptions (or Eigen variables). It was guaranteed that under the closed world assumption, all variables declared in such a context stood for closed LF expressions. Generalized context as defined in this section are much more general because of the regular world assumption. Recall that we reason about derivations that are "open" in regular extensions of the world, and therefore, assumptions declared in a generalized context may be open. Generalized contexts also describe the partial knowledge about the world at any point in a proof.

One question comes immediately to mind: Why represent all information in one generalized context? Wouldn't it be better to represent it in two different ones? One context represents assumptions, the other the current extension of the world? The answer is subtle: All information about assumptions and the world cannot be separated because of dependencies. Assumptions might occur in the types of the parameters, as we can see in the example above where $\underline{x} : \text{term } T$. And vice versa, parameters can occur in the types of assumptions. $E : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$ is such an example. Another example can be found in the proof of the diamond Lemma 4.6: after the first case analysis the left reduction is represented by $E^r : \underline{x} \stackrel{1}{\Longrightarrow} e^r$ uncovering a dependency.

Since variable blocks describe properties about the parameter contexts and assumptions live on an entirely different level, they are conceptually different, one could argue not to worry about dependencies at all. This argument is wrong, and we make one more observation that should clarify this question; our notion of generalized context cannot represent invalid parameter contexts.

Example 5.8 (Invalid parameter context) Consider the simply-typed λ-calculus from above and the following context schema:

$$(\text{SOME } T : \text{tp}, E : \text{term } T.\ \text{BLOCK } x : \text{term } T, u : E \stackrel{1}{\Longrightarrow} x)^L$$

The list

$$(x : \text{term } \ulcorner\tau\urcorner, u : y \stackrel{1}{\Longrightarrow} x)^L, (y : \text{term } \ulcorner\tau\urcorner, v : x \stackrel{1}{\Longrightarrow} y)^L$$

is not a parameter context, because the two blocks cannot be ordered in any way to respect dependencies.

By definition a context schema represents all *valid* parameter contexts, whose parameter blocks match the BLOCK declaration. Disregarding all dependencies would mean to permit *invalid* contexts, and reasoning about invalid contexts can lead to inconsistent results. Therefore we require that all dependencies of assumption and variable blocks declared in an generalized context are honored, and so invalid parameter contexts are excluded from our considerations. Committing to an order of variable blocks and assumption variables does not mean that we have committed to a particular order of declarations in the parameter context: none of the rules we introduce below will ever take advantage of this information. Consequently, representing assumption and variable blocks together in one generalized context only means that there exists at least one valid parameter context described by the generalized context. We start now with the formal presentation of generalized contexts.

$$\textit{Generalized Context:} \quad \Psi \quad ::= \quad \cdot \mid \Psi, x : A \mid \Psi, \rho^L$$

Since Ψ — when flattened out — is always a valid LF-context, we can use it also as context in LF judgments. In the rules below, we use the notation that $\Psi \vdash M : A$, which means, that after removing all labels from Ψ, the object M has type A in this newly obtained context. Moreover, our definition of regular worlds Φ is already contained in Ψ if Ψ contains nothing else but variable block declarations.

Generalized contexts are valid if assumptions and block variables are well-typed in the standard sense. A variable block ρ^L in the generalized context Ψ, ρ^L is valid, if it is an instance of a block schema SOME C_1. BLOCK C_2 as defined in Section 5.3.2.

Judgment

$$\textit{Validity of generalized contexts:} \quad \vdash \Psi \text{ abstract}$$

Note that we omit two important indices from the judgment: $\vdash \Psi$ abstract is actually indexed by the signature Σ and the context schema S. In pedantic detail, one would write

$$\vdash_{\Sigma;S} \Psi \text{ abstract}$$

for the validity judgment. But in order not to clutter the presentation and more than necessary, we omit these two indices. Because of the semantic validity of formulas and general formulas, it should be quite clear, that Σ, and S must be assumed constant throughout a proof. A similar remark holds for all other judgments which we introduce below in this section. Occasionally, we will remind the reader.

There are three rules which define the generalized contexts. First, the empty context is a generalized context, second generalized contexts can be extended by valid variable blocks or valid assumptions. Note that in this rule we use the LF typing judgment where we implicitly flatten out Ψ.

Rules

$$\frac{}{\vdash \cdot \text{ abstract}}\ \mathsf{vempty}$$

$$\frac{\vdash \Psi \text{ abstract} \qquad (\text{SOME } C_1.\ \text{BLOCK } C_2)^L \in S \qquad \Psi \vdash \sigma : C_1}{\vdash \Psi, ([\sigma]C_2)^L \text{ abstract}}\ \mathsf{vblock}$$

$$\frac{\vdash \Psi \text{ abstract} \qquad \Psi \vdash A : \text{type}}{\vdash \Psi, x : A \text{ abstract}}\,\textsf{vass}$$

5.4.2 Context Schemas

Context schemas are abstract descriptions of parameter contexts. A well-formed context schema consists of several labeled block schemas, each block schema is closed by itself, i.e. it cannot rely on any other assumptions but the ones introduced by the block. That means, if SOME C_1. BLOCK C_2 is a block schema, C_1, C_2 must form a context.

In this subsection we specify a set of inference rules for well-formed context schemas. By inspection of the definition of context schemas, it becomes immediately evident that this well-formed judgment is defined in terms of two auxiliary judgment: one judgment for well-formed block schemas and one for well-formedness of context forms.

Judgments

Well-formed context schemas:	$\vdash S$ Schema
Well-formed block schemas:	$\vdash B$ Block
Well-formed context forms:	$C_1 \vdash C_2$ Ctx

The rules defining these three judgments are entirely straightforward. The only thing to pay attention to is that for context blocks, we first have to check that the SOME-component is well-formed, and then that the BLOCK-Component is also well-formed. We tacitly assume that all labels are distinct.

Rules

$$\frac{}{\vdash \cdot \text{ Schema}} \qquad \frac{\vdash S \text{ Schema} \qquad \vdash B \text{ Block}}{\vdash S, B^L \text{ Schema}}$$

$$\frac{\cdot \vdash C_1 \text{ Ctx} \qquad C_1 \vdash C_2 \text{ Ctx}}{\vdash \text{SOME } C_1.\ \text{BLOCK } C_2 \text{ Block}}$$

$$\frac{}{C_1 \vdash \cdot \text{ Ctx}} \qquad \frac{C_1 \vdash C_2 \text{ Ctx} \qquad C_1, C_2 \vdash A \text{ type}}{C_1 \vdash C_2, x : A \text{ Ctx}}$$

Example 5.9 (Well-formed context schema) The context schema from Lemma 4.10 is well-formed:

$$\vdash (\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \overset{1}{\Longrightarrow} x)^{L_1}, (\text{BLOCK } a : \text{tp})^{L_2} \text{ Schema}$$

5.4.3 Formulas

In $\mathcal{M}_2^+$, there are two notions of formulas. First there are "formulas" that express properties, and second there are "general formulas". General formulas bind one context schema, which states the form of the extensions of the regular world. In order to judge if a general formula is well-formed, the inference rules have to ensure that the context schema is well-formed, and that every quantifier is well-typed.

Judgments:

$$\begin{array}{ll} \textit{Well-formed general formulas:} & \vdash G \text{ general} \\ \textit{Well-formed formulas:} & \Psi \vdash F \text{ formula} \end{array}$$

The judgment for generalized formulas is indexed by an LF signature, whereas the judgment for regular formulas is also indexed by a context schema. In addition, assumptions and variable blocks bound by universal quantifiers may occur anywhere in the body of the formula. Hence, the judgment for well-formed formulas is defined with respect to an generalized context.

Rules

$$\frac{\vdash_\Sigma S \text{ SCHEMA} \qquad \cdot \vdash_{\Sigma;S} F \text{ formula}}{\vdash_\Sigma \Box S.\, F \text{ general}}\,\mathsf{Vctx}$$

$$\frac{\Psi, x : A \vdash_{\Sigma;S} F \text{ formula}}{\Psi \vdash_{\Sigma;S} \forall x : A.\, F \text{ formula}}\,\mathsf{V}\forall \qquad \frac{\Psi, \rho^L \vdash_{\Sigma;S} F \text{ formula}}{\Psi \vdash_{\Sigma;S} \mathbf{\Pi}\rho^L.\, F \text{ formula}}\,\mathsf{V}\Pi \qquad \frac{\Psi, x : A \vdash_{\Sigma;S} F \text{ formula}}{\Psi \vdash_{\Sigma;S} \exists x : A.\, F \text{ formula}}\,\mathsf{V}\exists$$

$$\frac{\Psi \vdash_{\Sigma;S} F_1 \text{ formula} \qquad \Psi \vdash_{\Sigma;S} F_1 \text{ formula}}{\Psi \vdash_{\Sigma;S} F_1 \wedge F_2 \text{ formula}}\,\mathsf{V}\wedge \qquad \frac{\vdash_{\Sigma;S} \Psi \text{ abstract}}{\Psi \vdash_{\Sigma;S} \top \text{ formula}}\,\mathsf{Vtrue}$$

In the remainder of this thesis, we drop the subscript Σ and $\Sigma; S$ from these judgments and rules.

Example 5.10 (Well-formed formula) The formulation of the diamond lemma Lemma 4.6 is well-formed.

$$\begin{array}{l} \vdash \Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x. \\ \qquad \forall T : \text{tp.} \forall E : \text{term } T.\, \forall E^l : \text{term } T.\, \forall E^r : \text{term } T. \\ \qquad\quad \forall D^l : E \stackrel{1}{\Longrightarrow} E^l.\, \forall D^r : E \stackrel{1}{\Longrightarrow} E^r. \\ \qquad\quad \exists E' : \text{term } T.\, \exists R^l : E^l \stackrel{1}{\Longrightarrow} E'.\, \exists R^r : E^r \stackrel{1}{\Longrightarrow} E'.\, \top \text{ general} \end{array}$$

This concludes our presentation of well-formed formulas. We continue with the presentation of the proof system for $\mathcal{M}_2^+$.

5.4.4 $\mathcal{M}_2^+$-Calculus

The design of the proof calculus for $\mathcal{M}_2^+$ is inspired by a sequent calculus for first-order intuitionistic logic, but it is at the same time significantly different. It is similar in a sense, that there are left rules and right rules, and it is different in the sense that there is no cut-rule. In addition to the standard rules for formulas, there are also rules for general formulas. A particular drastic change to the original set of left rules is posed by the introduction of new parameters. Intuitively, introducing new parameter blocks corresponds in informal reasoning to a hypothetical argument. If assumptions are introduced in a proof, all reasoning steps are hypothetical until the newly assumed hypothesis are discharged. This observation will have a drastic impact on the form of the left rules. We will present the inference system in small steps, in this section, for example, we only present the basic notion of provability which we endow with proof terms in Section 5.5, recursion in Section 5.6, and lemmas in Section 5.7. The reader is asked to read this section very carefully and very attentively in order to capture the essential differences of $\mathcal{M}_2^+$ and the standard sequent calculus formulation.

Provability in $\mathcal{M}_2^+$ is expressed by two judgments $\vdash G$ and $\Psi; \Delta \vdash F$. The first judgment is indexed by Σ, the second by $\Sigma; S$. Ψ is the generalized context, describing all LF level assumptions and variable blocks in any given state in the proof. Δ stands for a list of formulas, representing all meta-assumptions during a proof. Informally, the Δ is the left hand side of the sequent symbol $\vdash$. Formally, it is defined as

$$\textit{Meta-assumptions:} \quad \Delta \quad ::= \quad \cdot \mid \Delta, F$$

and meta-assumptions are well-formed, if they satisfy the following judgment

Judgment

$$\textit{Well-formed meta-assumptions:} \quad \Psi \vdash \Delta \text{ meta}$$

which is defined by the following two rules.

Rules

$$\frac{\vdash \Psi \text{ abstract}}{\Psi \vdash \cdot \text{ meta}}\,\mathsf{vabstract} \qquad \frac{\Psi \vdash \Delta \text{ meta} \qquad \Psi \vdash F \text{ formula}}{\Psi \vdash \Delta, F \text{ meta}}\,\mathsf{vmeta}$$

Typical examples of formulas, which are represented by Δ, are for example the induction hypothesis, and subformulas of the induction hypothesis resulting of partial applications. In Section 5.5 we will revisit the list of meta-assumptions and assign names to them which are simply meta-variable names for the proof term calculus. But for the presentation of pure proof rules, it is enough to assume Δ to be a list of formulas.

Judgments

$$\begin{array}{ll} \textit{Provability of general formulas:} & \vdash_\Sigma G \\ \textit{Provability of formulas:} & \Psi; \Delta \vdash_{\Sigma;S} F \end{array}$$

The two provability judgments are not general enough to present the entire system of inference rules. As a matter of fact, they are only general enough to presen approximately half of the system, namely the right rules. The special case of the left rules is discussed below. For the sake of clarity, we omit index of the $\vdash$ symbol in the judgments for the definition of the rules below. It can be easily derived from the context.

$$\frac{\cdot; \cdot \vdash F}{\vdash \Box S.\, F}\,\mathsf{generalR}$$

This is only right rule for general formulas. The other right rules for the provability of formulas are almost straightforward. Ψ, interpreted as LF-context in the $\mathsf{R}\exists$ rule, provides all assumptions about LF objects known at the point of time when the rule is applied in a proof, and M is the witness object for the existential. Note that Δ does not change in any of these rules.

$$\frac{F \in \Delta}{\Psi; \Delta \vdash F}\,\mathsf{axvar}$$

$$\frac{\Psi, x : A; \Delta \vdash F}{\Psi; \Delta \vdash \forall x : A.\, F}\,\mathsf{R}\forall \qquad \frac{\Psi, \rho^L; \Delta \vdash F}{\Psi; \Delta \vdash \mathbf{\Pi}\rho^L.\, F}\,\mathsf{R}\Pi \qquad \frac{\Psi \vdash M : A \qquad \Psi; \Delta \vdash F[M/x]}{\Psi; \Delta \vdash \exists x : A.\, F}\,\mathsf{R}\exists$$

$$\frac{\Psi; \Delta \vdash F_1 \qquad \Psi; \Delta \vdash F_2}{\Psi; \Delta \vdash F_1 \wedge F_2}\,\mathsf{R}\wedge \qquad \frac{}{\Psi; \Delta \vdash \top}\,\mathsf{R}\top$$

The rule $\mathsf{R}\forall$ provides information about the existence of LF objects, and this information is stored in the generalized context. Similarly does the rule $\mathsf{R}\Pi$ provide information about the form of the parameter context.

Now to the left rules. Differently from the right rules, where the defining formula occurs in the conclusion to the right of the $\vdash$ symbol, the defining formula for the left rules occurs to the left, in the assumption list. And typically, there are as many rules as there are connectives. Applying a left rule in a backwards directed fashion means to extend Δ by new assumptions, resulting from manipulating this one formula. For example, if $\forall x : A.\, F$ is this formula in Δ, we can use it and for well-typed object M of type A, we can assert the new assumption $F[M/x]$. In a first attempt, let us define the left rule for $\forall$ to be:

$$\frac{\Psi \vdash M : A \qquad \Psi; \Delta_1, \forall x : A.\, F, F[M/x], \Delta_2 \vdash F}{\Psi; \Delta_1, \forall x : A.\, F, \Delta_2 \vdash F}\,\mathsf{L}\forall$$

How would we use this rule in a proof? Consider for example the proof of the reflexivity Lemma 4.3, and in this proof the case for app. Furthermore, assume that the induction hypothesis is already contained in Δ:

$$\forall T : \mathrm{tp}.\, \forall E : \mathrm{term}\ T.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top \in \Delta$$

Applying the induction hypothesis means to apply the rule $\mathsf{L}\forall$ bottom to top. In the example, assume that E_1 has type T_1, and E_2 has type T_2, and that all this information is represented by the generalized context Ψ:

$$\Psi = T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ T_1, E_2 : \mathrm{term}\ T_2$$

In the proof, we applied the induction hypothesis twice, once to T_1 and E_1, and once to T_2 and E_2. The $\mathsf{L}\forall$ rule provides exactly this functionality. After the first application, observe how the assumption list Δ grows.

$$\begin{array}{rcl} \Delta^{(1)} & = & \forall T : \mathrm{tp}.\, \forall E : \mathrm{term}\ T.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top, \\ & & \forall E : \mathrm{term}\ T_1.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top \end{array}$$

Then, after applying it a second time to the newly introduced assumption, we obtain:

$$\begin{array}{rcl} \Delta^{(2)} & = & \forall T : \mathrm{tp}.\, \forall E : \mathrm{term}\ T.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top, \\ & & \forall E : \mathrm{term}\ T_1.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top, \\ & & \exists D : E_1 \overset{1}{\Longrightarrow} E_1.\, \top \end{array}$$

Another application of the induction hypothesis, this time on T_2 and E_2 yields $\Delta^{(4)}$.

$$\begin{aligned}\Delta^{(4)} \;=\; & \forall T : \mathrm{tp}. \forall E : \mathrm{term}\; T. \exists D : E \stackrel{1}{\Longrightarrow} E. \top, \\ & \forall E : \mathrm{term}\; T_1. \exists D : E \stackrel{1}{\Longrightarrow} E. \top, \\ & \exists D : E_1 \stackrel{1}{\Longrightarrow} E_1. \top \\ & \forall E : \mathrm{term}\; T_2. \exists D : E \stackrel{1}{\Longrightarrow} E. \top, \\ & \exists D : E_2 \stackrel{1}{\Longrightarrow} E_2. \top\end{aligned}$$

Note, that in order to continue the proof, we have to extract the existentially quantified witness objects from the third and the fifth entry in $\Delta^{(4)}$ back into the generalized context. In natural deduction, this is done by the existential elimination rule, that corresponds in the sequent calculus to the existential left rule:

$$\frac{\Psi, x : A; \Delta_1, \exists x : A. F_1, F_1, \Delta_2 \vdash F}{\Psi; \Delta_1, \exists x : A. F_1, \Delta_2 \vdash F}\,\mathsf{L}\exists$$

Ψ has not changed while applying the $\mathsf{L}\forall$ rule, but it does when applying the $\mathsf{L}\exists$ rule which we must do twice: The first application extends the generalized context by the (true) assumption that $E_1 \stackrel{1}{\Longrightarrow} E_1$

$$\Psi^{(5)} = T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\; T_1, E_2 : \mathrm{term}\; T_2, P_1 : E_1 \stackrel{1}{\Longrightarrow} E_1$$

and the second by the (true) assumption that $E_2 \stackrel{1}{\Longrightarrow} E_2$:

$$\Psi^{(6)} = T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\; T_1, E_2 : \mathrm{term}\; T_2, P_1 : E_1 \stackrel{1}{\Longrightarrow} E_1, P_2 : E_2 \stackrel{1}{\Longrightarrow} E_2$$

All in all, the proof of the case can be finished by applying $\mathsf{R}\exists$ with $M = \mathrm{papp}\; P_1\; P_2$ followed by an application of $\mathsf{R}\top$.

Unfortunately, the two rules just presented do not apply to the hypothetical reasoning case. Consider for example the plam case in the proof the reflexivity Lemma 4.3 for parallel reduction: Before we apply the induction hypothesis, we have to assume the existence of a parameter block of the form $x : \mathrm{term}\; T, u : x \stackrel{1}{\Longrightarrow} x$. When are these assumptions discharged? Obviously, they can only be discharged after the induction hypothesis is applied to all arguments, and all witness objects are moved into the generalized context. From a formal point of view, this operation corresponds to several applications of the $\mathsf{L}\forall$-rule, followed by several applications by the $\mathsf{L}\exists$-rules.

Now it becomes difficult. We claim that we have to be very careful when to introduce and to discharge variable blocks! Just imagine two simultaneously applications to the induction hypothesis, where the first is hypothetical (that means it must extend the world by a new variable block), and the other isn't. Which formulas are valid in which world? The problem reduces to the question of proper scoping of world extensions. In a standard sequent calculus, the context of assumptions has intuitionistic properties, that means that once an assumption is introduced it is present in the context of all judgments in the premiss. The situation of world extensions on the other hand is different. A world is typically extended before an induction hypothesis is applied, and discharged afterwards. Thus, extensions to the world do not possess the standard intuitionistic properties.

Seemingly, we need to extend the world only for the purpose of induction hypothesis and lemma application. Our solution is to introduce a new judgment, that explicitly tracks world extensions. This judgment is defined exclusively in terms of the left rules since they are the ones needed for applying an induction hypothesis, precisely $\mathsf{L}\forall$ and $\mathsf{L}\exists$. While applying the left rules we do not record changes in Ψ immediately. Instead we collect all information, and add it to the intuitionistic context only after the last left rule is applied. This judgment entitles us to reason hypothetically. A special derivation rule which interfaces the standard derivability judgment with the new judgment extends Ψ accordingly. Back to the example of the proof of the reflexivity Lemma 4.3 for parallel reduction. This time, we consider the "lam"-case. Recall that we have to show that $E = \text{lam}\ (\lambda x : \text{term}\ T.\, E'\ x)$ reduces to itself. Formally, we have to construct an LF object of type $(\text{lam}\ (\lambda x : \text{term}\ T.\, E'\ x)) \stackrel{1}{\Longrightarrow} (\text{lam}\ (\lambda x : \text{term}\ T.\, E'\ x))$. This situation is summarized with the following generalized context:

$$\Psi = T : \text{tp}, T' : \text{tp}, E' : \text{term}\ T \to \text{term}\ T'$$

We begin now with a formal appeal to the induction hypothesis. First, we assume the existence of a new parameter block $\underline{x} : \text{term}\ T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$. Then we apply $\mathsf{L}\forall$ to T'.

$$\begin{array}{rcl} \Delta^{(1)} & = & \forall T : \text{tp}.\, \forall E : \text{term}\ T.\, \exists D : E \stackrel{1}{\Longrightarrow} E.\, \top, \\ & & \forall E : \text{term}\ T'.\, \exists D : E \stackrel{1}{\Longrightarrow} E.\, \top \end{array}$$

We then use the $\mathsf{L}\forall$ once again, this time on $(E'\ \underline{x})$ which has type term T'. Note that from an algorithmic point of view the type of E' already prompts for an extension of the world. Otherwise no induction hypothesis is applicable at all.

$$\begin{array}{rcl} \Delta^{(2)} & = & \forall T : \text{tp}.\, \forall E : \text{term}\ T.\, \exists D : E \stackrel{1}{\Longrightarrow} E.\, \top, \\ & & \forall E : \text{term}\ T.\, \exists D : E \stackrel{1}{\Longrightarrow} E.\, \top, \\ & & \exists D : (E'\ \underline{x}) \stackrel{1}{\Longrightarrow} (E'\ \underline{x}).\, \top \end{array}$$

And finally we apply $\mathsf{L}\exists$, and obtain a new assumption: $P : (E'\ \underline{x}) \stackrel{1}{\Longrightarrow} (E'\ \underline{x})$. Clearly, in order to add P to the generalized context, we have to abstract according to Equation (4.1) on page 76, and we obtain

$$\begin{array}{rcl} \Psi^{(3)} & = & T : \text{tp}, T' : \text{tp}, E' : \text{term}\ T \to \text{term}\ T', \\ & & P : \Pi x : \text{term}\ T.\, \Pi u : x \stackrel{1}{\Longrightarrow} x.\, (E'\ x) \stackrel{1}{\Longrightarrow} (E'\ x). \end{array}$$

Similarly, we abstract the new meta-assumptions in $\Delta^{(2)}$ and obtain

$$\begin{array}{rcl} \Delta^{(3)} & = & \forall T : \text{tp}.\, \forall E : \text{term}\ T.\, \exists D : E \stackrel{1}{\Longrightarrow} E.\, \top, \\ & & \mathbf{\Pi}(\underline{x} : \text{term}\ T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L.\, \forall E : \text{term}\ T.\, \exists D : E \stackrel{1}{\Longrightarrow} E.\, \top, \\ & & \mathbf{\Pi}(\underline{x} : \text{term}\ T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L.\, \exists D : (E'\ \underline{x}) \stackrel{1}{\Longrightarrow} (E'\ \underline{x}).\, \top \end{array}$$

How do we represent the extension of the world in the new to be defined judgment? The answer is that we simply extend the general context Ψ by the declaration of a new variable block. But in general this information is not enough to abstract the hypothetical assumptions after finishing applying the left rules, because there are possibly many variable blocks declared in Ψ, and many of them of them must not be discharged. Which variable blocks must be discharged and abstracted after a successful application of the left rules is represented by the derivation in $\mathcal{M}_2^+$: abstraction takes place while unraveling the trace of left rules. The left rules are defined via a new judgment which we call provability of declarations.

Judgment

Provability of declarations: $\Psi; \Delta \vdash \Psi'; \Delta'$

The generalized context Ψ, and the list of meta-assumptions Δ on the left carry the same meaning as in the judgments for provability and general provability above. Ψ is used to capture extensions of the current world. Ψ' and Δ' declared left of the $\vdash$ symbol represent a list new assumptions and new meta-assumptions which are synthesized during the application of the left rules, and which will be added eventually to the generalized context. Operationally interpreted, Ψ' and Δ' are constructed after all left rules are applied. The judgment can be read as a function: $\Psi; \Delta$ are input variables, and $\Psi'; \Delta'$ are output variables. Initially, in the example, before the induction hypothesis is applied, Ψ and Δ have the following form:

$$\begin{array}{rcl} \Psi^{(1)} & = & T : \mathrm{tp}, T' : \mathrm{tp}, E' : \mathrm{term}\ T \to \mathrm{term}\ T' \\ \Delta^{(1)} & = & \forall T : \mathrm{tp}. \forall E : \mathrm{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top \end{array}$$

We start now with the application of the induction hypothesis. First, a parameter block is introduced, and its existence is made visible by a variable block in the generalized context:

$$\begin{array}{rcl} \Psi^{(2)} & = & T : \mathrm{tp}, T' : \mathrm{tp}, E' : \mathrm{term}\ T \to \mathrm{term}\ T', (\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L \\ \Delta^{(2)} & = & \forall T : \mathrm{tp}. \forall E : \mathrm{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top \end{array}$$

Next, the induction hypothesis is applied to $T : \mathrm{tp}$ using a new version of the $\mathsf{L}\forall$-rule, which we introduce formally below.

$$\begin{array}{rcl} \Psi^{(3)} & = & T : \mathrm{tp}, T' : \mathrm{tp}, E' : \mathrm{term}\ T \to \mathrm{term}\ T', (\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L \\ \Delta^{(3)} & = & \forall T : \mathrm{tp}. \forall E : \mathrm{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top, \\ & & \forall E : \mathrm{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top, \end{array}$$

Another application of the rule $\mathsf{L}\forall$, this time to $E'\ x$ (well-typed in $\Psi^{(3)}$) yields

$$\begin{array}{rcl} \Psi^{(4)} & = & T : \mathrm{tp}, T' : \mathrm{tp}, E' : \mathrm{term}\ T \to \mathrm{term}\ T', (\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L \\ \Delta^{(4)} & = & \forall T : \mathrm{tp}. \forall E : \mathrm{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top, \\ & & \forall E : \mathrm{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top, \\ & & \exists D : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x). \top \end{array}$$

which allows us to assume the existence of a $P : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x)$ well-typed in $\Psi^{(4)}$ by rule $\mathsf{L}\exists$ (also defined below), and the body of the last formula in $\Delta^{(4)}$ becomes a meta-assumption.

$$\begin{array}{rcl} \Psi^{(5)} & = & T : \mathrm{tp}, T' : \mathrm{tp}, E' : \mathrm{term}\ T \to \mathrm{term}\ T', (\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L \\ & & P : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x) \\ \Delta^{(5)} & = & \forall T : \mathrm{tp}. \forall E : \mathrm{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top, \\ & & \forall E : \mathrm{term}\ T. \exists D : E \overset{1}{\Longrightarrow} E. \top, \\ & & \exists D : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x). \top, \\ & & \top \end{array}$$

The induction hypothesis is now completely applied, and we can begin to unravel the trace of left rule applications. Unraveling in this sense mean to step back through the call tree while discharging and abstracting hypothetical parameter blocks. Simultaneously, we construct the $\Psi'; \Delta'$, extensions of the original generalized context and meta-assumptions list. In the last step, nothing has been done, so both extensions are empty.

$$\begin{array}{rcl} \Psi'^{(5)} & = & \cdot \\ \Delta'^{(5)} & = & \cdot \end{array}$$

In the step before that, two assumptions were recorded, one in the generalized context, the other in Δ:

$$\begin{array}{rcl} \Psi'^{(4)} & = & P : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x) \\ \Delta'^{(4)} & = & \top \end{array}$$

Note, that $\Psi^{(4)}, \Psi'^{(4)}$ is a generalized context, and $\Delta^{(4)}, \Delta'^{(4)}$ is a meta-assumption list. Another step before, we applied the $\mathsf{L}\forall$ rule, and thus, we add the newly generated meta-assumption to the left.

$$\begin{array}{rcl} \Psi'^{(3)} & = & P : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x) \\ \Delta'^{(3)} & = & \exists D : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x).\, \top \\ & & \top \end{array}$$

Note, that we maintain the invariant that $\Delta^{(4)}, \Delta'^{(4)}$ is a valid meta-assumption list.

$$\begin{array}{rcl} \Psi'^{(2)} & = & P : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x) \\ \Delta'^{(2)} & = & \forall E : \mathrm{term}\ T.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top, \\ & & \exists D : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x).\, \top, \\ & & \top \end{array}$$

The last step in this example is the important step because it demonstrates how to discharge assumptions by abstracting and internalizing the newly assumed parameter block from the first application of the parameter introduction rule. Informally, we apply Equation (4.1) to all assumptions in $\Psi'^{(2)}$ in order to obtain $\Psi'^{(1)}$, and simply bind the new meta-level assumptions by $\mathbf{\Pi}$. The result is

$$\begin{array}{rcl} \Psi'^{(1)} & = & P : \Pi x : \mathrm{term}\ T.\, \Pi u : x \overset{1}{\Longrightarrow} x.\, (E'\ x) \overset{1}{\Longrightarrow} (E'\ x) \\ \Delta'^{(1)} & = & \mathbf{\Pi}(\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L.\, \forall E : \mathrm{term}\ T.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top, \\ & & \mathbf{\Pi}(\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L.\, \exists D : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x).\, \top, \\ & & \mathbf{\Pi}(\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L.\, \top \end{array}$$

and the proof can continue with $\Psi'^{(1)}, \Psi'^{(1)}; \Delta'^{(1)}, \Delta'^{(1)}$, that is:

$$\begin{array}{rcl} \Psi^{(1)}, \Psi'^{(1)} & = & T : \mathrm{tp}, T' : \mathrm{tp}, E' : \mathrm{term}\ T \to \mathrm{term}\ T' \\ & & P : \Pi x : \mathrm{term}\ T.\, \Pi u : x \overset{1}{\Longrightarrow} x.\, (E'\ x) \overset{1}{\Longrightarrow} (E'\ x) \\ \Delta^{(1)}, \Delta'^{(1)} & = & \forall T : \mathrm{tp}.\, \forall E : \mathrm{term}\ T.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top \\ & & \mathbf{\Pi}(\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L.\, \forall E : \mathrm{term}\ T.\, \exists D : E \overset{1}{\Longrightarrow} E.\, \top, \\ & & \mathbf{\Pi}(\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L.\, \exists D : (E'\ x) \overset{1}{\Longrightarrow} (E'\ x).\, \top, \\ & & \mathbf{\Pi}(\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L.\, \top \end{array}$$

This concludes our motivational example of how to formalize hypothetical reasoning. The skeptical user might wonder, why is is necessary to also maintain Δ'. As a matter of fact, it is not. But because partially applied lemmas are always available in a regular intuitionistic sequent calculus we have decided to also keep them in $\mathcal{M}_2^+$.

What remains to be done, is a formal definition of the rules that we have used in this example. We start with the presentation of the interface rule sel which triggers a sequence of left rules, in a very similar manner as shown in the example above.

$$\frac{\Psi; \Delta \vdash \Psi'; \Delta' \qquad \Psi, \Psi'; \Delta, \Delta' \vdash F}{\Psi; \Delta \vdash F}\,\mathsf{sel}$$

The first rule Ldone, is the rule which terminates a sequence of left rules. It is basically the complementary rule to sel, which can be seen as initiator of a sequence of left rules.

$$\frac{}{\Psi; \Delta \vdash \cdot; \cdot}\,\mathsf{Ldone}$$

The second rule Lnew supports the introduction of new parameters. $\Psi'; \Delta'$ are the returning extensions of the generalized context and the meta-assumptions, which are accordingly abstracted. Tentatively, as a first sketch, we write $\Pi\rho.\, A$ to abstract over a variable block.

$$\begin{array}{rcl} \Pi\cdot.\, A_2 & = & A_2 \\ \Pi(x : A_1, \rho).\, A_2 & = & \Pi x : A_1.\, (\Pi\rho.\, A_2) \end{array}$$

Note that this definition omits the underlines below x because the result of abstraction lives in LF. Our definition of abstraction can be easily generalized to lists of assumptions: $\Pi\rho^L.\, (\Psi'; \Delta')$. Note, that this Π is not a constructor, neither in LF nor in $\mathcal{M}_2^+$, it is merely an abbreviation for a function, that performs the abstraction on the fly. In Section 6.2.2 we refine abstractions to account only for variables declarations that may occur in the body of A. Declarations which cannot occur in A should be omitted.

$$\frac{(\text{SOME } C_1.\, \text{BLOCK } C_2)^L \in S \qquad \Psi \vdash \sigma : C_1 \qquad \Psi \vdash \rho \equiv_\alpha [\sigma]C_2 \qquad \Psi, \rho^L; \Delta \vdash \Psi'; \Delta'}{\Psi; \Delta \vdash \Pi\rho^L.\, (\Psi'; \Delta')}\,\mathsf{Lnew}$$

The $\mathsf{L}\forall$- and the $\mathsf{L}\Pi$ rule generalize the $\mathsf{L}\forall$ rule from above. Note that we must ensure, as premiss in $\mathsf{L}\forall$, that M is well-typed, and likewise, as premiss in $\mathsf{L}\Pi$, that ρ' is well-typed. In the former case we use the typing judgments from LF, in the latter, abstract type convertibility for variable blocks.

$$\frac{\Psi; \Delta \vdash \forall x : A.\, F \qquad \Psi \vdash M : A \qquad \Psi; \Delta, F[M/x] \vdash \Psi'; \Delta'}{\Psi; \Delta \vdash \Psi'; F[M/x], \Delta'}\,\mathsf{L}\forall$$

$$\frac{\Psi; \Delta \vdash \mathbf{\Pi}\rho^L.\, F \qquad \rho'^L \in \Psi \qquad \Psi \vdash \rho' \equiv_\alpha \rho \qquad \Psi; \Delta, F[\rho'/\rho] \vdash \Psi'; \Delta'}{\Psi; \Delta \vdash \Psi'; F[\rho'/\rho], \Delta'}\,\mathsf{L}\Pi$$

The $\mathsf{L}\exists$ rule is the only rule which extends the generalized context Ψ'.

$$\frac{\Psi; \Delta \vdash \exists x : A.\, F \qquad \Psi, x : A; \Delta, F \vdash \Psi'; \Delta'}{\Psi; \Delta \vdash x : A, \Psi'; F, \Delta'}\,\mathsf{L}\exists$$

Finally, there are to rules which project the left or right proof term from a conjunction.

$$\frac{\Psi; \Delta \vdash F_1 \wedge F_2 \quad \Psi; \Delta, F_1 \vdash \Psi'; \Delta'}{\Psi; \Delta \vdash \Psi'; F_1, \Delta'} \mathsf{L}\wedge_1 \qquad \frac{\Psi; \Delta \vdash F_1 \wedge F_2 \quad \Psi; \Delta, F_2 \vdash \Psi'; \Delta'}{\Psi; \Delta \vdash \Psi'; F_2, \Delta'} \mathsf{L}\wedge_2$$

This concludes our presentation of the proof system for $\mathcal{M}_2^+$. On the one hand, the proof system borrows many ideas and concepts from the sequent calculus for intuitionistic logic, on the other it is significantly different. In order to accommodate hypothetical reasoning, for example, the original judgments must be specialized. New parameter blocks are introduced by the rule Lnew which also abstracts the results of applying the induction hypothesis appropriately. The method of abstraction is not as straightforward as it may seem from the examples above, we postpone the detailed discussion until Section 6.2.2.

The remainder of this chapter is organized in three parts. First we add proof terms to $\mathcal{M}_2^+$, which formalize meta-proofs by summarizing entire $\mathcal{M}_2^+$-derivations and which form the basis of our soundness argument. Second, we add two rules in order to express case analysis and recursion in order to generalize the proof term calculus to a calculus for recursive functions and third we add lemmas to the meta-logic. Recursion and case analysis allow us to encode proofs "by induction" over higher-order encodings that may violate the positivity condition associated with standard inductive definitions.

5.5 Proof Term Calculus

In this section, we endow the proof calculus of $\mathcal{M}_2^+$ with proof terms. Proof terms are very concise representations of derivations in a formal system. As a matter of fact, given a proof term for a theorem, the original derivation can be unambiguously reconstructed. But this is not the only advantage of proof terms: In general it is possible to interpret them operationally. Consider the natural deduction calculus for propositional logic by Gentzen [Gen35, Pra65]. Each derivation of a formula can be uniquely represented by a simply-typed λ-term using the propositions-as-types principles. This observation goes back Howard [How69] and is commonly known as the Curry-Howard isomorphism. In this work, we interpret proof terms as recursive function, and by an argument of realizability interpretation we will eventually infer the soundness of $\mathcal{M}_2^+$. By moving the soundness argument of $\mathcal{M}_2^+$ form the logical level to the proof term level, we manage to avoid stating explicit induction principles for higher-order encodings. Instead, we argue that $\mathcal{M}_2^+$ is sound, because it only admits proof terms that guarantee complete case coverage and well-founded recursion.

We begin with the presentation of a proof term calculus for $\mathcal{M}_2^+$. In Chapter 6 we then define a type preserving operational semantics for it, and in Chapter 7 we will show that each function is total, yielding a soundness proof for $\mathcal{M}_2^+$. All recursive functions presented in Chapter 4 are proof terms. For improved readability, so far we have used some syntactic sugar in order to make proof terms more accessible to the user, and we omitted for example all implicit arguments in order to simplify the presentation, but in essence, the proof terms we present in this section have all been already discussed informally. As example, consider the proof of the reflexivity

Lemma 4.3.

$$
\begin{array}{l}
\textbf{fun refl } \underline{x} = \underline{u} \\
\quad \mid \textbf{refl } (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T.\, E'\ x)) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new } \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val} \quad P\ \underline{x}\ \underline{u} = \textbf{refl } (E'\ \underline{x}) \\
\qquad \textbf{in} \\
\qquad\quad \mathrm{plam}\ (\lambda x : \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, P\ x\ u) \\
\qquad \textbf{end} \\
\quad \mid \textbf{refl } (\mathrm{app}\ E_1\ E_2) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val } P_1 = \textbf{refl } E_1 \\
\qquad\quad \textbf{val } P_2 = \textbf{refl } E_2 \\
\qquad \textbf{in} \\
\qquad\quad \mathrm{papp}\ P_1\ P_2 \\
\qquad \textbf{end}
\end{array}
$$

In this section we concentrate on proof terms representating the body of each of the cases. The presentation of proof terms for pattern matching and recursion is postponed until the next Section 5.6. There are three kinds of proof terms: general proof terms for general provability judgments $\vdash G$, proof terms or programs for the right rules expressed by the judgment $\Psi; \Delta \vdash F$, and declarations for the left rules, expressed by the provability judgment $\Psi; \Delta \vdash \Psi'; \Delta'$. General proof terms are abbreviated with Q, proof terms with P and declarations with D. In order not confuse provability on the meta-level with typability on the logical framework level, we use $\in$ as the structural symbol in Δ and between proof terms and formulas.

Judgments

Provability of general formulas:	$\vdash Q \in G$
Provability of formulas:	$\Psi; \Delta \vdash P \in F$
Provability of declarations:	$\Psi; \Delta \vdash D \in \Psi'; \Delta'$

In the following three subsections, we define proof terms for each of the three judgments.

5.5.1 Provability of General Formulas

There is only one general formula. It is the closure operator, and it binds the context schema in which the formula makes sense. It is mandatory to represent the context schema on the level of proof terms since we cannot apply a lemma without validating the context schema of the called lemma.

$$\textit{General proof terms:} \quad Q \quad ::= \quad \mathbf{box}\ S.\, P$$

$$\frac{\cdot; \cdot \vdash P \in F}{\vdash \mathbf{box}\ S.\, P \in \Box S.\, F}\ \mathsf{generalR}$$

Note again that the judgment in the premiss of this rule is implicitly indexed by the context schema S. We discuss how to use proofs of generalized formulas as lemmas in Section 5.7.

5.5.2 Provability of Formulas

The proof terms for the provability judgment for formulas and inference rules provide an operational interpretation of derivations in $\mathcal{M}_2^+$. Recall that the provability judgment of formulas is defined by the right rules of the proof calculus of $\mathcal{M}_2^+$. Proof terms for the judgment whose validity is given by left rules are presented in Section 5.5.3. We start with endowing the axvar-rule with a proof term. As in any other proof term calculus [Gal93] assumptions are named and the name of an assumption is used as the proof term. Specifically, in our setting, meta-assumptions are labeled with variable names. Since this is already the third variable concept presented in this thesis, but the first for the meta-level, we call them meta-variables and use little bold Roman letters to denote them $(\mathbf{x}, \mathbf{y}, \mathbf{z})$. The list of meta-assumptions is generalized accordingly.

$$\textit{Meta-assumptions:} \quad \Delta \quad ::= \quad \cdot \mid \Delta, \mathbf{x} \in F$$

As usual, we assume that all meta-variable names among meta-assumptions in Δ are pairwise distinct. Assigning meta-variable names to meta-assumptions extends the rule vmeta slightly.

$$\frac{\vdash \Psi \text{ abstract}}{\Psi \vdash \cdot \text{ meta}}\,\mathsf{vabstract} \qquad \frac{\Psi \vdash \Delta \text{ meta} \qquad \Psi \vdash F \text{ formula}}{\Psi \vdash \Delta, \mathbf{x} \in F \text{ meta}}\,\mathsf{vmeta}$$

All meta-variables, defined in Δ are subject to instantiation. And instantiations of variables is best described by substitutions. In particular, in the case of meta-contexts and meta-variables, we introduce the notion of meta-substitution, and denote it with δ.

$$\textit{Meta-substitutions:} \quad \delta \quad ::= \quad \cdot \mid \delta, P/\mathbf{x}$$

The newly introduced meta-variables are used as proof terms for the rule axvar. If $\mathbf{x} \in F$ is an assumption in Δ, then $\mathbf{x}$ is a proof term for F.

$$\frac{(\mathbf{x} \in F) \in \Delta}{\Psi; \Delta \vdash \mathbf{x} \in F}\,\mathsf{axvar}$$

The proof term for the R$\forall$-rule is a simple abstraction, similar to the λ-abstraction in the standard simply-typed λ-calculus. The proof term has the form $\Lambda x : A.\, P$. Similarly, the proof term for RΠ is an abstraction over variable blocks, which can, at runtime, only be instantiated with other variable blocks. The proof term for RΠ has the form $\lambda \rho^L.\, P$.

$$\frac{\Psi, x : A; \Delta \vdash P \in F}{\Psi; \Delta \vdash \Lambda x : A.\, P \in \forall x : A.\, F}\,\mathsf{R}\forall \qquad \frac{\Psi, \rho^L; \Delta \vdash P \in F}{\Psi; \Delta \vdash \lambda \rho^L.\, P \in \mathbf{\Pi} \rho^L.\, F}\,\mathsf{R}\Pi$$

Not surprisingly, the proof term for the R$\exists$-rule looks like a pair $\langle M, P\rangle$, where M is a well-typed LF object — the witness object for the existential — and P is the proof term for the body of the existential formula. As a matter of fact, the proof term for the conjunction rule is very similar; it is also a pair, where each component is a proof term of the left and right formula, respectively. Its form is $\langle P_1, P_2\rangle$.

$$\frac{\Psi \vdash M : A \qquad \Psi; \Delta \vdash P \in F[M/x]}{\Psi; \Delta \vdash \langle M, P\rangle \in \exists x : A.\, F}\,\mathsf{R}\exists \qquad \frac{\Psi; \Delta \vdash P_1 \in F_1 \qquad \Psi; \Delta \vdash P_2 \in F_2}{\Psi; \Delta \vdash \langle P_1, P_2\rangle \in F_1 \wedge F_2}\,\mathsf{R}\wedge$$

The rule for $\top$ is endowed with the symbol $\langle\rangle$ as proof term. Clearly, $\langle\rangle$ does not expect any arguments.

$$\frac{}{\Psi;\Delta \vdash \langle\rangle \in \top}\,\mathsf{R}\top$$

And finally, there is a proof term for the interface rule sel. A derivation of the provability judgments for declarations corresponds exactly to the list of declarations in a **let**-expression, as it is depicted in the following excerpt from the proof of the reflexivity Lemma 4.3 for parallel reduction.

$$\begin{array}{l}\mathbf{let}\\ \quad \mathbf{new}\ \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x}\\ \quad \mathbf{val}\ \ P\ \underline{x}\ \underline{u} = \mathbf{refl}\ (E'\ \underline{x})\\ \mathbf{in}\\ \quad \mathrm{plam}\ (\lambda x : \mathrm{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, P\ x\ u)\\ \mathbf{end}\end{array}$$

The list of declarations is represented by declarations D, the proof term for the body is P. Together they form the arguments to a proof term for the sel-rule which we denote as **let** D **in** P.

$$\frac{\Psi;\Delta \vdash D \in \Psi';\Delta' \qquad \Psi,\Psi';\Delta,\Delta' \vdash P \in F}{\Psi;\Delta \vdash \mathbf{let}\ D\ \mathbf{in}\ P \in F}\,\mathsf{sel}$$

All in all, there are seven different proof term constructors, one for each rule, and all are different. That is, given a proof term, one can immediately reconstruct the derivation by decomposing a proof term into its components. Here is a complete list of all the proof terms for formulas.

$$\textit{Proof terms:}\quad P \quad ::= \quad \mathbf{x} \mid \Lambda x : A.\, P \mid \lambda\rho^L.\, P \mid \langle M, P\rangle \mid \langle P_1, P_2\rangle \mid \langle\rangle \mid \mathbf{let}\ D\ \mathbf{in}\ P$$

This concludes the presentation of proof terms for the provability judgment for formulas. On the one hand, this fragment is very weak, because it can neither apply induction hypotheses, nor lemmas, nor perform any kind of case analysis, but on the other, we can already represent small easy proofs. As example consider the following very simple lemma that states that the two single parallel reduction steps can be appended to a multi-step parallel reduction.

Lemma 5.11 (Append two single parallel reduction steps) *If* $\mathcal{D}_1 :: e_1 \overset{1}{\Longrightarrow} e_2$ *and* $\mathcal{D}_2 :: e_2 \overset{1}{\Longrightarrow} e_3$ *then there exists a* $\mathcal{P} :: e_1 \overset{*}{\Longrightarrow} e_3$.

Its formalization in $\mathcal{M}_2^+$ has the following form:

$$\begin{array}{l}\Box\cdot.\,\forall T : \mathrm{tp}.\,\forall E_1 : \mathrm{term}\ T.\,\forall E_2 : \mathrm{term}\ T.\,\forall E_3 : \mathrm{term}\ T.\\ \qquad\qquad \forall D_1 : E_1 \overset{1}{\Longrightarrow} E_2.\,\forall D_2 : E_2 \overset{1}{\Longrightarrow} E_3.\\ \qquad\qquad \exists P : E_1 \overset{*}{\Longrightarrow} E_3.\,\top\end{array}$$

And the proof is very simple:

$$\dfrac{\begin{array}{c}\mathcal{D}_1\\ E_1 \overset{1}{\Longrightarrow} E_2\end{array} \qquad \dfrac{\begin{array}{c}\mathcal{D}_2\\ E_2 \overset{1}{\Longrightarrow} E_3\end{array} \qquad \dfrac{}{E_3 \overset{*}{\Longrightarrow} E_3}\,\mathrm{pid}}{E_2 \overset{*}{\Longrightarrow} E_3}\,\mathrm{pstep}}{E_1 \overset{*}{\Longrightarrow} E_3}\,\mathrm{pstep}$$

and so is its representation as a proof term in $\mathcal{M}_2^+$.

$$\begin{array}{l}\textbf{box}\ \cdot.\, \Lambda T : \mathrm{tp}.\, \Lambda E_1 : \mathrm{term}\ T.\, \Lambda E_2 : \mathrm{term}\ T.\, \Lambda E_3 : \mathrm{term}\ T.\\ \quad \Lambda D_1 : E_1 \stackrel{1}{\Longrightarrow} E_2.\, \Lambda D_2 : E_2 \stackrel{1}{\Longrightarrow} E_3.\\ \quad \langle \mathrm{pstep}\ D_1\ (\mathrm{pstep}\ D_2\ \mathrm{pid}), \langle\rangle\rangle\end{array}$$

In order to show more interesting examples, we must decorate the left rules with declarations and add case distinction and recursion. Therefore, a complete proof term for the reflexivity lemma can only be given at the end of Section 5.6.

5.5.3 Provability of Declarations

The proof terms D for the provability judgment $\Psi; \Delta \vdash D \in \Psi'; \Delta'$ are called declarations, because they correspond directly to the sequence of declarations in a let statement. In this subsection we show how. Declarations are represented as a list. The simplest declaration is hence the empty list, and it is the proof term of Ldone. Following the line of empty contexts, empty signatures, and empty context schemas, we denote the empty proof term with "$\cdot$".

$$\frac{}{\Psi; \Delta \vdash \cdot \in \cdot; \cdot}\,\mathsf{Ldone}$$

The proof term for Lnew has the form $\nu\ \rho^L.\, D$, where ρ^L is a variable block representing the newly assumed parameter block, and D is the list of subsequent declarations.

$$\frac{S(L) = \mathrm{SOME}\ C_1.\, \mathrm{BLOCK}\ C_2 \quad \Psi \vdash \sigma : C_1 \quad \Psi \vdash \rho \equiv_\alpha [\sigma]C_2 \quad \Psi, \rho^L; \Delta \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash \nu\ \rho^L.\, D \in \Pi\rho^L.\, (\Psi'; \Delta')}\,\mathsf{Lnew}$$

The declarations for L$\forall$ and LΠ are very similar. In the first case, the declaration $\mathbf{y} \in F[M/x] = P\ M$, and in the second $\mathbf{y} \in F[\rho'/\rho] = P\ \rho'$ is added to the list of already determined declarations D. To judge by the form, P is a functional proof term in both cases, expecting an LF object M as argument in the first case, and expecting a variable block ρ' in the second.

$$\frac{\Psi; \Delta \vdash P \in \forall x : A.\, F \quad \Psi \vdash M : A \quad \Psi; \Delta, \mathbf{y} \in F[M/x] \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash (\mathbf{y} \in F[M/x] = P\ M, D) \in (\Psi'; \mathbf{y} \in F[M/x], \Delta')}\,\mathsf{L}\forall$$

$$\frac{\Psi; \Delta \vdash P \in \mathbf{\Pi}\rho^L.\, F \quad \rho'^L \in \Psi \quad \Psi \vdash \rho' \equiv_\alpha \rho \quad \Psi; \Delta, \mathbf{y} \in F[\rho'/\rho] \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash (\mathbf{y} \in F[\rho'/\rho] = P\ \rho', D) \in (\Psi'; \mathbf{y} \in F[\rho'/\rho], \Delta')}\,\mathsf{L}\Pi$$

The left rule for $\exists$ captures the result of an induction hypothesis and adds it to the generalized context. Formally, it is expressed by the declaration $\langle x : A, \mathbf{y} \in F\rangle = P$ where P is a proof term, which computes a pair $\langle M, P'\rangle$ and the declaration operations bind x to M and $\mathbf{y}$ to P'. And again, as we will see in the next chapter x and $\mathbf{y}$ must be explicitly typed.

$$\frac{\Psi; \Delta \vdash P \in \exists x : A.\, F \quad \Psi, x : A; \Delta, \mathbf{y} \in F \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash (\langle x : A, \mathbf{y} \in F\rangle = P, D) \in (x : A, \Psi'; \mathbf{y} \in F, \Delta')}\,\mathsf{L}\exists$$

Finally, there are two projection rules for conjunction on the left. Informally these rules are used to pick which induction hypothesis is supposed to be applied when proving a mutually inductive theorem. The declaration for selecting the left induction hypothesis is $\mathbf{x} \in F_1 = \pi_1\ P$, and he one of the right is not very surprisingly $\mathbf{x} \in F_2 = \pi_2\ P$ where P represents the proof of the mutual inductive theorem.

$$\frac{\Psi; \Delta \vdash P \in F_1 \wedge F_2 \qquad \Psi; \Delta, \mathbf{x} \in F_1 \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash (\mathbf{x} \in F_1 = \pi_1\ P, D) \in (\Psi'; \mathbf{x} \in F_1, \Delta')}\ \mathsf{L}\wedge_1$$

$$\frac{\Psi; \Delta \vdash P \in F_1 \wedge F_2 \qquad \Psi; \Delta, \mathbf{x} \in F_2 \vdash D \in \Psi'; \Delta'}{\Psi; \Delta \vdash (\mathbf{x} \in F_2 = \pi_2\ P, D) \in (\Psi'; \mathbf{x} \in F_2, \Delta')}\ \mathsf{L}\wedge_2$$

Alternatively, one could replace these two rules by one rule that introduces both projections simultaneously.

All in all, there are six different forms of declarations, each represents one rule. In particular, a proof term for a derivation in $\mathcal{M}_2^+$ is simply a series of declarations.

$$\begin{array}{llll} \textit{Declarations:} & D & ::= & \cdot \mid \nu\, \rho^L.\, D \mid \mathbf{x} \in F = P\ M, D \mid \mathbf{x} \in F = P\ \rho, D \\ & & & \mid \langle x : A, \mathbf{y} \in F \rangle = P, D \mid \mathbf{x} \in F = \pi_1\ P, D \mid \mathbf{x} \in F = \pi_2\ P, D \end{array}$$

This concludes the presentation of proof terms for the left rules of $\mathcal{M}_2^+$, and completes the presentation of proof terms for the core of the meta-logic $\mathcal{M}_2^+$. The proof term calculus is obviously not completely defined yet because none of the non-trivial left rules are applicable. The attentive reader might have already noticed that Δ must be empty since none of the right rules extends it. Hence, none of the left rules (except Ldone) is applicable in the system defined so far. This is going to change when we introduce recursion and case analysis operators in the next Section 5.6. In particular, there are no interesting examples we could develop in this version of $\mathcal{M}_2^+$, therefore we delay an example until the end of the next section.

5.6 Induction

As motivated in Section 4.1, induction is an important technique when it comes to reason about programming languages, logics, and type systems. Informally, reasoning by induction about programming languages is a not too difficult concept, but formalizing it in the presence of higher-order representations is problematic.

The main drawback of standard induction principles is the closed world assumption, which restricts the formalization of deductive systems to encodings that satisfy the positivity condition. The datatype defined must only occur in positive positions in its constructor types. Thus inductive definitions are very restrictive, in fact, they are too restrictive to handle higher-order encodings. The entire proof of the Church-Rosser theorem, for example, from Chapter 4 in all its elegance is simply not directly representable in a framework which only provides standard induction principles.

The goal and challenge of this section is to extend $\mathcal{M}_2^+$ by constructs to support the formalization of inductive arguments. Instead of trying to define induction principles for higher-order encodings, we propose a design based on a realizability interpretation of proof terms. In particular, the solution we are proposing in this thesis is to extend the proof term calculus to a recursive functional calculus, where all functions are total — i.e. realizers. Specifically, we are extending $\mathcal{M}_2^+$ by the two principles which are sufficient to formalize inductive arguments: well-founded recursion and complete case analysis.

Well-founded recursion as opposed to simple recursion guarantees that the computation of any recursive function is terminating. There cannot be any infinite chains of recursive calls. Recursion is discussed in Section 5.6.1.

Complete case analysis as opposed to simple case analysis guarantees that while executing a recursive function some case will be applicable. Therefore the execution of any recursive function can never get stuck. The technique of complete case analysis is discussed in Section 5.6.2. The exact definition of what it means to execute a recursive function, i.e. its operational semantics is presented in Chapter 6.

By guaranteeing well-founded recursion and complete case analysis, all recursive function in $\mathcal{M}_2^+$ are total, and consequently, it is a sound meta-logic based on a realizability interpretation of its proof terms.

5.6.1 Well-Founded Recursion

Well-founded recursion is expressed by the standard fixed-point rule with an open-ended side condition. The new proof term has the form $\mu\mathbf{x} \in F.\,P$. $\mathbf{x}$ is a meta-variable, and P the body of the fixed-point operator, where $\mathbf{x}$ may occur as a free variable. Informally, executing $\mu\mathbf{x} \in F.\,P$ means to replace all occurrences of $\mathbf{x}$ in P by $\mu\mathbf{x} \in F.\,P$, but this is discussed in the next Chapter.

$$\textit{Proof Term:}\quad P \quad ::= \quad \ldots \mid \mu\mathbf{x} \in F.\,P$$

The main emphasis of this investigation is how to enforce termination when executing the fixed-point operator. In our development, we assume that we have only one outermost fixed-point operator. If the fixed-point variable $\mathbf{x}$ occurs someplace else in the body, it is typically applied to some arguments.

The critical insight into the issue of termination is, that the vector of arguments to which $\mathbf{x}$ is applied is strictly smaller than the vector of arguments the function was originally called with. The "smaller" relation must be some well-founded order, i.e. the termination order we specified with each proof in Chapter 4. Naturally, this order must be fixed for all occurrences of $\mathbf{x}$. This way, we can guarantee that each chain of recursive calls is finite, and hence the execution of any recursive function must be terminating.

$$\frac{\Psi;\Delta,\mathbf{x} \in F \vdash P \in F}{\Psi;\Delta \vdash \mu\mathbf{x} \in F.\,P \in F}\,\mathsf{Rctx}$$

The typing rule for the fixed point is standard, but the side condition is not. For now, we leave it purposely informal, a more concise formulation is left to Section 7.2.

$$P \text{ terminates in } \mathbf{x} \tag{5.1}$$

5.6.2 Complete Case Analysis

Well-founded recursion and complete case analysis turn the proof term calculus into a calculus of total recursive functions. In particular, we discuss in this section of how to add a case operator to the meta-logic, and how to enforce that case analysis is always complete. What characterizes case analysis? In order to answer this question, we start the discussion with the reflexivity Lemma 4.3 for parallel reduction as example.

$$\Box(\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x)^L.$$
$$\forall T : \text{tp.} \forall E : \text{term } T. \exists D : E \stackrel{1}{\Longrightarrow} E. \top$$

In the proof we distinguished cases over the term e which is represented in LF as $E : \text{term} \ulcorner\tau\urcorner$. A closer look at e led us to consider three cases. One case was the global parameter case, the second the lam-case, and the third the app-case. It is this case analysis we would like to model in $\mathcal{M}_2^+$. We omit the leading context schema quantification.

Case: In the first case, the parameter context must contain at least one parameter block of the form $\underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$, that means, that we have to prove the formula

$$\forall T : \text{tp.} \mathbf{\Pi}(\underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L. \exists D : x \stackrel{1}{\Longrightarrow} x. \top$$

Case: In the second case, there is no parameter block, but there is a function representing the body of the λ-term.

$$\forall T_1 : \text{tp.} \forall T_2 : \text{tp.} \forall E' : \text{term } T_1 \to \text{term } T_2. \exists D : \text{lam } E' \stackrel{1}{\Longrightarrow} \text{lam } E'. \top$$

Case: In the last case, there are two new assumptions, one represents the function, and the second its argument.

$$\forall T_1 : \text{tp.} \forall T_2 : \text{tp.} \forall E_1 : \text{term } (T_2 \text{ arrow } T_1). \forall E_2 : \text{term } T_2.$$
$$\exists D : \text{app } E_1\ E_2 \stackrel{1}{\Longrightarrow} \text{app } E_1\ E_2. \top$$

The first observation is that case analysis is not local; in general we have to consider more than one assumption in Ψ. For example in all three cases above the formula $\exists D : E \stackrel{1}{\Longrightarrow} E. \top$ is refined by instantiating E with the concrete forms E takes in each case, "$\underline{x}$" in the first, "lam E'" in the second and "app E_1 E_2" in the third. In particular, if E occurred in the types of other universally quantified assumptions, these occurrences would be instantiated, too. Moreover, because of dependencies, consider cases over one assumption might partially instantiate others. To see that, consider the diamond Lemma 4.6.

$$\Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x.$$
$$\forall T : \text{tp.} \forall E : \text{term } T. \forall E^l : \text{term } T. \forall E^r : \text{term } T.$$
$$\forall D^l : E \stackrel{1}{\Longrightarrow} E^l. \forall D^r : E \stackrel{1}{\Longrightarrow} E^r.$$
$$\exists E' : \text{term } T. \exists R^l : E^l \stackrel{1}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{1}{\Longrightarrow} E'. \top$$

In its proof, the first proof operation we performed was a case analysis on D^l. The first case to be considered is that D^l is instantiated with a global parameter u. Since $\underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$, for $\underline{x} : \text{term } T$, declared by the same variable block, clearly E and E^l must equal x. The same holds for the next case that D^l is instantiated "pbeta D_1^l D_2^l". Because of dependencies, this means that E must have been instantiated with "app (lam E_1) E_2" for an E_1 of type term $T_1 \to$ term T for some type T_1. And E' must be the result of applying some E_1' (an LF-function) to some E_2'

(of appropriate type). While abstractly describing the case analysis, we do not know exactly what E_1' and E_2' are instantiated with, we only know that they must exist.

Formally, all universal assumptions in the examples are represented by a generalized context Ψ. Subsequently, each case analysis of Ψ leads to a new generalized context Ψ'. As example, consider the reflexivity Lemma 4.3.

Example 5.12 (Case analysis in reflexivity Lemma 4.3:) The form of the generalized context representing all universally quantified assumptions right *before* case analysis is

$$\Psi = T : \mathrm{tp}, E : \mathrm{term}\ T$$

and the form of the generalized context in each of the cases right *after* case analysis are

Case: $\Psi_1' = T : \mathrm{tp}, (\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L$

Case: $\Psi_2' = T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E' : \mathrm{term}\ T_1 \to \mathrm{term}\ T_2$

Case: $\Psi_3' = T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ (T_2\ \mathrm{arrow}\ T_1), E_2 : \mathrm{term}\ T_2$

As second example, consider the diamond Lemma 4.6.

Example 5.13 (Case analysis in diamond Lemma 4.6) The form of the generalized context representing all universally quantified assumptions right *before* the first case analysis is

$$\Psi = T : \mathrm{tp}, E : \mathrm{term}\ T, E^l : \mathrm{term}\ T, E^r : \mathrm{term}\ T, D^l : E \overset{1}{\Longrightarrow} E^l, D^r : E \overset{1}{\Longrightarrow} E^r$$

and the form of the generalized context in each of the cases right *after* case analysis on D^l are

Case: $\Psi_1' = T : \mathrm{tp}, (\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L, E^r : \mathrm{term}\ T, D^r : x \overset{1}{\Longrightarrow} E^r$

Case: $\Psi_2' = T : \mathrm{tp}, T_1 : \mathrm{tp}, E_1 : \mathrm{term}\ T_1 \to \mathrm{term}\ T, E_2 : \mathrm{term}\ T_1, E_1^l : \mathrm{term}\ T_1 \to \mathrm{term}\ T, E_2^l\ \mathrm{term}\ T_1, E^r : \mathrm{term}\ T, D_1^l : \Pi x : \mathrm{term}\ T_1.\, x \overset{1}{\Longrightarrow} x \to E_1\ x \overset{1}{\Longrightarrow} E_1^l\ x, D_2^l : E_2 \overset{1}{\Longrightarrow} E_2^l, D^r : (\mathrm{app}\ (\mathrm{lam}\ E_1)\ E_2) \overset{1}{\Longrightarrow} E^r$

Case: $\Psi_3' = T : \mathrm{tp}, T' : \mathrm{tp}, E : \mathrm{term}\ T \to \mathrm{term}\ T', E^l : \mathrm{term}\ T \to \mathrm{term}\ T', E^r : \mathrm{term}\ (T\ \mathrm{arrow}\ T'), D^l : \Pi x : \mathrm{term}\ T.\, x \overset{1}{\Longrightarrow} x \to E\ x \overset{1}{\Longrightarrow} E^l\ x, D^r : (\mathrm{lam}\ E) \overset{1}{\Longrightarrow} E^r$

Case: $\Psi_4' = T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1, E_2 : \mathrm{term}\ T_2, E_1^l : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1, E_2^l : \mathrm{term}\ T_2, E^r : \mathrm{term}\ T_1, D_1^l : E_1 \overset{1}{\Longrightarrow} E_1^l, D_2^l : E_2 \overset{1}{\Longrightarrow} E_2^l, D^r : \mathrm{app}\ E_1\ E_2 \overset{1}{\Longrightarrow} E^r$

Again, all assumptions in $\Psi_1', \ldots, \Psi_4'$ represent exactly the available assumptions after case analysis of the first parallel reduction D^l, as implicitly assumed in the informal presentation of the proof of Lemma 3.7. There the situation is slightly different, because we performed two case analysis at once, whereas here, we only present the one over D^l.

It is one of the major technical contributions of this thesis of how to design the case-distinction operator in order to capture this refinement of generalized contexts. Our solution employs *generalized substitutions* (defined for generalized contexts) whose definition we address in the following.

A generalized substitution is defined very similarly to LF level substitutions in Section 5.2. The main difference is, that its domain and its co-domain are generalized contexts, and therefore a special case for substituting context variables must be provided; only variable blocks can be substituted for variable blocks. Generalized substitutions are denoted by ψ.

$$\textit{Generalized substitutions:} \quad \psi \quad ::= \quad \cdot \mid \psi, M/x \mid \psi, \rho'/\rho$$

The composition of generalized substitutions is very similar to Definition 5.1 with one extra case for variable blocks.

Definition 5.14 (Composition of generalized substitutions)

$$\begin{array}{rcl} \cdot \circ \psi_2 & = & \psi_2 \\ (\psi_1, M/x) \circ \psi_2 & = & (\psi_1 \circ \psi_2), M[\psi_2]/x \\ (\psi_1, \rho'/\rho) \circ \psi_2 & = & (\psi_1 \circ \psi_2), [\psi_2]\rho'/\rho \end{array}$$

where we write $[\psi]\rho$ to apply a generalized substitution to a variable block, or more precise to the types declared within. The prefix notation indicates that the substitution is applied to a list of entities. It is an abbreviation for

$$\begin{array}{rcl} [\psi]\cdot & = & \cdot \\ [\psi](\underline{x} : A, \rho) & = & \underline{x} : A[\rho], [\psi, x/x]\rho \end{array}$$

Note that if $\rho = \underline{x_1} : A_1, \ldots, \underline{x_n} : A_n$ and $\rho' = \underline{y_1} : B_1, \ldots \underline{y_n} : B_n$, then ρ/ρ' is a substitution which substitutes $\underline{x_i}$ for $\underline{y_i}$ for all $i \leq n$. Specifically, $[\psi_2]\rho'$ updates only the type information in ρ' but leaves the variable names in ρ' untouched. In our shorthand notation $[\psi_2]\rho'/\rho$ denotes exactly the same generalized substitution as ρ'/ρ does, but the co-domain may be different.

Returning to the Example 5.12 of the reflexivity Lemma 4.3, there are three generalized substitutions ψ_1, ψ_2, ψ_3, one associated with each case:

Example 5.15 (Generalized substitutions and the reflexivity Lemma 4.3)

Case: The first case of the proof translates into $\psi_1 = T/T, x/E$, where Ψ is its domain and Ψ_1' its co-domain. The x/E in the substitution corresponds to the $\underline{x}$ in the informal presentation in Lemma 3.4, where x is declared of the variable block $\rho = \underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$. In this special case x is a binding occurrence of a parameter block. There can also be non-binding occurrences of variable blocks, which we encounter in the example about the diamond lemma below.

Case: The second case is also expressed by a simple substitution relating Ψ_2' to Ψ: $\psi_2 = (T_1 \text{ arrow } T_2)/T, (\text{lam } E')/E$

Case: And so is the third case: The domain of Ψ_3' is Ψ, and Ψ_3' is its co-domain. $\psi_3 = T_1/T, (\text{app } E_1\ E_2)/E$

The difference between binding and none-binding occurrences of variable blocks in a context is illustrated by the proof of the diamond Lemma 4.6. We put special emphasis on the first case:

Example 5.16 (Generalized substitution and the diamond Lemma 4.6:)
This example extends Lemma 5.13

Case: $\Psi_1' = T : \text{tp}, (\underline{x} : \text{term } T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L, E^r : \text{term } T, D^r : x \overset{1}{\Longrightarrow} E^r$:

The first case translates into the generalized substitution

$$\psi_1 = T/T, \underline{x}/E, \underline{x}/E^l, \underline{u}/D^l, D^r/D^r$$

The variable block $(\underline{x} : \text{term } T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L$ is a binding occurrence because, informally, when the case is executed, the instantiation of E, E^l, and D^l determine the parameter block in the context. In the original proof of the diamond Lemma 3.7, we discussed how to assume the existence of a second parameter block in order to distinguish cases over D^r. Because of typing constraints, the two variable blocks are constrained to be identical because as the left reduction, the right reduction starts in $\underline{x}$. In our system there are two options to express the second case analysis:

1. Define a second case analysis which is defined inside the scope of the first, with a new domain Ψ_1'' and a substitution ψ_1'

$$\begin{aligned} \Psi_1'' &= T : \text{tp}, (\underline{y} : \text{term } T, \underline{v} : \underline{y} \overset{1}{\Longrightarrow} \underline{y})^L \\ \psi_1' &= T/T, (\underline{y} : \text{term } T, \underline{v} : \underline{y} \overset{1}{\Longrightarrow} \underline{y})/(\underline{x} : \text{term } T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x}), \underline{y}/E^r, \underline{v}/D^r \end{aligned}$$

   The variable block $(\underline{y} : \text{term } T, \underline{v} : \underline{y} \overset{1}{\Longrightarrow} \underline{y})^L$ is a non-binding occurrence of a variable block. It is merely a renaming of $\underline{x}$ and $\underline{u}$ to $\underline{y}$ and $\underline{v}$.

2. Modify the generalized context Ψ_1'' and the generalized substitution ψ_1'' to also accommodate the second case analysis.

$$\begin{aligned} \Psi_1'' &= T : \text{tp}, (\underline{x} : \text{term } T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L \\ \psi_1'' &= T/T, \underline{x}/E^l, \underline{x}/E^r, \underline{u}/D^l, \underline{u}/D^r \end{aligned}$$

   and again is $(\underline{x} : \text{term } T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L$ a binding occurrence of a variable block.

It is possible to use either of these two representations, and the attentive reader might have noticed that ψ_1'' is nothing else but a composition of ψ_1 and ψ_1'.

Case: $\Psi_2' = T : \text{tp}, T_1 : \text{tp}, E_1 : \text{term } T_1 \to \text{term } T, E_2 : \text{term } T_1, E_1^l : \text{term } T_1 \to \text{term } T, E_2^l \text{ term } T_1, E^r : \text{term } T, D_1^l : \Pi x : \text{term } T_1. x \overset{1}{\Longrightarrow} x \to E_1\ x \overset{1}{\Longrightarrow} E_1^l\ x, D_2^l : E_2 \overset{1}{\Longrightarrow} E_2^l, D^r : (\text{app (lam } E_1)\ E_2) \overset{1}{\Longrightarrow} E^r$:

The substitution which expresses the relationship between Ψ and Ψ_2' results from a straightforward instantiation of assumption in Ψ:

$$\psi_2 = T/T, (\text{app (lam } E_1)\ E_2)/E, (E_1\ E_2)/E^l, E^r/E^r, (\text{pbeta } D_1^l\ D_2^l)/D^l, D^r/D^r$$

Case: $\Psi_3' = T : \text{tp}, T' : \text{tp}, E : \text{term } T \to \text{term } T', E^l : \text{term } T \to \text{term } T', E^r : \text{term } (T \text{ arrow } T'), D^l : \Pi x : \text{term } T. x \overset{1}{\Longrightarrow} x \to E\ x \overset{1}{\Longrightarrow} E^l\ x, D^r : (\text{lam } E) \overset{1}{\Longrightarrow} E^r$:

Analogously, the relationship between Ψ and Ψ_3' is expressed by the generalized substitution ψ_3.

$$\psi_3 = (T_1 \text{ arrow } T_2)/T, (\text{lam } E)/E, (\text{lam } E^l)/E^l, E^r/E^r, (\text{plam } D^l)/D^l, D^r/D^r$$

Case: $\Psi'_4 = T_1 : \text{tp}, T_2 : \text{tp}, E_1 : \text{term } T_2 \to \text{term } T_1, E_2 : \text{term } T_2, E^l_1 : \text{term } T_2 \to \text{term } T_1, E^l_2 : \text{term } T_2, E^r : \text{term } T_1, D^l_1 : E_1 \stackrel{1}{\Longrightarrow} E^l_1, D^l_2 : E_2 \stackrel{1}{\Longrightarrow} E^l_2, D^r : \text{app } E_1\ E_2 \stackrel{1}{\Longrightarrow} E^r$:

And finally, the relationship between Ψ and Ψ'_4 is expressed by the generalized substitution ψ_4.

$$\psi_4 = T/T, (\text{app } E_1\ E_2)/E, (\text{app } E^l_1\ E^l_2)/E^l, E^r/E^r, (\text{papp } D^l_1\ D^l_2)/D^l, D^r/D^r$$

These two example clearly demonstrate the general idea behind the design of the case construct. Each case is expressed by a substitution and its co-domain. The domain of the substitution is the context in which the original case-expression is valid (it therefore stays invariant for all the cases) , and the co-domain of the substitution is the context in which the body of a case is valid.

The subject of the case construct is hence not simply one LF object, instead it is a list of LF objects (a substitution) that instantiates all variables declared in the context simultaneously. In summary, we use the basic idea of explicit substitutions [DHKP96] to encode the case subject. An explicit substitution is a substitution which is turned into a first-class object of the calculus. We use such an explicit generalized substitutions in order to represent the case subject, that is ψ.

In order to make this presentation more uniform, we also use explicit meta-substitutions to capture the instantiation of meta-variables. These observations give rise to a new proof term, which is defined in terms of a list of cases.

$$\begin{array}{llll} \textit{Proof Terms} & P & ::= & \ldots \mid \text{case } (\psi; \delta) \text{ of } \Omega \\ \textit{Cases} & \Omega & ::= & \cdot \mid \Omega, (\Psi \triangleright \psi \mapsto P) \end{array}$$

The $(\psi; \delta)$ part of the new proof term is a pair of already discussed explicit substitutions, and Ω is a list of cases. Each case describes the substitution ψ in order to recognize if a case is applicable, its co-domain which describes all assumption and block variables available to the body of the case, and finally the body P of the case itself.

Operationally speaking, assume that at the time of execution case $(\psi; \delta)$ of Ω is given and is the term is closed (it doesn't contain any free variables). Consequently, ψ is ground substitution. A case $(\Psi' \triangleright \psi' \mapsto P) \in \Omega$ is applicable, if the system can construct a closed substitution ψ'' (the new environment) with domain Ψ' from ψ (the old environment), such that $\psi' \circ \psi'' = \psi$. If such a ψ'' exists, informally, the case is applicable, and the body P of the case can be executed after all variables from Ψ' have been replaced according to ψ''.

All proof terms are now defined and we can return to the reflexivity lemma 4.3, and illustrate its proof term. Proof terms in their internal formulation are very verbose, difficult to parse and painfully hard to interpret. We therefore opt to illustrate the internal version only once in the next example, and use the more familiar notation of proof terms (from Chapter 4) in the remainder of this thesis. In addition, this notation is easily definable as syntactic sugar.

Example 5.17 (Proof of the reflexivity Lemma 4.3) As derived by syntactic refinement in Section 4.2.2, the proof of the reflexivity lemma

$$\begin{array}{l} \Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x. \\ \qquad \forall T : \text{tp}. \forall E : \text{term } T. \exists D : E \stackrel{1}{\Longrightarrow} E. \top \end{array}$$

is a recursion. To the left is the version we have already seen, and to the right is the internal representation, where we omit some type and formula annotations.

$$
\begin{array}{l}
\mathbf{fun\ refl}\ \underline{x} = \underline{u} \\
\quad |\ \mathbf{refl}\ (\text{lam}\ (\lambda x : \text{term}\ T.\, E'\ x)) = \\
\qquad \mathbf{let} \\
\qquad\quad \mathbf{new}\ \underline{x} : \text{term}\ T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \mathbf{val}\ \ P\ \underline{x}\ \underline{u} = \mathbf{refl}\ (E'\ \underline{x}) \\
\qquad \mathbf{in} \\
\qquad\quad \text{plam}\ (\lambda x.\, \lambda u.\, P\ x\ u) \\
\qquad \mathbf{end} \\
\quad |\ \mathbf{refl}\ (\text{app}\ E_1\ E_2) = \\
\qquad \mathbf{let} \\
\qquad\quad \mathbf{val}\ P_1 = \mathbf{refl}\ E_1 \\
\qquad\quad \mathbf{val}\ P_2 = \mathbf{refl}\ E_2 \\
\qquad \mathbf{in} \\
\qquad\quad \text{papp}\ P_1\ P_2 \\
\qquad \mathbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{box}\ (\text{SOME}\ T : \text{tp}.\ \text{BLOCK}\ x : \text{term}\ T, u : x \stackrel{1}{\Longrightarrow} x)^L. \\
\mu\mathbf{refl}.\, \Lambda T : \text{tp}.\, \Lambda E : \text{term}\ T. \\
\text{case}\ (T/T, E/E; \mathbf{refl}/\mathbf{refl})\ \text{of} \\
\quad (T : \text{tp}, (\underline{x} : \text{term}\ T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L \triangleright T/T, x/E \\
\quad \mapsto \langle u, \langle\rangle\rangle), \\
\quad (T_1 : \text{tp}, T_2 : \text{tp}, E' : \text{term}\ T_1 \to \text{term}\ T_2 \\
\quad \triangleright (T_1\ \text{arrow}\ T_2)/T, (\text{lam}\ E')/E \\
\quad \mapsto \mathbf{let}\ \nu\ (\underline{x} : \text{term}\ T_1, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L. \\
\qquad\quad \mathbf{x}_1 = \mathbf{refl}\ T_2, \mathbf{x}_2 = \mathbf{x}_1\ (E'\ \underline{x}), \langle P, \mathbf{x}_3\rangle = \mathbf{x}_2 \\
\qquad \mathbf{in}\ \langle \text{plam}\ (\lambda x : \text{term}\ T_1.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, P\ x\ u), \langle\rangle\rangle), \\
\quad (T_1 : \text{tp}, T_2 : \text{tp}, E_1 : \text{term}\ (T_2\ \text{arrow}\ T_1), E_2 : \text{term}\ T_2 \\
\quad \triangleright T_1/T, (\text{app}\ E_1\ E_2)/E \\
\quad \mapsto \mathbf{let} \\
\qquad\quad \mathbf{x}_1 = \mathbf{refl}\ (T_2\ \text{arrow}\ T_1), \mathbf{x}_2 = \mathbf{x}_1\ E_1, \langle P_1, \mathbf{x}_3\rangle = \mathbf{x}_2, \\
\qquad\quad \mathbf{y}_1 = \mathbf{refl}\ T_2, \mathbf{y}_2 = \mathbf{y}_1\ E_2, \langle P_2, \mathbf{y}_3\rangle = \mathbf{y}_2 \\
\qquad \mathbf{in}\ \langle \text{papp}\ P_1\ P_2, \langle\rangle\rangle)
\end{array}
$$

Similar to LF-level substitutions from Section 5.2, generalized and meta-substitutions must be well-formed — we establish this property by two judgments. Generalized substitutions map generalized contexts into generalized contexts but generalized contexts themselves are already a prerequisite for meta-contexts. Consequently the definition of meta-substitutions relies on generalized substitutions.

The first of the two judgments is that of well-formed generalized substitutions, $\Psi' \vdash \psi \in \Psi$. Clearly, Ψ is the domain of the substitution and Ψ' is its co-domain.

Judgment:

$$\text{Well-typed generalized substitutions} \quad \Psi' \vdash \psi \in \Psi$$

The semantics of this judgment is defined by three inference rules. The empty generalized substitution is well-formed with an empty domain. Recall, that variable blocks are used to express the presence of a parameter block in the parameter context. Consequently, the image of a variable block must be a variable block. Finally there is the expected rule which allows a substitution of any well-typed LF-term for an assumption variable.

Rules:

$$\frac{\vdash \Psi'\ \text{abstract}}{\Psi' \vdash \cdot \in \cdot}\ \textsf{sempty}$$

$$\frac{\rho'^L \in \Psi \qquad \Psi' \vdash \rho' \equiv_\alpha [\psi]\rho \qquad \Psi' \vdash \psi \in \Psi}{\Psi' \vdash (\psi, \rho'/\rho) \in \Psi, \rho^L}\ \textsf{sblock} \qquad \frac{\Psi' \vdash M : A[\psi] \qquad \Psi' \vdash \psi \in \Psi}{\Psi' \vdash (\psi, M/x) \in \Psi, x : A}\ \textsf{sass}$$

Generalized substitution composition is well-defined.

Lemma 5.18 (Composition of generalized substitutions)

If $\mathcal{D}_1 :: \Psi_2 \vdash \psi_1 : \Psi_1$
and $\mathcal{D}_2 :: \Psi_3 \vdash \psi_2 : \Psi_2$
then $\Psi_3 \vdash \psi_1 \circ \psi_2 : \Psi_1$

Proof: by structural induction on $\mathcal{D}_1$. □

Generalized substitutions are a prerequisite for the definition of well-defined meta-substitutions. As a reminder, a meta-substitution replaces meta-variables by entire proof terms, as it is for example necessary when evaluating the fixed point-operator. Substitutions on meta-variables are used very often in the remainder of this thesis, for example in the substitution Lemma 6.20 which we prove in the next chapter. The judgment, expressing that a meta-substitution is well-formed, is in principle just an extension of the previous judgment.

Judgment:

$$\text{Well-typed meta-substitutions} \quad \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$$

In the spirit of extending the first judgment, the semantics of well-typed meta-substitutions is defined by two inference rules. The first rule coerces a standard well-formed generalized substitution to be a well-formed meta-substitution in the base case. The other expresses when non-trivial meta-substitutions are well-formed.

Rules:

$$\frac{\Psi' \vdash \psi \in \Psi}{\Psi'; \Delta' \vdash \psi; \cdot \in \Psi; \cdot}\,\mathsf{sabstract} \qquad \frac{\Psi'; \Delta' \vdash P \in F[\psi] \quad \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta}{\Psi'; \Delta' \vdash \psi; \delta, P/\mathbf{x} \in \Psi; \Delta, \mathbf{x} \in F}\,\mathsf{smeta}$$

In order to be perfectly precise, a precondition for the two judgments is, that all involved contexts are well-formed. That means that for the first judgment we can assume that $\vdash \Psi$ abstract and $\vdash \Psi'$ abstract, and for the second, we assume that $\Psi \vdash \Delta$ meta and $\Psi' \vdash \Delta'$ meta.

Meta-substitutions can be composed and we write $(\psi; \delta) \circ (\psi'; \delta') = (\psi''; \delta'')$ for the resulting substitution. It is defined in a straightforward way, where we assume the that meta-substitutions can be applied to a proof term $P[\psi; \delta]$.

Definition 5.19 (Composition of meta-substitutions)

$$\begin{array}{rcll} (\psi; \cdot) \circ (\psi'; \delta') & = & (\psi \circ \psi'; \cdot) & (\mathsf{cempty}) \\ (\psi; \delta, P/\mathbf{x}) \circ (\psi'; \delta') & = & (\psi''; \delta'', P[\psi'; \delta']/\mathbf{x}) & (\mathsf{cmeta}) \\ & & \textit{where } (\psi; \delta) \circ (\psi'; \delta') = (\psi''; \delta'') & \end{array}$$

Meta-substitution composition is well-defined, too. Since its proof relies on a substitution lemma for applying meta-substitution to programs, we postpone the proof this lemma until Section 6.2.4, Corollary 6.21.

Lemma 5.20 (Composition of meta-substitutions)

If $\mathcal{D}_1 :: \Psi_2; \Delta_2 \vdash \psi_1; \delta_1 \in \Psi_1; \Delta_1$
and $\mathcal{D}_2 :: \Psi_3; \Delta_3 \vdash \psi_2; \delta_2 \in \Psi_2; \Delta_2$
then $\Psi_3; \Delta_3 \vdash (\psi_1; \delta_1) \circ (\psi_2; \delta_2) \in \Psi_1; \Delta_1$

The special character of a meta-substitution to extend a generalized substitution is clearly exhibited by the observation that the underlying generalized substitution can easily be extracted form the meta-substitution.

Lemma 5.21 *If* $\mathcal{D} :: \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$
then $\Psi' \vdash \psi \in \Psi$.

Proof: by induction on $\mathcal{D}$. □

In many proofs below, we will encounter *identity substitutions*, i.e. substitutions whose domain and co-domain are equal, and every variable is mapped to itself. If an identity substitutions acts on an LF context Γ, we write id_Γ. Likewise a generalized identity substitution on Ψ is written as id_Ψ, and a meta identity substitutions on Δ as id_Δ. In the remainder of this thesis we take the freedom to simply omit identity substitutions from the formalism, if it does not contribute to the presentation of the material. For example the instead of $\psi; \mathrm{id}_\Delta$ we simply write ψ.

We continue this rather technical discussion, and present now the final extension to the inference rule system of $\mathcal{M}_2^+$. The rules will capture the essence of case analysis in order to define and formalize recursive functions in $\mathcal{M}_2^+$. Recall, from Example 5.17 that there is the case construct itself which takes as argument a list of cases Ω which must also be well-formed. Obviously, Ω's well-formedness requires a judgment by itself: $\Psi; \Delta \vdash \Omega \in F$, where F is the formula, and each case in Ω must be valid. Typically, F is an existentially quantified formula, such as $\exists D : E \overset{1}{\Longrightarrow} E. \top$ in the reflexivity lemma, or $\exists E' : \mathrm{term}\ T. \exists P^l : E^l \overset{1}{\Longrightarrow} E'. \exists P^r : E^r \overset{1}{\Longrightarrow} E'. \top$ in the diamond lemma.

Judgment

$$\text{Well-formed case lists:} \quad \Psi; \Delta \vdash \Omega \in F$$

The typing rule for case requires, that the case subject is a valid meta-substitution and that all cases are well-typed.

Rules

$$\frac{\Psi; \Delta \vdash \psi; \delta \in \Psi'; \Delta' \qquad \Psi'; \Delta' \vdash \Omega \in F}{\Psi; \Delta \vdash \mathrm{case}\ (\psi; \delta)\ \mathrm{of}\ \Omega \in F[\psi]}\ \mathsf{case}$$

Cases are well-formed, if each of the substitutions is well-formed with the associated generalized context as its co-domain. In addition, the proof term associated with each case must be well-formed in the same generalized context. There is a generalized substitution ψ that express how a case is being refined when it is successfully applied. The well-formedness proof can (and in most cases will) use the meta-assumptions given in Δ, but because case analysis might have distinguished cases over other variables which occur free in the formulas in Δ, the refinement must be reflected, written as $[\psi]\Delta$ in alt's premiss. Likewise, ψ must be applied to the formula in premiss of alt below. The precise definition of substitution application together with the associated properties are postponed until the next Chapter.

$$\frac{}{\Psi; \Delta \vdash \cdot \in F}\ \mathsf{base}$$

$$\frac{\Psi' \vdash \psi \in \Psi \qquad \Psi; \Delta \vdash \Omega \in F \qquad \Psi'; [\psi]\Delta \vdash P \in F[\psi]}{\Psi; \Delta \vdash \Omega, (\Psi' \triangleright \psi \mapsto P) \in F}\ \mathsf{alt}$$

This almost completes the presentation of the typing rules for case analysis. The only thing missing, is that the proof terms which are formalized in this system are realizers, which means that the case rule must guarantee that all cases are always covered, a property which is also referred to as *coverage* [Roh96]. Similarly to the side condition for termination 5.1, we endow the case rule with a side condition which enforces coverage.

Informally, the coverage condition guarantees that if the recursive function (the proof term) is executed in an environment possibly defined in a concrete parameter context, and all assumption and variable blocks in the generalized context are instantiated with LF objects and parameter blocks, then the case analysis can be successfully executed, and at least one case applies. Consider the following situation. We are presented with a well-typed term

$$\text{lam } (\lambda x : \text{term nat}.\, x) : \text{term (nat arrow nat)},$$

where we assume that nat : tp is a base type for natural numbers. The objective is to construct a term of type "lam $(\lambda x : \text{term nat}.\, x) \stackrel{1}{\Longrightarrow}$ lam $(\lambda x : \text{term nat}.\, x)$". This can be easily established by employing the recursive function **refl**, and applying it to the argument "lam λx : term nat. x". Once the evaluation reaches the point of case analysis, there is a case which applies: it is the second in Example 5.17.

But in general this is not necessarily the case. The rules defining the well-formedness of cases do not imply that a case is guaranteed to be applicable. Of course, this observation is not new. The same observation holds for any functional programming language, as for example ML [MTHM97] or Haskell [Tho99, Hud00] which employs pattern matching; in a situation where no case is applicable an exception is raised.

This solution is unacceptable for our situation. We must enforce that all recursive functions are realizers, that is evaluation must always make progress and eventually terminate. Termination is already informally guaranteed by side condition (5.1). It remains to guarantee that the evaluation of each recursive function makes progress under all circumstances.

In the quest for coverage, we first examine what it means for a rule to be applicable. At the interesting point in the evaluation, shortly before cases are analyzed, there exists a generalized substitution, (or better environments as they are called in functional programming languages) which has the following form:

$$\cdot \vdash \underbrace{((\text{nat arrow nat})/T, \text{lam } (\lambda x : \text{term nat}.\, x)/E)}_{\eta} \in (T : \text{tp}, E : \text{term } T)$$

Recall that the applicable case has the form

$$(T_1 : \text{tp}, T_2 : \text{tp}, E' : \text{term } T_1 \to \text{term } T_2) \triangleright \underbrace{((T_1 \text{ arrow } T_2)/T, (\text{lam } E')/E)}_{\psi} \mapsto \ldots$$

In detail, the rule is applicable, because the environment η is *decomposable* into a new environment, call it η', and ψ. We do not show how to calculate this new environment η' from η and ψ, this is left to the next chapter. Instead we simply state the result:

$$\cdot \vdash \underbrace{(\mathrm{nat}/T_1, \mathrm{nat}/T_2, (\lambda x : \mathrm{term\ nat}.\, x)/E')}_{\eta'} \in (T : \mathrm{tp}, E : \mathrm{term}\ T)$$

One can easily see, that η' is the right choice of environment since the composition of ψ and η' inevitably yields η:

$$\begin{array}{l}((T_1 \text{ arrow } T_2)/T, (\mathrm{lam}\ E')/E) \circ (\mathrm{nat}/T_1, \mathrm{nat}/T_2, (\lambda x : \mathrm{term\ nat}.\, x)/E') = \\ \quad ((\mathrm{nat\ arrow\ nat})/T, \mathrm{lam}\ (\lambda x : \mathrm{term\ nat}.\, x)/E)\end{array}$$

More formally, we say that a list of cases Ω covers all cases, if *any* environment η can be decomposed into η' and ψ for *some* case $(\Psi' \triangleright \psi \mapsto P) \in \Omega$.

$$\Omega \text{ is a complete case cover} \tag{5.2}$$

This side condition is associated with the case rule. The general problem of coverage is undecidable, but in Section 7.3, we will give a formal but sufficient criterion for coverage. It is semantic in nature; there is no feasible way to try every instantiation of ψ a priori. Semantic conditions are in general impossible to enforce directly. Therefore we present in Section 7.3 a syntactic criterion on Ω, which — when satisfied — guarantees complete case coverage. As we will see, the entire construction rests on the shoulders of the canonical form Theorem 2.6 for LF.

Side condition (5.2) enforces a condition on the refinement substitution ψ which are part of a case $(\Psi' \triangleright \psi \mapsto P) \in \Omega$. The rule alt guarantees that the substitution is well-typed: $\Psi' \vdash \psi \in \Psi$. The following example shows that Ψ' should not be unnecessarily large. If it were, an oracle would be necessary to assign an operational semantics to our proof term calculus. Consider the slightly extended lam-case of the reflexivity lemma (by adding $Q : \mathrm{term}\ T_1 \to\ \mathrm{term}\ T_1$).

$$\begin{array}{l}T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E' : \mathrm{term}\ T_1 \to \mathrm{term}\ T_2, Q : \mathrm{term}\ T_1 \to\ \mathrm{term}\ T_1 \\ \triangleright (T_1 \text{ arrow } T_2)/T, (\mathrm{lam}\ E')/E\ \mapsto \ldots\end{array}$$

After applying the decomposition rule to η above using ψ we arrive at an extension of η' which also instantiates Q. The value of Q cannot be determined from η itself since Q is not mentioned in any of the LF-objects used in η. It is hence entirely under-constrained, which gives rise to a possible non-deterministic choice: We simply choose $\lambda x : \mathrm{term}\ T_1.\, x$ for Q and complete the decomposition.

$$\cdot \vdash \mathrm{nat}/T_1, \mathrm{nat}/T_2, (\lambda x : \mathrm{term\ nat}.\, x)/E', (\lambda x : \mathrm{term}\ T_1.\, x)/Q \in T : \mathrm{tp}, E : \mathrm{term}\ T$$

The strange behavior associated with allowing unconstrained assumptions in Ψ' is even more clearly illustrated by the following extension of Ψ'; adding $Q : \mathrm{term}\ T_2 \to\ \mathrm{term}\ T_1$

$$\begin{array}{l}T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E' : \mathrm{term}\ T_1 \to \mathrm{term}\ T_2, Q : \mathrm{term}\ T_2 \to\ \mathrm{term}\ T_1 \\ \triangleright (T_1 \text{ arrow } T_2)/T, (\mathrm{lam}\ E')/E\ \mapsto \ldots\end{array}$$

renders the case inapplicable because there is no possible instantiation for Q. Such non-determinism therefore blurs the interpretation of proof terms as recursive functions. In general, we require that each η can be decomposed into a ψ and some η'. Moreover, we require that this decomposition is unique. Note, that η and η' need not to be closed; as usual they might be open in some parameter context, expressed abstractly by Φ. All these requirements are summarized by the following side condition which we associate with alt-rule.

$$\text{For all } \eta\ (\Phi \vdash \eta \in \Psi) \text{ there exists an } \eta'\ (\Phi \vdash \eta' \in \Psi') \text{ s.t. } \eta = \psi \circ \eta' \tag{5.3}$$

Similar to side condition (5.2) it is semantic, probably undecidable, but we present a sufficient syntactic criteria in Section 6.4.

This concludes our presentation of two new meta-level proof terms expressing well-founded recursion and complete case analysis which turn as we will see in Chapter 7 the proof term calculus of $\mathcal{M}_2^+$ into a calculus of total recursive functions, warranting the soundness of $\mathcal{M}_2^+$. We have established three semantic side conditions for the rules for which we present precise syntactic criteria in the chapters to follow. We conclude this chapter with a discussion of how to add lemma application to $\mathcal{M}_2^+$.

5.7 Lemmas

Theory and proof development without lemmas is unthinkable. Meta-logical arguments always consist of a sequence of lemmas as for example the development of the Church-Rosser theorem presented in Chapter 4. Using an auxiliary notion of parallel reduction, the proof of the Church-Rosser property of ordinary reduction is reduced to the Church-Rosser property for parallel reduction each of which is derived by a series of lemmas. In the discussion so far, we have presented all techniques necessary to formalize proofs which do not rely on other lemmas, the basic building blocks of a formal theory so to speak. We generalize this idea in this section by adding the ability to apply other lemmas to our system. With this technology at hand, we can formalize all lemmas and theorems from Chapter 4.

The reader may wish to skip this section in the first reading. If all lemmas in the development of a theory depend on *one but fixed world extension* this section does not contain any new ideas. In such a situation lemmas can simply be added as meta-assumptions to Δ. If on the other hand, the lemmas necessary for a development require *many possibly different world extensions*, the mechanism presented in this section apply.

This section is structured as follows. First we introduce the necessary basic definitions of lemmas in Section 5.7.1. As presented in Section 4.2.2, lemmas and theorems also take the shape of the parameter context into account. A criteria which expresses if one lemma can call another without violating the context schema restriction is presented in Section 5.7.2. In Section 5.7.3 we finally present the new proof rules extending the proof term calculus of $\mathcal{M}_2^+$.

5.7.1 Preliminaries

Lemmas are a very valuable and an important organizing force in the development of theories. Typically theories are built as hierarchies of lemmas. If well-chosen, this hierarchy can support the automated validation of changes to the underlying definition of a formal system. For example in Section 4.2.3, when we extend the simply-typed λ-calculus by polymorphism, all lemmas for the Church-Rosser theorem are still true (with a very minor modifications in the definition of context schemas, by adding a block schema for type variables).

What are lemmas? Lemmas are general formulas, i.e. they define a context schema and a formula whose proof possibly relies on *meta-hypotheses*, i.e. proofs of other lemmas which are assumed to be true. For example we can prove confluence Lemma 4.8 under the assumption that the strip Lemma 4.7 is true. Likewise, the proof of the strip lemma relies on the truth of the diamond Lemma 4.6, which itself depends on the truth of the substitution Lemma 4.5. It should be clear, that a general formula is only proven, if all of its meta-hypothesis are instantiated by real proofs. Formally, we first extend the notion of general proof term to allow meta-hypothesis.

$$\textit{General proof terms:}\quad Q \quad ::= \quad \ldots \mid \mathbf{x}$$

Meta-hypotheses are organized in form of a *lemma repository* which is very closely related to the list of meta-assumptions Δ and the instantiation of meta-hypothesis is described a substitution like structure, called a lemma instantiation.

$$\begin{array}{lll} \textit{Lemma repository:} & \Xi & ::= \quad \cdot \mid \Xi, \mathbf{x} \in G \\ \textit{Lemma instantiation:} & \xi & ::= \quad \cdot \mid \xi, Q/\mathbf{x} \end{array}$$

In addition, each judgment of the formal proof system $\mathcal{M}_2^+$ is being equipped with such a lemma repository. There are three such judgments expressing the provability of general formulas, formulas, and declarations.

Judgments

$$\begin{array}{ll} \textit{Provability of general formulas:} & \Xi \vdash Q \in G \\ \textit{Provability of formulas:} & \Psi; \Delta; \Xi \vdash P \in F \\ \textit{Provability of declarations:} & \Psi; \Delta; \Xi \vdash D \in \Psi'; \Delta' \end{array}$$

Ξ is not going to change during the proof of a meta-theorem. It only changes when meta-hypothesis are instantiated. Therefore, a meta-theorem G is proven if its proof is closed, that is formally, if there exists a proof Q such that $\cdot \vdash Q \in G$.

5.7.2 Context Schema Subsumption

One of the main characteristics of a lemma is the form of the world extension for which it is defined. World extensions are described by the context schema. The need of context schemas has been motivated and discussed in Section 4.2.2 in great detail. In particular, context schemas are necessary in order to express properties of deductions which are not necessarily closed. The diamond Lemma 4.6 for example contains the declaration of the context schema

$$(\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x)^L$$

which serves as a quantifier over all regularly formed parameter contexts of the form:

$$(\underline{x_1} : \text{term } T_1, \underline{u_1} : \underline{x_1} \stackrel{1}{\Longrightarrow} \underline{x_1})^L, \ldots (\underline{x_n} : \text{term} T_n, \underline{u_n} : \underline{x_n} \stackrel{1}{\Longrightarrow} \underline{x_n})^L$$

The question, if the proof of the diamond lemma can use to the transitivity Lemma 4.4. Surely, if the transitivity lemma is proven for the same world extension as required by the diamond lemma, the application is sound. If it isn't it may not be sound. Which lemmas can be applied from within a meta-proof and which can not is determined by a relation between the context schema of the lemma to be proven to the context schema of the lemma to be applied which we call *subsumption relation.*

More abstractly, if a formula is to be proven for any parameter context $\Phi \in [\![S]\!]$, and one is tempted to apply lemma $\Box S'. F$, then such an appeal is admissible if $\Phi \in [\![S']\!]$. This is a very strong requirement, and without doubt, it can be relaxed. We postpone the discussion on more sophisticated context subsumptions until Section 9.1.3.

Definition 5.22 (Context subsumption) *We say that context schema S' subsumes context schema S iff $\Phi \in [\![S]\!]$ implies that $\Phi \in [\![S']\!]$.*

Context subsumption is a semantic criteria and in this, it is very similar to termination, coverage, and strictness. A very simple minded syntactic criterion for context will be presented in Chapter 6.

It is clear, that the diamond Lemma 4.6 can appeal to the substitution Lemma 4.5, because both context schemas are the same, and hence the subsumption condition is trivially satisfied.

5.7.3 Proof Rules

The concepts of lemmas requires two additional proof rules for $\mathcal{M}_2^+$ (Section 5.5), one to type meta-hypothesis, and the other to express lemma application. The first rule extends the provability judgment on general formulas, and the second the judgments of provability of declarations. The complete and final set of proof rules for $\mathcal{M}_2^+$ is presented in Appendix A.

So far, a general formula is considered proved if its body is provable from no other assumptions. Since we have extended the meta-logic by meta-hypothesis, we must add one more rule. Each meta-hypothesis is a proof.

$$\frac{\mathbf{x} \in G \in \Xi}{\Xi \vdash \mathbf{x} \in G}\ \mathsf{mhyp}$$

Next to the new left rule. So far, the only two application rules where $\mathsf{L}\forall$ and $\mathsf{L}\forall$, which pick a meta-assumption from Δ, and apply them to either an LF-term or a block-variable to the meta-assumption, respectively. Likewise, if Q is a general proof term, it can be considered for application. Recall that the judgment for the provability of formulas is indexed by a context schema S and a signature Σ.

$$\frac{\Xi \vdash Q \in \Box S'. F \qquad \Psi; \Delta, \mathbf{y} \in F; \Xi \vdash D \in \Psi'; \Delta'}{\Psi; \Delta; \Xi \vdash \mathbf{y} \in F = \text{lemma } Q, D \in \Psi'; \mathbf{y} \in F, \Delta'}\ \mathsf{L}\Xi$$

Of course, as side condition, we must require that the context schema of the callee S' subsumes the context schema S of the caller.

$$S' \text{ subsumes } S \tag{5.4}$$

The Church-Rosser theorem is provable under the meta-hypothesis, that the confluence property holds; there is a proof term $Q_{\mathbf{cr}}$, which one obtains by desugaring the proof term in Figure 4.7,

$$
\begin{aligned}
\mathbf{conf} \in \Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x.\\
\forall T : \text{tp}. \forall E : \text{term } T. \forall E^l : \text{term } T. \forall E^r : \text{term } T.\\
\forall D^l : E \stackrel{*}{\Longrightarrow} E^l. \forall D^r : E \stackrel{*}{\Longrightarrow} E^r.\\
\exists E' : \text{term } T. \exists R^l : E^l \stackrel{*}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{*}{\Longrightarrow} E'. \top
\end{aligned}
$$

$$
\begin{aligned}
\vdash \quad Q_{\mathbf{cr}} \in \Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x.\\
\forall T : \text{tp}. \forall E^l : \text{term } T. \forall E^r : \text{term } T.\\
\forall D : E^l \Longleftrightarrow E^r.\\
\exists E' : \text{term } T. \exists R^l : E^l \stackrel{*}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{*}{\Longrightarrow} E'. \top
\end{aligned}
$$

Similarly, the confluence lemma is provable under the meta-hypothesis that there is a proof of the strip lemma: $Q_{\mathbf{conf}}$ is the desugared version of the proof term in Figure 4.6.

$$
\begin{aligned}
\mathbf{strip} \in \Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x.\\
\forall T : \text{tp}. \forall E : \text{term } T. \forall E^l : \text{term } T. \forall E^r : \text{term } T.\\
\forall D^l : E \stackrel{1}{\Longrightarrow} E^l. \forall D^r : E \stackrel{*}{\Longrightarrow} E^r.\\
\exists E' : \text{term } T. \exists E^l : R^l \stackrel{*}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{1}{\Longrightarrow} E'. \top
\end{aligned}
$$

$$
\begin{aligned}
\vdash \quad Q_{\mathbf{conf}} \in \Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x.\\
\forall T : \text{tp}. \forall E : \text{term } T. \forall E^l : \text{term } T. \forall E^r : \text{term } T.\\
\forall D^l : E \stackrel{*}{\Longrightarrow} E^l. \forall D^r : E \stackrel{*}{\Longrightarrow} E^r.\\
\exists E' : \text{term } T. \exists R^l : E^l \stackrel{*}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{*}{\Longrightarrow} E'. \top
\end{aligned}
$$

The strip Lemma 3.8 is based on the diamond Lemma 4.6, and its proof term $Q_{\mathbf{strip}}$ is the desugared version of the proof term in Figure 4.5.

$$
\begin{aligned}
\mathbf{dia} \in \Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x.\\
\forall T : \text{tp}. \forall E : \text{term } T. \forall E^l : \text{term } T. \forall E^r : \text{term } T.\\
\forall D^l : E \stackrel{1}{\Longrightarrow} E^l. \forall D^r : E \stackrel{*}{\Longrightarrow} E^r.\\
\exists E' : \text{term } T. \exists R^l : E^l \stackrel{*}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{1}{\Longrightarrow} E'. \top
\end{aligned}
$$

$$
\begin{aligned}
\vdash \quad Q_{\mathbf{strip}} \in \Box\text{SOME } T : \text{tp. BLOCK } x : \text{term } T, u : x \stackrel{1}{\Longrightarrow} x.\\
\forall T : \text{tp}. \forall E : \text{term } T. \forall E^l : \text{term } T. \forall E^r : \text{term } T.\\
\forall D^l : E \stackrel{1}{\Longrightarrow} E^l. \forall D^r : E \stackrel{*}{\Longrightarrow} E^r.\\
\exists E' : \text{term } T. \exists R^l : E^l \stackrel{*}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{1}{\Longrightarrow} E'. \top
\end{aligned}
$$

On the other hand the diamond Lemma 4.6 is provable using the substitution as meta-hypothesis. The proof term $Q_{\mathbf{dia}}$ is the desugared version of the proof term given in Figure4.4.

$$
\begin{array}{l}
\mathbf{subst} \in \Box\mathrm{SOME}\ T : \mathrm{tp}.\ \mathrm{BLOCK}\ x : \mathrm{term}\ T, u : x \overset{1}{\Longrightarrow} x. \\
\quad \forall T_1 : \mathrm{tp}. \forall T_2 : \mathrm{tp}. \forall E_1 : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1. \forall E_1' : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1. \\
\quad \forall E_2 : \mathrm{term}\ T_2. \forall E_2' : \mathrm{term}\ T_2. \\
\quad\quad \forall D_1 : (\Pi y : \mathrm{term}\ T_2. y \overset{1}{\Longrightarrow} y \to E_1\ y \overset{1}{\Longrightarrow} E_1'\ y). \forall D_2 : E_2 \overset{1}{\Longrightarrow} E_2'. \\
\quad\quad \exists P : E_1\ E_2 \overset{1}{\Longrightarrow} E_1'\ E_2'. \top
\end{array}
$$

$$
\begin{array}{l}
\vdash\ Q_{\mathbf{dia}} \in \Box\mathrm{SOME}\ T : \mathrm{tp}.\ \mathrm{BLOCK}\ x : \mathrm{term}\ T, u : x \overset{1}{\Longrightarrow} x. \\
\quad \forall T : \mathrm{tp}. \forall E : \mathrm{term}\ T. \forall E^l : \mathrm{term}\ T. \forall E^r : \mathrm{term}\ T. \\
\quad\quad \forall D^l : E \overset{1}{\Longrightarrow} E^l. \forall D^r : E \overset{*}{\Longrightarrow} E^r. \\
\quad\quad \exists E' : \mathrm{term}\ T. \exists R^l : E^l \overset{*}{\Longrightarrow} E'. \exists R^r : E^r \overset{1}{\Longrightarrow} E'. \top
\end{array}
$$

Finally, the substitution lemma is directly provable. The proof is formalized by the proof term $Q_{\mathbf{subst}}$, the desugared version of the proof term given in Figure 4.3

$$
\begin{array}{l}
\cdot \vdash\ Q_{\mathbf{subst}} \in \Box\mathrm{SOME}\ T : \mathrm{tp}.\ \mathrm{BLOCK}\ x : \mathrm{term}\ T, u : x \overset{1}{\Longrightarrow} x. \\
\quad \forall T_1 : \mathrm{tp}. \forall T_2 : \mathrm{tp}. \forall E_1 : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1. \forall E_1' : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1. \\
\quad \forall E_2 : \mathrm{term}\ T_2. \forall E_2' : \mathrm{term}\ T_2. \\
\quad\quad \forall D_1 : (\Pi y : \mathrm{term}\ T_2. y \overset{1}{\Longrightarrow} y \to E_1\ y \overset{1}{\Longrightarrow} E_1'\ y). \forall D_2 : E_2 \overset{1}{\Longrightarrow} E_2'. \\
\quad\quad \exists P : E_1\ E_2 \overset{1}{\Longrightarrow} E_1'\ E_2'. \top
\end{array}
$$

How can we obtain a closed proof the Church-Rosser theorem? By using lemma instantiations. Lemma instantiations act as substitutions on the meta-level. Entire proofs are substituted into proof terms, hereby gradually instantiating meta-assumptions. Naturally, lemma instantiations must be well-formed.

Judgment

$$\textit{Well-formed lemma instantiations:}\quad \Xi' \vdash Q \in \Xi$$

Intuitively, a lemma instantiation is well-formed, if it is either empty, or if the general programs Q are really proofs of the formulas the claim to be proofs of.

$$
\frac{}{\Xi' \vdash \cdot \in \cdot}\ \mathsf{sabstract} \qquad \frac{\vdash Q \in G \qquad \Xi' \vdash \xi \in \Xi}{\Xi' \vdash \xi, Q/\mathbf{x} \in \Xi, \mathbf{x} \in G}\ \mathsf{smeta}
$$

Similar to substitution we write $Q[\xi]$ in order to apply a lemma instantiation ξ to a general proof term Q. Lemma instantiations can be composed the same way, substitutions can.

Definition 5.23 (Composition of lemma instantiations)

$$
\begin{array}{rcl}
\cdot \circ \xi_2 & = & \xi_2 \\
(\xi_1, Q/x) \circ \xi_2 & = & (\xi_1 \circ \xi_2), Q[\xi_2]/x
\end{array}
$$

Provided, that there is a substitution lemma for lemma instantiations (which we prove in Chapter 6), we can prove the validity of lemma instantiation composition.

Lemma 5.24 (Composition of lemma instantiations)

If $\mathcal{D}_1 :: \Xi_2 \vdash \xi_1 : \Xi_1$
and $\mathcal{D}_2 :: \Xi_3 \vdash \xi_2 : \Xi_2$
then $\Xi_3 \vdash \xi_1 \circ \xi_2 : \Xi_1$

As example, consider the a combined proof of the Church-Rosser theorem for parallel reduction,

$$\begin{array}{ll} \cdot \vdash & Q_{\mathbf{cr}}[Q_{\mathbf{conf}}[Q_{\mathbf{strip}}[Q_{\mathbf{dia}}[Q_{\mathbf{subst}}/\mathbf{subst}]/\mathbf{dia}]/\mathbf{strip}]/\mathbf{conf}] \\ & \in \Box\mathrm{SOME}\ T : \mathrm{tp}.\ \mathrm{BLOCK}\ x : \mathrm{term}\ T, u : x \stackrel{1}{\Longrightarrow} x. \\ & \qquad\qquad \forall T : \mathrm{tp}. \forall E^l : \mathrm{term}\ T. \forall E^r : \mathrm{term}\ T. \\ & \qquad\qquad \forall D : E^l \iff E^r. \\ & \qquad\qquad \exists E' : \mathrm{term}\ T. \exists R^l : E^l \stackrel{*}{\Longrightarrow} E'. \exists R^r : E^r \stackrel{*}{\Longrightarrow} E'. \top \end{array}$$

By easy inspection, if follows that all involved lemma instantiations are well-formed, and by the meta-theory which we start to describe in the next Chapter that this proof term indeed formalizes the proof of the Church-Rosser theorem.

5.8 Summary

In this Section we have described the meta-logic $\mathcal{M}_2^+$ and an appropriate proof term calculus which formalizes meta-proofs as recursive functions. Among the many rules, there are four rules, which have side conditions in order to guarantee that the proof term calculus is indeed a calculus of realizers. There is a termination side condition (5.1), which enforces that any evaluation of a recursive function eventually terminates, the coverage side condition (5.2), which ensures that all cases are always covered, the strictness side condition (5.3), which enforces determinacy, and eventually the subsumption side condition (5.4), which guarantees soundness of lemma application. A summary of all rules can be found in Appendix A.

The meta-logic is general enough to represent any of the proof terms from Chapter 3 and Chapter 4, such as the entire development of the Church-Rosser proof of ordinary reduction. It is powerful enough to represent the theory of cut-elimination, meta-theoretic properties of programming languages, especially functional and logic programming languages, compiler correctness, and examples from category theory. Not only are all theorems representable in the meta-logic, but they are also automatically derivable, as we will discuss in Chapter 8. A more detailed account on which theorems have been proven automatically will be given in Section 8.5.

$\mathcal{M}_2^+$ has several limitations. The first limitation stems from the observation that the representation power of the meta-logic is directly connected to the representation power of the underlying logical framework. Reasoning about imperative programming languages is not very well supported by the logical framework LF due to the lack of an elegant encoding of state. First promising results have been achieved with an extension of LF to a linear logical framework LLF [CP96], which treats memory cells as resources. Resources disappear whenever accessed. It is during the reassumption phase, that the value of a resource can be changed, which makes the linear logical framework a prime candidate for modeling imperative languages. A generalization of $\mathcal{M}_2^+$ to a meta-logic for a linear logical framework such as LLF has not been carried out yet, but it will be discussed briefly in Section 9.1.2.

A second limitation of the meta-logic $\mathcal{M}_2^+$ is that it currently cannot represent any meta-logical arguments which require a proof by logical relations (also Tait's method). When applying this method, one normally defines semantically a relation P, and in order to show that a judgment J can be transformed into a judgment J', we show that each derivation of J satisfies P and furthermore that each derivation satisfying P can be transformed into a derivation satisfying J'. This technique is used for example in the canonical form theorem for the simply-typed λ-calculus. $\mathcal{M}_2^+$ lacks mechanisms such as for example quantification over substitutions to express commonly used logical relations P.

A third limitation is that $\mathcal{M}_2^+$ is restricted to Π_2-formulas, and that it offers only a limited number of logical connectives. Many theorems have natural formulations, which fall outside this fragment, prompting the user for auxiliary constructions.

This concludes the presentation of the meta-logic $\mathcal{M}_2^+$, and we continue with the presentation of a type-preserving operational semantics, which we use to show that all proof terms are total functions.

Chapter 6

Operational Semantics for $\mathcal{M}_2^+$

6.1 Introduction

The proof term calculus $\mathcal{M}_2^+$ is designed with the idea in mind that all proof terms correspond to total recursive functions called realizers, summarizing derivations and witnessing the soundness of the meta-logic $\mathcal{M}_2^+$. The soundness proof itself is long and introduces many definitions, a sophisticated matching algorithm, a big-step and a small-step semantics. Therefore we have decided to break it up into two chapters. This is the first chapter, and its goal is to demonstrate how proof terms are interpreted as recursive functions and how they can be executed. In future work we will investigate independent applications of $\mathcal{M}_2^+$ as a programming language. In the next chapter we show that all functions in $\mathcal{M}_2^+$ are realizers when satisfying the termination side condition (5.1), and the coverage side condition (5.2). The reader who is more interested in the practical applications and results is invited to skip these two chapters and to continue reading Chapter 8 which discusses an implementation of $\mathcal{M}_2^+$ as part of the Twelf system.

This Chapter is organized as follows: In Section 6.2, we directly begin with the technical discussion; we formally introduce substitutions, abstractions, subordination, and other necessary concepts, and we derive basic properties such as weakening and substitution lemmas. In Section 6.3 then, we present a syntactic criterion for context schema subsumption necessary for sound lemma invocations. The matching algorithm for case constructs is defined in Section 6.4 as part of the big-step semantics which is described in Section 6.5. Finally, we conclude this chapter with a summary in Section 6.6.

6.2 Preliminaries

Proof terms are recursive functions and they operate on LF objects. Because of the different variable concepts used to define $\mathcal{M}_2^+$, there are many different notions of substitutions and substitution applications to be considered. Generalized substitutions for example enjoy the same properties LF substitutions introduced in Section 5.2 enjoy; assumptions variables correspond directly to LF variables, and variable blocks are mapped to lists of LF variables. The main difference to LF substitutions is that generalized substitutions carry additional information about the boundaries of variable blocks.

This section is organized as follows: We first discuss basic properties of standard LF substitutions in Section 6.2.1 and issues related to hypothetical arguments in Section 6.2.2. We then derive a set of weakening lemmas, for formulas, proof terms, generalized, and meta-contexts in Section 6.2.3 which are required for the upcoming technical discussions. Likewise we prove a variety of substitution lemmas for formulas and proof terms in Section 6.2.4.

6.2.1 LF

The construction of $\mathcal{M}_2^+$ relies on the fundamental property of LF that canonical forms exist, as shown in Theorem 2.6. But there are also other properties, which are equally important for the sake of the formal development. The first property is the weakening property. An object M (type family A) remains well-typed (well-kinded) under any extension of the context Γ. Formally, we write $\Gamma \leq \Gamma'$ if Γ' results from interspersing Γ with arbitrary (but always well-typed) variable declarations. Note, that we implicitly assume that $\vdash \Gamma$ ctx and $\vdash \Gamma'$ ctx holds.

Lemma 6.1 (Weakening for LF)

1. *If* $\Gamma \vdash M : A$
   and $\Gamma \leq \Gamma'$
   then $\Gamma' \vdash M : A$

2. *If* $\Gamma \vdash A : K$
   and $\Gamma \leq \Gamma'$
   then $\Gamma' \vdash A : K$

Proof: by induction on the typing derivations. □

Similarly, there is a substitution lemma, which expresses that the typing relation is stable under substitution application. The definition of substitution application to LF objects, LF types, and LF kinds is omitted from this thesis.

Lemma 6.2 (Substitution property of LF)

1. *If* $\Gamma \vdash M : A$
   and $\Gamma' \vdash \sigma : \Gamma$
   then $\Gamma' \vdash M[\sigma] : A[\sigma]$

2. *If* $\Gamma \vdash A : K$
   and $\Gamma' \vdash \sigma : \Gamma$
   then $\Gamma' \vdash A[\sigma] : K[\sigma]$

Proof: by induction on the typing derivations. □

This concludes the presentation of all properties of the logical framework LF necessary to carry out the formal analysis of $\mathcal{M}_2^+$.

6.2.2 Abstraction

Abstraction is an operation which is used for example in the definition of the Lnew-rule in Section 5.4.4.

$$\frac{(\text{SOME } C_1.\, \text{BLOCK } C_2)^L \in S \quad \Psi \vdash \sigma : C_1 \quad \Psi \vdash \rho \equiv_\alpha [\sigma]C_2 \quad \Psi, \rho^L; \Delta \vdash \Psi'; \Delta'}{\Psi; \Delta \vdash \Pi\rho^L.\, (\Psi'; \Delta')}\,\mathsf{Lnew}$$

Abstraction formalizes how results of applying the induction hypothesis are interpreted after an extension of the world is discharged. Hypothetical arguments typically first introduce new assumptions, then apply induction hypotheses or possibly lemmas, and eventually discharge the new assumptions. It is the goal of this subsection to give a formal account on how to interpret, for example, the result of the induction hypothesis after the last step.

Recall the walk through the proof of the reflexivity Lemma 4.3 in Section 5.4.4. In order to prove the case for "lam" we had to introduce new assumptions. More precisely, we introduced a new parameter block in form of a variable block $(\underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L$. After a few further reasoning steps, we demonstrated the existence of an LF object $P : (E' \ \underline{x}) \stackrel{1}{\Longrightarrow} (E' \ \underline{x})$, and three meta-assumptions, represented as the meta-assumption list $\Delta'^{(2)}$ on page 117.

$$\begin{array}{rcl}
\Psi'^{(2)} & = & P : (E' \ \underline{x}) \stackrel{1}{\Longrightarrow} (E' \ \underline{x}) \\
\Delta'^{(2)} & = & \mathbf{x}_0 \in \forall E : \text{term } T.\, \exists D : E \stackrel{1}{\Longrightarrow} E.\, \top, \\
& & \mathbf{x}_1 \in \exists D : (E' \ \underline{x}) \stackrel{1}{\Longrightarrow} (E' \ \underline{x}).\, \top, \\
& & \mathbf{x}_2 \in \top
\end{array}$$

Let us first concentrate only on $\Psi'^{(2)}$. What does it mean to reason hypothetically? It simply means, that P is the representation image of a derivation $\mathcal{P}$, which possibly uses two assumptions $\underline{x}$ and $\underline{u}$ represented as $\ulcorner x \urcorner = x$ and $\ulcorner u \urcorner = u$ as already shown in Equation (4.1), keeping in mind that $(\underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L$ is assumed.

$$\left\ulcorner \begin{array}{c} \overline{x \stackrel{1}{\Longrightarrow} x}\,u \\ \mathcal{P} \\ e' \stackrel{1}{\Longrightarrow} e' \end{array} \right\urcorner = \Pi x : \text{term } \ulcorner \tau_1 \urcorner.\, \Pi u : x \stackrel{1}{\Longrightarrow} x.\, (E' \ x) \stackrel{1}{\Longrightarrow} (E' \ x)$$

By abstraction we refer to the process that calculates the right hand side of this equation from the variable block $(\underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L$ and the type of the new assumption $(E' \ \underline{x}) \stackrel{1}{\Longrightarrow} (E' \ \underline{x})$: We write $P : \Pi(\underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}).\, (E' \ \underline{x}) \stackrel{1}{\Longrightarrow} (E' \ \underline{x})$ for this operation. Here is a preliminary definition of the abstraction operation $\Pi\rho.\, A$, the same we have already presented earlier.

$$\begin{array}{rcl}
\Pi\cdot.\, A_2 & = & A_2 \\
\Pi(\underline{x} : A_1, \rho).\, A_2 & = & \Pi x : A_1.\, (\Pi\rho.\, A_2)
\end{array}$$

Note, that while the abstraction algorithm executes, the hypothesis $\underline{x}$ and $\underline{u}$ are transformed into LF variables x and u. Thus, abstraction simultaneously and implicitly removes the "underlines" from variable names. This operation is rather conservative. On the one hand, it safe because no variable cannot escape its scope. On the other hand, it is conservative, because it is possible that x can never occur in $\Pi\rho.\, A_2$; as for example no term x can ever occur in the definition of a type "tp". This is impossible as an easy inspection of the signature in Figure 2.2 shows. Abstracting over x in this situation certainly does not lead to an unsoundness, but it may lead to an incompleteness, as we can easily demonstrate using the proof of Lemma 4.12. Thus, abstraction should be able to strengthen the variable block that is abstracted by removing all declarations that cannot occur in the type.

In the example the hypothetical argument introduces two related parameters: $\underline{x}$ is an atomic term and $\underline{y}$ is a term of the same type. Therefore, the regular world extension consists of parameter blocks of the following form: $(\underline{x} : \text{atm } T_1, \underline{y} : \text{term } T_1)^L$. In the proof of Lemma 4.12 the induction hypothesis is applied in an extension of the current world with the result that there exists a term of type T_2. T_2's existence is hypothetical, therefore we use the abstraction algorithm to abstract, $\underline{x}$ and $\underline{y}$. However, using the algorithm in its current form, the result is an object of type "atm $T_1 \to$ term $T_1 \to$ term T_2", even though $\underline{x}$ is a semantically meaningless abstraction. Consequently, we will refine the abstraction algorithm to ignore any semantically meaningless assumption. Only if abstraction ignores the "$\underline{x}$: atm T_1" hypothesis, the proof can be easily completed as we have already informally argued at the end of Section 4.2.3.

The static analysis of the signature which summarizes which objects of which type can occur as subobjects in objects of some other type is satisfactorily summarized by the dependency relation, or subordination relation [Roh96, Vir99]. Virga has shown that if a type A_1 is not subordinate to type A_2, then it is impossible, that any object of type A_1 occurs as a subobject in any object of type A_2. As a matter of fact, we can partition type families into equivalence classes modulo subordination and define a partial order on those classes based on subordination. For our purposes, we completely adopt the definition and notation of the subordination relation from Virga [Vir99], Chapter 5, and we write $A_1 \prec_\Sigma A_2$ iff $A_1 \prec^\tau_\Sigma A_2$ or $A_1 \prec^t_\Sigma A_2$. Indeed by Corollary 5.2.2 in [Vir99] we learn that if $A_1 \not\prec_\Sigma A_2$ then no variable $x : A_1$ can occur freely in any object of type A_2. Translated into our setting we obtain the following lemma.

Lemma 6.3 (Subordination)

1. *If* $A_1 \not\prec_\Sigma A_2$
   and $\Gamma_1, x : A_1, \Gamma_2 \vdash M : A$
   then $\Gamma_1, \Gamma_2 \vdash M : A$

2. *If* $A_1 \not\prec_\Sigma A_2$
   and $\Gamma_1, x : A_1, \Gamma_2 \vdash A : K$
   then $\Gamma_1, \Gamma_2 \vdash A : K$

Proof: see [Vir99], Corollary 5.2.2. □

This lemma only holds for objects which are valid in regular world extensions that conform with the subordination relation. In our situation it is hence important, that the signature and the context schema bound by any general formula do not invalidate the subordination relation. All one has to do is to check all dependencies introduced by the LF-types of the BLOCK-component of a block schema. More precisely, we write $\prec_S$ for the subordination relation induced by a context schema S. In order to guarantee soundness of $\mathcal{M}_2^+$ we must attach a side condition to the rule generalR.

$$(\prec_S) \subset (\prec_\Sigma) \tag{6.1}$$

Without loss of generality, we can assume that this side condition is always satisfied for the set of meta-theorems and proofs, we are interested in, and hence we drop the subscript and write only $\prec$ instead of $\prec_\Sigma$. Inspired by Lemma 6.3, we refine the abstraction operation from above by defining it for arbitrary LF (sub-)contexts. The reader should keep in mind, that abstraction is an LF-level operation which expects an LF type as argument and computes a new abstracted LF-type. Likewise we define an abstraction operation for objects by building λ-closures in a very similar way.

Definition 6.4 (Abstraction)

1. *Type-level abstraction:*

$$\begin{array}{rcll} \Pi\cdot.\, A_2 & = & A_2 & \\ \Pi(x : A_1, \Gamma).\, A_2 & = & \Pi\Gamma.\, A_2 & \text{if } A_1 \not\prec A_2 \\ \Pi(x : A_1, \Gamma).\, A_2 & = & \Pi x : A_1.\, (\Pi\Gamma.\, A_2) & \text{if } A_1 \prec A_2 \end{array}$$

2. *Object-level abstraction: Let M be well-typed of type A_2*

$$\begin{array}{rcll} \lambda\cdot.\, M & = & M & \\ \lambda(x : A_1, \Gamma).\, M & = & \lambda\Gamma.\, M & \text{if } A_1 \not\prec A_2 \\ \lambda(x : A_1, \Gamma).\, M & = & \lambda x : A_1.\, (\lambda\Gamma.\, M) & \text{if } A_1 \prec A_2 \end{array}$$

It remains to show that the abstraction algorithm is well-defined. But this is an easy consequence from Virga's results. The statement of the theorem relies on

Lemma 6.5 (Single assumption abstraction)

1. *For all contexts Γ_1*
   if $\Gamma_1, \Gamma_2 \vdash A : type$
   then $\Gamma_1 \vdash \Pi\Gamma_2.\, A : type$

2. *For all contexts Γ_1*
   if $\Gamma_1, \Gamma_2 \vdash M : A$
   then $\Gamma_1 \vdash \lambda\Gamma_2.\, M : \Pi\Gamma_2.\, A$

Proof: by induction over Γ_2 (in part 1) and Γ_2 (in part 2), using Lemma 6.3. A detailed proof can be found in Appendix B.1.1. □

LF-level abstraction ignores semantically meaningless parameter declarations, which can provably never occur in the subject of abstraction. Meta-level assumptions on the other hand are treated differently. In the example above recall that the assumptions in $\Delta'^{(2)}$ are also being abstracted over the same block variable. But on the meta-level, at least in this thesis, we respect parameter block boundaries and do not omit any semantically meaningless parameter declarations. Thus, abstraction translates directly into the application of the inference rule $\mathsf{R}\Pi$.

In this subsection so far we have described how to execute abstraction for single LF-assumptions and for single meta-assumptions. In the remainder of this subsection we generalize abstraction to extensions $\Psi'; \Delta'$ as used in the Lnew-rule. As example consider again the reflexivity lemma for parallel reduction from above. In this special case we can simply iterate through $\Psi'^{(2)}; \Delta'^{(2)}$ and repeatedly apply the abstraction operation which eventually results in $\Psi'^{(1)}; \Delta'^{(1)}$. The reader should be warned, the general case is more complicated.

$$\begin{array}{rcl}
\Psi'^{(1)} & = & P : \Pi x : \text{term}\, T.\, \Pi u : x \stackrel{1}{\Longrightarrow} x.\, (E'\ x) \stackrel{1}{\Longrightarrow} (E'\ x) \\
\Delta'^{(1)} & = & \mathbf{x}_0 \in \mathbf{\Pi}(\underline{x} : \text{term}\, T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L.\, \forall E : \text{term}\, T.\, \exists D : E \stackrel{1}{\Longrightarrow} E.\, \top, \\
 & & \mathbf{x}_1 \in \mathbf{\Pi}(\underline{x} : \text{term}\, T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L.\, \exists D : (E'\ \underline{x}) \stackrel{1}{\Longrightarrow} (E'\ \underline{x}).\, \top, \\
 & & \mathbf{x}_2 \in \mathbf{\Pi}(\underline{x} : \text{term}\, T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L.\, \top
\end{array}$$

Formally, we write $\Pi\rho^L.\, (\Psi'^{(2)}; \Delta'^{(2)}) = \Psi'^{(1)}; \Delta'^{(1)}$ for this abstraction operation. Clearly, the abstraction of an assumption variable implicitly changes its type, and this change must be reflected at the locations the variable in a type. In the example above, the only new declaration in Ψ is P, and by the formulation of the theorem, P does not occur in any other type as index variable. Thus this example is only a special case.

We encounter the general case in the proof of the diamond Lemma 4.6. In the pbeta/pbeta-case for example, we assume the existence of a parameter block (which happens to have the same form as above: $\underline{x} : \text{term}\, T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$). Here is a snapshot of the additional assumptions immediately before the abstraction operation is about to take place. For brevity, we only present the extension to the abstract context.

$$\begin{array}{rcl}
\Psi'^{(2)} & = & E' : \text{term}\, T_2, R^l : (E^l\ x) \stackrel{1}{\Longrightarrow} E', R^r : (E^r\ x) \stackrel{1}{\Longrightarrow} E' \\
\Delta'^{(2)} & = & \ldots
\end{array}$$

Abstraction considers the declaration of E' first. Because of the subordination relation, u is guaranteed not to occur as a subterm of E'. Using the abstraction operation from Definition 6.4 we obtain as new type for E': "term $T \to$ term T_2". It should be clear, that all occurrences of E' must be replaced by the abstracted version of E', namely $E'\ x$.

Next R^l is abstracted, and a quick inspection of the subordination relation reveals that it may depend on x and u. Consequently, R^l's new type is $\Pi x : \text{term}\, T.\, \Pi u : x \stackrel{1}{\Longrightarrow} x.\, (E^l\ x) \stackrel{1}{\Longrightarrow} (E'\ x)$ and any occurrences of R^l would have to be replaced by $R^l\ x\ u$, but there aren't any. Similarly, R^r's abstracted type is $\Pi x : \text{term}\, T.\, \Pi u : x \stackrel{1}{\Longrightarrow} x.\, (E^r\ x) \stackrel{1}{\Longrightarrow} (E'\ x)$.

In summary, after abstraction we must obtain a new abstract context extension, for which we write

$$\begin{array}{rcl}
\Psi'^{(1)} & = & E' : \text{term}\, T \to \text{term}\, T_2, \\
 & & R^l : \Pi x : \text{term}\, T.\, \Pi u : x \stackrel{1}{\Longrightarrow} x.\, (E^l\ x) \stackrel{1}{\Longrightarrow} (E'\ x), \\
 & & R^r : \Pi x : \text{term}\, T.\, \Pi u : x \stackrel{1}{\Longrightarrow} x.\, (E^r\ x) \stackrel{1}{\Longrightarrow} (E'\ x) \\
\Delta'^{(1)} & = & \ldots
\end{array}$$

and naturally, $\Delta'^{(1)}$ follows from $\Delta'^{(2)}$ by replacing all occurrences of E' by $(E'\ x)$, followed by the standard abstraction step for formula as described in Section 6.2.2.

In the general case, the occurrence of a variable might be abstracted over several variables, possibly over all variables declared by the new parameter block which satisfy the subordination condition we have described in Section 6.2.2. In the example above, for $\rho = \underline{x} : \text{term}\, T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow}$

$\underline{x}$ we write $(E'\ \rho)/E$. The notation of $E\ \rho$ is introduced to facilitate the presentation. Again, we loose the underlines of the parameter variables when execution this variable application. It is defined as as follows.

Definition 6.6 (Variable application) *Let E' be well-typed of type A_2*

$$\begin{array}{rcll} E'\ \cdot & = & E' & \\ E'\ (\underline{x} : A_1, \rho) & = & (E'\ x)\ \rho & \textit{if } A_1 \prec A_2 \\ E'\ (\underline{x} : A_1, \rho) & = & E'\ \rho & \textit{if } A_1 \not\prec A_2 \end{array}$$

We begin now with the formal definition of the $\Pi\rho^L.\,(\Psi;\Delta) = \Psi';\Delta'$ relation. The reader should be aware, that neither $\Psi;\Delta$ nor $\Psi';\Delta'$ are meta-contexts by themselves, they are merely valid extensions of some meta-context $\Psi_0;\Delta_0$. Formally, it always holds that

$$\begin{array}{l} \vdash \Psi_0, \rho^L, \Psi \text{ abstract} \\ \Psi_0, \rho^L, \Psi \vdash \Delta_0, \Delta \text{ meta} \end{array}$$

and the same for the abstracted versions:

$$\begin{array}{l} \vdash \Psi_0, \Psi' \text{ abstract} \\ \Psi_0, \Psi' \vdash \Delta_0, \Delta' \text{ meta} \end{array}$$

The basic idea of the definition of $\Pi\rho^L.\,(\Psi;\Delta)$ is therefore to traverse Ψ, abstract it to Ψ', and simultaneously, replace all occurrences of abstracted variables by the their abstracted counterparts in the rest of Ψ and in Δ.

Judgment

$$\text{Meta-context abstraction:}\quad \Pi\rho^L.\,(\Psi;\Delta) = \Psi';\Delta'$$

Rules

$$\frac{}{\Pi\rho^L.\,(\cdot;\cdot) = \cdot;\cdot}\ \mathsf{rempty}$$

$$\frac{\Pi\rho^L.\,([(x\ \rho)/x]\Psi; [(x\ \rho)/x]\Delta)) = \Psi';\Delta'}{\Pi\rho^L.\,(x : A, \Psi;\Delta) = x : \Pi\rho.\,A, \Psi';\Delta'}\ \mathsf{rass} \qquad \frac{\Pi\rho^L.\,(\cdot;\Delta) = \cdot;\Delta'}{\Pi\rho^L.\,(\cdot;\mathbf{x} \in F, \Delta) = \cdot;\mathbf{x} \in \mathbf{\Pi}\rho^L.\,F, \Delta'}\ \mathsf{rmeta}$$

Meta-context abstraction is used in the definition of the meta logic $\mathcal{M}_2^+$, specifically, for the definition of the Lnew-rule. When executing a proof term, we calculate an instantiation for those variable declarations, as we discuss in Section 6.5, and those instantiations must clearly be abstracted accordingly. For obvious reasons, we call this operation meta-substitution abstraction, and write $\lambda\rho^L.\,(\psi;\delta) = \psi';\delta'$. Note that the instantiation is only a tail of real meta-substitutions, i.e. they are partial in the same sense as meta-contexts are extensions of real meta-contexts, too.

Judgment

$$\text{Meta-substitution abstraction:}\quad \lambda\rho^L.\,(\psi;\delta) = \psi';\delta'$$

Rules

$$\frac{}{\lambda\rho^L.\,(\cdot;\cdot) = \cdot;\cdot}\ \mathsf{rpempty}$$

$$\frac{\lambda\rho^L.\,(\psi;\delta) = \psi';\delta'}{\lambda\rho^L.\,(M/x,\psi;\delta) = \lambda\rho.\,M/x,\psi';\delta'}\ \mathsf{rpass} \qquad \frac{\lambda\rho^L.\,(\cdot;\delta) = \cdot;\delta'}{\lambda\rho^L.\,(\cdot;P/\mathbf{x},\delta) = \cdot;\lambda\rho^L.\,P/\mathbf{x},\delta'}\ \mathsf{rpmeta}$$

The reader might already suspect that if $\Psi';\Delta'$ is a valid meta-context and in $\Psi',\rho^L;\Delta'$ the meta-substitution extension $\psi;\delta$ instantiates the meta-context extension $\Psi;\Delta$ then we can safely abstract the variable block ρ^L. As result we obtain a new meta-substitution extension $\lambda\rho^L.\,(\psi;\delta)$ declared for $\Pi\rho^L.\,(\Psi;\Delta)$. This result is one of the basic ingredients to the proofs of type preservation for the operational semantics.

Lemma 6.7 (Extension abstraction)

1. *If* $\mathcal{E} :: \Psi_0,\rho^L;\cdot \vdash \psi_1,\rho/\rho,\psi;\delta \in \Psi_1,\rho^L,\Psi;\Delta$
   and $\mathcal{D} :: \Psi_0;\cdot \vdash \psi_1;\cdot \in \Psi_1;\cdot$
   then $\Psi_0;\cdot \vdash \psi_1,\psi';\delta' \in \Psi_1,\Psi';\Delta'$
   and $\psi';\delta' = \lambda\rho^L.\,(\psi;\delta)$
   and $\Psi';\Delta' = \Pi\rho^L.\,(\Psi;\Delta)$

2. *If* $\Psi_0,\rho^L;\cdot \vdash \psi_1,\rho/\rho;\delta \in \Psi_1,\rho^L;\Delta$
   and $\mathcal{D} :: \Psi_0;\cdot \vdash \psi_1;\cdot \in \Psi_1;\cdot$
   then $\Psi_0;\cdot \vdash \psi_1;\delta' \in \Psi_1;\Delta'$
   and $\cdot;\delta' = \lambda\rho^L.\,(\cdot;\delta)$
   and $\cdot;\Delta' = \Pi\rho^L.\,(\cdot;\Delta)$

Proof: by induction on $\Psi(1)$, $\Delta(2)$, using Lemma 6.5. A detailed proof can be found in Appendix B.1.1. □

This concludes our discussion about abstraction and we continue with the presentation of a few weakening results.

6.2.3 Weakening

The weakening results for LF from Section 6.2.1 generalize directly to weakening results for meta-level constructs such as generalized contexts, formulas, meta contexts, and proof terms. To establish these results is the goal of this subsection.

We begin with the presentation of a weakening result for generalized substitutions. If a generalized substitution has co-domain Ψ and Ψ' extends Ψ then — as expected — the same substitution is still well-defined only in the extended co-domain Ψ'. Similarly to Section 6.2.1, we write $\Psi \leq \Psi'$ for Ψ' extends Ψ and again, we implicitly assume that $\vdash \Psi$ abstract and $\vdash \Psi'$ abstract. Ψ' stems from Ψ by inserting new assumption variables and variable block declarations.

Lemma 6.8 (Weakening of generalized substitutions)
If $\mathcal{D} :: \Psi' \vdash \psi \in \Psi$
and $\Psi' \leq \Psi''$
then $\Psi'' \vdash \psi \in \Psi$

Proof: by induction on $\mathcal{D}$ using Lemma 6.1. □

Recall from Section 5.4.3, that the well-formedness judgment for formulas is defined with respect to a generalized context Ψ. Naturally a weakening result for proof terms implicitly requires that weakening of meta contexts is admissible which itself relies on a weakening result for formulas. The last lemma can be easily proven by induction on the structure of the formula.

Lemma 6.9 (Weakening of formulas)
If $\mathcal{D} :: \Psi \vdash F$ *formula*
and $\Psi \leq \Psi'$
then $\Psi' \vdash F$ *formula*

Proof: by induction on $\mathcal{D}$, using Lemma 6.1. □

The next goal is to establish a similar weakening result for proof terms. Proof terms may be open with respect to $\Psi; \Delta$. In particular, proof terms are defined in terms of declarations D and explicit meta substitutions $\psi; \delta$ for which we show the weakening property first. From the definition of meta contexts in Section 5.5.2, it follows immediately, that Ψ is a generalized context. How shall we define context extensions of meta contexts? We follow the same pattern as above and say that $\Psi'; \Delta'$ extends $\Psi; \Delta$, if $\Psi \leq \Psi'$ and Δ' results from inserting new meta-assumptions of the form $\mathbf{x} \in F$ into Δ. In this case we write $\Psi; \Delta \leq \Psi'; \Delta'$, where we always implicitly assume that the left and right hand sides of this notation are all well-formed meta-contexts. Naturally, the argument that this construction works relies on the shoulders of the weakening property for meta-contexts.

Lemma 6.10 (Weakening of meta-contexts)
If $\mathcal{D} :: \Psi \vdash \Delta$ *meta*
and $\Psi \leq \Psi'$
then $\Psi' \vdash \Delta$ *meta*

Proof: by induction on $\mathcal{D}$, using Lemma 6.9. □

The weakening lemma for proof terms cannot be proven directly, since they are mutually dependent on declarations and explicit meta substitutions. Consequently, the generalized form of the theorem must provide extra cases for those two constructs.

Lemma 6.11 (Weakening of proof terms)

1. *If* $\mathcal{D} :: \Psi; \Delta; \Xi \vdash P \in F$
   and $\Psi; \Delta \leq \Psi'; \Delta'$
   then $\Psi'; \Delta'; \Xi \vdash P \in F$

2. *If* $\mathcal{D} :: \Psi; \Delta; \Xi \vdash D \in \Psi''; \Delta''$
   and $\Psi; \Delta \leq \Psi'; \Delta'$
   then $\Psi'; \Delta'; \Xi \vdash D \in \Psi''; \Delta''$

3. *If* $\mathcal{D} :: \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$
   and $\Psi'; \Delta' \leq \Psi''; \Delta''$
   then $\Psi''; \Delta'' \vdash \psi; \delta \in \Psi; \Delta$

Proof: by induction on $\mathcal{D}(1)$, $\mathcal{D}(2)$, and $\mathcal{D}(3)$. □

Weakening is an essential property which is used implicitly and explicitly over and over throughout the entire theoretical investigation of this thesis, especially when we examine the interaction of substitutions and derivations in the meta-logic $\mathcal{M}_2^+$ which will be discussed in the next subsection.

6.2.4 Substitution

Substitutions are omnipresent in our investigation. The subject of the case construct in Section 5.6.2, for example, is defined by a pair of explicit substitutions; one which collects instantiations for assumptions and variable blocks, and another which explicitly tracks instantiations of meta-variables. The first substitution is an generalized substitution, and the second a meta substitution. Third, there are lemma instantiations. Recall that any proof is parametrized by a lemma repository Ξ which contains a list of lemmas, not necessarily proven yet, but which may be used during a meta-proof. All in all, there are three variables concepts and consequently three different notions of substitutions. In this subsection we are concerned with the application and interaction of the different kind of substitutions with context schemas, formulas, abstractions, and proof terms.

Context schemas

Context schemas are abstract descriptions of regularly formed parameter contexts. Every theorem is quantified by one outermost context schema. In Section 5.3 for example, we have specified a precise criterion of how to judge if a parameter block is an instance of a block schema SOME C_1. BLOCK C_2. First, all SOME-parameters must be instantiated by well-typed objects, well-typed in some generalized context Ψ. This process is referred to as SOME-instantiation. The parameter block in question must then be α-equivalent to the BLOCK-construction of this block schema. These two constructions are used in the Lnew-rule. It is this setting for which we need a substitution property.

Lemma 6.12 (Substitution lemma for context schemas)

1. *If* $\mathcal{D}_1 :: \Psi \vdash \sigma : C_1$
   and $\Psi' \vdash \psi \in \Psi$
   then $\Psi' \vdash \sigma \circ \psi : C_1$

2. *If* $\Psi \vdash [\sigma]C \equiv \rho$
   and $\Psi' \vdash \psi \in \Psi$
   then $\Psi' \vdash [\sigma \circ \psi]C \equiv [\psi]\rho$

Proof: by structural induction on $\mathcal{D}(1)$ and $C(2)$ using Lemma 6.2. □

Formulas

The application of generalized substitutions to formulas $F[\psi] = F'$ is easily defined.

$$\begin{array}{rcll}
(\forall x : A.\, F)[\psi] & = & \forall x : A[\psi].\, F[\psi, x/x] & (\mathsf{sAll}) \\
(\mathbf{\Pi}\rho^L.\, F)[\psi] & = & \mathbf{\Pi}\rho^L[\psi].\, F[\psi, \rho[\psi]/\rho] & (\mathsf{sAllP}) \\
(\exists x : A.\, F)[\psi] & = & \exists x : A[\psi].\, F[\psi, x/x] & (\mathsf{sEx}) \\
(\top)[\psi] & = & \top & (\mathsf{sTrue}) \\
(F_1 \wedge F_2)[\psi] & = & F_1[\psi] \wedge F_2[\psi] & (\mathsf{sAnd})
\end{array}$$

It is similarly easy to see that substitution application is sound.

Lemma 6.13 (Substitution lemma for formulas)
If $\mathcal{D} :: \Psi \vdash F$ *formula*
and $\mathcal{P} :: \Psi' \vdash \psi \in \Psi$
then $\Psi' \vdash F[\psi]$ *formula.*

Proof: by induction on $\mathcal{D}$ using Lemma 6.2. □

Note, that general formulas are always closed. Therefore they do not have to be considered for any kind of substitution operation. The careful reader will undoubtedly have noticed, that substitutions as used for example in the $\exists\mathsf{R}$ or $\forall\mathsf{L}$ rules are not completely specified. In the rule $\exists\mathsf{R}$, for example we write M/x as substitution, but we really mean $\mathrm{id}_\Psi, M/x$. We have committed to this simplification in order to keep this discussion short and accessible. Finally, we derive a limited commutativity property for substitutions.

Lemma 6.14 (Properties of substitution)

1. $F[M/x][\psi] = F[\psi, x/x][M[\psi]/x]$
2. $F[\rho'/\rho][\psi] = F[\psi, \rho/\rho][[\psi]\rho'/\rho]$

Proof: by induction on F. □

Meta assumptions

Meta assumptions lists are lists of possibly open formulas. They are defined with respect to a generalized context $\Psi \vdash \Delta$ meta. The notion of substitution application to formulas can be easily generalized to those lists for which we write $[\psi]\Delta = \Delta'$.

$$\begin{array}{rcll}
[\psi]\cdot & = & \cdot & (\mathsf{sassempty}) \\
[\psi](\mathbf{x} \in F, \Delta) & = & \mathbf{x} \in F[\psi], [\psi]\Delta & (\mathsf{sasscons})
\end{array}$$

In this definition we use another simple trick in order to facilitate the presentation. Even though assumption list typically grow to the right, we treat them in the definition as if they do grow to the left. Even though not necessary here, this trick makes subsequent definitions structural.

Lemma 6.15 (Substitution lemma for assumptions)
If $\mathcal{D} :: \Psi \vdash \Delta$ *meta*
and $\mathcal{P} :: \Psi' \vdash \psi \in \Psi$
then $\Psi' \vdash [\psi]\Delta$ *meta.*

Proof: by induction on $\mathcal{D}$ using Lemma 6.13. □

Any meta substitution can be extended in such a way that it acts as identity substitution on any domain extension. Note, that the co-domain must be extended accordingly. This lemma is trivially true, but it requires some work and a few generalizations because of the complicated definition of meta-substitutions.

Lemma 6.16 (Identity extension for declarations)

1. *If* $\Psi'' \vdash \psi \in \Psi$
   and $\vdash \Psi, \Psi'$ *meta*
   then $\Psi'', [\psi]\Psi' \vdash \psi, id_{\Psi'} \in \Psi, \Psi'$

2. *If* $\mathcal{D} :: \Psi''; \Delta'' \vdash \psi; \delta \in \Psi; \Delta$
   and $\vdash \Psi, \Psi'$ *meta*
   then $\Psi'', [\psi]\Psi'; \Delta'' \vdash \psi, id_{\Psi'}; \delta \in \Psi, \Psi'; \Delta$

3. *If* $\Psi''; \Delta'' \vdash \psi; \delta \in \Psi; \Delta$
   and $\vdash \Psi, \Psi'$ *meta*
   and $\Psi, \Psi' \vdash \Delta, \Delta'$ *abstract*
   then $\Psi'', [\psi]\Psi'; \Delta'', [\psi, id_{\Psi'}]\Delta' \vdash \psi, id_{\Psi'}; \delta, id_{\Delta'} \in \Psi, \Psi'; \Delta, \Delta'$

Proof: by structural induction on $\Psi'(1)$, $\mathcal{D}(2)$, and $\Delta'(3)$, using Lemma 6.9, Lemma 6.11 (3), and sabstract, and Lemma 6.13. □

Back in Section 6.2.2 we have discussed how to abstract new meta-assumptions. How does abstraction interact with substitution application? Essentially, the answer is a generalization of Lemma 6.14.

Lemma 6.17 (Substitution lemma and abstraction)

$$[\psi](\mathbf{\Pi}\rho^L.\,(\Psi''; \Delta'')) = \mathbf{\Pi}([\psi]\rho)^L.\,([\psi, [\psi]\rho/\rho]\Psi''; [\psi, [\psi]\rho/\rho, id_{\Psi''}]\Delta'')$$

Proof: by structural induction on ρ. □

This concludes our presentation of substitution properties for formulas. We continue the discussion and investigate of how substitutions can be applied to proof terms.

Proof terms

There are two entirely independent notions of substitution application associated with proof terms. First, there is lemma instantiation. Before a program can be executed, we must guarantee that it doesn't contain any free meta-hypotheses. Meta-hypothesis can only be instantiated by general proof terms. Second, there is meta-substitution application which is used for example when applying a proof term to some argument object. The operational semantics we define below immediately carries out substitution application; in doing so, it is different from previous versions of $\mathcal{M}_2^+$ [SP98] where the operational semantics is defined via environments.

The idea behind lemma instantiation has already been explained in Section 5.7. In the following we discuss how it is carried out. A lemma repository consists of free meta-hypotheses.

By instantiating them with closed general proof terms, we can turn a hypothetical into a non-hypothetical meta-proof. Formally, we write $Q[\xi] = Q'$, $P[\xi] = P'$, and $D[\xi] = D'$ to apply the lemma instantiation ξ to a general proof term, proof term, and to a list of declarations, respectively.

General proof terms:

$$
\begin{array}{rcll}
\mathbf{x}[\xi] & = & \xi(\mathbf{x}) & (\mathsf{iHyp}) \\
(\mathbf{box}\ S.\, P)[\xi] & = & \mathbf{box}\ S.\, P & (\mathsf{iCtx})
\end{array}
$$

Proof terms:

$$
\begin{array}{rcll}
\mathsf{iVar})\mathbf{x}[\xi] & = & \mathbf{x} & \\
(\Lambda x : A.\, P)[\xi] & = & \Lambda x : A.\, P[\xi] & (\mathsf{iFun}) \\
(\lambda \rho^L.\, P)[\xi] & = & \lambda \rho^L.\, P[\xi] & (\mathsf{iFunP}) \\
\langle M, P\rangle[\xi] & = & \langle M, P[\xi]\rangle & (\mathsf{iInx}) \\
\langle\rangle[\xi] & = & \langle\rangle & (\mathsf{iUnit}) \\
(\mathbf{let}\ D\ \mathbf{in}\ P)[\xi] & = & \mathbf{let}\ D[\xi]\ \mathbf{in}\ P[\xi] & (\mathsf{iLet}) \\
(\mu \mathbf{x} \in F.\, P)[\xi] & = & \mu \mathbf{x} \in F.\, P[\xi] & (\mathsf{iRec}) \\
\langle P_1, P_2\rangle[\xi] & = & \langle P_1[\xi], P_2[\xi]\rangle & (\mathsf{iPair}) \\
(\text{case } (\psi'; \delta') \text{ of } \Omega)[\xi] & = & \text{case } (\psi'; \delta') \text{ of } \Omega[\xi] & (\mathsf{iCase})
\end{array}
$$

Declarations:

$$
\begin{array}{rcll}
\cdot[\xi] & = & \cdot & (\mathsf{iDone}) \\
(\langle x : A, \mathbf{y} \in F\rangle = P, D)[\xi] & = & \langle x : A, \mathbf{y} \in F\rangle = P[\xi], D[\xi] & (\mathsf{iSplit}) \\
(\mathbf{x} \in F = P\ M, D)[\xi] & = & \mathbf{x} \in F = P[\xi]\ M, D[\xi] & (\mathsf{iApp}) \\
(\mathbf{x} \in F = P\ \rho, D)[\xi] & = & \mathbf{x} \in F = P[\xi]\ \rho, D[\xi] & (\mathsf{iAppP}) \\
(\nu\, \rho^L.\, D)[\xi] & = & \nu\, \rho^L.\, D[\xi] & (\mathsf{iNew}) \\
(\mathbf{x} \in F = \pi_1\ P, D)[\xi] & = & \mathbf{x} \in F = \pi_1\ P[\xi], D[\xi] & (\mathsf{iPil}) \\
(\mathbf{x} \in F = \pi_2\ P, D)[\xi] & = & \mathbf{x} \in F = \pi_2\ P[\xi], D[\xi] & (\mathsf{iPir}) \\
(\mathbf{y} \in F = \text{lemma } Q, D)[\xi] & = & \mathbf{y} \in F = \text{lemma } Q[\xi], D[\xi] & (\mathsf{iLem})
\end{array}
$$

And, as one might already expect, the application of lemma instantiations is sound:

Lemma 6.18 (Soundness of lemmas instantiation)
If $\mathcal{D} :: \Psi; \Delta; \Xi \vdash P \in F$
and $\mathcal{P} :: \Xi' \vdash \xi \in \Xi$
then $\Psi; \Delta; \Xi' \vdash P[\xi] \in F$.

Proof: by induction on $\mathcal{D}$. □

Hypothetical meta-proofs can be turned non-hypothetical by providing general proof terms for each meta-hypothesis. As a matter of fact, all future considerations involving the operational semantics require Ξ to be empty. In particular, only if they are defined with respect to an empty lemma repository, programs P and general programs Q are executable.

In the remainder of this subsection we are concerned with the application of a meta-substitution $\psi; \delta$, which replaces variables in Ψ and meta-assumptions in Δ simultaneously. Meta-substitutions can only be applied to programs and declarations. Clearly, there is no need to apply them to general programs since they are always closed by definition. In addition they need not be applied to cases Ω because of the choice of case subjects; a case subject is an explicit substitution which absorbs all substitution applications by composition while shielding the list of cases Ω from substitution application. Only when a case construct is operationally executed, i.e. one of its cases is selected and matched against (see Section 6.4), the newly derived matching substitution is applied to its body. For the application of a meta-substitution to programs we write $P[\psi;\delta] = P'$ and to declarations $D[\psi;\delta] = D'$. Both judgments are mutual recursive and defined by the following rules.

The construction of $\mathrm{id}_{\Psi'}$ and $\mathrm{id}_{\Delta'}$ in the rule sLet can be easily calculated while applying $\psi;\delta$ to D. In essence, it summarizes all newly introduced assumptions and meta-assumptions of $D[\psi;\delta]$. Alternatively, we could have made the calculation of $\mathrm{id}_{\Psi'}$ and $\mathrm{id}_{\Delta'}$ explicit which would have noticeably cluttered the presentation. Note the use of meta-substitution composition in the rule sCase, as described above. The composition itself is described by Definition 5.19. The subject of case in rule sCase is an explicit substitution, and substituting into a case object reduces to substitution composition.

$$
\begin{array}{rcll}
\mathbf{x}[\psi;\delta] & = & \delta(\mathbf{x}) & (\mathsf{sVar})\\
(\Lambda x : A.\, P)[\psi;\delta] & = & \Lambda x : A[\psi].\, P[\psi, x/x;\delta] & (\mathsf{sFun})\\
(\lambda\rho^L.\, P)[\psi;\delta] & = & \lambda([\psi]\rho)^L.\, P[\psi, [\psi]\rho/\rho;\delta] & (\mathsf{sFunP})\\
\langle M, P\rangle[\psi;\delta] & = & \langle M[\![\psi]\!], P[\psi;\delta]\rangle & (\mathsf{sInx})\\
\langle\rangle[\psi;\delta] & = & \langle\rangle & (\mathsf{sUnit})\\
(\mathbf{let}\ D\ \mathbf{in}\ P)[\psi;\delta] & = & \mathbf{let}\ D[\psi;\delta]\ \mathbf{in}\ P[\psi, \mathrm{id}_{\Psi'};\delta, \mathrm{id}_{\Delta'}] & (\mathsf{sLet})\\
 & & \text{where } \Psi';\Delta' \text{ are newly introduced assumptions by } D & \\
(\mu\mathbf{x} \in F.\, P)[\psi;\delta] & = & \mu\mathbf{x} \in F[\psi].\, P[\psi;\delta, \mathbf{x}/\mathbf{x}] & (\mathsf{sRec})\\
\langle P_1, P_2\rangle[\psi;\delta] & = & \langle P_1[\psi;\delta], P_2[\psi;\delta]\rangle & (\mathsf{sPair})\\
(\text{case } (\psi';\delta') \text{ of } \Omega)[\psi;\delta] & = & \text{case } (\psi';\delta') \circ (\psi;\delta) \text{ of } \Omega & (\mathsf{sCase})\\
\\
\cdot[\psi;\delta] & = & \cdot & (\mathsf{sDone})\\
(\langle x : A, \mathbf{y} \in F\rangle = P, D)[\psi;\delta] & = & (\langle x : A[\psi], \mathbf{y} \in F[\psi, x/x]\rangle = P[\psi;\delta], D[\psi, x/x;\delta, \mathbf{y}/\mathbf{y}]) & (\mathsf{sSplit})\\
(\mathbf{x} \in F = P\ M, D)[\psi;\delta] & = & (\mathbf{x} \in F[\psi] = P[\psi;\delta]\ M[\psi], D[\psi;\delta, \mathbf{x}/\mathbf{x}]) & (\mathsf{sApp})\\
(\mathbf{x} \in F = P\ \rho, D)[\psi;\delta] & = & (\mathbf{x} \in F[\psi] = P[\psi;\delta]\ [\psi]\rho, D[\psi;\delta, \mathbf{x}/\mathbf{x}]) & (\mathsf{sAppP})\\
(\nu\ \rho^L.\, D)[\psi;\delta] & = & \nu\ ([\psi]\rho)^L.\, D[\psi, [\psi]\rho/\rho;\delta] & (\mathsf{sNew})\\
(\mathbf{x} \in F = \pi_1\ P, D)[\psi;\delta] & = & (\mathbf{x} \in F[\psi] = \pi_1\ P[\psi;\delta], D[\psi;\delta, \mathbf{x}/\mathbf{x}]) & (\mathsf{sPil})\\
(\mathbf{x} \in F = \pi_2\ P, D)[\psi;\delta] & = & (\mathbf{x} \in F[\psi] = \pi_2\ P[\psi;\delta], D[\psi;\delta, \mathbf{x}/\mathbf{x}]) & (\mathsf{sPir})\\
(\mathbf{y} \in F = \text{lemma } Q, D)[\psi;\delta] & = & (\mathbf{y} \in F = \text{lemma } Q, D[\psi;\delta, \mathbf{y}/\mathbf{y}]) & (\mathsf{sLem})
\end{array}
$$

Clearly, closely related to the soundness property of meta-substitution application is the soundness of meta-substitution composition. But before we address the formulation of the substitution lemma, we state some very trivial facts on how to access variable blocks and proof terms in a meta-substitutions.

Lemma 6.19 (Lookup)

1. *If* $\mathcal{D} :: \Psi';\Delta' \vdash \psi;\delta \in \Psi;\Delta$
   and $\Delta(\mathbf{x}) = F$
   then $\Psi';\Delta' \vdash \delta(\mathbf{x}) \in F[\psi]$

2. *If* $\mathcal{D} :: \Psi';\Delta' \vdash \psi;\delta \in \Psi;\Delta$
   and $\rho^L \in \Psi$
   then there exists a $\rho'^L \in \Psi'$
   and $[\psi]\rho = \rho'$.

Proof: by induction on $\mathcal{D}(1)$ and $\mathcal{D}(2)$. □

Everything is prepared for the proof of the substitution lemma for proof terms: If a proof term P is well-typed in some meta context $\Psi; \Delta$ and there exists a meta-substitution $\psi; \delta$ with the same domain then it is applicable and $P[\psi; \delta]$ is a well-typed proof term in the meta-logic. Clearly, this property is not directly provable, since we must first generalize the lemma to also apply to declarations and to substitution composition.

With the machinery developed so far at hand, the proof of the generalized substitution lemma is a simple induction on the various typing derivations.

Lemma 6.20 (Substitution lemma for proof-terms)

1. *If* $\mathcal{D} :: \Psi; \Delta \vdash P \in F$
   and $\mathcal{P} :: \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$
   then $\Psi'; \Delta' \vdash P[\psi; \delta] \in F[\psi]$.

2. *If* $\mathcal{D} :: \Psi; \Delta \vdash D \in \Psi''; \Delta''$
   and $\mathcal{P} :: \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$
   then $\Psi'; \Delta' \vdash D[\psi; \delta] \in [\psi](\Psi''; \Delta'')$.

3. *If* $\mathcal{D}_1 :: \Psi_2; \Delta_2 \vdash \psi_1; \delta_1 \in \Psi_1; \Delta_1$
   and $\mathcal{D}_2 :: \Psi_3; \Delta_3 \vdash \psi_2; \delta_2 \in \Psi_2; \Delta_2$
   then $\Psi_3; \Delta_3 \vdash (\psi_1; \delta_1) \circ (\psi_2; \delta_2) \in \Psi_1; \Delta_1$
   and $(\psi_1; \delta_1) \circ (\psi_2; \delta_2) = (\psi_1 \circ \psi_2, \delta')$ *for some meta-substitution* δ'

Proof: by induction on $\mathcal{D}(1), \mathcal{D}(2), \mathcal{D}_1(3)$ using Lemma 6.19, Lemma 6.16, Lemma 5.21, Lemma 6.2, Lemma 6.14, Lemma 6.12, Lemma 6.17, Lemma 6.23, and Lemma 5.18 A detailed proof can be found in Appendix B.1.2. □

The third part of this lemma guarantees that the composition of two meta-substitutions as defined in Definition 5.19 is well-defined.

Corollary 6.21 (Compositions of meta-substitutions)

If $\mathcal{D}_1 :: \Psi_2; \Delta_2 \vdash \psi_1; \delta_1 \in \Psi_1; \Delta_1$
and $\mathcal{D}_2 :: \Psi_3; \Delta_3 \vdash \psi_2; \delta_2 \in \Psi_2; \Delta_2$
then $\Psi_3; \Delta_3 \vdash (\psi_1; \delta_1) \circ (\psi_2; \delta_2) \in \Psi_1; \Delta_1$

Proof: Follows directly from Lemma 6.20. □

The next few lemmas are of technical nature. They summarize simple properties needed for the type preservation proof which are described below. The first of these technical lemmas guarantees the existence of an identity meta-substitution.

Lemma 6.22 (Identity meta-substitution)
If $\vdash \Psi$ *abstract*
and $\Psi \vdash \Delta$ *meta*
then $\Psi; \Delta \vdash id_\Psi; \cdot \in \Psi; \cdot$

Proof: follows directly from the rule sabstract. □

And the second technical lemma is a substitution lemma for variable blocks. In essence, it is a generalization of Lemma 6.2.

Lemma 6.23 (variable blocks convertibility under substitution)
If $\Psi' \vdash \psi \in \Psi$
and $\mathcal{D} :: \Psi \vdash \rho \equiv \rho'$
then $\Psi' \vdash [\psi]\rho \equiv [\psi]\rho'$

Proof: by structural induction over $\mathcal{D}$, using Lemma 6.2. □

This concludes our description of substitution properties for the various syntactical concepts defined in $\mathcal{M}_2^+$. Before we begin with the specification of its operational semantics, we discuss context schema subsumption and matching. Context schema subsumption judges if a lemma is applicable by examining if the regular worlds in which the caller and the callee are defined are compatible. Matching is a technique that selects a case from Ω and effectively applies it. Simultaneously, we provide syntactic criteria for two of the altogether four side conditions of the proof calculus of $\mathcal{M}_2^+$.

6.3 Subsumption

Proof terms can be interpreted as recursive functions and thus appeals to lemmas corresponds to functions calls. This feature is supported by $\mathcal{M}_2^+$ and has been discussed in depth in Section 5.7. But not every proof can apply any lemma; we must first check, if the regular world extensions of the caller and callee are compatible: the context schema of the calling realizer must subsume the context schema of the called realizers, as expressed by side Condition (5.4). In this thesis, we specify a very simply syntactic criterion for context subsumption, and we leave the design of more sophisticated criteria to future work.

In general, subsumption is undecidable. The criterion specified here is expressed in form of a judgment $S_1 \subset S_2$ and two inference rules. S_1 is the context schema of the caller, S_2 the context schema of the callee.

$$\frac{}{\cdot \subset S}\,\mathsf{subempty} \qquad \frac{}{S \subset S}\,\mathsf{subtriv}$$

If a property is to be proven for closed objects then any other lemma can be applied, and if the property is to be proven for open objects then only lemmas can be applied which are defined with exactly the same context schema. Of course, this condition is quite restrictive, but it is powerful enough to allow the formalization of all lemmas we have encountered in Chapter 4 and many more.

Lemma 6.24 (Soundness)
If $\mathcal{D} :: S_1 \subset S_2$
then $[\![S_1]\!] \subset [\![S_2]\!]$

Proof: by case analysis of $\mathcal{D}$. □

Any refinement of the subsumption relation has to satisfy this soundness property. In the next section we present another syntactic criterion, called strictness.

6.4 Matching

One of fundamental operations necessary for executing proof terms is matching. Once the operational semantics encounters a case statement, it must select a case that is applicable. In Section 5.6.2 we have already informally discussed how this operation is executed. In essence we have defined a pattern matching operation, where patterns are expressed by substitutions. A case is applicable, if the pattern matches the current environment.

Having defined pattern-matching for recursive functions in this generality raises the question, how we can be sure that we can decide if a case is applicable or not. This might sound like a small technicality, but it is not! In particular, we cannot allow the body of a case to depend on variables that will not be instantiated by pattern-matching. Informally, we have already addressed this issue in Section 5.6.2 which led us to the side condition (5.3) associated with the alt-rule. A case is only then valid, if all variables that may occur in the body are instantiated by pattern-matching. The side condition (5.3) unfortunately defines only a semantic criterion for which we develop a syntactic criterion on substitutions called *strictness*. Intuitively, a substitution is strict, if each variable from its co-domain occurs in a strict position in the substitution. Strictness extends the pattern condition as defined by Miller [Mil91] in a straightforward way.

Consider for example an execution trace of a recursive function, where the executing machine is deciding if the case $(\Psi' \triangleright \psi \mapsto P) \in \Omega$ is applicable. If η is the current environment, according to Condition (5.3) we can decide if ψ matches η or not. Furthermore, if it matches all variables in Ψ' will be instantiated. If all variables declared in Ψ' occur in ψ in form of a pattern, i.e. each variable is applied to pairwise distinct local parameters only, the substitution is strict since the more general operation of pattern unification is decidable [Mil91, DHKP96].

Unfortunately, in our setting, the substitution ψ is in general not a pattern substitution. As an example consider Example 5.16 (page 129). ψ_2 is not a pattern substitution, because $(E_1\ E_2)$ is not a pattern; and it is not a pattern because E_2 is not a local parameter. As already pointed out by Virga [Vir99], this observation is quite common when one uses higher-order representation techniques. For this reason, the decidability results from [Mil91] are not directly applicable to our setting.

A possible generalization of patterns is already suggested implicitly by Example 5.16. Even though $E_1\ E_2$ is not a pattern on its own, the variables E_1 and E_2 occur elsewhere: specifically, they occur in form of patterns in the object (app (lam $(\lambda x : \text{term}\, T.\, E_1\ x)$) E_2) which is to be substituted for E. Matching this term with any other term will either fail (due to a constant clash), or it will succeed and thereby properly instantiating E_1 and E_2. We call these occurrences of E_1 and E_2 in ψ_2 *strict occurrences*. In the case of success, the non-pattern $E_1\ E_2$ becomes then instantiated, it β-reduces, and the matching algorithm can proceed. In summary, even though ψ_2 contains non-pattern occurrences, it can be seen as such as long as there are other pattern occurrences of the same variable in ψ_2. A constraint mechanism allows us to locally reorder matching goals, in order to guarantee that strict occurrences of variables are matched before non-strict occurrences. This way, we can indeed enforce the decidability of matching, as long as every variable declared in Ψ' occurs in a strict position in ψ.

This section is organized as follows. First, in Section 6.4.1 we introduce an alternative formulation of LF based on the spine calculus inspired by [CP97b]. Using spine notation we introduce a constraint based matching algorithm in Section 6.4.2 and a precise formal definition of strictness (as syntactic criterion for side condition (5.3)) in Section 6.4.3. Following this discussion we demonstrate that the matching algorithm is sound (in Section 6.4.4) and complete (in Section 6.4.5) provided that ψ, the generalized substitution describing a case, is strict with respect to its co-domain. Finally we assess results in Section 6.4.6.

6.4.1 Spine Calculus

One of the main drawbacks of the standard formulation of LF for the purpose of matching and unification is that it is difficult to describe what the head of a term is. Typically, the head is buried under several applications. However, the rules defining unification or matching algorithms depends crucially on the head of a term. For instance, failure due to a constant clash is triggered by examining the head of a term and not its arguments.

Consider an attempt to match two terms (lam $(\lambda x : \text{term } T.\, E\ x)$) and ((app E_1) E_2) (we intentionally insert all typically omitted parentheses). In order to see that these two terms do not unify we have to traverse several applications written as juxtaposition in order to reach the heads of the terms. As simplification, it is conceivable to adopt an alternative formulation of objects, where the head of an atomic object is explicitly exposed, and the arguments are given in form of a spine. Usually, this notation is used informally. "(lam $(\lambda x : \text{term } T.\, E\ x)$)" for example is written in this formulation as "lam $\cdot$ $((\lambda x : \text{term } T.\, E\ x); \textsc{nil})$", where NIL is the empty spine, and "((app E_1) E_2)" is written as "app $\cdot$ $(E_1; E_2; \textsc{nil})$". In this subsection we presuppose the equivalence of the standard and the spine formulation of LF. For a detailed presentation of spines and many proofs, the interested reader is invited to consult [CP97b]. Canonical forms of LF as described in Section 2.4.3 are expressible in spine notation by the following grammar.

$$
\begin{array}{llcl}
\textit{Kinds:} & K & ::= & \text{type} \mid \Pi x : A.\, K \\
\textit{Types:} & A & ::= & a \cdot S \mid \Pi x : A_1.\, A_2 \\
\textit{Objects:} & M & ::= & c \cdot S \mid x \cdot S \mid \lambda x : A.\, M \\
\textit{Spines:} & S & ::= & \textsc{nil} \mid S; M
\end{array}
$$

Intuitively, every canonical form can be easily represented in spine notation, but the inverse does not necessarily hold. The attentive reader might have noticed, that LF terms in spine notation are always in β-normal but not necessarily in η-long form. On the other hand, it is a simple algorithm which transforms a term in spine notation into η-long form. For the remainder of this subsection we assume all objects, types, and kinds in spine notation to be images of canonical forms.

6.4.2 Algorithm

Using spine notation, it is now quite straightforward to devise a matching algorithm modulo constraints. Following [Mil91] we express a matching problem by a state formula, and the matching algorithm is specified by a set of transition rules. As running example throughout this subsection, consider the proof of the diamond Lemma 4.6 for parallel reduction.

What happens if we apply **dia** to the term $((\lambda y : \text{nat}.\, y)\ x)$ and twice to the derivation $\mathcal{D}$ which we define below? Note, that this λ is the one we have introduced in Chapter 2, and not the λ defined by the logical framework. To make this example more concrete, we assume that natural numbers are defined. $((\lambda y : \text{nat}.\, y)\ x)$ is valid term with respect to an assumption list $x :: \text{term nat}, u :: x \stackrel{1}{\Longrightarrow} x$.

$$\mathcal{D} :: \dfrac{\dfrac{\dfrac{}{y \overset{1}{\Longrightarrow} y}\,v}{\lambda y : \mathrm{nat}.\, y \overset{1}{\Longrightarrow} \lambda y : \mathrm{nat}.\, y}\,\mathrm{plam}^v \qquad \dfrac{}{x \overset{1}{\Longrightarrow} x}\,u}{(\lambda y : \mathrm{nat}.\, y)\; x \overset{1}{\Longrightarrow} x}\,\mathrm{pbeta}$$

Eventually, **dia** will terminate and return the common reduct e', and two derivations $\mathcal{R}^l$ and $\mathcal{R}^r$ as the following diagram shows.

$$\begin{tikzcd}
 & (\lambda y : \mathrm{nat}.\, y)\; x \arrow[dl, "\mathcal{D}"'] \arrow[dr, "\mathcal{D}"] & \\
x \arrow[dr, dotted, "\mathcal{R}^l"'] & & x \arrow[dl, dotted, "\mathcal{R}^r"] \\
 & e' &
\end{tikzcd}$$

Not too surprisingly, the result is $e' = x$ and $\mathcal{R}^l = \mathcal{R}^r = u$. In order to understand the subtleties and details of this evaluation, we shift our point of view to LF, and follow the evaluation trace. First we represent the arguments in LF.

$$\begin{array}{rcl}
\ulcorner(\lambda y : \mathrm{nat}.\, y)\; x\urcorner & = & \mathrm{app}\;(\mathrm{lam}\;(\lambda y : \mathrm{nat}.\, y))\;\underline{x} \\[2ex]
\ulcorner \begin{array}{c}\mathcal{D} \\ (\lambda y : \mathrm{nat}.\, y)\; x \overset{1}{\Longrightarrow} x\end{array} \urcorner & = & \mathrm{pbeta}\;(\mathrm{plam}\;(\lambda y : \mathrm{term\ nat}.\, \lambda v : y \overset{1}{\Longrightarrow} y.\, v))\;\underline{u}
\end{array}$$

Once the evaluation of

$$\begin{array}{l}
\mathbf{dia}\;(\mathrm{app}\;(\mathrm{lam}\;(\lambda y : \mathrm{nat}.\, y))\;\underline{x}), \\
\quad \mathrm{pbeta}\;(\mathrm{plam}\;(\lambda y : \mathrm{term\ nat}.\, \lambda v : y \overset{1}{\Longrightarrow} y.\, v))\;\underline{u}, \\
\quad \mathrm{pbeta}\;(\mathrm{plam}\;(\lambda y : \mathrm{term\ nat}.\, \lambda v : y \overset{1}{\Longrightarrow} y.\, v))\;\underline{u})
\end{array}$$

has begun, it immediately invokes the matching algorithm described below. As a matter of fact, with a little insight it is easy to derive from Example 5.16 that the only applicable case is the case containing ψ_2. The other three are not applicable because of constant clashes.

We motivate now how the matching algorithm works. The evaluation takes part in a world that has the following form

$$\Phi = (\underline{x} : \mathrm{term\ nat}, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L.$$

In addition recall from Example 5.13, that the case statement is valid in the generalized context

$$\Psi = T : \mathrm{tp}, E : \mathrm{term}\ T, E^l : \mathrm{term}\ T, E^r : \mathrm{term}\ T, D^l : E \overset{1}{\Longrightarrow} E^l, D^r : E \overset{1}{\Longrightarrow} E^r.$$

Thus, during execution all variables in Ψ become instantiated, and the instantiation is summarized in form of a substitution. As a matter of fact, this substitution is the case subject and takes the rôle of a local environment. For the scope of this section we denote it with η in order not to confuse it with the other substitutions which come up at numerous occasions. We continue to denote the substitution describing a case with ψ. To make this example more concrete, we assume that x has type nat.

$$\begin{aligned}\Phi \vdash \eta = \mathrm{nat}/T, & (\mathrm{app}\ (\mathrm{lam}\ (\lambda y : \mathrm{nat}.\, y))\ \underline{x})/E, \underline{x}/E^l, \underline{x}/E^r,\\ & (\mathrm{pbeta}\ (\mathrm{plam}\ (\lambda y : \mathrm{term}\ \mathrm{nat}.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ \underline{u})/D^l,\\ & (\mathrm{pbeta}\ (\mathrm{plam}\ (\lambda y : \mathrm{term}\ \mathrm{nat}.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ \underline{u})/D^r \in \Psi\end{aligned}$$

Once executed, the operational semantics searches through all cases to find one which is applicable. For the purpose of this example, we consider only two cases from Example 5.16: The first case, which is not applicable, is defined by

$$\psi_1 = T/T, \underline{x}/E, \underline{x}/E^l, \underline{u}/D^l, D^r/D^r$$

and the second case, which is, is defined by

$$\psi_2 = T/T, (\mathrm{app}\ (\mathrm{lam}\ E_1)\ E_2)/E, (E_1\ E_2)/E^l, E^r/E^r, (\mathrm{pbeta}\ D_1^l\ D_2^l)/D^l, D^r/D^r.$$

The three generalized substitutions η, ψ_1, and ψ_2 have all the same domain, but quite different co-domains. The challenge for the matching algorithm is to select an applicable case, i.e. a case whose co-domain variables can be instantiated by another substitution η', in such a way, that the pair ψ, η' is a valid decomposition of the original environment $\psi \circ \eta' = \eta$. Clearly η' is a generalized substitution whose domain is Φ, i.e. it can use the same parameters for instantiations as η, and its domain is the co-domain of ψ. In our example, the matching algorithm must construct a $\Phi \vdash \eta' \in \Psi_2'$, since ψ_2 is the only applicable case. Recall from Example 5.13 that

$$\begin{aligned}\Psi_2' = T : \mathrm{tp}, & T_1 : \mathrm{tp}, E_1 : \mathrm{term}\ T_1 \to \mathrm{term}\ T, E_2\ \mathrm{term}\ T_1,\\ & E_1^l : \mathrm{term}\ T_1 \to \mathrm{term}\ T, E_2^l\ \mathrm{term}\ T_1, E^r : \mathrm{term}\ T,\\ & D_1^l : \Pi x : \mathrm{term}\ T_1.\, x \stackrel{1}{\Longrightarrow} x \to E_1\ x \stackrel{1}{\Longrightarrow} E_1^l\ x, D_2^l : E_2 \stackrel{1}{\Longrightarrow} E_2^l,\\ & D^r : (\mathrm{app}\ (\mathrm{lam}\ E_1)\ E_2) \stackrel{1}{\Longrightarrow} E^r.\end{aligned}$$

and consequently

$$\begin{aligned}\eta' = \mathrm{nat}/T, & \mathrm{nat}/T_1, (\lambda y : \mathrm{nat}.\, y)/E_1, \underline{x}/E_2, (\lambda y : \mathrm{nat}.\, y)/E_1^l, \underline{x}/E_2^l, \underline{x}/E^r,\\ & (\mathrm{plam}\ (\lambda y : \mathrm{term}\ \mathrm{nat}.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))/D_1^l, \underline{u}/D_2^l,\\ & (\mathrm{pbeta}\ (\mathrm{plam}\ (\lambda y : \mathrm{term}\ \mathrm{nat}.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ \underline{u})/D^r\end{aligned}$$

In the remainder of this subsection we present the matching algorithm which computes such an η' if it exists, and it reports *failure*, if it does not. As an example for the later case, consider the generalized substitution ψ_1 from above; the attempt to match

$$\psi_1 = T/T, \underline{x}/E, \underline{x}/E^l, \underline{u}/D^l, D^r/D^r$$

with

$$\begin{aligned}\eta = \mathrm{nat}/T, & (\mathrm{app}\ (\mathrm{lam}\ (\lambda y : \mathrm{nat}.\, y))\ \underline{x})/E, \underline{x}/E^l, \underline{x}/E^r,\\ & (\mathrm{pbeta}\ (\mathrm{plam}\ (\lambda y : \mathrm{term}\ \mathrm{nat}.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ \underline{u})/D^l,\\ & (\mathrm{pbeta}\ (\mathrm{plam}\ (\lambda y : \mathrm{term}\ \mathrm{nat}.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ \underline{u})/D^r\end{aligned}$$

fails because of clashes in two places (indicated by the grey backgrounds), and hence an η' cannot be constructed. Recall, that $\underline{x}$ and $\underline{u}$ are not existential variables. They represent a parameter block and therefore, this case is clearly not applicable.

We begin now with the definition of the matching algorithm which is essentially defined via a transition relation on state formulas E in a very similar way to [Mil91]. State formulas are not to be confused with formulas F of the meta-logic $\mathcal{M}_2^+$ as defined in Section 5.3.1. In the example above, the matching algorithms starts with a state formula $\exists\Psi'. \psi_2 \approx \eta\{\top\}$. The left-hand side of the derivation can mention the existential variables defined in Ψ' whereas the right-hand side is closed with respect to a generalized parameter context Φ. More specifically, we use the notation $\Phi \triangleright E$ to denote a specific state of the matching algorithm.

The matching algorithm begins then to decompose ψ and η in order to match its components. This gives rise to new state formulas, which we call universal state formulas and which we denote with U. Universal state formulas are a conjunction of equations to be solved, equations defined on objects $M_1 \approx M_2$, spines $S_1 \approx S_2$, and types $A_1 \approx A_2$.

The part $\{\top\}$ in the state formula $\exists\Psi'. \psi_2 \approx \eta\{\top\}$ represents the here empty constraint store. Since the matching algorithm postpones goals that lie outside the pattern fragment as constraints, they must be stored in a special place. A list of constraints is simply a list of still to be resolved equations, and is consequently represented as the universal formula $\{U\}$.

$$
\begin{array}{llll}
\text{Universal State formulas:} & U & ::= & (\psi \approx \eta) \wedge U \\
 & & & \mid (\forall\Gamma. M_1 \approx M_2) \wedge U \\
 & & & \mid (\forall\Gamma. S_1 \approx S_2) \wedge U \\
 & & & \mid (\forall\Gamma. A_1 \approx A_2) \wedge U \\
 & & & \mid \top \\
\text{Existential State formulas:} & E & ::= & \exists x : A. E \mid \exists\rho^L. E \mid U_1\{U_2\} \\
\text{State:} & T & ::= & \Phi \triangleright E
\end{array}
$$

The universally quantified context Γ proceeding equations of the form $M_1 \approx M_2$, $S_1 \approx S_2$, and $A_1 \approx A_2$ is used to represent local parameters and hence the universal quantifier carries exactly the same meaning as in [Mil91]. M_1, M_2, S_1, S_2, A_1, and A_2 are all valid in Γ. The Φ never changes throughout the algorithm, but it is necessary since it characterizes all parameters that have been introduced by an extension to the world.

The matching algorithm is expressed by a judgment $T_1 \implies T_2$, which reads as state T_1 is transformed into state T_2. As for standard matching and unification algorithms, these rules are successively applied beginning at an initial state, until a solved state is reached. Overall, this solved state is $\Phi \triangleright \top\{\top\}$ meaning, that all equations and all constraints have been satisfactorily resolved.

Judgment

$$\text{Match state:} \quad T_1 \implies T_2$$

Rules

$$
\begin{array}{lll}
\mathsf{mconst} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, c \cdot S_1 \approx c \cdot S_2) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, S_1 \approx S_2) \wedge\ U_1\{U_2\} \\
\mathsf{mlam} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, \lambda x : A_1.\, M_1 \approx \lambda x : A_2.\, M_2) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, A_1 \approx A_2) \wedge\ (\forall\Gamma, x : A_1.\, M_1 \approx M_2) \wedge\ U_1\{U_2\} \\
\mathsf{mlocal} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, x \cdot S_1 \approx x \cdot S_2) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, S_1 \approx S_2) \wedge\ U_1\{U_2\} \\
& & \text{if } x : A \in \Gamma \\
\mathsf{mglobal} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, x \cdot S_1 \approx \underline{x} \cdot S_2) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, S_1 \approx S_2) \wedge\ U_1\{U_2\} \\
& & \text{if } \underline{x} : A \in \rho \text{ and } \rho^L \in \Phi \\
\mathsf{mpat} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, x \cdot S_1 \approx M) \wedge U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi_1.\,[\lambda x_1 : A_1 \ldots x_n : A_n.\, M/x](\exists\Psi_2.\, U_1\{U_2\}) \\
& & \text{if } \Psi = \Psi_1, x : A, \Psi_2 \text{ and } S_1 = (x_1 \ldots x_n) \text{ pattern} \\
& & \text{and } x_i : A_i \in \Gamma, \text{ for all } 1 \leq i \leq n \\
& & \text{and all free variables in } M \text{ are among } x_1 \ldots x_n \\
\mathsf{mnopat} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, x \cdot S_1 \approx M) \wedge U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\, U_1\{(\forall\Gamma.\, x \cdot S_1 \approx M) \wedge U_2\} \\
& & \text{if } \Psi = \Psi_1, x : A, \Psi_2 \text{ and } S_1 \text{ is not a pattern} \\
\mathsf{mpar} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, \underline{x_i} \cdot S_1 \approx \underline{y_i} \cdot S_2) \wedge U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi_1.\,[\rho'/\rho](\exists\Psi_2.\, A_1 \approx A'_1 \ldots \wedge A_n \approx A'_n \wedge \forall\Gamma.\, S_1 \approx S_2 \wedge U_1\{U_2\}) \\
& & \text{if } \Psi = \Psi_1, \rho^L, \Psi_2 \text{ and } \rho = \underline{x_1} : A_1 \ldots \underline{x_n} : A_n \\
& & \text{and } \Phi = \Phi_1, \rho'^L, \Phi_2 \text{ and } \rho' = \underline{y_1} : A'_1 \ldots \underline{y_n} : A'_n \\
& & \text{and } 1 \leq i \leq n
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{mfam} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, a \cdot S_1 \approx a \cdot S_2) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, S_1 \approx S_2) \wedge\ U_1\{U_2\} \\
\mathsf{mpi} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, \Pi x : A_1.\, A'_1 \approx \Pi x : A_2.\, A'_2) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, A_1 \approx A_2) \wedge\ (\forall\Gamma, x : A_1.\, A'_1 \approx A'_2) \wedge\ U_1\{U_2\}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{mnil} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, \text{NIL} \approx \text{NIL}) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\, U_1\{U_2\} \\
\mathsf{mapp} & :: & \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, M_1; S_1 \approx M_2; S_2) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\,(\forall\Gamma.\, M_1 \approx M_2) \wedge\ (\forall\Gamma.\, S_1 \approx S_2) \wedge\ U_1\{U_2\}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{mempty} & :: & \Phi \triangleright \exists\Psi.\, \cdot \approx \cdot \wedge U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\, U_1\{U_2\} \\
\mathsf{mcons} & :: & \Phi \triangleright \exists\Psi.\,(\psi, M_1/x \approx \eta, M_2/x) \wedge\ U_1\{U_2\} \\
& & \Longrightarrow\ \Phi \triangleright \exists\Psi.\, \psi \approx \eta \wedge\ M_1 \approx M_2 \wedge\ U_1\{U_2\}
\end{array}
$$

Note, that all but three rules are direct reformulations of the pattern-matching (pattern-unification) rules as presented in [Mil91]. First, mglobal is a new rule, because of the presence of parameter blocks, which were not present in Miller's investigation. Second, since we are concerned with a standard matching problem no explicit pruning rule is necessary. Instead, pruning is hard-wired into the mpat-rule. And third, since our matching algorithm is applicable to problems outside the pattern fragment, the mnopat rule is designed to postpone matching goals. If S is not a pattern, the equation $x \cdot S \approx M$ is postponed and added to the constraint store. The reflexive and transitive closure of single transition steps is denoted by $\stackrel{*}{\Longrightarrow}$.

$$\frac{}{T \stackrel{*}{\Longrightarrow} T}\,\textsf{mrefl} \qquad \frac{T_1 \Longrightarrow T_2 \qquad T_2 \stackrel{*}{\Longrightarrow} T_3}{T_1 \stackrel{*}{\Longrightarrow} T_3}\,\textsf{mtrans}$$

Due to the presence of constraints, we are proposing a two-step matching algorithm. First, the matching algorithm is invoked with the initial state until it reaches state $\top\{U\}$ for some universal formula U representing constraints. Second, the matching algorithm starts in $U\{\top\}$ and continues until the solved state $\top\{\top\}$ is reached. Formally, we write $\vdash T$ matchable, if such a sequence of transition steps exists.

$$\frac{T \stackrel{*}{\Longrightarrow} \Phi \triangleright \top\{U\} \qquad \Phi \triangleright U\{\top\} \stackrel{*}{\Longrightarrow} \Phi \triangleright \top\{\top\}}{\vdash T \text{ matchable}}\,\textsf{msuccess}$$

Clearly, the solved state $\top\{\top\}$ does not contain any information about the form of the solution substitution — that is the η' in the example above — but once it is formally derived that $\vdash T$ matchable holds, it is simple to transform its derivation into a matching substitution as we discuss in Section 6.4.4. More general, we say that a state is *solvable* iff there exists a substitution η' which makes all equations contained in the state equal using $\beta\eta$ equality.

Definition 6.25 (Solution)

η' is a solution of $\psi \approx \eta \wedge U$ iff $\psi \circ \eta' = \eta$ and η' is a solution of U

η' is a solution of $(\forall\Gamma.\, M_1 \approx M_2) \wedge U$ iff $M_1[\eta', id_\Gamma] = M_2$ and η' is a solution of U

η' is a solution of $(\forall\Gamma.\, A_1 \approx A_2) \wedge U$ iff $A_1[\eta', id_\Gamma] = A_2$ and η' is a solution of U

η' is a solution of $(\forall\Gamma.\, S_1 \approx S_2) \wedge U$ iff $S_1[\eta', id_\Gamma] = S_2$ and η' is a solution of U

η' is always a solution of $\top$

η' is a solution of $\exists\Psi.\, U_1\{U_2\}$ iff $\Phi \vdash \eta' \in \Psi$ and and η' is a solution of $U_1 \wedge U_2$.

η' is a solution of $\Phi \triangleright E$ iff $\Phi \vdash \eta' \in \Psi$ and and η' is a solution of E.

As we discuss in Section 6.4.4 the matching algorithm is sound, i.e. if $T = \Phi \triangleright \exists\Psi'.\, \psi \approx \eta \wedge \top\{\top\}$ and $\vdash T$ matchable than T is also solvable, but it is not necessarily complete. On the other hand, if we restrict T to be a strict matching problem (see Section 6.4.3), completeness also holds. This is shown in Section 6.4.5.

We return to the example from above and show the matching algorithm in operation. Recall, that we try to match ψ_2 with η. In order to simplify the presentation, we elide the prefix $\Phi \triangleright \exists \Psi_2'$. throughout this exposition.

$$\begin{array}{lcl} T/T, & & \ulcorner\tau\urcorner/T, \\ (\text{app } (\text{lam } E_1)\ E_2)/E, & & (\text{app } (\text{lam } (\lambda y : \ulcorner\tau\urcorner.\, y))\ x)/E, \\ (E_1\ E_2)/E^l, & \approx & x/E^l, \\ E^r/E^r, & & x/E^r, \\ (\text{pbeta } D_1^l\ D_2^l)/D^l, & & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u)/D^l, \\ D^r/D^r & & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u)/D^r \end{array} \quad \{\top\}$$

After repeated applications of mcons, the substitutions is decomposed into several smaller equations.

$$\begin{array}{lrcl} & T & \approx & \ulcorner\tau\urcorner \\ \wedge & (\text{app } (\text{lam } E_1)\ E_2) & \approx & (\text{app } (\text{lam } (\lambda y : \ulcorner\tau\urcorner.\, y))\ x) \\ \wedge & (E_1\ E_2) & \approx & x \\ \wedge & E^r & \approx & x \\ \wedge & (\text{pbeta } D_1^l\ D_2^l) & \approx & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u) \\ \wedge & D^r & \approx & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u)\{\top\} \end{array} \tag{6.2}$$

Starting from top to bottom, each equation is solved. The first equation is removed by mpat with $\ulcorner\tau\urcorner/T$.

$$\begin{array}{lrcl} & (\text{app } (\text{lam } E_1)\ E_2) & \approx & (\text{app } (\text{lam } (\lambda y : \ulcorner\tau\urcorner.\, y))\ x) \\ \wedge & (E_1\ E_2) & \approx & x \\ \wedge & E^r & \approx & x \\ \wedge & (\text{pbeta } D_1^l\ D_2^l) & \approx & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u) \\ \wedge & D^r & \approx & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u)\{\top\} \end{array}$$

Likewise after a few applications of mconst, mnil, mapp — ignoring the spine notation — and mpat, the new first equation is solved yielding $(\lambda y : \ulcorner\tau\urcorner.\, y)/E_1$ and x/E_2 which simplifies the matching problem to

$$\begin{array}{lrcl} & x & \approx & x \\ \wedge & E^r & \approx & x \\ \wedge & (\text{pbeta } D_1^l\ D_2^l) & \approx & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u) \\ \wedge & D^r & \approx & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u)\{\top\}. \end{array} \tag{6.3}$$

One transition of mglobal yields

$$\begin{array}{lrcl} & E^r & \approx & x \\ \wedge & (\text{pbeta } D_1^l\ D_2^l) & \approx & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u) \\ \wedge & D^r & \approx & (\text{pbeta } (\text{plam } (\lambda y : \text{term } \ulcorner\tau\urcorner.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v))\ u)\{\top\}. \end{array}$$

and several applications of mpat, mconst eventually solve the entire matching problem. Note that even though $(E_1\ E_2)$ in state (6.2) is not a pattern the instantiation of E_1 and E_2 in state (6.3) brings it back into the pattern fragment. That's why a solution of the problem is possible without the generation of any constraints. The situation is entirely different if the equations in the state (6.2) are reordered, a scenario which cannot be excluded.

$$\begin{array}{lrcl}
 & T & \approx & \ulcorner\tau\urcorner \\
\wedge & (E_1\ E_2) & \approx & x \\
\wedge & (\text{app}\ (\text{lam}\ E_1)\ E_2) & \approx & (\text{app}\ (\text{lam}\ (\lambda y : \ulcorner\tau\urcorner .\, y))\ x) \\
\wedge & E^r & \approx & x \\
\wedge & (\text{pbeta}\ D_1^l\ D_2^l) & \approx & (\text{pbeta}\ (\text{plam}\ (\lambda y : \text{term}\ \ulcorner\tau\urcorner .\, \lambda v : y \stackrel{1}{\Longrightarrow} y .\, v))\ u) \\
\wedge & D^r & \approx & (\text{pbeta}\ (\text{plam}\ (\lambda y : \text{term}\ \ulcorner\tau\urcorner .\, \lambda v : y \stackrel{1}{\Longrightarrow} y .\, v))\ u)\{\top\}
\end{array}$$

The first equation is solvable by mpat, as above, but the second is not. As a matter of fact, the matching algorithm will postpone it as constraint using the rule mnopat.

$$\begin{array}{lrcl}
 & (\text{app}\ (\text{lam}\ E_1)\ E_2) & \approx & (\text{app}\ (\text{lam}\ (\lambda y : \ulcorner\tau\urcorner .\, y))\ x) \\
\wedge & E^r & \approx & x \\
\wedge & (\text{pbeta}\ D_1^l\ D_2^l) & \approx & (\text{pbeta}\ (\text{plam}\ (\lambda y : \text{term}\ \ulcorner\tau\urcorner .\, \lambda v : y \stackrel{1}{\Longrightarrow} y .\, v))\ u) \\
\wedge & D^r & \approx & (\text{pbeta}\ (\text{plam}\ (\lambda y : \text{term}\ \ulcorner\tau\urcorner .\, \lambda v : y \stackrel{1}{\Longrightarrow} y .\, v))\ u)\{(E_1\ E_2) \approx x\}
\end{array}$$

Eventually, it will continue as above, solving all other equations by successively instantiating existential variables until it arrives in state

$$\top\{x \approx x\}.$$

What is the matching algorithm trying next? Obviously this state is not in solved form because the constraint list not empty. In order to solve it, the algorithm attempts to solve

$$x \approx x\{\top\}$$

and certainly it succeeds by mglobal. Finally, as expected, by msuccess we deduce that the original matching problem is solvable.

Clearly, in the general case, the matching algorithm cannot be complete. This hinges on the fact, that in the second pass, when the matching algorithm attempts to solve the constraints, new constraints might arise. Even though in theory possible, this situation does not come up in any of our examples and experiments. There are at least two ways to rectify this incompleteness. First, one could try to generalize the matching algorithm to a bigger set of matching problems, but the reader should be warned that this is not a simple endeavor: higher-order matching problems only up to third order are known to be decidable [Dow92]. Second, one can restrict the set of matching problems. On the one hand by only considering problems from the pattern fragment is too restrictive as we have seen in this section. Higher-order encodings typically fall out of this fragment. On the other hand, the strict fragment of matching problems that we discuss in the following section accommodates significantly more and for our purposes sufficiently enough matching problems. Below we characterize this strict fragment and show that the matching algorithm restricted to this fragment is decidable, sound, and complete. Unification on the other hand might not be decidable any more.

6.4.3 Strictness

How can we restrict the set of matching problems in order to make the matching algorithm from Section 6.4.2 complete? Recall that the algorithm proceeds in two phases. In the first phase it tries to solve all immediate goals and it postpones all non-pattern goals as constraints. In the second phase it then attempts to solve those constraints one after the other.

In the first phase existential variables are instantiated by the mpat-rule. One of its preconditions is that the variable is the head of a pattern. Its spine must be a list of pairwise different local parameters. In order to simplify this presentation we introduce an abbreviation for pattern spines and write $\Gamma \vdash S$ pattern. If the existential variable does not occur in form of a pattern, the matching goal is postponed as a constraint by the mnopat-rule.

On which problems is the matching algorithm incomplete? The answer is easy. There is absolutely no guarantee that the second phase does not introduce new constraints whose solution would require a third pass. Likewise a fourth or fifth pass might be necessary in order to resolve all constraints. Some constraints can never be resolved.

One way to avoid multiple (more than two) runs of the algorithm is to impose restrictions on the matching problems to be considered. Miller for example has established the pattern restriction on unification problems, which guarantees decidability of pattern matching and pattern unification by enforcing that constraints can never occur. Therefore only *one* pass of the matching algorithm is necessary. In this work, we relax the pattern restriction to *strictness*, where we allow constraints to occur, but we require that after the first pass of the matching algorithm all existential variables are instantiated, which in turn means that after the first pass is completed all constraints are ground as the example in the previous section shows.

For the pattern fragment, it is required that every occurrence of *every* existential variable occurs as the head of a pattern. For the strict fragment, we only require, that there is *at least* one occurrence of every existential variable which occurs as the head of a pattern. Note that this is a dramatic generalization of the pattern fragment. Intuitively matching against this one occurrence is guaranteed to succeed and, as a side effect, all other occurrences of the same variable are instantiated thus removing all non-pattern occurrences of the same variable by the reduction rules defined for LF.

The idea behind the strictness restriction is hence as follows: Consider a case $(\Psi' \triangleright \psi \mapsto P)$ in a list of cases Ω. We say that ψ is strict in Ψ', if every variable $x : A$ occurs as a pattern some place in ψ. Moreover to identify variable blocks, at least one parameter $\underline{x} : A$ of every $(\rho^L) \in \Psi'$ must also occur as pattern somewhere in ψ. Note, that generalized substitutions only allow variable blocks to be replaced by variable blocks, and therefore one strict parameter occurrence of an entire parameter block already signifies a match of the others.

Informally a proof of strictness of ψ exposes the path from the root of a term to a strict occurrence of each variable in Ψ'. This path leads through the substitution ψ, possibly through LF-objects, LF-types, and very likely through LF-spines. By following this path one eventually arrives at a variable occurrence which is guaranteed to satisfy the side condition of the mpat-rule.

Example 6.26 (Strict variable occurrences) Consider the two substitutions from Example 5.16, ψ_1 and ψ_2. The grey backgrounds behind the variables denote strict occurrences. In addition, there are strict occurrences of other variables which occur implicitly in λ-binders or in omitted arguments which we do not show here.

$$\psi_1 = \boxed{T}/T, \boxed{\underline{x}}/E, \boxed{\underline{x}}/E^l, \boxed{\underline{u}}/D^l, \boxed{D^r}/D^r$$

$$\psi_2 = \boxed{T}/T, (\text{app } (\text{lam } \boxed{E_1}) \ \boxed{E_2})/E, (E_1 \ E_2)/E^l, \boxed{E^r}/E^r, (\text{pbeta } \boxed{D_1^l} \ \boxed{D_2^l})/D^l, \boxed{D^r}/D^r$$

In general there is no unique proof that a substitution ψ is strict in its co-domain.

The main consequences of the strictness restriction are that pattern-matching is sound, complete, decidable, and yields sound solutions. That is, for the strict fragment of matching problems we can indeed guarantee that side condition (5.3) is satisfied.

$$\text{For all } \eta \ (\Phi \vdash \eta \in \Psi) \text{ there exists a unique } \eta' \ (\Phi \vdash \eta' \in \Psi') \text{ s.t. } \eta = \psi \circ \eta'$$

Section 6.4.4 and Section 6.4.5 prepare the proof of this result which is summarized in Theorem 6.36. We begin now with the formal presentation of the strictness criterion.

On the top-level we write $\Psi \vdash \psi$ strict in order to express that ψ is strict in Ψ as described above. Top-level strictness is expressed in terms of substitution-level strictness for which we require that a variable x occurs in a strict position in ψ. It is expressed by the judgment $\vdash_x \psi$ strict.

The three judgments defining strict variable occurrences in objects, types, and spines are mutually recursive. Their definition is very similar to [PS99a]. Each of the judgments is defined relative to a context of local parameters Γ. We write $\Gamma \vdash_x M$ strict, $\Gamma \vdash_x A$ strict, and $\Gamma \vdash_x S$ strict for the strict occurrence of a variable x in M, A, and S, respectively.

Judgment

Top-level strictness:	$\Psi_1 \vdash (\Psi_2 \rhd \psi)$ strict
Substitution-level strictness:	$\vdash_x \psi$ strict
Generalized context strictness:	$\vdash_x \Psi$ strict
LF-level strictness for objects	$\Gamma \vdash_x M$ strict
LF-level strictness for types	$\Gamma \vdash_x A$ strict
LF-level strictness for spines	$\Gamma \vdash_x S$ strict

Rules The top-level strictness judgment iterates through all declarations in Ψ, and guarantees that each assumption variable occurs in a strict position in ψ (stass), and at least one parameter declaration of every variable block also has a strict occurrence (stblock). stdone is the base case of the iteration.

$$\frac{}{\cdot \vdash (\Psi \rhd \psi) \ \text{ strict}}\ \textsf{stdone}$$

$$\frac{\Psi_1 \vdash (x : A, \Psi_2 \rhd \psi) \text{ strict} \qquad \vdash_x \psi \text{ strict}}{\Psi_1, x : A \vdash (\Psi_2 \rhd \psi) \text{ strict}}\ \textsf{stass}$$

$$\frac{\Psi_1 \vdash (x : A, \Psi_2 \rhd \psi) \text{ strict} \qquad \vdash_x \Psi_2 \text{ strict}}{\Psi_1, x : A \vdash (\Psi_2 \rhd \psi) \text{ strict}}\ \textsf{stass}'$$

$$\frac{(\underline{x} : A) \in \rho \qquad \Psi_1 \vdash (\rho^L, \Psi_2 \rhd \psi) \text{ strict} \qquad \vdash_{\underline{x}} \psi \text{ strict}}{\Psi_1, \rho^L \vdash (\Psi_2 \rhd \psi) \text{ strict}}\ \textsf{stblock}$$

$$\frac{(\underline{x} : A) \in \rho \qquad \Psi_1 \vdash (\rho^L, \Psi_2 \rhd \psi) \text{ strict} \qquad \vdash_{\underline{x}} \Psi_2 \text{ strict}}{\Psi_1, \rho^L \vdash (\Psi_2 \rhd \psi) \text{ strict}}\ \textsf{stblock}'$$

Each deduction of the remaining four judgments witnesses a strict occurrence of a variable x indexing the judgment. x occurs in a strict position in a substitution if it either occurs in a declaration of the form M/x (stsubassyes) or ρ'/ρ (stsubblockyes).

$$\frac{\cdot \vdash_x M \text{ strict}}{\vdash_x \psi, M/y \text{ strict}}\,\textsf{stsubassyes} \qquad \frac{\vdash_x \psi \text{ strict}}{\vdash_x \psi, M/y \text{ strict}}\,\textsf{stsubassno}$$

$$\frac{(\underline{x} : A) \in \rho'}{\vdash_{\underline{x}} \psi, \rho'/\rho \text{ strict}}\,\textsf{stsubblockyes} \qquad \frac{\vdash_x \psi \text{ strict}}{\vdash_x \psi, \rho'/\rho \text{ strict}}\,\textsf{stsubblockno}$$

x occurs in a strict position in a generalized context if it occurs in the type of some declaration.

$$\frac{\cdot \vdash_x A \text{ strict}}{\vdash_x \Psi, y : A \text{ strict}}\,\textsf{stctxassyes} \qquad \frac{\vdash_x \Psi \text{ strict}}{\vdash_x \Psi, y : A \text{ strict}}\,\textsf{stctxassno}$$

$$\frac{(\underline{x} : A) \in \rho \qquad \cdot \vdash_x A \text{ strict}}{\vdash_{\underline{x}} \Psi, \rho^L \text{ strict}}\,\textsf{stctxblockyes} \qquad \frac{\vdash_x \Psi \text{ strict}}{\vdash_x \Psi, \rho^L \text{ strict}}\,\textsf{stctxblockno}$$

The inference rules defining the remaining three judgments are all mutual recursive. x is a strict occurrence in an object, if it is either the head of a pattern (stocc) or if it occurs in the spine as argument to a constant (stconst) or to a local parameter (stlocal). As expected, x is a strict occurrence in a λ-term, if it occurs strictly in either the binder (stlamdec) or the body (stlambody).

$$\frac{}{\Gamma \vdash_x x \cdot S \text{ strict}}\,\textsf{stocc} \qquad \text{Side condition: } \Gamma \vdash S \ \text{ pattern}$$

$$\frac{\Gamma \vdash_x S \text{ strict}}{\Gamma \vdash_x c \cdot S \text{ strict}}\,\textsf{stconst} \qquad \frac{y : A \in \Gamma \qquad \Gamma \vdash_x S \text{ strict}}{\Gamma \vdash_x y \cdot S \text{ strict}}\,\textsf{stlocal}$$

$$\frac{\Gamma \vdash_x A \text{ strict}}{\Gamma \vdash_x \lambda y : A.\, M \text{ strict}}\,\textsf{stlamdec} \qquad \frac{\Gamma, y : A \vdash_x M \text{ strict}}{\Gamma \vdash_x \lambda y : A.\, M \text{ strict}}\,\textsf{stlambody}$$

On the type level, a variable x occurs strictly in an atomic type if it occurs in any of its arguments (stfam). Likewise it occurs strictly in a Π-type, if it either occurs strictly in its binder (stpidec) or its body (stlambody).

$$\frac{\Gamma \vdash_x S \text{ strict}}{\Gamma \vdash_x a \cdot S \text{ strict}}\,\textsf{stfam}$$

$$\frac{\Gamma \vdash_x A_1 \text{ strict}}{\Gamma \vdash_x \Pi y : A_1.\, A_2 \text{ strict}}\,\textsf{stpidec} \qquad \frac{\Gamma, y : A_1 \vdash_x A_2 \text{ strict}}{\Gamma \vdash_x \Pi y : A_1.\, A_2 \text{ strict}}\,\textsf{stlambody}$$

Finally, a variable x occurs strictly in a spine, if it occurs strictly in at least one of its arguments (stthis).

$$\frac{\Gamma \vdash_x M \text{ strict}}{\Gamma \vdash_x M; S \text{ strict}}\,\textsf{stthis} \qquad \frac{\Gamma \vdash_x S \text{ strict}}{\Gamma \vdash_x M; S \text{ strict}}\,\textsf{stnext}$$

One of the main results of this section is that if we require the ψ in the alt-rule from Section 5.6 to be strict in its co-domain then it automatically satisfies side condition (5.3). The argumentation rests on the observation that any matching problem of a strict ψ against a ground η (which as usual might be open with respect to a well formed parameter context Φ) is decidable.

In the remainder of this section we give a detailed account of this argument, but first we have to generalize strictness to matching problems. Informally, strictness is also a property of a state formula which is preserved during execution as we show in Section 6.4.5. We say, that a matching problem $\Phi \triangleright \exists \Psi. U_1\{U_2\}$ is *strict* iff U_1 is strict in Ψ which we denote with the judgment $\Psi \vdash U_1$ strict.

Judgments

Top-level strictness for universal formulas:	$\Psi_1 \vdash (\Psi_2 \triangleright U)$ strict
Universal formula-level strictness:	$\vdash_x U$ strict

Rules The top-level judgment serves as iterator, iterating through all assumption variables (ustass) and variable blocks (ustblock) in Ψ and ensuring that each declaration has at least one strict occurrence in any of the left-hand sides of one of the possibly many equations in U. Recall that the left hand sides of any equation $\psi \approx \eta$ or $M_1 \approx M_2$ in U may contain free existential variables still to be matched whereas the right hand sides are ground.

$$\frac{}{\cdot \vdash (\Psi_2 \triangleright U)\ \text{ strict}}\ \mathsf{ustdone}$$

$$\frac{\Psi_1 \vdash (x : A, \Psi_2 \triangleright U)\ \text{strict} \qquad \vdash_x U\ \text{strict}}{\Psi_1, x : A \vdash (\Psi_2 \triangleright U)\ \text{strict}}\ \mathsf{ustass}$$

$$\frac{\Psi_1 \vdash (x : A, \Psi_2 \triangleright U)\ \text{strict} \qquad \vdash_x \Psi_2\ \text{strict}}{\Psi_1, x : A \vdash (\Psi_2 \triangleright U)\ \text{strict}}\ \mathsf{ustass'}$$

$$\frac{(\underline{x} : A) \in \rho \qquad \Psi_1 \vdash (\rho^L, \Psi_2 \triangleright U)\ \text{strict} \qquad \vdash_{\underline{x}} U\ \text{strict}}{\Psi_1, \rho^L \vdash (\Psi_2 \triangleright U)\ \text{strict}}\ \mathsf{ustblock}$$

$$\frac{(\underline{x} : A) \in \rho \qquad \Psi_1 \vdash (\rho^L, \Psi_2 \triangleright U)\ \text{strict} \qquad \vdash_{\underline{x}} \Psi_2\ \text{strict}}{\Psi_1, \rho^L \vdash (\Psi_2 \triangleright U)\ \text{strict}}\ \mathsf{ustblock'}$$

The second strictness judgment for universal state formulas is indexed by a variable x. It is derivable if x occurs in a strict position in any of the left hand sides of the equations contained in U.

$$\frac{\vdash_x \psi \text{ strict}}{\vdash_x \psi \approx \eta \wedge U \text{ strict}}\,\mathsf{ustsubl} \qquad \frac{\vdash_x U \text{ strict}}{\vdash_x \psi \approx \eta \wedge U \text{ strict}}\,\mathsf{ustsubr}$$

$$\frac{\Gamma \vdash_x M_1 \text{ strict}}{\vdash_x (\forall\Gamma.\, M_1 \approx M_2) \wedge U \text{ strict}}\,\mathsf{ustobjl} \qquad \frac{\vdash_x U \text{ strict}}{\vdash_x (\forall\Gamma.\, M_1 \approx M_2) \wedge U \text{ strict}}\,\mathsf{ustobjr}$$

$$\frac{\Gamma \vdash_x A_1 \text{ strict}}{\vdash_x (\forall\Gamma.\, A_1 \approx A_2) \wedge U \text{ strict}}\,\mathsf{usttypel} \qquad \frac{\vdash_x U \text{ strict}}{\vdash_x (\forall\Gamma.\, A_1 \approx A_2) \wedge U \text{ strict}}\,\mathsf{usttyper}$$

$$\frac{\Gamma \vdash_x S_1 \text{ strict}}{\vdash_x (\forall\Gamma.\, S_1 \approx S_2) \wedge U \text{ strict}}\,\mathsf{ustspinel} \qquad \frac{\vdash_x U \text{ strict}}{\vdash_x (\forall\Gamma.\, S_1 \approx S_2) \wedge U \text{ strict}}\,\mathsf{ustspiner}$$

The generalization of the strictness predicate to state formulas is straightforward. Moreover, no additional information is involved in this construction. In particular, we show that the initial state $\Phi \triangleright \exists\Psi.\,(\psi \approx \eta)$ constructed from a strict ψ is also strict in Ψ. Thus if a case proof term in $\mathcal{M}_2^+$ satisfies the side condition (5.3), the initial state of the matching process is also strict.

Lemma 6.27
If $\mathcal{D} :: \Psi' \vdash \psi$ strict
and $\Phi \vdash \eta \in \Psi'$
then $\Psi' \vdash (\psi \approx \eta \wedge True)$ strict

Proof: by induction on $\mathcal{D}$. □

In Section 6.4.5 we show as part of the completeness argument that strictness is preserved throughout the run of the matching algorithm. The main result of this section is that strict matching problems are decidable with the matching algorithm from Section 6.4.2 being the decision procedure. It is this observation which justifies the choice of strictness to warrant side condition (5.3).

6.4.4 Soundness

The matching algorithm is a sound procedure for all strict matching problems. As a matter of fact, the result is more general. It applies also to matching problems which are not necessarily strict. Given a successful trace of the matching algorithm, the deduction of $\vdash T$ matchable contains enough information to extract the desired matching substitution.

Recall that the matching algorithm proceeds in two phases. Informally, the first phase starts with the matching problem $\psi \approx \eta$ and terminates in a state $\top\{U\}$, where U is a list (or better conjunction) of all constraints postponed during the run. In the second phase the algorithm solves all constraints until it reaches $\top\{\top\}$ as final state. In order to show soundness we extract from these two traces the matching substitution η', which satisfies $\psi \circ \eta' = \eta$.

The soundness argument is presented in three steps. First, we show that if the matching algorithm makes one step from state T_1 to state T_2, any matching substitution η_2 for T_2 can be extended in a unique way to a matching substitution for T_1. Clearly, by applying this argument successively, we can generalize this result to T_1 and T_2 being several steps apart.

Lemma 6.28 (Solution preservation)

1. *If* $\Phi \triangleright \exists\Psi.\, U_1\{U_2\} \implies \Phi \triangleright \exists\Psi'.\, U_1'\{U_2'\}$
and η' $(\Phi \vdash \eta' \in \Psi')$ *is a solution of* $\exists\Psi'.\, U_1'\{U_2'\}$
then there exists a unique η $(\Phi \vdash \eta \in \Psi)$ *which is a solution of* $\exists\Psi.\, U_1\{U_2\}$

2. *If* $\mathcal{D} :: \Phi \triangleright \exists\Psi.\, U_1\{U_2\} \overset{*}{\implies} \Phi \triangleright \exists\Psi'.\, U_1'\{U_2'\}$
and η' $(\Phi \vdash \eta' \in \Psi')$ *is a solution of* $\exists\Psi'.\, U_1'\{U_2'\}$
then there exists a unique η $(\Phi \vdash \eta \in \Psi)$ *which is a solution of* $\exists\Psi.\, U_1\{U_2\}$

Proof: 1. direct by inspection of the rules, and 2. by induction on $\mathcal{D}$. □

In order to apply the local soundness lemma, we must know, that there exists a matching substitution for the termination state $\top\{\top\}$. And indeed, not very surprisingly, there is one namely the empty substitution.

Lemma 6.29 (Initial soundness)
The substitution $\cdot$ *is a trivial solution of* $\top\{\top\}$.

Proof: follows directly from Definition 6.25. □

From the two traces of the matching algorithm we construct a solution of the original matching problem in the following way. Starting with the second trace, and the lemma about initial soundness, there must be a solution of $\top\{U\}$ by the local soundness lemma. By definition this solution is also a solution for $U\{\top\}$. Another application of the local soundness lemma immediately results in a matching substitution for the original matching problem.

Lemma 6.30 (Soundness)
If $\mathcal{D} :: \vdash \Phi \triangleright \exists\Psi'.\, \psi \approx \eta\{\top\}$ *matchable*
then there exists a unique η', $\Phi \vdash \eta' \in \Psi'$ *and* $\psi \circ \eta' = \eta$

Proof: direct by Lemma 6.29 and Lemma 6.28. A detailed proof can be found in Appendix B.2. □

In summary, it follows that the matching algorithm is sound by extracting the matching substitution from the trace in a right to left fashion. The completeness result of the matching algorithm for strict matching problems is discussed next.

6.4.5 Completeness

The matching algorithm from Section 6.4.2 is sound for strict matching problems as shown in the previous section. Whenever the algorithm terminates and reports *yes*, there is indeed a solution for the initially posed matching problem. In this section we show, that we can also trust the answers of the matching algorithm. In particular, given a solvable problem, the algorithm will terminate and report *yes*. Moreover, we show that the matching algorithm always terminates, which makes it an appropriate decision procedure.

We begin this technical discussion with the definition of a particular well-founded ordering, which guarantees that the matching algorithm always terminates. A state of the matching algorithm is formally defined as $\Phi \triangleright \exists\Psi.\, U_1\{U_2\}$ for two universal state formulas U_1 and U_2. As the matching algorithm progresses, it either instantiates variables of Ψ, or it decomposes the left hand side of some equation in U_1. As usual, we write $|\Psi|$ for the length of Ψ and we define $|U_1|$ as the sum of all LF-subobjects of all left hand sides of equations in U_1 where we count $|\underline{x}|$ as 1+ the number of all LF subobjects of all types of that ρ the parameter variable $\underline{x}$ is defined in. Clearly, $0 \leq |\Psi|$ and $0 \leq |U_1|$.

As termination ordering for the matching algorithm, we choose the lexicographic ordering on pairs of non-negative integer numbers $(|\Psi|, |U_1|)$. Recall, that the lexicographic ordering is defined as follows.

$$(n_1, m_1) <_{\text{lex}} (n_2, m_2) \quad \text{iff} \quad n_1 < n_2 \text{ or } (n_1 = n_2 \text{ and } m_1 < m_2)$$

The *measure* of a state $\Phi \triangleright \exists\Psi.\, U_1\{U_2\}$ is defined to be the pair $(|\Psi|, |U_1|)$. Back to the completeness argument. It hinges on the fact, that there is a transition for every state whose measure is different from $(n, 0)$. The algorithm terminates if $U_1 = \top$ or — in measures — if $|U_1| = 0$. Can Ψ still contain existential variable declarations once the matching algorithm comes to a halt with $U_1 = \top$? The answer is no, and the reason is deeply connected with the strictness requirement. If we can guarantee (and we can!) that strictness is preserved during the execution of the algorithm, the final state $\Phi \triangleright \exists\Psi.\, \top\{U_2\}$ must still be strict, and it hence follows by inversion that $\Psi = \cdot$. If Ψ is not empty, there must be at least one strict occurrence of in variable in $\top$ which cannot be the case. As invariant, we infer that each strict state satisfies $|\Psi| \leq |U_1|$.

Therefore, and because the ordering is well-founded the algorithm terminates in a state whose measure is equal to $(0, 0)$ or it reports *failure*. Which state can this be? It must be $\Phi \triangleright \top\{U_2\}$ for any U_2.

Lemma 6.31 (Measure)

1. *If* $T = \Phi \triangleright \exists\Psi.\, U_1\{U_2\}$
   and $\mathcal{D} :: \Psi \vdash U_1$ *strict*
   then $|\Psi| \leq |U_1|$

2. *If* $T = \Phi \triangleright \exists\Psi.\, U_1\{U_2\}$ *is given*
   and $|\Psi| = 0$ *and* $|U_1| = 0$
   then $T = \Phi \triangleright \top\{U_2\}$

Proof: 1. by induction on $\mathcal{D}$, 2. by definition of $|U|$. Every other syntactical construction for U has at least one LF subobject. □

We start now with the discussion of the completeness proof itself. It is split into two parts according to the two phases of the matching algorithm. Assume, that the there is a solution for the initial state T. First we prove, that if $T \neq \Phi \triangleright \top\{U_2\}$ then the matching algorithm can perform another step. In particular, this step preserves strictness and it reduces the measure of a state. A slight generalization reveals, that when running it on any strict and solvable initial state, the matching algorithm eventually terminates in $\Phi \triangleright \top\{U\}$ for some U. All existential variables are instantiated, and therefore U must be ground (with respect to Φ, naturally).

Lemma 6.32 (Completeness I)

1. *If* $U_1 \neq \top$
   and $\Phi \triangleright \exists \Psi.\, U_1\{U_2\}$ *is given*
   and η $(\Phi \vdash \eta \in \Psi)$ *is a solution of* $\exists \Psi.\, U_1\{U_2\}$
   and $\Psi \vdash U_1$ *strict*
   then $\Phi \triangleright \exists \Psi.\, U_1\{U_2\} \implies \Phi \triangleright \exists \Psi'.\, U_1'\{U_2'\}$
   and there exists an η' $(\Phi \vdash \eta' \in \Psi')$ *which is a solution of* $\exists \Psi'.\, U_1'\{U_2'\}$
   and $\Psi' \vdash U_1'$ *strict*
   and $(|\Psi'|, |U_1'|) <_{lex} (|\Psi|, |U_1|)$.

2. *If* $T = \Phi \triangleright \exists \Psi.\, U_1\{U_2\}$ *is given*
   and $\Psi \vdash U_1$ *strict*
   then $T \overset{*}{\implies} \Phi \triangleright \top\{U\}$ *for some* U.

Proof: 1. by inspection of the rules, 2. by induction on $(|\Psi|, |U_1|)$ using Lemma 6.31. A detailed proof can be found in Appendix B.2. □

In the second phase, we start the matching algorithm on $\Phi \triangleright U\{\top\}$, where the U is the list of constraints resulting from the first phase. As already noted, U is ground, i.e. it does not contain any free existential variables. Informally, U is nothing else but a set of equations to be checked for convertibility.

How can we convince ourselves that the algorithm terminates in $\top\{\top\}$? First, note that some of the rules defining the matching algorithm can never apply. mpat and mnopat for example can never be applied because there are no existential variables. Therefore, the set of constraints never changes and thus it is remains empty ($= \top$) during the entire second phase. That U eventually ends up being empty, too, follows by the same argument used for the first completeness lemma.

Lemma 6.33 (Completeness II)

1. *If* $U \neq \top$
   and $\Phi \triangleright U\{\top\}$ *is given*
   and $\cdot$ $(\Phi \vdash \cdot \in \cdot)$ *is a trivial solution for* $U\{\top\}$
   then $\Phi \triangleright U\{\top\} \implies \Phi \triangleright U'\{\top\}$
   and $\cdot$ $(\Phi \vdash \cdot \in \cdot)$ *is a trivial solution for* $U'\{\top\}$
   and $|U'| < |U|$.

2. *If* $T = \Phi \triangleright U\{\top\}$ *is given matching state*
   then $\Phi \triangleright U\{\top\} \overset{*}{\implies} \Phi \triangleright \top\{\top\}$

Proof: 1. by inspection of the rules, 2. by induction on $|U|$ using Lemma 6.31. A detailed proof can be found in Appendix B.2. □

An easy combination of these two previous results yields the completeness lemma. If a strict matching problem has a solution, the matching algorithm eventually terminates and reports *yes*.

Theorem 6.34 (Completeness)
If $T = \Phi \triangleright \exists \Psi.\, U_1\{U_2\}$
and $\mathcal{D} :: \Psi \vdash U_1$ *strict*
and η $(\Phi \vdash \eta \in \Psi)$ *is a solution of* $\exists \Psi.\, U_1\{U_2\}$
then $\vdash T$ *matchable*

Proof: direct A detailed proof can be found in Appendix B.2. □

As a side result of the completeness argument follows the termination property of the matching algorithm. Each run terminates, because the measure of the successive states strictly decreases. Since the termination ordering is well-founded, the matching algorithm must come to a final state. If it is $\top\{\top\}$ it reports *yes*, if not it reports *no*.

Corollary 6.35 (Termination)
Any sequence of $\Longrightarrow$ -reduction steps is finite.

In summary we have shown that matching is sound, complete, and decidable, provided that the matching problem is strict. The solution that matching computes is unique. Therefore side condition (5.3) follows if we require the case defining ψ from rule alt to be strict.

Theorem 6.36 (Determinacy)
If $\mathcal{D} :: \Psi' \vdash \psi \in \Psi$
and $\mathcal{E} :: \Psi' \vdash \psi$ strict
and $\mathcal{F} :: \Phi \vdash \eta \in \Psi$
then there exists a (unique) η' $(\Phi \vdash \eta' \in \Psi')$ s.t. $\psi \circ \eta' = \eta$
or not.

Proof: direct. A detailed proof can be found in Appendix B.2. □

In conclusion, we stipulate

$$\Psi' \vdash \psi \text{ strict}$$

as syntactic criterion for side condition (5.3).

6.4.6 Results

The main difficulty in designing a matching algorithm for $\mathcal{M}_2^+$ lies in the fact that in general matching for LF is not known to be decidable. It is only known to be decidable on fragments, as for example the pattern fragment defined by Miller. But unfortunately, this fragment is too weak for our purposes. Already the matching problems associated with our examples lie outside the pattern fragment.

As solution to this problem we characterize an extension of the pattern fragment which we call the *strict* fragment. The main result is, that strict matching problems are decidable and yield unique matching substitutions. The matching algorithm defined in this section is the appropriate decision procedure. Strict unification problems for the strict fragment on the other hand might not be decidable. We leave a further investigation to future research.

6.5 Big-Step Semantics

There are several ways to show the consistency of a logic. As already motivated earlier the way we have chosen to show the consistency of $\mathcal{M}_2^+$ is to assign an operational meaning to its proof terms and to show that each proof term corresponds to a total function — a realizer. In this section, we define such an operational semantics. In particular, it is a big-step semantics which we refine in the next chapter to a small-step semantics. Why defining two different semantics?

The big-step semantics is only relates proof terms with the result of their computations, it is easier to describe, and therefore it is more accessible than the small-step semantics. Nevertheless, for the soundness proof of $\mathcal{M}_2^+$, we need the small-step semantics. If a proof term cannot be related to any result of a computation, be it because the function does not terminate, or because its evaluation gets stuck, the big-step operational semantics is not fine-grained enough to support any further investigation. Based on a state transition machine very similar to the CPM machine [Pfe00], the small-step semantics on the other hand allows us to express properties such as termination and progress. We leave this discussion entirely to the next chapter.

The operational semantics assigns a computational interpretation to the proof terms of the meta-logic $\mathcal{M}_2^+$. Any proof of any theorem can calculate from any well-typed input (a set of objects instantiating the universal quantified variables) a well-typed output (a set of objects which witness the existential quantifiers). The input and the output arguments are possibly open with respect to a given regular world extension Φ. Therefore we index the evaluation judgment by Φ.

The operational semantics itself is defined with respect to three different judgments, one for proof terms P, one for declarations D, and a last one for cases Ω. Proof terms are evaluated via the judgment $\Phi \vdash P \hookrightarrow V$, where we denote the outcome of the evaluation by V. V is a proof term itself, but in addition it is a value which cannot be evaluated any further. Values are potentially open with respect to the regular world extension.

$$\textit{Values} \quad V \quad ::= \quad \langle\rangle \mid \langle M, V\rangle \mid \Lambda x : A.\, P \mid \lambda\rho^L.\, P \mid \langle V_1, V_2\rangle$$

Declarations declare extensions of meta contexts $\Psi; \Delta$ in which they are defined. Recall from rule sel, that these context extensions are denoted by $\Psi'; \Delta'$. The attentive reader might already suspect, that the result of evaluating a list of declarations, results in a list of LF objects for Ψ' and proof terms for Δ'. The vehicle we like to use in order to express these resulting terms are substitution extensions denoted by $(\psi; \delta)$. Formally, we write $\Phi \vdash D \hookrightarrow (\psi; \delta)$ for the evaluation relation of declarations. Finally, we need to evaluate cases. A case in Ω is triggered if it matches the current "environment" — the explicit substitution as case subject — $\psi; \delta$ using the matching algorithm defined in Section 6.4.2. Case analysis is expressed by the judgment $\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V$. And again V denotes a proof term.

Judgments

Evaluation of programs	$\Phi \vdash P \hookrightarrow V$
Evaluation of declarations	$\Phi \vdash D \hookrightarrow \eta; \delta$
Selection	$\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V$

Rules Evaluation, assumption and selection obey the laws of a call-by-value semantics. The only non-standard rules are ev_let and ev_case. ev_let evaluates first the list of declarations and obtains a meta-substitution extension $\psi; \delta$, which is in turn applied to the body of the let construct. The ev_case rule selects a matching case by invoking selection.

$$\frac{}{\Phi \vdash \Lambda x : A.\, P \hookrightarrow \Lambda x : A.\, P}\ \textsf{ev\_Lam} \qquad \frac{}{\Phi \vdash \lambda\rho^L.\, P \hookrightarrow \lambda\rho^L.\, P}\ \textsf{ev\_lam}$$

$$\frac{\Phi \vdash P \hookrightarrow V}{\Phi \vdash \langle M, P\rangle \hookrightarrow \langle M, V\rangle}\ \textsf{ev\_inx} \qquad \frac{\Phi \vdash P_1 \hookrightarrow V_1 \quad \Phi \vdash P_2 \hookrightarrow V_2}{\Phi \vdash \langle P_1, P_2\rangle \hookrightarrow \langle V_1, V_2\rangle}\ \textsf{ev\_pair} \qquad \frac{}{\Phi \vdash \langle\rangle \hookrightarrow \langle\rangle}\ \textsf{ev\_unit}$$

$$\frac{\Phi \vdash D \hookrightarrow \psi;\delta \qquad \Phi \vdash P[\mathrm{id}_\Phi, \psi;\delta] \hookrightarrow V}{\Phi \vdash \mathbf{let}\ D\ \mathbf{in}\ P \hookrightarrow V}\ \mathsf{ev\_let}$$

$$\frac{\Phi \vdash P[\mu\mathbf{x} \in F.\, P/\mathbf{x}] \hookrightarrow V}{\Phi \vdash \mu\mathbf{x} \in F.\, P \hookrightarrow V}\ \mathsf{ev\_rec} \qquad \frac{\Phi \vdash (\psi;\delta) \sim \Omega \hookrightarrow V}{\Phi \vdash \mathrm{case}\ (\psi;\delta)\ \mathrm{of}\ \Omega \hookrightarrow V}\ \mathsf{ev\_case}$$

The evaluation rules for declarations return meta-substitution extensions $(\psi';\delta')$. The formulation of these rules is are non-standard since they follow the provability rules of $\mathcal{M}_2^+$ in Section 5.4.4. The definition of ev_new uses meta-substitution abstraction as defined in Section 6.6.

$$\frac{}{\Phi \vdash \cdot \hookrightarrow (\cdot;\cdot)}\ \mathsf{ev\_empty}$$

$$\frac{\Phi \vdash P \hookrightarrow \langle M, V\rangle \qquad \Phi \vdash D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}] \hookrightarrow (\psi';\delta')}{\Phi \vdash (\langle x : A, \mathbf{y} \in F\rangle = P, D) \hookrightarrow (M/x, \psi'; V/\mathbf{y}, \delta')}\ \mathsf{ev\_split}$$

$$\frac{\Phi \vdash P \hookrightarrow \Lambda x : A.\, P' \qquad \Phi \vdash P'[\mathrm{id}_\Phi, M/x] \hookrightarrow V \qquad \Phi \vdash D[V/\mathbf{y}] \hookrightarrow (\psi';\delta')}{\Phi \vdash (\mathbf{y} \in F = P\ M, D) \hookrightarrow (\psi'; V/\mathbf{y}, \delta')}\ \mathsf{ev\_App}$$

$$\frac{\Phi \vdash P \hookrightarrow \lambda\rho'^L.\, P' \qquad \Phi \vdash P'[\mathrm{id}_\Phi, \rho/\rho'] \hookrightarrow V \qquad \Phi \vdash D[V/\mathbf{y}] \hookrightarrow (\psi';\delta')}{\Phi \vdash (\mathbf{y} \in F = P\ \rho, D) \hookrightarrow (\psi'; V/\mathbf{y}, \delta')}\ \mathsf{ev\_app}$$

$$\frac{\Phi, \rho^L \vdash D \hookrightarrow (\psi';\delta')}{\Phi \vdash \nu\ \rho^L.\, D \hookrightarrow (\lambda\rho^L.\, (\psi';\delta'))}\ \mathsf{ev\_new}$$

$$\frac{\Phi \vdash P \hookrightarrow \langle V_1, V_2\rangle \qquad \Phi \vdash D[V_1/\mathbf{x}] \hookrightarrow (\psi';\delta')}{\Phi \vdash \mathbf{x} \in F_1 = \pi_1\ P, D \hookrightarrow (\psi'; V_1/\mathbf{x}, \delta')}\ \mathsf{ev\_fst}$$

$$\frac{\Phi \vdash P \hookrightarrow \langle V_1, V_2\rangle \qquad \Phi \vdash D[V_2/\mathbf{x}] \hookrightarrow (\psi';\delta')}{\Phi \vdash (\mathbf{x} \in F_2 = \pi_2\ P, D) \hookrightarrow (\psi'; V_2/\mathbf{x}, \delta')}\ \mathsf{ev\_snd}$$

Finally, there are two rules defining the selection of cases. Because of Theorem 6.36 it is always decidable if a case in Ω applies, and in addition the new environment is effectively computable by the matching algorithm presented in Section 6.4.2. The situation when a case matches is expressed by rule ev_yes. If it does not match, ev_no tries the next case. Note, that there is no rule for $\Omega = \cdot$. If this case is encountered, it follows that the function currently executed is not a realizer because the evaluation would terminate without returning a value. A careful study why such a situation cannot occur will be the focus of the next chapter.

$$\frac{\Phi \vdash P[\psi''; \delta] \hookrightarrow V}{\Phi \vdash (\psi; \delta) \sim (\Omega, (\Psi \triangleright \psi' \mapsto P)) \hookrightarrow V}\ \textsf{ev\_yes}$$

if there exists a ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$

$$\frac{\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V}{\Phi \vdash (\psi; \delta) \sim (\Omega, (\Psi \triangleright \psi' \mapsto P)) \hookrightarrow V}\ \textsf{ev\_no}$$

if there is no ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$

The operational semantics is type preserving. Specifically, its proof relies on a technical but easy to prove lemma which we have dubbed context lemma.

Lemma 6.37 (Context)

1. *If* $\mathcal{D} :: \Phi; \cdot \vdash id_\Phi, \psi; \delta \in \Phi, [id_\Phi, M/x]\Psi; [id_\Phi, M/x, id_\Psi]\Delta$
   and $\mathcal{E} :: [\![\Phi]\!] \vdash M : A$
   and $\mathcal{P} :: \Phi; \cdot \vdash V \in F[id_\Phi, M/x]$
   then $\Phi; \cdot \vdash (id_\Phi, M/x, \psi; V/\mathbf{y}, \delta) \in (\Phi, x : A, \Psi; \mathbf{y} \in F, \Delta)$

2. *If* $\mathcal{D} :: \Phi; \cdot \vdash id_\Phi, \psi; \delta \in \Phi, \Psi; \Delta$
   and $\mathcal{P} :: \Phi; \cdot \vdash V \in F$
   then $\Phi; \cdot \vdash (id_\Phi; \psi; V/\mathbf{y}, \delta) \in (\Phi, \Psi; \mathbf{y} \in F, \Delta)$

Proof: direct in both cases using Lemma 6.21. A detailed proof can be found in Appendix B.3. □

Based on the context lemma, the type preservation theorem follows. Clearly, the evaluation relation is mutually dependent on the evaluation relation of declarations and cases, and the type preservation theorem must hence be accordingly generalized.

Theorem 6.38 (Type-preservation)

1. *If* $\mathcal{D} :: \Phi \vdash P \hookrightarrow V$
   and $\mathcal{E} :: \Phi; \cdot \vdash P \in F$
   then $\Phi; \cdot \vdash V \in F$

2. *If* $\mathcal{D} :: \Phi \vdash D \hookrightarrow \psi; \delta$
   and $\mathcal{E} :: \Phi; \cdot \vdash D \in \Psi; \Delta$
   then $\Phi; \cdot \vdash (id_\Phi, \psi; \delta) \in (\Phi, \Psi; \Delta)$
   which extends $\Phi; \cdot \vdash (id_\Phi; \cdot) \in (\Phi; \cdot)$

3. *If* $\mathcal{D} :: \Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V$
   and $\mathcal{F} :: \Phi; \cdot \vdash \psi; \delta \in \Psi; \Delta$
   and $\mathcal{E} :: \Psi; \Delta \vdash \Omega \in F$
   then $\Phi; \cdot \vdash V \in F[\psi]$

Proof: by simultaneous induction over $\mathcal{D}(1), \mathcal{D}(2), \mathcal{D}(3)$ using Lemma 6.20, Lemma 6.22, Lemma 6.37, Lemma 6.11, Lemma 6.7. A detailed proof can be found in Appendix B.3. □

Note, that the proof of the type preservation theorem does not rely on the termination side condition (5.1) or the coverage side condition (5.2). In fact, even without these two side conditions, the operational semantics is type-preserving. In other words, every function that corresponds to a well-typed proof term in $\mathcal{M}_2^+$ is partially correct.

This already concludes the presentation of the operational big-step semantics. It assigns an operational meaning to a proof term by relating it to the result value of its computation. Moreover this computation is type preserving. The main draw back of this kind of operational semantics is that it does not offer any fine-grained control on how the evaluation is being conducted. Using only the big-step semantics we cannot express that a proof term interpreted as function is total and applied to LF-objects always computes a value.

6.6 Summary

In this chapter we have introduced and proved properties about all major basic concepts needed for an in depth analysis of the meta logic $\mathcal{M}_2^+$. On the basis of generalized substitutions, meta-substitutions, lemma instantiations, context schema subsumption, matching, and strictness, we have defined an operational big-step semantics which assigns an operational meaning to proof terms, justifying the intuitive idea from Section 2.6, that proof terms correspond to recursive functions.

Chapter 7

Realizability

From a functional perspective, the proof terms of $\mathcal{M}_2^+$ correspond to partial recursive functions suitable for programming, but from a logical perspective we must require from these functions to be total to be considered proofs. Unlike the soundness proof of $FO\lambda^{\Delta I\!N}$ which is based on a cut-elimination argument in the presence of natural number induction, the soundness proof for $\mathcal{M}_2^+$ is based on realizability; each proof term in $\mathcal{M}_2^+$ realizes a proof. That means, that upon application to arguments, proof terms evaluate always to some result, in particular their computation makes always progress and will eventually terminate. Under the regular world assumption, we have argued that the big-step operational semantics is not fine-grained enough to study properties such as termination and coverage. Thus we refine the big-step semantics to a trace-based continuation style small-step semantics and establish two syntactic criteria on the form of proof terms that imply progress and termination of their computations. Together, the two criteria guarantee that $\mathcal{M}_2^+$'s proof terms are total, entailing the soundness of the meta-logic $\mathcal{M}_2^+$. We stress that the criteria are syntactic, which makes them good candidates for proof generation and proof checking in Twelf (see Chapter 8).

The soundness proof of a functional calculus as complicated as $\mathcal{M}_2^+$ involves a lot of work. Sadly, the operational big-step semantics is inappropriate to reason about proof terms the way we need to. In particular, $\Phi \vdash P \hookrightarrow V$ is not an algorithmic description of how to evaluate proof terms, it is merely the definition of a relation, relating a program to be executed with the result of its computation. This means that if a computation does not terminate or cannot make progress because not all cases are covered, $\Phi \vdash P \hookrightarrow V$ is simply not derivable for any V. On the other hand, if we had a more algorithmic specification of the operational semantics we could formulate termination and coverage properties. Specifically, it would allow us to express the non-termination of the program "$\mu \mathbf{x} \in \langle\rangle.\, \mathbf{x}$", because once this program begins to execute, it deterministically transforms to itself after each execution step. Similarly, it would allow us to express that the evaluation of "case $(\mathrm{lam}(\lambda y : \mathrm{term}\ \ulcorner\tau\urcorner.\, y)/x; \cdot)$ of $\cdot$" does not make progress, since no cases are applicable. Therefore, we refine the big-step operational semantics to a small-step semantics that is executable on an abstract machine. Only by observing the computation trace of an abstract machine it is possible to investigate if a function is total or not.

This chapter is organized as follows. In Section 7.1 we define a small-step semantics for the proof terms of $\mathcal{M}_2^+$. This semantics is trace-based, and can be executed on a continuation based abstract machine, very similar to the CPM machine [Pfe00]. In addition, it refines the big-step semantics from Section 6.5, but we skip the proof. We then address the question of termination in Section 7.2 and coverage in Section 7.3. Termination and coverage are the two necessary properties for totality. Therefore, proof terms in $\mathcal{M}_2^+$ are realizers and the meta-logic $\mathcal{M}_2^+$ is sound, which we show in Section 7.4. Finally, we summarize the results of this Chapter in Section 7.5.

7.1 Small-Step Semantics

In the previous chapter, we formally expressed the evaluation of a program P to a value V by the judgment $\Phi \vdash P \hookrightarrow V$ where Φ is the parameter context modelling the regular world assumption. In order to appreciate the new and refined operational semantics, we have to look at evaluation from a different point of view. We say that P evaluates to V if and only if there exists a trace from an initial state which corresponds to P to a final state which correspond to V. The trace is to be understood as a sequence of state transitions performed by some — still to be defined — abstract machine.

This idea can be visualized by the following diagram: The derivation $\mathcal{P}$ stands for compiling P into a program contained in the state S_0, $\mathcal{E}$ is the trace of the abstract machine, and S_n is the final state reached say after n steps. $\mathcal{Q}$ then extracts the result of the computation from S_n and decompiles it into a value in the sense of Section 6.5.

$$\begin{tikzcd}
P \arrow[r, "\mathcal{D} :: P \hookrightarrow V"] \arrow[d, "\mathcal{P} :: [P] = S_0"'] & V \\
S_0 \arrow[r, "\mathcal{E} :: S_0 \overset{*}{\Longrightarrow} S_n"'] & S_n \arrow[u, "\mathcal{Q} ::]S_n[= V"']
\end{tikzcd}$$

Consider the proof term **dia** formalizing the proof of the diamond Lemma 4.6. Given that all universal quantifiers are instantiated by concrete LF objects, the residual proof term P is of existential type. It guarantees that the common reduct and the two derivations closing the diamond exist. Upon completion of the abstract machine, we extract those three witness objects from the result of the computation.

In this section, we discuss various aspects of the new small-step operational semantics and the underlying abstract machine. The fact is, that the meta-logic $\mathcal{M}_2^+$ is sound, even if the diagram above does not commute. All that matters is that by executing the proof term applied to LF objects, the result are well-typed objects, witnessing that the types of the variables bound by existential quantifiers are inhabited. It does not matter, if the big-step semantics evaluates to a different value or not. On the other hand, even though we will not provide a proof for this claim, we conjecture that the small-step semantics is a refinement of the big-step semantics, i.e. that both are observationally equivalent: For any derivation $\mathcal{D}$ evaluating P to V, the small-step operational semantics will compute V from $[P]$, and the reverse also holds. For any program P, value V, and computation trace $\mathcal{E}$ that starts in $[P]$ and ends in $[V]$ it is possible to reconstruct a derivation $\mathcal{D}$ of $P \hookrightarrow V$.

The kind of abstract machine employed in this thesis is a continuation stack based state machine. It is inspired by the CPM machine [Pfe00], and the CPS machine [FSDF93]. A *state* of the computation contains three bits of information. There is a continuation stack which represents delayed computations necessary to compute the overall value V, a generalized parameter context Φ, and, naturally, the program to be executed.

This section is organized as follows: we begin with the definition of programs and values and describe the process of compilation in Section 7.1.1. Based on programs we define execution states in Section 7.1.2. The abstract machine, executing the programs of the small-step semantics is discussed in Section 7.1.3, for which we define a typing discipline and prove type preservation in Section 7.1.4.

7.1.1 Programs

The small-step operational semantics is based on the idea that proof terms of $\mathcal{M}_2^+$ are compiled into programs suitable for execution on an abstract machine. In this subsection we discuss the form of programs. We also explain how to compile proof terms into programs.

What are programs? Traditionally, a program is a list of instructions to be executed on the abstract machine. In this thesis however, we avoid sequentializing a proof term into a list of instruction but we interpret proof terms unaltered as programs. Consider, for example, a proof term of the form $\langle P_1, P_2\rangle$. When executing it on our abstract machine it will eventually evaluate to the value $\langle V_1, V_2\rangle$ (see the definition of the big-step operational semantics). But intuitively, during computation, the abstract machine will inevitably encounter the program $\langle V_1, P_2\rangle$, if we consider a left to right computation of pairs. Note, that in this example, all three programs are syntactically the same proof term, but conversely, each of them carries a different computational meaning. Thus, each proof term gives raise to several programs, depending on which of the arguments have been instantiated by values (values are proof terms that do not need to be computed any further, since they wouldn't change). Throughout this chapter we distinguish the different off-springs of a proof term by different variable names of its arguments. P stand for programs, and V for values.

What forms do values take? Differently from Section 6.5 where we describe values only as the outcome of the evaluation of a proof term, in this setting values can take additional shapes; meta-substitution extensions must be considered as values because they may be the outcome of a list of declarations.

$$\textit{Values:}\quad V \quad ::= \quad \Lambda x : A.\, P \mid \lambda\rho^L.\, P \mid \langle M, V\rangle \mid \langle V_1, V_2\rangle \mid \langle\rangle \mid (\eta; \delta)$$

Thus, in summary, programs are proof terms or values. This information can be easily inferred and need not be represented explicitly. In accordance with this observation there is no need to introduce a new syntactic class for programs. Instead, we use P to denote proof terms and programs simultaneously.

What about compilation? Compiling a proof term means to tag it by information which subterm is already a value and which is not. Symmetrically, decompiling a program means to remove those tags. The compilation and decompilation operation $[\cdot]$ and $]\cdot[$ can be safely omitted. Proof terms are defined in form of declaration and cases. What was said in this subsection about compilation also holds for the declarations and cases, to which we sloppily refer as programs.

7.1.2 States

A state of the abstract machine is a triplet consisting of a parameter context Φ, a continuation stack C, and a the program to be executed. In this setting programs are either proof terms, declarations, cases or substitutions. Therefore, one possible form of a state is $\Phi; C \triangleright P$. In general, states are denoted by S, and should not be confused with spines from Section 6.4. Their precise definition is given at the end of this subsection.

Continuation stacks C are stacks of delayed computations necessary to eventually compute the overall value of a program. Each continuation is a function expecting one argument, namely the result of the previous computation. There are different styles in presenting continuations; one can either represent each continuation by a leading λ-abstraction, binding the one argument [Pfe00] and assigning a name to it, or alternatively, one can represent each variable by a special symbol [FSDF93]. In this presentation we have decided to follow the latter style and we write $\bullet$ for each hole representing a variable.

Returning to the small example from above, we consider a trace which computes the value of a pair $\langle P_1, P_2\rangle$. It starts with computing value V_1 of its first component P_1. But before the abstract machine starts with the computation of P_1, it has to memorize how to interpret its result. This is done by adding an appropriate continuation to the continuation stack. The form of the continuation is $\langle \bullet, P_2\rangle$, meaning that once P_1 is executed, the resulting value belongs into the first position of the pair $\langle V_1, P_2\rangle$. Similarly, the evaluation of the second component requires the continuation $\langle V_1, \bullet\rangle$ to be added to the continuation stack. Putting it all together, informally, here are a few snapshots along the trace of the abstract machine. We write $S_1 \Longrightarrow S_2$ for a step of the abstract machine.

$$
\begin{array}{ll}
 & \ldots \\
\Longrightarrow & \Phi; C \triangleright \langle P_1, P_2\rangle \\
\Longrightarrow & \Phi; C, \langle \bullet, P_2\rangle \triangleright P_1 \\
\Longrightarrow & \ldots \\
\Longrightarrow & \Phi; C, \langle \bullet, P_2\rangle \triangleright V_1 \\
\Longrightarrow & \Phi; C \triangleright \langle V_1, P_2\rangle \\
\Longrightarrow & \Phi; C, \langle V_1, \bullet\rangle \triangleright P_2 \\
\Longrightarrow & \ldots \\
\Longrightarrow & \Phi; C, \langle V_1, \bullet\rangle \triangleright V_2 \\
\Longrightarrow & \Phi; C \triangleright \langle V_1, V_2\rangle \\
\Longrightarrow & \ldots
\end{array}
$$

This example shows not only the idea behind continuation stacks in particular, but also how the abstract machine works in general. The different programs contained in a state determine the next computation step. If needed, the continuation stack is extended. Once a value in the computation is reached, the top continuation of the computation stack is awakened, and the $\bullet$ is replaced by the value. The computation resumes with the new state. Therefore, in the general case, the computation of a program starts with an empty continuation stack, and the abstract machine halts with a value if the continuation stack is empty again. This value is the return value of the computation.

Since the proof terms of $\mathcal{M}_2^+$ are recursive functions defined via case analysis, the abstract machine can reach a state where it has to select an applicable case from the given list of cases Ω, and continue with the execution of its body. We call these states *match states* and write $\Phi; C \triangleright (\psi; \delta) \sim \Omega$. A transition to a match state means to select an applicable case from Ω, to instantiate a few variables, and to execute the program in its body. The subject of the case is an explicit substitution $(\psi; \delta)$ against which the machine has to match the cases in Ω using the algorithm we have developed in Section 6.4.

There are two possible states we have not accounted for yet. Programs compute values, but declarations compute substitutions. Thus, the state which prompts the abstract machine to execute a declaration is $\Phi; C \triangleright D$ whereas the result of this evaluation prompts for a special kind of state, suitable to return the result substitution $\Phi; C \triangleright \psi; \delta$. In addition, the continuations introduced by declarations contain two $\bullet$'s. In the case such a continuation is awakened and applied to $(\psi; \delta)$, the left most $\bullet$ is replaced by ψ and the right most $\bullet$ by δ. We denote the empty continuation stack with $\star$.

$$
\begin{array}{lcll}
\textit{Continuations:} & C & ::= & \star \mid C, \mathbf{let} \bullet \mathbf{in}\ P \\
 & & & \mid C, \langle \bullet, P \rangle \mid C, \langle V, \bullet \rangle \mid C, \langle M, \bullet \rangle \\
 & & & \mid C, (\mathbf{x} \in F = \pi_1 \bullet, D) \mid C, (\mathbf{x} \in F = \pi_2 \bullet, D) \\
 & & & \mid C, (\bullet; V/\mathbf{x}, \bullet) \\
 & & & \mid C, (\langle x : A, \mathbf{y} \in F \rangle = \bullet, D) \mid C, (M/x, \bullet; V/\mathbf{y}, \bullet) \\
 & & & \mid C, (\mathbf{x} \in F = \bullet\ M, D) \\
 & & & \mid C, (\mathbf{x} \in F = \bullet\ \rho, D) \\
 & & & \mid C, (\mathbf{x} \in F = \bullet, D) \\
 & & & \mid C, (\lambda \rho^L. (\bullet; \bullet)) \\
\textit{States:} & S & ::= & \Phi; C \triangleright P \mid \Phi; C \triangleright D \mid \Phi; C \triangleright (\psi; \delta) \sim \Omega \mid \Phi; C \triangleright \psi; \delta
\end{array}
$$

The first transition into a match state is depicted by the following snapshot of a particular trace of the abstract machine.

$$\ldots \Longrightarrow \Phi; C \triangleright \text{case } (\psi; \delta) \text{ of } \Omega \Longrightarrow \Phi; C \triangleright (\psi; \delta) \sim \Omega \Longrightarrow \ldots$$

Operationally speaking, the abstract machine attempts to find an applicable case in Ω, and makes a transition to a regular state, unless such a case does not exist. It is the one of the results of this chapter, that an applicable case always exists, given that the side condition (5.2) is satisfied. We continue this exposition with the definition of the abstract machine.

7.1.3 Abstract Machine

Following examples from the literature, we specify the operational small-step semantics as a transition relation between states. The single step relation is denoted by $\Longrightarrow$, and its transitive closure by $\stackrel{*}{\Longrightarrow}$.

$$
\begin{array}{ll}
\text{One-step reduction} & \vdash S_1 \Longrightarrow S_2 \\
\text{Multi-step reduction} & \vdash S_1 \stackrel{*}{\Longrightarrow} S_2
\end{array}
$$

The rules are a direct translation of the rules from Section 6.5 where we make extensive use of postponing the computation of subprograms as continuations. Every transition rule introduces at most one continuation. For each such rule, there is a dual rule that defines how the continuation is to be applied to a value. In the let case, for example, there is a rule that defines how the program **let** D **in** P is computed — D is computed while storing the overall goal of the computation **let** $\bullet$ **in** P on the continuation stack. The second rule then says, that V must be a meta-substitution extension (because of typing which we define in Section 7.1.4). Consequently, by reawakening the top continuation we obtain a new program embedded in a new state. Without loss of generality — as we discuss below in more detail — we can assume that every program computes a value otherwise some other rule is applicable.

$$
\begin{array}{ll}
\mathsf{trlet} & :: \Phi; C \triangleright \mathbf{let}\ D\ \mathbf{in}\ P \implies \Phi; C, \mathbf{let}\ \bullet\ \mathbf{in}\ P \triangleright D \\
\mathsf{trletC} & :: \Phi; C, \mathbf{let}\ \bullet\ \mathbf{in}\ P \triangleright (\psi; \delta) \implies \Phi; C \triangleright P[\mathrm{id}_\Phi, \psi; \delta]
\end{array}
$$

Similarly the definition of the next four transition rules for conjunction account for the computation of pairs — as discussed above. The first two rules define the computation of a pair, where both components are non-values, the second two rules define the computation if the left component is a value.

$$
\begin{array}{ll}
\mathsf{trpair} & :: \Phi; C \triangleright \langle P_1, P_2 \rangle \implies \Phi; C, \langle \bullet, P_2 \rangle \triangleright P_1 \\
\mathsf{trpairC} & :: \Phi; C, \langle \bullet, P_2 \rangle \triangleright V \implies \Phi; C \triangleright \langle V, P_2 \rangle \\
\mathsf{trmix} & :: \Phi; C \triangleright \langle V_1, P_2 \rangle \implies \Phi; C, \langle V_1, \bullet \rangle \triangleright P_2 \\
\mathsf{trmixC} & :: \Phi; C, \langle V_1, \bullet \rangle \triangleright V \implies \Phi; C \triangleright \langle V_1, V \rangle
\end{array}
$$

Dually to the rules for pairs, there are rules that define the computation of projections. Recall, that projections are proof terms for left rules, and therefore they take the shape of declarations. Computing the result of a projection is standard. The first transition initiates the computation of the program from which we project, while storing on the continuation stack the information what to do with the result: $\mathbf{x} \in F = \pi_1\ \bullet, D$. The second rule is activated once this continuation lies on top of the continuation stack. By inversion it follows that the form of the result value $\langle V_1, V_2 \rangle$. Next, the machine continues to compute the rest of the declarations, and we can expect that it terminates in a value, precisely in a meta-substitution extension. This substitution must be extended by $V_1/\mathbf{x}$.

$$
\begin{array}{ll}
\mathsf{trfst} & :: \Phi; C \triangleright \mathbf{x} \in F = \pi_1\ P, D \implies \Phi; C, \mathbf{x} \in F = \pi_1\ \bullet, D \triangleright P \\
\mathsf{trfstC} & :: \Phi; C, \mathbf{x} \in F = \pi_1\ \bullet, D \triangleright \langle V_1, V_2 \rangle \implies \Phi; C, (\bullet; V_1/\mathbf{x}, \bullet) \triangleright D[V_1/\mathbf{x}] \\
\mathsf{trsnd} & :: \Phi; C \triangleright \mathbf{x} \in F = \pi_2\ P, D \implies \Phi; C, \mathbf{x} \in F = \pi_2\ \bullet, D \triangleright P \\
\mathsf{trsndC} & :: \Phi; C, \mathbf{x} \in F = \pi_2\ \bullet, D \triangleright \langle V_1, V_2 \rangle \implies \Phi; C, (\bullet; V_2/\mathbf{x}, \bullet) \triangleright D[V_2/\mathbf{x}] \\
\mathsf{trmeta} & :: \Phi; C, (\bullet; V/\mathbf{x}, \bullet) \triangleright (\psi; \delta) \implies \Phi; C \triangleright (\psi; V/\mathbf{x}, \delta)
\end{array}
$$

Very similar, the proof term of an existential formula is computed via two rules trinx and trinxC which can be expected to return $\langle M, P \rangle$. Likewise three more rules ($\mathsf{trsplit}$, $\mathsf{trsplitC}$, and $\mathsf{trsubst}$) define the computation of a splitting operation. The $\mathsf{trsubst}$ rule plays the rôle of trmeta from above, except — since this is a splitting operation — the witness object itself is part of the final result.

$$
\begin{array}{ll}
\mathsf{trinx} & :: \Phi; C \triangleright \langle M, P \rangle \implies \Phi; C, \langle M, \bullet \rangle \triangleright P \\
\mathsf{trinxC} & :: \Phi; C, \langle M, \bullet \rangle \triangleright V \implies \Phi; C \triangleright \langle M, V \rangle \\
\mathsf{trsplit} & :: \Phi; C \triangleright \langle x : A, \mathbf{y} \in F \rangle = P, D \implies \Phi; C, \langle x : A, \mathbf{y} \in F \rangle = \bullet, D \triangleright P \\
\mathsf{trsplitC} & :: \Phi; C, \langle x : A, \mathbf{y} \in F \rangle = \bullet, D \triangleright \langle M, V \rangle \implies \Phi; C, (M/x, \bullet; V/\mathbf{y}, \bullet) \triangleright D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}] \\
\mathsf{trsubst} & :: \Phi; C, (M/x, \bullet; V/\mathbf{y}, \bullet) \triangleright (\psi; \delta) \implies \Phi; C \triangleright (M/x, \psi; V/\mathbf{y}, \delta)
\end{array}
$$

The following eight rules form the remaining transition rules for declarations. The rule $\mathsf{trempty}$ marks the computation of the end-of-declarations symbol. Naturally, it returns an empty meta-substitution extension. trApp, and trAppC compute the result of a redex. They correspond to the $\mathsf{ev\_app}$ rule from Section 6.5. The first rule computes the value of the recursive function to be executed, and the second rule applies it. Similarly, for variable block applications we define two rules: trapp and trappC. The trnew rule introduces a new parameter block of assumptions. It then computes the value of its body D. This computation can be expected to return a meta-substitution extension $(\psi; \delta)$ that is to be abstracted over the new parameter block as discussed in Section 6.2.2 by trnewC.

$$
\begin{array}{ll}
\mathsf{trempty} & :: \Phi; C \triangleright \cdot \implies \Phi; C \triangleright \cdot; \cdot \\
\mathsf{trApp} & :: \Phi; C \triangleright \mathbf{x} \in F = P\ M, D \implies \Phi; C, \mathbf{x} \in F = \bullet\ M, D \triangleright P \\
\mathsf{trAppC} & :: \Phi; C, \mathbf{x} \in F = \bullet\ M, D \triangleright \Lambda x : A.\, P \implies \Phi; C, \mathbf{x} \in F = \bullet, D \triangleright P[\mathrm{id}_\Phi, M/x] \\
\mathsf{trapp} & :: \Phi; C \triangleright \mathbf{x} \in F = P\ \rho, D \implies \Phi; C, \mathbf{x} \in F = \bullet\ \rho, D \triangleright P \\
\mathsf{trappC} & :: \Phi; C, \mathbf{x} \in F = \bullet\ \rho', D \triangleright \lambda \rho^L.\, P \implies \Phi; C, \mathbf{x} \in F = \bullet, D \triangleright P[\mathrm{id}_\Phi, \rho'/\rho] \\
\mathsf{trassign} & :: \Phi; C, \mathbf{x} \in F = \bullet, D \triangleright V \implies \Phi; C, (\bullet; V/\mathbf{x}, \bullet) \triangleright D[V/\mathbf{x}] \\
\mathsf{trnew} & :: \Phi; C \triangleright \nu\ \rho^L.\, D \implies \Phi, \rho^L; C, (\lambda \rho^L.\, (\bullet; \bullet)) \triangleright D \\
\mathsf{trnewC} & :: \Phi, \rho^L; C, (\lambda \rho^L.\, (\bullet; \bullet)) \triangleright \psi; \delta \implies \Phi; C \triangleright \lambda \rho^L.\, (\psi; \delta)
\end{array}
$$

The final set of rules defines the execution of the two operations recursion and case analysis. The recursion rule does not introduce any new continuations. Similar to the $\mathsf{ev\_rec}$ rule, it merely computes the value of its body, given that all occurrences of the recursion variable are replaced by the program itself.

$$\mathsf{trrec} :: \Phi; C \triangleright \mu \mathbf{x} \in F.\, P \implies \Phi; C \triangleright P[\mu \mathbf{x} \in F.\, P/\mathbf{x}]$$

Case analysis on the other hand is slightly more complicated. It uses the matching algorithm defined in Section 6.4.2. First, as shown in the example above, a analysis of cases is initiated by the case-program. The subject of the case construct — the explicit substitution — is matched against each case. Because we can assume that each case is well-typed, each substitution ψ' is strict in both cases. Therefore, by Theorem 6.36 the matching problem $\Phi \triangleright \psi \approx \psi'$ is decidable. And thus it is clearly decidable if tryes or trno are applicable. As side remark, no new continuations must be pushed onto the continuation stack for the purpose of case analysis.

$$
\begin{array}{ll}
\mathsf{trcase} & :: \Phi; C \triangleright \mathrm{case}\ (\psi; \delta)\ \mathrm{of}\ \Omega \implies \Phi; C \triangleright (\psi; \delta) \sim \Omega \\
\mathsf{tryes} & :: \Phi; C \triangleright (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P)) \implies \Phi; C \triangleright P[\psi''; \delta] \\
& \quad \text{if there exists a } \psi'' \text{ s.t. } (\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta) \\
\mathsf{trno} & :: \Phi; C \triangleright (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P)) \implies \Phi; C \triangleright (\psi; \delta) \sim \Omega \\
& \quad \text{if there is no } \psi'' \text{ s.t. } (\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)
\end{array}
$$

The rule tryes and trno are quite simple, and we will see, the theoretical results associated with this construction reuses many of the results we have shown in the previous Chapter. The final two rules define the reflexive and transitive closure of traces.

$$
\frac{}{S \overset{*}{\implies} S}\ \mathsf{trid}
\qquad
\frac{S_1 \implies S_2 \quad S_2 \overset{*}{\implies} S_3}{S_1 \overset{*}{\implies} S_3}\ \mathsf{trstep}
$$

This concludes our presentation of the operational semantics of the abstract machine. We conjecture that the small-step operational semantics is sound and complete with respect to the big-step operational semantics. Whenever there is computation of V from P, then V is the result of evaluating P.

Conjecture 7.1 (Soundness)
If $\mathcal{D} :: \Phi; \star \triangleright P \overset{*}{\implies} \Phi; \star \triangleright V$
then $\mathcal{E} :: \Phi \vdash P \hookrightarrow V$.

Vice versa, each evaluation of any program P to a value V can be simulated and executed on the abstract machine while preserving results.

Conjecture 7.2 (Completeness)
If $\mathcal{D} :: \Phi \vdash P \hookrightarrow V$
then $\mathcal{E} :: \Phi; \star \triangleright P \stackrel{*}{\Longrightarrow} \Phi; \star \triangleright V$.

In summary, we have defined an abstract machine which executes the proof terms of $\mathcal{M}_2^+$. The small-step semantics is the appropriate tool to reason about totality. In the next section, we show that it is also type preserving.

7.1.4 Validity

A small-step operational semantics is said to be type preserving if all states in a computation trace are valid and are of the same type! Therefore the task of designing an appropriate typing discipline for this operational semantics reduces to establishing a suitable typing discipline on states.

Internally, states consist of parameter contexts, continuation stacks, and programs. Programs are proof terms and they therefore inherit the typing discipline from Section 5.4. From this assumption we can already guarantee that throughout a computation, the parameter context Φ remains well-typed given that it is initially well-typed what we always presuppose. The reason is that Φ is only modified by the trnew and the trnewC rule — and this clearly in a sound way.

Therefore, the main challenge in designing a type system for states hinges on an appropriate definition of the type systems for continuation stacks which we develop in this subsection. We proceed as follows: Continuations are parametric meta-level functions different enough from standard meta-level functions we have described in Chapter 4 to warrant a new definition of continuation types. More precisely, there are two different continuation types. First, there is one that expresses that a continuation expects a value of some formula F_1 as input and returns some value of formula F_2 as result. Second, the other type expresses that the continuation expects as input a meta-substitution extension of "type" $(\Psi; \Delta)$ and returns a value of a formula F as result.

$$\text{Continuation types:} \quad T \quad ::= \quad F_1 \Rightarrow F_2 \mid (\Psi; \Delta) \Rightarrow F$$

From a logical point of view, both continuation types are implications; however, the meta-logic $\mathcal{M}_2^+$ is relatively impoverished with respect to the standard propositional connectives such as negation and disjunction and, in addition, we cannot quantify over substitutions. Thus, we cannot use the meta-logic itself to provide the notion of continuation types.

The typing discipline for continuation stacks extends the one for continuations in the standard way. Note, that because of the regular world assumption continuation stacks are not necessarily closed. New parameter blocks are inserted into the parameter contexts by trnew and retracted by trnewC. Therefore, the typing judgment for continuation stacks must take a parameter context Φ into account.

$$\textit{Valid continuation stacks:} \quad \Phi \vdash C \in T$$

The semantics of this judgment is defined by a set of inference rules. The empty continuation stack, for example, acts as the identity function.

$$\frac{}{\Phi \vdash \star \in F \Rightarrow F}\ \mathsf{tcdone}$$

A continuation with a let continuation as top element has type $(\Psi; \Delta) \Rightarrow F$, if its body P has type F_1 in the meta-context $\Phi, \Psi; \Delta$. The remaining continuation on the other hand must be of type $F_1 \Rightarrow F$.

$$\frac{\Phi \vdash C \in F_1 \Rightarrow F \qquad \Phi, \Psi; \Delta \vdash P \in F_1}{\Phi \vdash C, \mathbf{let} \bullet \mathbf{in}\ P \in (\Psi; \Delta) \Rightarrow F}\ \mathsf{tclet}$$

A stack whose top-level continuation is a pair with one $\bullet$ as first component has type $F_1 \Rightarrow F$, if the rest continuation stack has type $F_1 \wedge F_2 \Rightarrow F$ and the second component of the pair is well-typed.

$$\frac{\Phi \vdash C \in F_1 \wedge F_2 \Rightarrow F \qquad \Phi; \cdot \vdash P \in F_2}{\Phi \vdash C, \langle \bullet, P \rangle \in F_1 \Rightarrow F}\ \mathsf{tcpair}$$

Similarly, if the top continuation on the stack is a pair with a $\bullet$ as second component, its type if $F_2 \Rightarrow F$ given that the rest of the continuation stack has type $F_1 \wedge F_2 \Rightarrow F$ and the first component of the pair is well-typed.

$$\frac{\Phi \vdash C \in F_1 \wedge F_2 \Rightarrow F \qquad \Phi; \cdot \vdash V \in F_1}{\Phi \vdash C, \langle V, \bullet \rangle \in F_2 \Rightarrow F}\ \mathsf{tcmix}$$

Along the same lines, if $\langle M, \bullet \rangle$ lies on the continuation stack, the continuation stack must be of type $\exists x : A. F_1 \Rightarrow F$. Consequently, the stack as a whole has type $F_1[\mathrm{id}_\Phi, M/x] \Rightarrow F$ then.

$$\frac{\Phi \vdash C \in \exists x : A. F_1 \Rightarrow F \qquad [\![\Phi]\!] \vdash M : A}{\Phi \vdash C, \langle M, \bullet \rangle \in F_1[\mathrm{id}_\Phi, M/x] \Rightarrow F}\ \mathsf{tcinx}$$

We consider now the case that $(\mathbf{x} \in F_1 = \pi_1\ \bullet, D)$ lies on top of the continuation stack. It expects as argument a value of type $F_1 \wedge F_2$, given that D has type $(\Psi; \Delta)$. Thus, the continuation expects a meta-substitution extension of $\Psi, \mathbf{x} \in F_1; \Delta$ as input. A side remark: These rules are the reason why we need to label each occurrence of $\mathbf{x}$ explicitly by its formula otherwise we could not express the right premiss of tcfst. The typing rule for the symmetric projection is defined analogously.

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F \qquad \Phi; \mathbf{x} \in F_1 \vdash D : \Psi; \Delta}{\Phi \vdash C, (\mathbf{x} \in F_1 = \pi_1\ \bullet, D) \in F_1 \wedge F_2 \Rightarrow F}\ \mathsf{tcfst}$$

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_2, \Delta) \Rightarrow F \qquad \Phi; \mathbf{x} \in F_2 \vdash D : \Psi; \Delta}{\Phi \vdash C, (\mathbf{x} \in F_2 = \pi_2\ \bullet, D) \in F_1 \wedge F_2 \Rightarrow F}\ \mathsf{tcsnd}$$

The continuation $(\mathbf{x} \in F_1 = \bullet, D)$ captures the assignment of the current value to $\mathbf{x}$ in D and to subsequent occurrences of $\mathbf{x}$ in the proof term. Naturally, its type is $F_1 \Rightarrow F$, where F is the type of the overall computation.

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F \qquad \Phi; \mathbf{x} \in F_1 \vdash D : \Psi; \Delta}{\Phi \vdash C, (\mathbf{x} \in F_1 = \bullet, D) \in F_1 \Rightarrow F}\ \mathsf{tcassign}$$

Following a very similar line of argument, we derive that the type of the continuation used in the trmeta-rule has type $(\Psi; \Delta) \Rightarrow F$.

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F \qquad \Phi; \cdot \vdash V \in F_1}{\Phi \vdash C, (\bullet; V/\mathbf{x}, \bullet) \in (\Psi; \Delta) \Rightarrow F}\,\mathsf{tcmeta}$$

The previous three rules were concerned with typing continuation which may occur while evaluating a pair. Analogously, the following two rules assign types to continuations which may occur during the evaluation of a proof term of existential type.

$$\frac{\Phi \vdash C \in (x : A, \Psi; \mathbf{y} \in F_1, \Delta) \Rightarrow F \qquad \Phi, x : A; \mathbf{y} \in F_1 \vdash D \in \Psi; \Delta}{\Phi \vdash C, (\langle x : A, \mathbf{y} \in F_1 \rangle = \bullet, D) \in \exists x : A.\, F_1 \Rightarrow F}\,\mathsf{tcsplit}$$

$$\frac{\Phi \vdash C \in (x : A, \Psi; \mathbf{y} \in F_1, \Delta) \Rightarrow F \qquad \Phi \vdash M : A \qquad \Phi; \cdot \vdash V \in F_1[\mathrm{id}_\Phi, M/x]}{\Phi \vdash C, (M/x, \bullet; V/\mathbf{y}, \bullet) \in [\mathrm{id}_\Phi, M/x](\Psi; \Delta) \Rightarrow F}\,\mathsf{tcsubst}$$

While executing a declaration that applies a function to an LF object or a variable block, a new continuation is pushed on the continuation stack; in the first case, the continuation is $(\mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D)$, and in the second it is $(\mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D)$. The types of the resulting continuation stacks are $\forall x : A.\, F_1 \Rightarrow F$ and $\mathbf{\Pi}\rho^L.\, F_1 \Rightarrow F$, respectively.

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x], \Delta) \Rightarrow F \qquad \Phi \vdash M : A \qquad \Phi; \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] \vdash D \in \Psi; \Delta}{\Phi \vdash C, (\mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D) \in \forall x : A.\, F_1 \Rightarrow F}\,\mathsf{tcApp}$$

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho], \Delta) \Rightarrow F \qquad \Phi \vdash \rho \equiv \rho' \qquad \Phi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] \vdash D \in \Psi; \Delta}{\Phi \vdash C, (\mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D) \in \mathbf{\Pi}\rho^L.\, F_1 \Rightarrow F}\,\mathsf{tcapp}$$

And finally, the case of the **new**-continuation $C, (\lambda\rho^L.\, (\bullet; \bullet))$ has type $(\Psi; \Delta) \Rightarrow F$ only if C expects the abstracted version of $(\Psi; \Delta)$ as input, and returns a value of type F. In this rule Π is not a constructor, it is the function which abstracts $\Psi; \Delta$, accordingly.

$$\frac{\Phi \vdash C \in \Pi\rho^L.\, (\Psi; \Delta) \Rightarrow F}{\Phi, \rho^L \vdash C, (\lambda\rho^L.\, (\bullet; \bullet)) \in (\Psi; \Delta) \Rightarrow F}\,\mathsf{tcnew}$$

This concludes our presentation of the typing rules for continuation stacks. The next step in this development leads to a typing discipline on states. We write $\vdash S \in F$ if S has type F.

Valid states: $\vdash S \in F$

Since we distinguish four different kind of states (according to if its body is a program, a list of declarations, a list of cases, or a meta-substitution), there are four different typing rules; tsprg, tsdec, tscase, and tssub. The design of all rules is inspired by the form of a cut-rule in the sense that F_1 and $(\Psi; \Delta)$ does not occur in the conclusion, respectively.

$$\frac{\Phi \vdash C \in F_1 \Rightarrow F \qquad \Phi; \cdot \vdash P \in F_1}{\vdash (\Phi; C \triangleright P) \in F}\,\mathsf{tsprg}$$

$$\frac{\Phi \vdash C \in (\Psi; \Delta) \Rightarrow F \qquad \Phi; \cdot \vdash D \in \Psi; \Delta}{\vdash (\Phi; C \triangleright D) \in F}\,\textsf{tsdec}$$

$$\frac{\Phi \vdash C \in F_1[\psi] \Rightarrow F \qquad \Phi; \cdot \vdash \psi; \delta : \Psi; \Delta \qquad \Psi; \Delta \vdash \Omega \in F_1}{\vdash (\Phi; C \triangleright (\psi; \delta) \sim \Omega) \in F}\,\textsf{tscase}$$

$$\frac{\Phi \vdash C \in (\Psi; \Delta) \Rightarrow F \qquad \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi; \Delta}{\vdash (\Phi; C \triangleright \psi; \delta) \in F}\,\textsf{tssub}$$

In summary, we have successfully established a typing discipline for states. What remains to be shown, is that any execution on the abstract machine preserves types. Specifically, we prove, that the type of a state remains invariant during computation.

Theorem 7.3 (Local type preservation for small-step semantics)

If $\mathcal{D} :: \vdash S \in F$
and $\mathcal{E} :: S \implies S'$
then $\vdash S' \in F$.

Proof: by case analysis on $\mathcal{E}$ using Lemma 6.20, Lemma 6.37, Lemma 6.22, Lemma 6.7. A detailed proof can be found in Appendix C. □

By a simple induction over the length of a computation trace, we generalize this result to the transitive closure of the transition relation.

Theorem 7.4 (Type preservation for small-step semantics)

If $\mathcal{D} :: S \overset{*}{\implies} S'$
and $\mathcal{E} :: \vdash S \in F$
then $\vdash S' \in F$.

Proof: by induction on $\mathcal{D}$ using Lemma 7.3. A detailed proof can be found in Appendix C. □

Note, that in that similarly to the proof of Theorem 6.38, neither this proof relies on the termination side condition (5.1) or the coverage side condition (5.2). In fact, even without these two side conditions, the small-step operational semantics is type-preserving.

In general, we are not interested in computations that terminate prematurely but only in computations that terminate with a legitimate return value. We will show in Section 7.3 that no computation can terminate prematurely. Legitimate states which mark the end of a computation are called *final states*. Their main characteristic is that their continuation stack is empty, and their program a value, which cannot be evaluated any further.

Definition 7.5 (Final state) *Let* Φ *be a parameter context and* V *be a value.* $\Phi; \star \triangleright V$ *is called a* final *state.*

Any computation trace, which starts in an initial state ($\Phi; \star \triangleright P$, for any regular Φ, and any P) ends in a final state whose parameter context is Φ again. The main insight is that trnew and trnewC are the only two rules that can insert and retract parameter blocks from Φ.

Lemma 7.6 (Parameter context preservation)
If $\Phi; \star \triangleright P \stackrel{*}{\Longrightarrow} \Phi'; \star \triangleright V$
then $\Phi = \Phi'$

Proof: by inspection of the transition rules. □

In summary, we have defined a small-step operational semantics that can be executed on an abstract machine. The semantics is type preserving (i.e. partially correct), which means that once a well-typed program is executed, the return value will also be well-typed. $\mathcal{M}_2^+$ proof terms can be directly executed on this machine, and without proof we conjecture that it computes exactly the same value one would expect when examining a proof term with the big-step operational semantics defined in Section 6.5.

Independent of this conjecture, we continue our analysis of termination and progress properties with the small-step semantics. Once shown that these two properties hold we have proven that $\mathcal{M}_2^+$ is sound.

7.2 Termination

In order for $\mathcal{M}_2^+$ to be a sound logic, all its proof terms must be realizers, that is that once applied to arguments, their computation eventually terminates and return the appropriate existentially quantified witness objects. Recall that recursion and case analysis are designed in order to support reasoning by induction over higher-order encodings in $\mathcal{M}_2^+$.

It is obvious, that without side condition (5.1) on Rctx not every computation terminates. For example,

$$\textbf{fun trans } E_1\ E_2 = \textbf{trans } E_1\ E_2$$

has type

$$\forall T : \mathrm{tp}. \forall E_1 : \mathrm{term}\ T. \forall E_2 : \mathrm{term}\ T. \exists D : E_1 \stackrel{1}{\Longrightarrow} E_2. \top$$

but it is not a realizer for this theorem because it will never terminate once applied to any type T, and well-typed terms E_1 and E_2. Moreover, this claim should not have a proof at all, since it is clearly false; there are well-typed terms that do not parallel reduce to each other. Thus, that all recursive functions in $\mathcal{M}_2^+$ terminate is a necessary precondition for the soundness of $\mathcal{M}_2^+$.

Why doesn't this function terminate? It doesn't terminate because in the recursive call the argument vector $E_1\ E_2$ is not smaller than the vector of arguments the function is initially called with. Instead, we must require for it to be strictly smaller, according to some well-founded ordering. This observation is the fundamental insight which allows us to design an appropriate syntactical criterion for side condition 5.1. Recall that in Section 5.6.1 where we introduced the rule Rctx, we already established a semantic termination condition. In essence, we can directly apply the results of previous work by Rohwedder and Pfenning [RP96] where they analyze lexicographic and simultaneous extensions of the subterm ordering. Other more complex terminations orderings are still work in progress.

In this section we discuss the matter of termination which we split into three parts. We first impose a restriction on the form of proof terms in Section 7.2.1 before we define a syntactic criterion for side condition 5.1 in Section 7.2.2. We then argue in Section 7.2.3 that the computation of any proof term on the abstract machine is always terminating.

7.2.1 Syntactic Restriction on Proof Terms

In order to simplify this presentation, we pose a few more syntactic restrictions on the form of proof terms. First, we consider only proof terms that start with a leading $\mu \mathbf{x} \in F.\, P$ where P itself does not contain any other occurrences of recursion operators. Therefore, we can easily localize the occurrences of the recursion variable $\mathbf{x}$ in the proof term. Because of an earlier restriction, we only examine proof terms which are in the Π_2-fragment of $\mathcal{M}_2^+$. Thus we can easily analyze all complete argument vectors in the recursive calls and compare them to the originally given one. Unfortunately, because of the distributed character of function calls arguments might be splattered all over the term. It goes without saying, that reasoning about proof terms in this generality is extremely complicated and convoluted. Therefore — as second restriction — we also restrict the form of recursive calls.

Typically, a recursive call is expressed via a list of declarations. To simplify matters, we only consider proof terms that use one **let** constructor to describe each recursive call and each call to a subroutine. Naturally, each declaration block may be preceded by several parameter block introductions. As example, consider the pbeta/pbeta-case in the proof of the diamond Lemma 3.7.

$$
\begin{array}{l}
\mid \ \mathbf{dia}\ \ (\text{pbeta}\ (\lambda x:\ \text{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l) \\
\qquad\quad (\text{pbeta}\ (\lambda x:\ \text{term}\ T.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^r\ x\ u)\ D_2^r) = \\
\quad \mathbf{let} \\
\quad\quad \mathbf{new}\ \underline{x} : \text{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\quad\quad \mathbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \mathbf{dia}\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1^r\ \underline{x}\ \underline{u}) \\
\quad \mathbf{in} \\
\quad\quad \mathbf{let} \\
\quad\quad\quad \mathbf{val}\ (Q_1, Q_2) = \mathbf{dia}\ D_2^l\ D_2^r \\
\quad\quad\quad \mathbf{val}\ E_1 = \mathbf{subst}\ P_1\ Q_1 \\
\quad\quad\quad \mathbf{val}\ E_2 = \mathbf{subst}\ P_2\ Q_2 \\
\quad\quad \mathbf{in} \\
\quad\quad\quad (E_1, E_2) \\
\quad\quad \mathbf{end} \\
\quad \mathbf{end}
\end{array}
$$

The first application of the induction hypothesis can be represented in $\mathcal{M}_2^+$ in many different ways, but for the purpose of this thesis, we require that it has the following form (we omit all type and formula annotations):

$$
\begin{array}{l}
\textbf{let} \\
\quad \nu\ (\underline{x} : \text{term}\ T_2, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L. \\
\quad \mathbf{y}_1 = \textbf{dia}\ T_1, \\
\quad \mathbf{y}_2 = \mathbf{y}_1\ E_1, \\
\quad \mathbf{y}_3 = \mathbf{y}_2\ E_1^l, \\
\quad \mathbf{y}_4 = \mathbf{y}_3\ E_1^r, \\
\quad \mathbf{y}_5 = \mathbf{y}_4\ (D_1^l\ \underline{x}\ \underline{u}), \\
\quad \mathbf{y}_6 = \mathbf{y}_5\ (D_1^r\ \underline{x}\ \underline{u}), \\
\quad \langle E_1', \mathbf{y}_7 \rangle = \mathbf{y}_6, \\
\quad \langle P_1, \mathbf{y}_8 \rangle = \mathbf{y}_7, \\
\quad \langle P_2, \mathbf{y}_9 \rangle = \mathbf{y}_8 \\
\textbf{in} \\
\quad \ldots
\end{array}
$$

$\mathbf{y}_1 ... \mathbf{y}_9$ are only auxiliary variables which bind partially instantiated proof terms. Throughout the remainder of this chapter, we only consider functions in $\mathcal{M}_2^+$ which are of this form and in addition which never reuse any of the auxiliary variables elsewhere in the proof term. In order to have a convenient form of revering to a recursive call in a proof term, we use the following shorthand notation.

$$\nu\ (\underline{x} : \text{term}\ T_2, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L. (E_1'\ \underline{x}, P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \textbf{dia}\ T_1\ E_1\ E_1^l\ E_1^r\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1^l\ \underline{x}\ \underline{u})$$

It seems as if these two restrictions limit the expressive power of $\mathcal{M}_2^+$. This is not true. There are two reasons: First, the subset we are considering is certainly superset of the surjective image of the naive formulation of the desugaring function. Thus, our termination argument applies to every function that we can write down in ML-notation. Second, we strongly believe, that any program $P \in F$ can be transformed into a program $P' \in F$ which lies within this fragment of $\mathcal{M}_2^+$. An exact investigation of this issue is left to future research, when we extend termination orderings beyond extensions of the subterm ordering and allow arbitrary well-founded termination orderings.

7.2.2 Syntactic Termination Criterion

We begin now with the discussion of a syntactic termination criterion for side condition (5.1). As example, consider the proof of diamond Lemma 3.7. The corresponding $\mathcal{M}_2^+$ function expects six arguments, and it has therefore the following form:

$$
\begin{array}{l}
\mu\textbf{dia}. \Lambda T : \text{tp}. \Lambda E : \text{term}\ T. \Lambda E^l : \text{term}\ T. \Lambda E^r : \text{term}\ T. \\
\quad \Lambda D^l : E \stackrel{1}{\Longrightarrow} E^l. \Lambda D^r : E \stackrel{1}{\Longrightarrow} E^r. \ldots
\end{array}
$$

In several places in the body of this function (indicated by ...), **dia** is used as the head of a recursive call. After adding syntactic sugar, the recursive calls in the pbeta/pbeta case (see Figure above), for example, are of the form

$$\textbf{dia}\ (D_1^l\ x\ u)\ (D_1^r\ x\ u)$$

and

$$\textbf{dia}\ D_2^l\ D_2^r$$

where the variables in the head of the function have been instantiated as follows.

$$\begin{array}{rcl} T & = & T_1 \\ E & = & (\text{app}\ (\text{lam}\ E_1)\ E_2) \\ E^l & = & (E_1^l\ E_2^l) \\ E^r & = & (E_1^r\ E_2^r) \\ D^l & = & \text{pbeta}\ (\lambda x : \text{term} T_2.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l \\ D^r & = & \text{pbeta}\ (\lambda x : \text{term} T_2.\, \lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^r\ x\ u)\ D_2^r \end{array}$$

Clearly, the argument vector decreases with each recursive call, because D_1^l, D_2^l are subterms of D^l, and D_1^r, D_2^r are subterms of D^r. The same observation holds for all other occurrences of recursive calls, and therefore **dia** terminates overall.

In this work, we also consider simultaneous and lexicographic extensions of the subterm ordering. Simultaneous orderings, on the other hand, also extend subterm orderings. The proof of the diamond lemma, for example, satisfies the simultaneous ordering $[D^l\ D^r]$ which means that either D^l or D^r becomes smaller in every recursive call. But unlike for lexicographic orderings neither D^l nor D^r can ever become bigger.

Another example is the cut-elimination theorem for classical or intuitionistic logic. It requires a termination ordering which is lexicographic in the cut-formula A, and simultaneous in the left derivation D^l and the right derivation D^r [Pfe95]. Formally, this ordering is written as $\{A\ [D^l\ D^r]\}$.

For mutually recursive functions we have to precisely track how the different functions call each other. We do this by introducing positions which are lists of variable names, bound in the different mutually recursive parts of a theorem, as we used it for the proof of Lemma 4.11.

Definition 7.7 (Termination order)

$$\begin{array}{llcl} \textit{Position:} & P & ::= & (x_1, \ldots, x_n) \\ \textit{Termination order:} & O & ::= & P \mid \{O_1, \ldots, O_n\} \mid [O_1, \ldots, O_n] \end{array}$$

For simplicity, we omit the mutual recursive case from this discussion. It is entirely orthogonal to this development an can be easily added. We write 'order $(O, M_1 \ldots M_n)$' to extract the vector of significant arguments for the well-founded ordering. The comparison functions '$<_O$' and '$\leq_O$' on those vectors are defined as in [RP96], and $<_O$ is well-founded for any ordering O. We specify a syntactic termination condition for Rctx inspired by [RP96].

Definition 7.8 (Termination condition for Rctx) *Let* $\mu \mathbf{x} \in F.\, \Lambda x_1 : A_1. \ldots \Lambda x_n : A_n.\, P$ *a proof term,* O *a termination order,* $\nu\ \rho_1^L. \ldots \nu\ \rho_m^L. \ldots = \mathbf{x}\ M_1 \ldots M_n$ *a recursive call in* P*, valid in* $\Psi; \Delta$ *and* ψ *a substitution* $\Psi \vdash \psi \in x_1 : A_1, \ldots, x_n : A_n$*.*

We say, that this recursive calls satisfies the termination condition *if and only if*

$$\textit{order}\ (O, M_1 \ldots M_n) <_O \textit{order}\ (O, x_1[\psi] \ldots x_n[\psi])$$

One important property of Rohwedder and Pfenning's termination order is that the well-foundedness of $<_O$ is preserved under substitution. Formally, if

$$\text{order}\ (O, M_1 \ldots M_n) <_O \text{order}\ (O, x_1[\psi] \ldots x_n[\psi])$$

in Ψ then

$$\text{order } (O, M_1[\psi'] \ldots M_n[\psi']) <_O \text{order } (O, x_1[\psi \circ \psi'] \ldots x_n[\psi \circ \psi'])$$

for all ψ' for which $\Psi' \vdash \psi' \in \Psi$ holds.

The choice of the termination order is very important especially for automated theorem proving. It dictates the general form of the induction hypothesis. To see that the proof term **dia** terminates, for example, it is enough to check all recursive calls via the termination order $[D^l]$ (see Figure 4.4)! For proof generation on the other hand, we can use the termination order to encode additional information about recursive calls. There are infinitely many recursive calls satisfying $[D^l]$ because D^r remains unrestricted. On the other hand, if we use the termination order $[D^l\ D^r]$, there are only very few recursive call satisfying this termination order.

7.2.3 Termination Theorem

With the syntactic side condition specified in Definition 7.8, all recursive functions in $\mathcal{M}_2^+$ are terminating. Under the regular world assumption, a program is computed with respect to a regularly formed parameter context Φ. Apart from this, the program must be closed with respect to meta-variables and LF-variables. We assume that all meta-variables acting as sub-routine calls have been instantiated by terminating proof terms.

The proof of the termination theorem is based on the following idea: We record the argument vector the function is called with. When the abstract machine executes a recursive call, we compare it to the argument vector of the recursion variable. By unfolding Condition 7.8 we ensure, that the new argument vector is smaller. In addition, in all other situation the abstract machine continues by executing a subterm of the current program (or a list of cases) and therefore, the computation must eventually terminate.

Theorem 7.9 (Termination) *We consider the evaluation of a function of type* $\forall x_1 : A_1. \ldots \forall x_n : A_n. \exists y_1 : A'_1. \ldots \exists y_m : A'_m. \top$ *applied to arguments* $M_1, \ldots, M_n$ *in a parameter context* Φ*. The termination order is* O *and all procedures (used as lemmas) terminate.*

1. *If* $S = \Phi; C \triangleright P$ *and* P *is not a value*
   then $S \stackrel{*}{\Longrightarrow} \Phi; C \triangleright V$
   or the computation terminates prematurely.

2. *If* $S = \Phi; C \triangleright (\psi; \delta) \sim \Omega$
   then $S \stackrel{*}{\Longrightarrow} \Phi; C \triangleright V$
   or the computation terminates prematurely.

Proof: by induction lexicographically on 'order $(O, M_1 \ldots M_n)$' and $(P(2)$ and $\Omega(3))$. A detailed proof can be found in Appendix C. □

The evaluation of any program which — syntactically speaking — introduces only one outermost recursion variable and whose recursive calls and appeals to lemmas appear only in their natural form (without unnecessary reuses of auxiliary introduced meta-variables) must terminate given that side condition 5.1 is satisfied. The computation may still terminate prematurely because of an inexhaustive match exception from a case statement, but from the results discussed in the next section we learn that this cannot happen.

7.3 Coverage

In this section we show that the evaluation of any program that is not a value always makes progress. By inspecting the transition rules defining the small-step operating semantics, we can easily recognize, that all but three rules are defined in a way that they must make progress. For every right hand side of such a transition rule there is another rule whose left hand side matches it. Moreover, for every program, there is a rule that matches it. The three exceptions are the rules `trcase`, `tryes`, and `trno`. What we have to guarantee is that once `trcase` is applied, there is a case in Ω which triggers `tryes`.

What would happen if there is no such case in Ω? In such a situation `trno` would be applied for all cases in Ω until $\Omega = \cdot$, and consequently, the evaluation would get stuck because there is no rule which applies to an empty Ω! This situation must never occur. It would violate the important progress property.

Thus, we need to show that given side condition 5.2 is satisfied Ω *covers* all possible cases. We first introduce the notion of most general unifier in Section 7.3.1 whose existence is crucial in our argument. In Section 7.3.2 we specify a syntactical and machine checkable coverage criterion for side condition 5.2, and finally, we show in Section 7.3.3, that every program that satisfies this syntactic criterion makes always progress.

7.3.1 Motivation

Our coverage analysis relies on the fact that canonical forms in LF are inductive [HHP93, Coq91]. Since every well-typed object in LF possesses a canonical form, case analysis of an object reduces to inductive reasoning about the canonical forms of its type. Unlike in Coq or Isabelle where datatypes are defined by a set of finitely many constructors whose type satisfy the positivity condition, the situation is different in our setting. We are considering arbitrary higher-order LF objects well-typed under the regular world assumption. Our design is sound, because we know that every object in LF has a canonical form, independently if it is a function or not.

To illustrate the idea behind the coverage, we return to Example 5.12, and observe, that under the regular world assumption, the E : term T can take three different (most general) forms:

1. $E' = \underline{x} : \text{term } T$
2. $E'' = \text{lam } (\lambda x : \text{term } T_1 .\, E_1\ x) : \text{term } (T_1 \text{ arrow } T_2)$
3. $E''' = \text{app } E_1\ E_2 : \text{term } T_1$

The goal of coverage is to decide, if the list of cases given in Example 5.12 really covers all the cases and indeed, intuitively, in this situation it does. Consider the following list of cases Ω. E' is being matched by the first case, E'' by the second, and E''' by the third.

$$\begin{array}{lll}
T : \text{tp}, (\underline{x} : \text{term } T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L & \triangleright T/T, \underline{x}/E & \mapsto \ldots \\
T_1 : \text{tp}, T_2 : \text{tp}, E' : \text{term } T_1 \to \text{term } T_2 & \triangleright (T_1 \text{ arrow } T_2)/T, \text{lam } (\lambda x : \text{term } T_1 .\, E_1\ x)/E & \mapsto \ldots \\
T_1 : \text{tp}, T_2 : \text{tp}, E_1 : \text{term } (T_2 \text{ arrow } T_1), & & \\
E_2 : \text{term } T_2 & \triangleright T_1/T, \text{app } E_1\ E_2/E & \mapsto \ldots
\end{array}$$

But how exactly do we decide this property? The answer lies in the proper analysis of generalized substitutions as part of each case declared in Ω. For this purpose of this example, consider an environment η (also a generalized substitution), where we presuppose a type 'nat' for natural numbers.

$$(\underline{x}' : \text{term nat}, \underline{u}' : \underline{x}' \stackrel{1}{\Longrightarrow} \underline{x}')^L \vdash \text{nat}/T, \underline{x}'/E \in T : \text{tp}, E : \text{term } T$$

To see that the first case covers this environment one has to provide a new environment η', such that $(T/T, \underline{x}/E) \circ \eta' = \eta$. Indeed such an environment exists and it has the following form.

$$(\underline{x}' : \text{term nat}, \underline{u}' : \underline{x}' \stackrel{1}{\Longrightarrow} \underline{x}')^L \vdash \text{nat}/T, (\underline{x}', \underline{u}')/(\underline{x}, \underline{u}) \in T : \text{tp}, (\underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L$$

Therefore, in the general case if we want to guarantee that a list of cases Ω contains an applicable case $(\Psi' \triangleright \psi \mapsto P)$ we have to ensure, that for *every* $\Psi_0 \vdash \psi' \in \Psi$, *there exists a* substitution $\Psi_0 \vdash \psi'' \in \Psi'$ such that $\psi \circ \psi'' = \psi'$.

This small example already exhibits several important aspects of our design. First, the list of cases from Example 5.12 is a result of analyzing cases over *one* LF object E : term T. In general, this need not to be the case, because in Example 5.13 we distinguish cases over *two* LF objects simultaneously. Second, pattern matching and coverage analysis are very closely connected to generalized substitutions, their decompositions, and as we will discuss below, unification of types. Third, our coverage criterion excludes impossible cases. For example, if we return to the formalization of the diamond Lemma 4.6 we first analyzed four cases for D^l, but only one for D^r, namely the parameter case. We deduced from typing constraints that the other three candidates for D^r cannot occur.

In order to develop a formal coverage criterion, we must develop a complete algorithm to generate all possible forms of ψ' above. Our algorithm is defined by iteration, that expects one substitution together with its co-domain as input and generates a list of refining substitutions.

Starting from the identity substitution id_Ψ it successively applies a refinement step by non-deterministically picking one variable declaration from the co-domain of the current environment, and analyzes its forms in a most general fashion. The result is a set of generalized substitutions describing all possible shapes of Ψ.

Intuitively, the algorithm computes a list of "forms" describing the most general form of the environment, and by construction the coverage criterion is guaranteed to be satisfied. The algorithm terminates if the following criterion is satisfied: *every* substitution in the returned from the algorithm matches *some* case in Ω.

In this presentation, however, we are slightly more restrictive and expect Ω not only to match the set of substitutions calculated by the coverage algorithm, but "to be equal" to it. That means, we only allow Ω's whose embedded substitutions can be generated by our coverage algorithm. This restriction can be easily lifted, but because we are predominantly interested in automation of case analysis we leave this issue to future work.

The workings of the coverage algorithm is best illustrated by an example. First we consider the identity generalized substitution:

$$T' : \text{tp}, E' : \text{term } T' \vdash T'/T, E'/E \in T : \text{tp}, E : \text{term } T$$

What are the possible forms of E'? There are the three possibilities as already mentioned above. First $E' = \underline{x}$:

$$T'' : \text{tp}, (\underline{x} : \text{term } T'', \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L \vdash \underline{x} : \text{term } T''$$

Since $E' = \underline{x}$, their types must be convertible (term T' = term T''), and therefore $T'' = T'$!

$$T'' : \text{tp}, (\underline{x} : \text{term } T'', \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L \vdash T''/T, \underline{x}/E \in T : \text{tp}, E : \text{term } T$$

The second possibility is that E is instantiated with a term starting with lam.

$$T_1'' : \text{tp}, T_2'' : \text{tp}, E'' : \text{term } T_1'' \to \text{term } T_2'' \vdash \text{lam } (\lambda x : \text{term } T_1''.\, E'' x) : \text{term } (T_1'' \text{ arrow } T_2'')$$

By using the same argument as above, we obtain that $E = \text{lam } (\lambda x : \text{term } T_1.\, E'' x)$, and that the $T = T_1''$ arrow T_2''.

$$\begin{array}{l} T_1'' : \text{tp}, T_2'' : \text{tp}, E'' : \text{term } T_1'' \to \text{term } T_2'' \\ \quad \vdash (T_1'' \text{ arrow } T_2'')/T, (\text{lam } (\lambda x : \text{term } T_1''.\, E'' x))/E \in T : \text{tp}, E : \text{term } T \end{array}$$

The third substitution for $E = \text{app } E_1\ E_2$ follows from a similar argument. It is easy to see that all substitutions are strict in their co-domain.

When we compare E : term T and lam $(\lambda x : \text{term } T_1''.\, E'' x)$: term $(T_1''$ arrow $T_2'')$ we conclude $T = T_1''$ arrow T_2''. Technically speaking, the operation hidden behind this comparison is unification. It cannot be matching, because in a different example E's type might be more constrained. For example it could be E : term $((T_1$ arrow $T_2)$ arrow $T_3)$, and then $T_1'' = (T_1$ arrow $T_2)$ and $T_2'' = T_3$.

The second case of this example exhibits that the unification problems in question are certainly not first-order, but higher-order due to the higher-order character of the underlying representation. Higher-order unification problems are in general undecidable [Hue73], and therefore we must restrict our considerations to problems which are decidable namely those which guarantee the existence of *one* most general unifiers. Certainly, one can generalize this work to unification problems that have *finite* complete sets of general unifiers, but in all our experiments the a one element complete set of general unifiers suffices.

Most unification problems we are dealing with lie in fact in the Miller's pattern fragment [Mil91] which guarantee most general unifiers if they exist, but some of our unification problems fall outside the pattern fragment. Those unification problems have in our experience non pattern occurrences of the form "existential variable applied to existential variable", but in most cases they are still decidable by reordering the unification goals, in a way very similar to the way how we extended pattern matching to strict matching in Section 6.4.2. Unlike in matching, even if all existential variables occur in some strict position in the unification problem, it still need not guarantee decidability. This is clearly exemplified by the following example

$$\text{c } X_1\ X_2\ M_1\ M_2\ (X_1\ M_1) \approx \text{c } Y_1\ Y_2\ N_1\ N_2\ (Y_2\ N_2)$$

where 'c' is a constant and $X_1, X_2, M_1, M_2, Y_1, Y_2, N_1, N_2$ are existential variables, all occurring in strict positions in this equation. It is easy to see, that this unification problem does not have a most general unifier because it reduces to a flex-flex pair $X_1\ M_1 \approx Y_2\ N_2$.

Therefore, we restrict the following discussion to unification problems for which most general unifiers exists. In addition, we expect the each variable in the co-domain of the unifying substitution occurs strictly in it, otherwise we might not be able to execute the proof term on our abstract machine. This is not just a technical restriction but it also has practical consequences. Case analyses which fall outside this fragment are simply not valid in $\mathcal{M}_2^+$ and can hence not be expressed. In this work we do not present a unification algorithm for this fragment, we merely presuppose the existence of most general unifiers.

Definition 7.10 (Unifiers) *We follow standard practice and write $M_1 \approx M_2$, and $A_1 \approx A_2$ for equations describing a unification problem N. ψ is a solution for N if $M_1[\psi] = M_2[\psi]$ for each equation $M_1 \approx M_2 \in N$ and $A_1[\psi] = A_2[\psi]$ for each equation $A_1 \approx A_2 \in N$. Formally, we write $\psi \in unify\,(N)$. We say that σ is a most general unifier of N if for every other solution ψ, there exists a ψ' s.t. $\sigma \circ \psi' = \psi$. Formally we write $\sigma \in mgu\,(N)$.*

We return to the proof of the substitution Lemma 4.5 in order to determine the exact form of the unification problem. It clearly shows how declarations of functional type must be split. We consider a snapshot of the context shortly before the case analysis over D_1 takes place.

$$\begin{array}{l} T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1, E_1' : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1, E_2 : \mathrm{term}\ T_2, E_2' : \mathrm{term}\ T_2, \\ D_1 : \Pi y : \mathrm{term}\ T_2.\, y \stackrel{1}{\Longrightarrow} y \to (E_1\ y) \stackrel{1}{\Longrightarrow} (E_1'\ y), D_2 : E_2 \stackrel{1}{\Longrightarrow} E_2' \end{array}$$

D_1 is of functional type. What possible forms can it take? We extract this list from the proof term from Figure 4.3.

$$\begin{array}{lcl}
D_1^{(1)} & : & \Pi y : \mathrm{term}\ T_2.\, y \stackrel{1}{\Longrightarrow} y \to \underline{x} \stackrel{1}{\Longrightarrow} \underline{x} \\
& = & \lambda y : \mathrm{term}\ T_2.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \underline{u} \\
D_1^{(2)} & : & \Pi y : \mathrm{term}\ T_2.\, y \stackrel{1}{\Longrightarrow} y \to y \stackrel{1}{\Longrightarrow} y \\
& = & \lambda y : \mathrm{term}\ T_2.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, v \\
D_1^{(3)} & : & \Pi y : \mathrm{term}\ T_2.\, y \stackrel{1}{\Longrightarrow} y \to (\mathrm{app}\ (\mathrm{lam}\ (\lambda z.\, \hat{E}_1\ y\ z))\ (\hat{E}_2\ y)) \stackrel{1}{\Longrightarrow} (\hat{E}_1'\ y\ (\hat{E}_2'\ y)) \\
& = & \lambda y : \mathrm{term}\ T_2.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \mathrm{pbeta}\ (\lambda x : \mathrm{term}\ T_1.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, \hat{D}_1\ y\ v\ x\ u)\ (\hat{D}_2\ y\ v) \\
D_1^{(4)} & : & \Pi y : \mathrm{term}\ T_2.\, y \stackrel{1}{\Longrightarrow} y \to (\mathrm{lam}\ (\lambda z.\, \hat{E}_1\ y\ z)) \stackrel{1}{\Longrightarrow} (\mathrm{lam}\ (\lambda z.\, \hat{E}_1'\ y\ z)) \\
& = & \lambda y : \mathrm{term}\ T_2.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \mathrm{plam}\ (\lambda x : \mathrm{term}\ T_1.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, \hat{D}_1\ y\ v\ x\ u) \\
D_1^{(5)} & : & \Pi y : \mathrm{term}\ T_2.\, y \stackrel{1}{\Longrightarrow} y \to (\mathrm{app}\ (\hat{E}_1\ y)\ (\hat{E}_2\ y)) \stackrel{1}{\Longrightarrow} (\mathrm{app}\ (\hat{E}_1'\ y)\ (\hat{E}_2'\ y)) \\
& = & \lambda y : \mathrm{term}\ T_2.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \mathrm{papp}\ (\hat{D}_1\ y\ v)\ (\hat{D}_2\ y\ v)
\end{array}$$

The variables marked with a hat such as $\hat{E}$ and $\hat{D}$ are new. Going back to the algorithm, which splits variables from the co-domain of a substitution, we must therefore pay special attention to variables of functional type. Also in this example, we initialize the coverage algorithm with the identity substitution.

$$\begin{array}{l}
T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1, E_1' : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1, E_2 : \mathrm{term}\ T_2, E_2' : \mathrm{term}\ T_2, \\
\quad D_1 : \Pi y : \mathrm{term}\ T_2.\, y \stackrel{1}{\Longrightarrow} y \to (E_1\ y) \stackrel{1}{\Longrightarrow} (E_1'\ y), D_2 : E_2 \stackrel{1}{\Longrightarrow} E_2' \\
\quad \vdash T_1/T_1, T_2/T_2, E_1/E_1, E_1'/E_1', E_2/E_2, E_2'/E_2', D_1/D_1, D_2/D_2 \\
\quad \in T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1, E_1' : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1, E_2 : \mathrm{term}\ T_2, E_2' : \mathrm{term}\ T_2, \\
\quad\quad D_1 : \Pi y : \mathrm{term}\ T_2.\, y \stackrel{1}{\Longrightarrow} y \to (E_1\ y) \stackrel{1}{\Longrightarrow} (E_1'\ y), D_2 : E_2 \stackrel{1}{\Longrightarrow} E_2'
\end{array}$$

The coverage algorithm picks D_1 for splitting, and from its type it deduces, that there are two local parameters, $y : \mathrm{term}\ T_2$ and $v : y \stackrel{1}{\Longrightarrow} y$ which may occur free in its body. This means, that when we consider refining D_1 to app $\hat{D}_1\ \hat{D}_2$ we must let $\hat{D}_1$ and $\hat{D}_2$ depend on y, v! Therefore, we rely on another algorithm which raises the types of the arguments by introducing those new dependencies. According to its intention, the algorithm is called *raising*. Given the type A of a constructor, raise (Γ, A) generates a list of raised argument variables in form of a generalized context and a refined target type. Note, that in order to execute raising on the LF level we actually invoke the abstraction function defined in Section 6.2.2 which accounts for subordination. In the example of papp above,

$$\begin{array}{rl}
& \text{raise}\ (y : \text{term}\ T_2, v : y \overset{1}{\Longrightarrow} y, \\
& \quad \Pi t_1 : \text{tp}.\, \Pi t_2 : \text{tp}. \\
& \quad \Pi e_1 : \text{term}\ (t_2\ \text{arrow}\ t_1).\, \Pi e_1' : \text{term}\ (t_2\ \text{arrow}\ t_1).\, \Pi e_2 : \text{term}\ t_2.\, \Pi e_2' : \text{term}\ t_2. \\
& \quad\quad e_1 \overset{1}{\Longrightarrow} e_1' \to e_2 \overset{1}{\Longrightarrow} e_2' \to (\text{app}\ e_1\ e_2) \overset{1}{\Longrightarrow} (\text{app}\ e_1'\ e_2')) \\
= & \hat{T}_1 : \text{tp}, \hat{T}_2 : \text{tp}, \\
& \hat{E}_1 : \text{term}\ T_2 \to \text{term}\ (\hat{T}_2\ \text{arrow}\ \hat{T}_1), \hat{E}_1' : \text{term}\ T_2 \to \text{term}\ (\hat{T}_2\ \text{arrow}\ \hat{T}_1), \\
& \hat{E}_2 : \text{term}\ T_2 \to \text{term}\ \hat{T}_2, \hat{E}_2' : \text{term}\ T_2 \to \text{term}\ \hat{T}_2, \\
& \hat{D}_1 : \Pi y : \text{term}\ T_2.\, y \overset{1}{\Longrightarrow} y \to (\hat{E}_1\ y) \overset{1}{\Longrightarrow} (\hat{E}_1'\ y), \\
& \hat{D}_2 : \Pi y : \text{term}\ T_2.\, y \overset{1}{\Longrightarrow} y \to (\hat{E}_2\ y) \overset{1}{\Longrightarrow} (\hat{E}_2'\ y) \\
& \triangleright (\text{app}\ (\hat{E}_1\ y)\ (\hat{E}_2\ y)) \overset{1}{\Longrightarrow} (\text{app}\ (\hat{E}_1'\ y)\ (\hat{E}_2'\ y))
\end{array}$$

As discussed already the subordination relation satisfies that

$$\begin{array}{lcl}
\text{term} & \not\prec & \text{tp} \\
\text{arrow} & \not\prec & \text{tp} \\
\text{arrow} & \not\prec & \text{term}
\end{array}$$

and therefore

$$\begin{array}{lcl}
\Pi(y : \text{term}\ T_2, v : y \overset{1}{\Longrightarrow} y).\, \text{tp} & = & \text{tp} \\
\Pi(y : \text{term}\ T_2, v : y \overset{1}{\Longrightarrow} y).\, \text{term}\ T_2' & = & \text{term}\ T_2 \to \text{term}\ T_2'
\end{array}$$

which explains the form of the raised version of the type of papp. In general, raising is defined by the judgment

$$\textit{Raising:}\quad \Psi_1 \vdash \text{raise}\ (\Gamma, A_1) = (\Psi_2 \triangleright A_2)$$

and two rules specify the behavior of the raising algorithm.

$$\frac{}{\Psi \vdash \text{raise}\ (\Gamma, B) = (\cdot \triangleright \Pi\Gamma.\, B)}\ \textsf{raisebase}$$

$$\frac{\Psi_1, x : \Pi\Gamma.\, A_1 \vdash \text{raise}\ (\Gamma, A_2[x\ \Gamma/x]) = (\Psi_2 \triangleright A')}{\Psi_1 \vdash \text{raise}\ (\Gamma, \Pi x : A_1.\, A_2) = (x : \Pi\Gamma.\, A_1, \Psi_2 \triangleright A')}\ \textsf{raisepi}$$

Lemma 7.11 (Properties of Raising)
If $\Psi \vdash M : A$
and $\Psi \vdash \textit{raise}\ (\Gamma) A = (\Psi' \triangleright B)$
then $\Psi, \Psi' \vdash \lambda\Gamma.\, M\ (\Psi'\ \Gamma) : \Pi\Gamma.\, B$

Proof: by induction on A. □

Once the raised version of papp is calculated, we have to unify the originally picked

$$D_1 : \Pi y : \text{term}\ T_2.\, y \overset{1}{\Longrightarrow} y \to (E_1\ y) \overset{1}{\Longrightarrow} (E_1'\ y)$$

with the refined version (where we are overly explicit in the arguments applied to the constant).

$$\begin{array}{l}\lambda y : \text{term } T_2.\, \lambda v : y \overset{1}{\Longrightarrow} y.\, \text{papp } \hat{T}_1\ \hat{T}_2\ (\hat{E}_1\ y)\ (\hat{E}'_1\ y)\ (\hat{E}_2\ y)\ (\hat{E}'_2\ y)\ (\hat{D}_1\ y\ v)\ (\hat{D}_1\ y\ v)\\ : \Pi y : \text{term } T_2.\, y \overset{1}{\Longrightarrow} y \to (\text{app } (\hat{E}_1\ y)\ (\hat{E}_2\ y)) \overset{1}{\Longrightarrow} (\text{app } (\hat{E}'_1\ y)\ (\hat{E}'_2\ y))\end{array}$$

As result we obtain the following two equations:

$$D_1 \approx \begin{array}{l}\lambda y : \text{term } T_2.\, \lambda v : y \overset{1}{\Longrightarrow} y.\\ \quad \text{papp } \hat{T}_1\ \hat{T}_2\ (\hat{E}_1\ y)\ (\hat{E}'_1\ y)\ (\hat{E}_2\ y)\ (\hat{E}'_2\ y)\\ \qquad (\hat{D}_1\ y\ v)\ (\hat{D}_1\ y\ v)\end{array} \tag{7.1}$$

$$\begin{array}{l}\Pi y : \text{term } T_2.\, y \overset{1}{\Longrightarrow} y\\ \quad \to (E_1\ y) \overset{1}{\Longrightarrow} (E'_1\ y)\end{array} \approx \begin{array}{l}\Pi y : \text{term } T_2.\, y \overset{1}{\Longrightarrow} y\\ \quad \to (\text{app } (\hat{E}_1\ y)\ (\hat{E}_2\ y)) \overset{1}{\Longrightarrow} (\text{app } (\hat{E}'_1\ y)\ (\hat{E}'_2\ y))\end{array} \tag{7.2}$$

Equation (7.1) binds D_1, and Equation (7.2) expresses that the types of both participating objects must be unifiable.

In this situation, the unification problem has one most general solution. We show a version of the solution substitution whose domain equals the co-domain of the substitution modeling the environment we have originally started with.

$$\psi = \begin{array}{l}\hat{T}_1/T_1, T_2/T_2, (\text{app } (\hat{E}_1\ y)\ (\hat{E}_2\ y))/E_1, (\text{app } (\hat{E}'_1\ y)\ (\hat{E}'_2\ y))/E'_1, E_2/E_2, E'_2/E'_2,\\ \lambda y : \text{term } T_2.\, \lambda v : y \overset{1}{\Longrightarrow} y.\, \text{papp } \hat{T}_1\ \hat{T}_2\ (\hat{E}_1\ y)\ (\hat{E}'_1\ y)\ (\hat{E}_2\ y)\ (\hat{E}'_2\ y)\ (\hat{D}_1\ y\ v)\ (\hat{D}_1\ y\ v)/D_1,\\ D_2/D_2\end{array}$$

The co-domain of this substitution can now be easily read out of substitution itself.

$$\begin{array}{l}\Psi = \hat{T}_1 : \text{tp}, \hat{T}_2 : \text{tp}, T_2 : \text{tp},\\ \hat{E}_1 : \text{term } T_2 \to \text{term } (\hat{T}_2 \text{ arrow } \hat{T}_1), \hat{E}'_1 : \text{term } T_2 \to \text{term } (\hat{T}_2 \text{ arrow } \hat{T}_1),\\ \hat{E}_2 : \text{term } T_2 \to \text{term } \hat{T}_2, \hat{E}'_2 : \text{term } T_2 \to \text{term } \hat{T}_2,\\ E_2 : \text{term } T_2, E'_2 : \text{term} T_2,\\ \hat{D}_1 : \Pi y : \text{term } T_2.\, y \overset{1}{\Longrightarrow} y \to (\hat{E}_1\ y) \overset{1}{\Longrightarrow} (\hat{E}'_1\ y),\\ \hat{D}_2 : \Pi y : \text{term } T_2.\, y \overset{1}{\Longrightarrow} y \to (\hat{E}_2\ y) \overset{1}{\Longrightarrow} (\hat{E}'_2\ y),\\ D_2 : E_2 \overset{1}{\Longrightarrow} E'_2\end{array}$$

Note that by construction Ψ is strict in ψ as required. We want to stress, that while not all variables are involved in the unification operation their types might be. The substitution ψ, for example, acts as identity on T_2, E_2, E'_2 and D_2, and the change in types is best observed directly in the proof of the diamond Lemma 4.6. There, while splitting the left reduction of $E \overset{1}{\Longrightarrow} E^l$, case analysis instantiates E with another term, e.g. a parameter $\underline{x}$ in the global parameter case, and therefore the type of the right reduction $D^r : E \overset{1}{\Longrightarrow} E^r$ changes to $D^r : \underline{x} \overset{1}{\Longrightarrow} E^r$ — even though D^r itself does not change. New information acquired during case analysis may therefore be recorded in the types of other assumptions.

Back to our example. One can easily verify that the substitution ψ is most general. Given any solution η of the unification problem specified by Equation (7.1) and Equation (7.2), it can be rewritten as $\psi \circ \eta'$. As a matter of fact, this observation already provides the basic insight into our coverage argument, and the reader is invited to look out how this fact is used in the proof.

Naturally, in general, the unification problem may not have a solution. There are a few ways how unification in Equation (7.2) can fail: There can be a constant/constant clash, a constant/variable clash (where the variable is locally bound in the term), a constant/parameter clash (where the parameter is represented by a parameter variable in the generalized context) a variable/variable clash, a variable/parameter clash, or a parameter/parameter clash. If any of these clashes should occur when analyzing cases, one can be sure — because of the uniqueness of typing — that the affected case does not apply. The user is invited to himself/herself, that the algorithm also returns the other four substitutions necessary to cover $D_1^{(1)}, \ldots, D_1^{(4)}$.

In summary, we have motivated an algorithm to compute a complete list of substitutions which describe all possible instantiations of any generalized context Ψ. In the next section we formally define this algorithm in form of a syntactical criterion for side condition 5.2, and in Section 7.3.3 we prove, that it indeed guarantees correctness. $\mathcal{M}_2^+$ owes the feasibility of this approach to the logical framework LF and pattern unification, in particular, the existence of canonical forms, and the the existence of most general unifiers.

7.3.2 Coverage Condition

We begin now with the formal presentation of the coverage algorithm. The goal is to establish a criterion on the case rule that guarantees that any program of the form 'case $(\psi; \delta)$ of Ω' covers all possible cases. For the sake of this exposition, we are very restrictive about the form of Ω; specifically, we require that all cases in Ω are in one-to-one correspondence with the outcome of the coverage algorithm sketched in the previous section. The reason is, that the implementation described in Chapter 8 follows the outline of this algorithm *to generate* cases hereby trivially guaranteeing its correctness. On the other hand, one can easily extend this criterion *to check* if a given set of cases really covers all possibility, but we leave this issue to future investigation.

Recall that Ω is defined as a list of cases, and each case has the form $(\Psi \triangleright \psi \mapsto P)$. For this discussion the form of the different P's is unimportant. What is important are the Ψ's and the ψ's, the main players in the coverage algorithm sketched above. As we will see below, the outcome of the algorithm is guaranteed to cover all cases, and thus we introduce as an auxiliary construct a list of pairs $(\Psi \triangleright \psi)$ which we call a *cover* for Ω.

$$\textit{Cover:} \quad \omega \quad ::= \quad \cdot \mid \omega, (\Psi \triangleright \psi)$$

Next, we formalize the coverage algorithm. Several judgments are involved in its definition and we discuss each one and its implementation in turn. According to the canonical form theorem, there are three possible head constructors for each object of a type; a constant defined in the signature (Constant Coverage), a local parameter introduced by a λ-binder in the case that it is a function (Local Parameter Coverage), or a global parameter which is part of the parameter context (due to the regular closed world assumption). All three cases are treated in a very similar way, special attention has to be paid to the last case: The form of parameter contexts is regular, and it is inductively described by context schemas; context schemas consist of several context blocks, and each context block of several BLOCK-declarations. Every possibility has to be accounted for, that means we have to traverse and examine each declaration in every BLOCK-block (Global Parameter Coverage) of the specified context schema (Schematic Coverage).

There are two special judgments combining partial coverage results; first, there is 'single coverage' which combines the results of splitting one single variable, and there is a judgment that allows the algorithm to successively and non-deterministically pick one variable from the co-domain of an already computed cover and split it. The result is a new, further refined cover.

Judgments

Constant Coverage:	$\Psi_1; x : \Pi\Gamma_x. B_x; \Psi_2 \vdash \Sigma \gg \omega$ cover
Local Parameter Coverage:	$\Psi_1; x : \Pi\Gamma_x. B_x; \Psi_2 \vdash \Gamma \gg \omega$ cover
Global Parameter Coverage:	$\Psi_1; x : \Pi\Gamma_x. B_x; \Psi_2; \Psi_3 \vdash \rho \gg \omega$ cover
Schematic Coverage:	$\Psi_1; x : \Pi\Gamma_x. B_x; \Psi_2 \vdash S \gg \omega$ cover
Single Coverage:	$\Psi \vdash \omega$ cover
Multiple Coverage:	$\Psi \vdash \omega$ cover*

These six judgments specify the coverage algorithm and its operational meaning is defined by inference rules. The first three judgments are defined by three rules each. For example, there is a rule (whose name ends in empty) for the empty signature, the empty local context, and the empty BLOCK-block. Depending on if a most-general unifier can be determined using the construction sketched above, there is a rule (whose name ends in unify) which adds a new entry to the cover. If a most-general unifier does not and cannot exist there is a rule (whose name ends in skip) which skips ahead and examines the next constant/local parameter/global parameter.

As representation invariant, in first two and the fourth judgment, $\Psi_1, x : \Pi\Gamma_x. B_x, \Psi_2$ is a valid generalized context, and x is the assumption for which cases are to be considered. Similarly, in the third judgment, $\Psi_1; x : \Pi\Gamma_x. B_x; \Psi_2; \Psi_3$ is a valid generalized context. What exactly Ψ_3 stands for will be discussed below.

Rules for Constant Coverage

The left hand side of the judgment for constant coverage, consists of three parts. In fact, if one replaces the ';' by a ',' the left hand side forms a general context. Enclosed by the two ';' is the declaration which we want to split over. The $\Pi\Gamma_x. B_x$ is a short hand for the type of x that may be a functional introducing local parameters $\Gamma_x = y_1 : A_1, \ldots, y_n : A_n$. If we were perfectly explicit, we would write

$$\Pi\Gamma_x. B_x = \Pi y_1 : A_1. \ldots \Pi y_n : A_n. B_x$$

where B_x is atomic, i.e. not a function type.

The Σ in between the $\vdash$ and the $\gg$ is the structure we are examining for possible head constructors of x. For constants, it is the signature, for local variable, it is some context Γ, and for global variables we simply use a variable block. Finally, ω is being returned, if we attach an operational (bottom-up) reading to the rules. Therefore, if $\Sigma = \cdot$, the returning cover will be empty.

$$\frac{}{\Psi_1; x : \Pi\Gamma_x. B_x; \Psi_2 \vdash \cdot \gg \cdot \text{ cover}}\ \text{ccempty}$$

The two following rules are best described by their operational interpretation. The coverage algorithm examines a non-empty signature $\Sigma, c : \Pi\Gamma_c. B_c$. In the example above, c would be papp, and $\Pi\Gamma_c. B_c$ its type. Is c a valid head constructor for x? The answer is yes, as long the type of c after raising it and applying it to the right argument (exactly the same way as we have sketched it in the previous section) and the type of x have a most general unifier. We write Ψ'

for the new set of variables which are the co-domain of the substitution. Ψ_c correspond to the list of variables marked by a hat as the outcome of the raising operation in the previous section. If such a unifier exist, we call it ψ, then we calculate a cover ω for the remaining signature Σ, and return $\omega, (\Psi' \triangleright \psi)$.

$$\frac{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma \gg \omega \text{ cover}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma, c : \Pi\Gamma_c.\, B_c \gg \omega, (\Psi' \triangleright \psi|_\Psi) \text{ cover}}\ \textsf{ccunify}$$

$$\begin{array}{l}
\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2 \\
\Psi \vdash \text{raise}\ (\Gamma_x, \Pi\Gamma_c.\, B_c) = (\Psi_c \triangleright \Pi\Gamma_x.\, B'_c) \\
\Psi' \vdash \psi = \text{mgu}\ (\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_c, x \approx \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x)) : \Psi, \Psi_c \\
\Psi' \vdash (\cdot \triangleright \psi) \text{ strict}
\end{array}$$

In addition, we require that the new general context Ψ' is strict in ψ. We are very much convinced, that the way we have implemented unification in Chapter 8 only returns pairs $(\Psi' \triangleright \psi)$ that are strict, but because we do not present the unification algorithm in this thesis, we stipulate strictness as a requirement, until we will have described the unification algorithm in future work. Should there be a non-strict substitution as result of the unification operation, there cannot be a cover.

Dually to this rule, if the unification algorithm fails because of a clash, we simply return the cover ω calculated for the remaining signature Σ.

$$\frac{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma \gg \omega \text{ cover}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma, c : \Pi\Gamma_c.\, B_c \gg \omega \text{ cover}}\ \textsf{ccskip}$$

$$\begin{array}{l}
\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2 \\
\Psi \vdash \text{raise}\ (\Gamma_x, \Pi\Gamma_c.\, B_c) = (\Psi_c \triangleright \Pi\Gamma_x.\, B'_c) \\
\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_c, x \approx \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x) \text{ do not unify}
\end{array}$$

If the unification problem fails for other reasons but a clash, the coverage algorithm will not return an answer. Non-strict substitutions ψ' and non-clash failures are a strong indication, that it cannot be decided if a unification problem has most general unifiers or not.

Rules for Local Parameter Coverage

The rules for local parameter coverage cover cases such as $D_1^{(2)}$ in the example in the previous section. D_1 can be of the form of a function, which uses the locally introduced argument v in the body.

$$\begin{array}{rcl}
D_1^{(2)} & : & \Pi y : \text{term}\ T_2.\, y \overset{1}{\Longrightarrow} y \to y \overset{1}{\Longrightarrow} y \\
& = & \lambda y : \text{term}\ T_2.\, \lambda v : y \overset{1}{\Longrightarrow} y.\, v
\end{array}$$

From the perspective of the coverage algorithm, semantically, there is no difference between a constant and a locally introduced parameter. The parameter is essentially seen as a dynamically introduced constant, and this view is reflected in the three rules below. The left hand side of the $\vdash$ symbol in the judgment for local parameter coverage is the generalized context, which exposes the variable x and its type $\Pi\Gamma_x.\, B_x$. As above, x is instantiated by an object that expects as arguments Γ_x. The algorithm considers all cases with parameters declared in Γ_x as head. Thus, the judgment iterates through Γ_x and in order to avoid confusion, we denote the intermediate versions of this context with Γ. If Γ is empty, an empty cover is returned.

$$\frac{}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \cdot \gg \cdot \text{ cover}}\ \textsf{lcempty}$$

If Γ is not empty but contains a declaration of a parameter $p : \Pi\Gamma_p.\, B_p$, then we use exactly the same technique as above, by raising p's type by Γ_x and trying to unify the types. Should the unification process terminate successfully, with a strict most general unifier ψ and co-domain Ψ', we proceed as above and return an extended cover.

$$\frac{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma \gg \omega \text{ cover}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma, p : \Pi\Gamma_p.\, B_p \gg \omega, (\Psi' \triangleright \psi|_\Psi) \text{ cover}}\ \textsf{lcunify}$$

$$\begin{array}{l}
\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2 \\
\Psi \vdash \text{raise } (\Gamma_x, \Pi\Gamma_p.\, B_p) = (\Psi_p \triangleright \Pi\Gamma_x.\, B'_p) \\
\Psi' \vdash \psi = \text{mgu } (\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p, x \approx \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x)) : \Psi, \Psi_p \\
\Psi' \vdash (\cdot \triangleright \psi) \text{ strict}
\end{array}$$

Should such a unifier not exist (and again, unification must have failed with some kind of clash indicating that it is really impossible to unify these two types) then the coverage algorithm returns the cover it has calculated for the remaining list of parameters.

$$\frac{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma \gg \omega \text{ cover}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma, p : \Pi\Gamma_p.\, B_p \gg \omega \text{ cover}}\ \textsf{lcskip}$$

$$\begin{array}{l}
\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2 \\
\Psi \vdash \text{raise } (\Gamma_x, \Pi\Gamma_p.\, B_p) = (\Psi_p \triangleright \Pi\Gamma_x.\, B'_p) \\
\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p, x \approx \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x) \text{ do not unify}
\end{array}$$

With these two operations and under the closed world assumption, we can already calculate a complete cover if we split a variable of arbitrary type.

Rules for Global Parameter Coverage

Under the regular world assumption, however, we must also consider the case that $x : \Pi\Gamma_x.\, B_x$ is instantiated with a parameter from the parameter context. Recall, that these parameter contexts are in general finite, but arbitrarily long, and therefore it is infeasible to introduce a case for *each* possible parameter from the parameter context. In Section 4, we have motivated that under the regular world assumption, parameter contexts are regularly formed, and each parameter block is an instance of one of finitely many block schemas. In addition, we have introduced a new variable concept that can range over those blocks, and we called them variable blocks ρ. The regular structure of parameter contexts is a priori defined by the context schema, which is part of any general formula. Thus, with the help of the context schemas and variable blocks, we can in fact examine coverage. Since we know that the parameter context is regularly formed, we simply examine each possible block schema, and observe if it contains a parameter which might be the head of x! In the example from the previous section, for example, the body of x could be $\underline{u}$.

$$\begin{array}{rcl} D_1^{(1)} & : & \Pi y : \text{term } T_2.\, y \stackrel{1}{\Longrightarrow} y \to \underline{x} \stackrel{1}{\Longrightarrow} \underline{x} \\ & = & \lambda y : \text{term } T_2.\, \lambda v : y \stackrel{1}{\Longrightarrow} y.\, \underline{u} \end{array}$$

Therefore, it should not come as a surprise, that the structure of the rules for global parameter coverage resemble the two blocks of rules already discussed. The only difference is that in these rules, we range in addition over a variable block, and check each single declaration.

$$\frac{}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \cdot \gg \cdot \text{ cover}}\ \mathsf{gcempty}$$

Because of the form of context blocks and additional dependencies, we augment the left hand side of the judgment with a third partial generalized context Ψ_3, which captures all SOME-variables, and the entire variable block itself. We need Ψ_3, because $\Psi, \Psi_3 \vdash \Pi\Gamma_g.\, B_g : \text{type}$; otherwise the appeal to raise type in the side condition below is not well-formed.

Consider the case that $g : \Pi\Gamma_g.\, B_g$ is one of the parameter variables declared in ρ. Following the two sets of rules above, we make all arguments of g dependent on Γ_x, and then try to unify the raised base type of g with $\Pi\Gamma_x.\, B_x$. If the unification algorithm returns with a most general and strict substitution, we return the new cover.

$$\frac{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \rho \gg \omega \text{ cover}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \rho, g : \Pi\Gamma_g.\, B_g \gg \omega, (\Psi' \rhd \psi|_\Psi) \text{ cover}}\ \mathsf{gcunify}$$

$$\begin{array}{l} \Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2 \\ \Psi, \Psi_3 \vdash \text{raise } (\Gamma_x, \Pi\Gamma_g.\, B_g) = (\Psi_g \rhd \Pi\Gamma_x.\, B_g') \\ \Psi' \vdash \psi = \text{mgu } (\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B_g', x \approx \lambda\Gamma_x.\, g\ (\Psi_g\ \Gamma_x)) : \Psi, \Psi_3, \Psi_g \\ \Psi' \vdash (\cdot \rhd \psi) \text{ strict} \end{array}$$

And exactly as above, if the unification fails, g cannot be the head of the x, and therefore we do not have to add it to the cover.

$$\frac{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \rho \gg \omega \text{ cover}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \rho, g : \Pi\Gamma_g.\, B_g \gg \omega \text{ cover}}\ \mathsf{gcskip}$$

$$\begin{array}{l} \Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2 \\ \Psi, \Psi_3 \vdash \text{raise } (\Gamma_x, \Pi\Gamma_g.\, B_g) = (\Psi_g \rhd \Pi\Gamma_x.\, B_g') \\ \Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B_g', x \approx \lambda\Gamma_x.\, g\ (\Psi_g\ \Gamma_x) \text{ do not unify} \end{array}$$

Rules for Schematic Coverage

Unlike the two first set of rules, the coverage algorithm must also traverse the context schema, and check each block schema if it contributes new cases. In most of our examples, we dealt only with one context block, but in practice, theorems are very likely to rely on many. We have already seen in Section 4.2.3 one example of multi-block context schemas, when we added polymorphism to the simply-typed λ-calculus.

The rules for schematic coverage, require renaming substitutions σ that map schema contexts C to generalized substitutions Ψ: $\Psi \vdash \sigma \in C$. They are defined in a straightforward way.

We start with the definition of schematic coverage. If the context schema is empty, we return the empty cover

$$\frac{}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \cdot \gg \cdot \text{ cover}}\ \textsf{scempty}$$

otherwise, there must be a context block SOME C_1. BLOCK C_2. We proceed by renaming C_1 into a generalized context Ψ_3, and α-convert C_2 to a new variable block ρ. Then we examine all cases which arise from ρ, using the global parameter judgment, and return the newly found cases.

$$\frac{\begin{array}{l}\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash S \gg \omega_1 \text{ cover}\\ \Psi_3 \vdash \sigma \in C_1\\ \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2, \Psi_3 \vdash \rho \equiv_\alpha [\sigma]C_2\\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3, \rho^L \vdash \rho \gg \omega_2 \text{ cover}\end{array}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash S, (\text{SOME } C_1.\ \text{BLOCK } C_2)^L \gg \omega_1, \omega_2 \text{ cover}}\ \textsf{scnext}$$

Rules for Single Coverage

All is prepared to combine the three parts of the coverage algorithm described above. The overall coverage algorithm non-deterministically and successively picks variables from the generalized context Ψ, and splits them. This part of the algorithm is defined by two judgments, called single coverage, and multiple coverage. Single coverage means, that ω covers all cases by refining one variable.

$$\frac{\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2 \qquad \begin{array}{l}\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma_x \gg \omega_1 \text{ cover}\\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma \gg \omega_2 \text{ cover}\\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \mathcal{S} \gg \omega_3 \text{ cover}\end{array}}{\Psi \vdash \omega_1, \omega_2, \omega_3 \ \ \text{cover}}\ \textsf{single}$$

Rules for Multiple Coverage

The judgment for multiple coverage calls single coverage repeatedly and combines the results by an easy substitution composition.

$$\frac{\Psi \vdash \omega \text{ cover}}{\Psi \vdash \omega \text{ cover}^*}\ \textsf{multiempty}$$

$$\frac{\Psi \vdash \omega_1, (\Psi' \triangleright \psi), \omega_2 \ \ \text{cover}^* \qquad \Psi' \vdash \omega' \ \ \text{cover}}{\Psi \vdash \omega_1, \psi \circ \omega', \omega_2 \ \ \text{cover}^*}\ \textsf{multicons}$$

where

$$\begin{array}{rcl}\psi \circ \cdot & = & \cdot\\ \psi \circ (\omega', (\Psi'' \triangleright \psi')) & = & \psi \circ \omega', (\Psi'' \triangleright \psi \circ \psi')\end{array}$$

The coverage algorithm is designed to formulate a syntactical criterion for side condition (5.2) attached to the case-rule in Section 5.6.2:

$$\frac{\Psi; \Delta \vdash \psi; \delta \in \Psi'; \Delta' \qquad \Psi'; \Delta' \vdash \Omega \in F}{\Psi; \Delta \vdash \text{case } (\psi; \delta) \text{ of } \Omega \in F[\psi]}\ \textsf{case}$$

Informally, Ω is said to cover all cases, if it can be guaranteed, that the stripped version of Ω are in fact generated by the coverage algorithm.

Definition 7.12 (Syntactic coverage criterion)

$$\Psi' \vdash \mathit{strip}\ (\Omega)\ \mathit{cover}^*$$

where we understand as *stripping* the operation that removes all $\mapsto P$ from Ω.

Definition 7.13 (Stripping)

$$\begin{array}{rcl} \mathit{strip}\ (\cdot) & = & \cdot \\ \mathit{strip}\ (\Omega, (\Psi' \triangleright \psi \mapsto P)) & = & \mathit{strip}\ (\Omega), (\Psi' \triangleright \psi) \end{array}$$

In summary, we have presented a sophisticated algorithm to characterize the coverage property of case analysis as a syntactic property of a proof term. That the coverage algorithm indeed returns a *complete* set of covers is the main result that we present in the next Section. From an experimental point of view, we want to point out that all examples from Chapter 3, and their formalizations in Chapter 4 satisfy this criterion. The side condition itself is syntactic, which means that it is easy to implement. Moreover, in the implementation of the meta-theorem prover in the Twelf system, we use the coverage algorithm to generate the different forms of Ω, a process which we call *Splitting*. The Twelf system is described in detail in Chapter 8. From a theoretical point of view, the design of this criterion is the final step in our quest to turn $\mathcal{M}_2^+$ into a calculus of realizers. That it satisfies the necessary properties is discussed in the next section.

7.3.3 Meta-Theory

We begin now with the discussion of the theoretical properties of the coverage algorithm presented in the previous subsection. Any valid case analysis satisfying the syntactic criterion is guaranteed to cover all cases, and in particular, when executing it on the abstract machine defined in Section 7.1.3 the computation never runs into a state where it cannot make progress.

This subsection is organized as follows: First, we discuss some general properties about substitutions. Second, we show that independent of the current environment, the abstract machine always finds a case in Ω, when it executes a case instruction. This property is called liveness. Finally, we generalize liveness to progress.

Preliminaries

Well-formed generalized substitutions ψ satisfy $\Psi' \vdash \psi \in \Psi$. Each substitution of this form can be easily restricted to an initial fragment of Ψ. Consider, for example, $\Psi = \Psi_1, \Psi_2$. By several inversions on the typing derivation of ψ, we can easily deduce that $\psi = \psi_1, \psi_2$, and moreover $\Psi' \vdash \psi_1 \in \Psi_1$. This simple property of substitutions is used several times in the proofs presented in this section. Following common practice, we write $\psi_1 = \psi|_{\Psi_1}$ in order to restrict ψ to Ψ_1.

Lemma 7.14 (Restricting substitutions)
If $\Psi' \vdash \psi \in \Psi_1, \Psi_2$
then $\Psi' \vdash \psi|_{\Psi_1} \in \Psi_1$

Proof: by induction on Ψ_2 □

Similarly, if we restrict the composition of two substitutions to a generalized context Ψ, the result is the same as if we had restricted the left substitution to Ψ before composing it with the right.

Lemma 7.15 (Restricting compositions)
If $\Psi_2 \vdash \psi_2 \in \Psi_1$
and $\Psi_1 \vdash \psi_1 \in \Psi, \Psi'$
then $(\psi_1 \circ \psi_2)|_\Psi = \psi_1|_\Psi \circ \psi_2$

Proof: by induction on the definition of substitution composition. □

A second concept, which is important in the proofs below, is that given a generalized substitution, we can transform it easily into a meta-substitution. This process is called *factorization*, because we can write $(\psi; \delta)$ as composition of two substitutions given that we know how to factor ψ.

Lemma 7.16 (Factorization)
If $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$
and $\Phi; \cdot \vdash \psi; \delta \in \Psi; \Delta$
then $\Phi; \cdot \vdash \psi_1; \delta \in \Psi_0; [\psi_0]\Delta$
and $\Psi_0; [\psi_0]\Delta \vdash \psi_0; id_\Delta \in \Psi; \Delta$
and $\Phi; \cdot \vdash (\psi_0; id_\Delta) \circ (\psi_1; \delta) = (\psi; \delta) \in \Psi; \Delta$.

Proof: direct. □

A last useful property is *projection*. Given a meta substitution, we can extract the underlying generalized substitution.

Lemma 7.17 (Projection)
If $\mathcal{D} :: \Phi; \cdot \vdash \psi; \delta \in \Psi; \Delta$
then $\Phi \vdash \psi \in \Psi$.

Proof: by induction on $\mathcal{D}$. □

Liveness

Liveness expresses that every case statement satisfying the coverage condition can be successfully executed without starving the computation. Only under the assumption that liveness holds can we prove progress. More precisely, we must show that "case $(\psi; \delta)$ of Ω" provides a case $(\Psi \triangleright \psi' \mapsto P)$ in Ω such that there exists a ψ'' that satisfies $\psi = \psi' \circ \psi''$. The proof of this property is split into several lemmas, closely following the definition of the coverage condition in the previous section. For example, there is a liveness property for constant covers, for local parameter covers, for global parameter covers and for schematic covers, and naturally for single and multiple coverage.

We begin the presentation with a liveness lemma for constant covers. This lemma expresses that if there exists a (in general arbitrary) unifier of $(\Pi\Gamma_x. B_x \approx \Pi\Gamma_x. B'_c, x \approx \lambda\Gamma_x. c\ (\Psi_c\ \Gamma_x))$, where Ψ_c, B'_c are the result of raising the type of a constant c declared in Σ, then the case in Ω generated by c (by ccunify) is applicable, and ev_yes would fire if Ω were executed. We construct the substitution which solves this unification problem in the proof of Lemma 7.22 from the explicit substitution in the case subject.

Lemma 7.18 (Liveness of constant covers)

If $\mathcal{D} :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma \gg \omega$ *cover*
and $\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2$
and $\Sigma(c) = \Pi\Gamma_c.\, B_c$
and $\Psi \vdash$ *raise* $(\Gamma_x, \Pi\Gamma_c.\, B_c) = (\Psi_c \triangleright \Pi\Gamma_x.\, B'_c)$
and $\mathcal{F} :: \Phi; \cdot \vdash \psi; \delta \in \Psi, \Psi_c; \Delta$
and $\psi \in$ *unify* $(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_c, x \approx \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x))$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$

Proof: by induction on $\mathcal{D}$, using Lemma 7.14 and by Lemma 7.15. A detailed proof can be found in Appendix C. □

Similarly, if we have a solution for the unification problem unify $(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p, x \approx \lambda\Gamma_x.p\ (\Psi_p\ \Gamma_x))$ generated by a local parameter in Γ (which we also construct in the proof of Lemma 7.22), then the corresponding case (generated by p) in Ω is applicable, and ev_yes would fire if Ω were executed.

Lemma 7.19 (Liveness of local parameter covers)

If $\mathcal{D} :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma \gg \omega$ *cover*
and $\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2$
and $\Gamma(p) = \Pi\Gamma_p.\, B_p$
and $\Psi \vdash$ *raise* $(\Gamma_x, \Pi\Gamma_p.\, B_p) = (\Psi_p \triangleright \Pi\Gamma_x.\, B'_p)$
and $\mathcal{F} :: \Phi; \cdot \vdash \psi; \delta \in \Psi, \Psi_p; \Delta$
and $\psi \in$ *unify* $(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p, x \approx \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x))$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$

Proof: by induction on $\mathcal{D}$, using by Lemma 7.14 and by Lemma 7.15. A detailed proof can be found in Appendix C. □

And finally, if we have a solution for the unification problem unify $(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_y, x \approx \lambda\Gamma_x.p\ (\Psi_y\ \Gamma_x))$ this time to be constructed in the proof of Lemma 7.21, the corresponding case (generated by y) in ρ is applicable, and ev_yes would fire if Ω were executed.

Lemma 7.20 (Liveness of global parameter covers)

If $\mathcal{D} :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \rho \gg \omega$ *cover*
and $\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2$
and $\rho(y) = \Pi\Gamma_y.\, B_y$
and $\Psi \vdash$ *raise* $(\Gamma_x, \Pi\Gamma_y.\, B_y) = (\Psi_y \triangleright \Pi\Gamma_x.\, B'_y)$
and $\mathcal{F} :: \Phi \vdash \psi \in \Psi, \Psi_3, \Psi_y$
and $\psi \in$ *unify* $(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_y, x \approx \lambda\Gamma_x.\, p\ (\Psi_y\ \Gamma_x))$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$

Proof: by induction on $\mathcal{D}$, using Lemma 7.14 Lemma 7.15. A detailed proof can be found in Appendix C. □

Recall that the coverage condition for global parameters was defined by two judgments in the previous subsection. We have a judgment for global parameter coverage and one for schematic coverage. Correspondingly, there is a lemma for liveness of global parameter covers which we have already discussed, and there is one for the liveness of schematic coverage which we discuss now. Consider a case analysis on x. We must show that for every possible form of x, there is a case in Ω, and for this lemma, we assume that x is bound to an object whose head constructor is a parameter variable g declared in the parameter context: $\psi(x) = \lambda\Gamma_x.\, g\ M_1..M_n$. g must be declared in a block schema which is part of the overall declared context schema. From this information alone, we can construct a solution to the unification problem in Lemma 7.20 which proves the claim immediately.

Lemma 7.21 (Liveness of schematic coverage)
If $\mathcal{D} :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash S \gg \omega$
and $\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2$
and $\mathcal{F} :: \Phi \vdash \psi \in \Psi$
and $\psi(x) = \lambda\Gamma_x.\, g\ M_1..M_n$
and $\rho^L \in \Phi$
and $\rho(g) = \Pi\Gamma_g.\, B_g$
and $S(L) = \text{SOME } C_1.\ \text{BLOCK } C_2$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0; \circ\psi_1 = \psi \in \Psi$

Proof: by induction on $\mathcal{D}$, using Lemma 7.11, Lemma 6.7, Lemma 2.7, and Lemma 7.20. A detailed proof can be found in Appendix C. □

The substitution ψ may map x to an object whose head constructor is not necessarily a global parameter. It could be either a local parameter or a constant since there are only three possibilities! By a very similar construction as in the previous argument, we construct a solution to the unification problem from Lemma 7.18 and Lemma 7.19, respectively. The claim follows immediately.

Lemma 7.22 (Liveness of single coverage)
If $\mathcal{D} :: \Psi \vdash \omega$ *cover*
and $\mathcal{E} :: \Phi \vdash \psi \in \Psi$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$

Proof: by case analysis of $\mathcal{D}$, using Theorem 2.6, Lemma 7.11, Lemma 6.7, Lemma 2.7, Lemma 7.18, Lemma 7.19, Lemma 7.21. A detailed proof can be found in Appendix C. □

Our approach to complete case analysis allows several splitting steps of different variables; in the coverage condition, this is expressed by the multicons-rule. In order to show that there is always one applicable case in Ω, we have to consider successive splits over several variables in ψ according to the cover* relation.

Lemma 7.23 (Liveness of multiple coverage)
If $\mathcal{D} :: \Psi \vdash \omega$ *cover**
and $\Phi \vdash \psi \in \Psi$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$

Proof: by induction on $\mathcal{D}$ using Lemma 7.22 and Lemma 5.2. A detailed proof can be found in Appendix C. □

Finally, by factoring and projecting meta-substitutions, we obtain the formal result that any case statement which satisfies the coverage condition defines one case in Ω that keeps the computation running on the abstract machine. The decomposition of the substitution, which is guaranteed to exist by the next lemma, is a formalization of the side condition of ev_yes.

Lemma 7.24 (Liveness)
If $\mathcal{D} :: \Psi \vdash \omega$ *cover**
and $\mathcal{E} :: \Phi; \cdot \vdash \psi; \delta \in \Psi; \Delta$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$*, and a* ψ_1*, s.t.* $\Phi; \cdot \vdash \psi_1; \delta \in \Psi_0; [\psi_0]\Delta$
and $\Psi_0; [\psi_0]\Delta \vdash \psi_0; id_\Delta \in \Psi; \Delta$
and $\Phi; \cdot \vdash (\psi_0; id_\Delta) \circ (\psi_1; \delta) = (\psi; \delta) \in \Psi; \Delta$

Proof:
direct, by Lemma 7.17, Lemma 7.23, and Lemma 7.16. A detailed proof can be found in Appendix C. □

The stage is set for the proof that every function in $\mathcal{M}_2^+$ under the two side conditions is a realizer.

Progress

Liveness is a property attached to case statements. In essence, it expresses, that for any case subject, the side condition attached to tryes is fulfilled for at least one case in Ω. This observation guarantees that the abstract machine, once started on a match state containing Ω will transition into a non-match state after finitely many steps (by applying tryes). Therefore the computation of cases can never get stuck.

Lemma 7.25 (Progress for case)
If $S = \Phi; C \triangleright (\psi; \delta) \sim \Omega$
and there exists a $((\Psi_0 \triangleright \psi_0) \mapsto P) \in \Omega$, *and a* ψ_1
s.t. $\Phi; \cdot \vdash \psi_1; \delta \in \Psi_0; [\psi_0]\Delta$
and $\Psi_0; [\psi_0]\Delta \vdash \psi_0; id_\Delta \in \Psi; \Delta$
and $\Phi; \cdot \vdash (\psi_0; id_\Delta) \circ (\psi_1; \delta) = (\psi; \delta) \in \Psi; \Delta$
then there exists an S'
and $S \stackrel{*}{\Longrightarrow} S'$
and S' *is not a match state*

Proof: by induction over Ω. A detailed proof can be found in Appendix C. □

This result generalizes directly to the progress theorem. In every state (except a final state) the abstract machine can make one transition step to the next state. Thus, by induction it follows that in any situation the machine can make progress. In a situation where the current state contains a case statement, the claim follows from the progress lemma for case, in all others directly from the form of the rules.

Theorem 7.26 (Progress)
If S *is a state, but not a match state*
and $S \neq \Phi; \cdot \triangleright V$
and $\mathcal{D} :: \vdash S \in F$
then there exist an S' *and an* S'' *which is not a match state*
and $S \Longrightarrow S' \stackrel{*}{\Longrightarrow} S''$

Proof: by case analysis of S,using Lemma 7.24 and Lemma 7.25. A detailed proof can be found in Appendix C. □

Therefore, any computation executed on the abstract machine can never get stuck until it reaches a final state that we interpret as the result of the computation.

7.4 Soundness of $\mathcal{M}_2^+$

All proof terms of the fragment of $\mathcal{M}_2^+$ specified in Section 7.2.1 are total on under the operational interpretation via a small-step semantics. We conjecture that this claim holds for all functions in the Π_2-fragment of $\mathcal{M}_2^+$ but we leave the proof of this claim to future work.

When those proof terms (encoded as states) are executed on the abstract machine, the computation makes progress and will eventually terminate. Technically we can extract the value of the computation out of the final state. Thus, all proof terms in $\mathcal{M}_2^+$ witness the provability of a theorem and are therefore called realizers.

Theorem 7.27 (Realizability)
If $\Phi; \cdot \vdash P \in F$
then there exists a V
s.t. $\Phi; \cdot \vdash V \in F$
and $\Phi; \star \triangleright P \overset{*}{\Longrightarrow} \Phi; \star \triangleright V$

Proof: direct, using Theorem 7.9, Lemma 7.4, and Theorem 7.26. A detailed proof can be found in Appendix C. □

All functions in $\mathcal{M}_2^+$ are realizers, and by executing them we construct the witness objects for the existentials from given instantiations of the universals. Moreover, we can now give a formal proof of the soundness of $\mathcal{M}_2^+$ with respect to the semantics we have specified in Definition 5.7. Any Π_2-formula which is 'inhabited' by a value V is semantically valid. The proof is an easy induction over the structure of formulas.

Theorem 7.28 (Soundness of $\mathcal{M}_2^+$)

1. *If* $\mathcal{D} :: \vdash Q \in G$
   then $\models G$.

2. *If* $\mathcal{D} :: \Phi; \cdot \vdash V \in F$
   then $\Phi \models F$.

Proof: (1) direct, (2) by induction on the size of formulas F, using Lemma 6.11, Theorem 7.27, Lemma 6.22, and Lemma 6.20. A detailed proof can be found in Appendix C. □

7.5 Summary

Thus, we conclude this Section by reiterating the main theoretical results of this thesis. $\mathcal{M}_2^+$ is a sound intuitionistic meta-logic, because all recursive functions are realizers. It elegantly combines higher-order representation techniques with the formalization of inductive arguments. Unlike purely logical systems, which are designed to be complete, we cannot hope for $\mathcal{M}_2^+$ to be complete because of its expressiveness (even though it is restricted to the Π_2-fragment). We have not carried out the argument, but we speculate that $\mathcal{M}_2^+$ could theoretically be represented in LF, which exposes it to Gödel's incompleteness theorem [Göd31].

If preferred, $\mathcal{M}_2^+$ can be seen as type theory whose datatypes take full advantage of LF's representational power; i.e. dependent types and higher-order representation techniques. In this it differs significantly from inductive definitions that rely on the positivity condition. Without coverage and termination side condition, $\mathcal{M}_2^+$ is a type theory for recursive functions, but with them, $\mathcal{M}_2^+$ can be seen as a sound meta-logic for LF.

In addition, this type theory inherits many of the properties associated with hypothetical judgments such as substitution, contraction, weakening, and exchange. Those properties need not be explicitly represented in a proof term which makes them in general short, concise, and amenable for automatic construction which we discuss in the next Chapter 8. We leave an investigation of how to turn $\mathcal{M}_2^+$ into a full-fledged programming language to future research but discuss the basic ideas in Section 9.1.5.

Part III

Implementation

Chapter 8

Twelf

8.1 Introduction

Twelf is a meta-logical framework for the specification, implementation, and meta-theory of deductive systems from the theory of programming languages and logics. For example, Twelf has been successfully employed to derive various properties such as type preservation and progress of various operational semantics, the consistency of logics, and the admissibility of new inference rules. Other results include automatic proofs of the Church-Rosser theorem, cut-elimination for various logics, soundness and completeness of uniform proof search and resolution. It relies on the LF type theory and the judgments-as-types methodology for specification [HHP93], a constraint logic programming interpreter for implementation [Pfe91], and the meta-logic $\mathcal{M}_2^+$ for reasoning about object languages encoded in LF under the regular world assumption. It is a significant extension and complete reimplementation of the Elf system [Pfe94].

Specification. Twelf employs the representation methodology and underlying type theory of the LF logical framework discussed in Chapter 2. Expressions are represented as LF objects using the technique of *higher-order abstract syntax* and *hypothetical judgments* whereby variables of an object language are mapped to variables in the meta-language. This means that common operations, such as renaming of bound variables or capture-avoiding substitutions are directly supported by the framework and do not need to be programmed anew for each object language.

For semantic specification LF uses the *judgments-as-types* representation technique. This means that a derivation is coded as an object whose type represents the judgment it establishes. Checking the correctness of a derivation is thereby reduced to type-checking its representation in the logical framework and therefore in Twelf (which is efficiently decidable).

Algorithms. Generally, specification is followed by implementation of algorithms manipulating expressions or derivations. Twelf supports the implementation of such algorithms by a constraint logic programming interpretation of LF signatures, a slight variant of the one originally proposed in [Pfe91] and implemented in Elf [Pfe94]. The operational semantics is based on goal-directed, backtracking search for an object of a given type. For the purpose of this thesis we will not discuss this feature here. The interested reader is invited to consult [PS98] for a detailed discussion.

Meta-Theory. Using the regular world assumption Twelf offers an experimental automatic meta-theorem proving component based on the meta-logic $\mathcal{M}_2^+$ presented in Chapter 5. It expects as input a Π_2-statement describing a property of LF objects over a fixed signature, a fixed context schema, and a termination ordering and searches for an inductive proof by constructing a realizer in $\mathcal{M}_2^+$. Even though a number of the theorems in the example suites described below can be proven automatically, we consider the meta-theorem prover to be in a preliminary state.

Twelf is written in Standard ML and runs under SML of New Jersey and MLWorks on Unix and Window platforms. The current version is distributed with a complete manual, example suites, a tutorial in the form of on-line lecture notes [Pfe00], and an Emacs interface. Source and binary distributions are accessible via the Twelf home page `http://www.twelf.org`.

While Twelf is implemented in ML it is executed as a stand-alone program rather than within the ML top-level loop. This is feasible, since meta-programming is carried out in type theory itself via a logic programming interpretation, rather than in ML as in many other proof development environments. The most effective way to interact with Twelf is as an inferior process to Emacs. The Emacs interface, which has been tested under XEmacs, FSF Emacs, and NT Emacs, provides an editing mode for Twelf source files and commands for incremental type checking, logic program execution, and theorem proving. Moreover it provides utilities for jumping to error locations and tagging and maintaining configurations of source files.

In this Chapter we sketch a theorem prover for LF implemented in the Twelf system in Section 8.2, and we describe the meta-theorem prover in Section 8.3, in particular the three basic operations *Filling*, *Splitting*, *Recursion*, the non-standard treatment of lemmas and we remark on the correctness of the implementation. In Section 8.4 we give a brief overview of how to use Twelf and its meta-theorem prover, and we demonstrate its power by presenting a formalization of the Church-Rosser example. In Section 8.5 we report on other experiments we have conducted with the meta-theorem prover, and we summarize the results of this Chapter in Section 8.6.

8.2 Theorem Prover for LF

The overall goal of this thesis is to develop a tool to automate the meta-theory of deductive systems. This tool is designed to automate the reasoning processes as we have used them to convince ourselves of the correctness of the substitution Lemma 3.6 and the diamond Lemma 3.7 for the simply-typed λ-calculus in Section 2.2. It lies in the very nature of this goal that reasoning about a deductive system is connected to reasoning inside the formal system; in all example proofs, we have used the rules defined with the deductive system to complete a case in the proof such as for example 'plam' and 'papp' in the proof of Lemma 3.4. Because the representation of the formal systems defining parallel reduction and well-typed terms in LF are adequate, i.e. there is a one-to-one correspondence between derivations in the deductive system and their representation as objects in the type theory, we can carry out the following development purely in type theory. We use the proof of the reflexivity Lemma 4.3 as example.

$$
\begin{array}{l}
\textbf{fun refl } \underline{x} = \underline{u} \\
\quad \mid \textbf{refl } (\text{lam } (\lambda x : \text{term } T.\, E'\ x)) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{new } \underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x} \\
\qquad\quad \textbf{val} \quad P\ \underline{x}\ \underline{u} = \textbf{refl } (E'\ \underline{x}) \\
\qquad \textbf{in} \\
\qquad\quad \text{plam } (\lambda x : \text{term } T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, P\ x\ u) \\
\qquad \textbf{end} \\
\quad \mid \textbf{refl } (\text{app } E_1\ E_2) = \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val } P_1 = \textbf{refl } E_1 \\
\qquad\quad \textbf{val } P_2 = \textbf{refl } E_2 \\
\qquad \textbf{in} \\
\qquad\quad \text{papp } P_1\ P_2 \\
\qquad \textbf{end}
\end{array}
$$

This is the proof a theorem *about* deductive systems, but at three occasions we reason *inside* the deductive system. In the first case, we have to search for an LF object M of type $\underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$, and such an M clearly exists, because we assumed the existence of $\underline{u}$. In the second case where the argument to **refl** is 'lam $(\lambda x : \text{term } T.\, E'\ x)$', we have to apply 'plam' to the result of the induction hypothesis in order to construct a derivation of type 'lam $(\lambda x : \text{term } T.\, E'\ x) \stackrel{1}{\Longrightarrow}$ lam $(\lambda x : \text{term } T.\, E'\ x)$'. And finally, in the third case we have to apply 'papp' to the result of the two calls to the induction hypotheses P_1 and P_2.

Therefore, the meta-theorem prover that is designed to reason *about* deductive systems relies on the ability to reason *within* it. In short, we distinguish the *LF-theorem prover* that searches for proofs *within* a deductive system from the *meta-theorem prover* that searches for proofs *about* formal systems. We sketch the design of the LF-theorem prover as it is implemented in the Twelf system in this section, and postpone the design of the meta-theorem prover until Section 8.3.

8.2.1 Basic Operations

The objective of the LF theorem prover is to search for an LF object of given LF type from a set of assumptions Ψ and a set of constants declared in the signature Σ. The context Ψ contains all information about the currently valid extension of the regular world. In the implementation we use meta-variables to signify holes in an LF object (see also [Muñ97]), and in this section, we simply write $\boxed{P}$ for a meta-variable with the name P, omitting all details.

Example 8.1 (Parameter case) Given

$$\Psi = T : \text{tp}, (\underline{x} : \text{term } T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L$$

the LF theorem prover can construct an object

$$\boxed{P} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$$

in the following way. First, it detects that no constant in the signature Σ can instantiate $\boxed{P}$ because none of their types unify with $\underline{x} \stackrel{1}{\Longrightarrow} \underline{x}$. $\underline{x}$ are parameters that cannot be unified with constants. Second, it locates the one declaration in Ψ whose type unifies: the parameter $\underline{u}$. Therefore

$$\boxed{P} = \underline{u}$$

successfully instantiates $\boxed{P}$.

Example 8.2 (lam-case) Given the context

$$\Psi = T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E' : \mathrm{term}\ T_1 \to \mathrm{term}\ T_2, P : \Pi x : \mathrm{term}\ T_1.\, x \stackrel{1}{\Longrightarrow} x \to (E'\ x) \stackrel{1}{\Longrightarrow} (E'\ x)$$

where P is the result of applying the induction hypothesis after extending the world, the LF theorem prover constructs an object

$$\boxed{P'} : (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T_1.\, E'\ x)) \stackrel{1}{\Longrightarrow} (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T_1.\, E'\ x))$$

the following way. After examining the entire signature and the context, the LF theorem prover determines that there is only one possible choice to instantiate $\boxed{P'}$, namely 'plam'. Since 'plam' is of functional type, it needs to be applied to another LF object signified by $\boxed{P''} : (E'\ x) \stackrel{1}{\Longrightarrow} (E'\ x)$:

$$\boxed{P'} = \mathrm{plam}\ (\lambda x : \mathrm{term}\ T_1.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, \boxed{P''})$$

The search continues, this time for $\boxed{P''}$. Note, that the search must take place in an extended context, because $\boxed{P''}$ may depend on x and u.

$$\begin{array}{l} T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E' : \mathrm{term}\ T_1 \to \mathrm{term}\ T_2, P : \Pi x : \mathrm{term}\ T_1.\, x \stackrel{1}{\Longrightarrow} x \to (E'\ x) \stackrel{1}{\Longrightarrow} (E'\ x), \\ x : \mathrm{term}\ T_1, u : x \stackrel{1}{\Longrightarrow} x. \end{array}$$

Eventually, LF theorem prover successfully terminates with a valid instantiation $P\ x\ u$ for $\boxed{P''}$ and returns the overall search result:

$$\boxed{P'} = \mathrm{plam}\ (\lambda x : \mathrm{term}\ T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, P\ x\ u)$$

Example 8.3 (app-case) In the third case the LF theorem prover is given the context Ψ

$$\begin{array}{l} T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ (T_2\ \mathrm{arrow}\ T_1), E_2 : \mathrm{term}\ T_2 \\ P_1 : E_1 \stackrel{1}{\Longrightarrow} E_1, P_2 : E_2 \stackrel{1}{\Longrightarrow} E_2 \end{array}$$

where P_1 and P_2 are the results of applying the induction hypothesis and it is asked to construct an object $\boxed{P'}$ of type $(\mathrm{app}\ E_1\ E_2) \stackrel{1}{\Longrightarrow} (\mathrm{app}\ E_1\ E_2)$. There is only one constant in the signature that does not violate any typing constraints; 'papp' applied to two new meta-variables is therefore a possible instantiation for $\boxed{P'}$.

$$\boxed{P'} = \mathrm{papp}\ \boxed{P_1'}\,\boxed{P_2'} : (\mathrm{app}\ E_1\ E_2) \stackrel{1}{\Longrightarrow} (\mathrm{app}\ E_1\ E_2)$$

where $\boxed{P_1'}$ and $\boxed{P_2'}$ are two new meta-variables.

$$\begin{array}{lcl}\boxed{P_1'} & : & E_1 \stackrel{1}{\Longrightarrow} E_1 \\ \boxed{P_2'} & : & E_2 \stackrel{1}{\Longrightarrow} E_2\end{array}$$

Next, in the same context Ψ, they are instantiated by P_1 and P_2, respectively, and hence $\text{papp } P_1\ P_2 : (\text{app } E_1\ E_2) \stackrel{1}{\Longrightarrow} (\text{app } E_1\ E_2)$ is a solution for $\boxed{P'}$.

These three examples demonstrate how the LF theorem prover works. Starting with one meta-variable, the system searches for an instantiation of a variable hereby possibly introducing new meta-variables. Only if all meta-variables are instantiated, the theorem prover stops and signals success. Meta-variables of functional type can be lowered by moving the additional functional parameters into the context, a trick we have used in Example 8.2.

Naturally, the search space for objects of a certain type may not always be finite. The LF theorem prover therefore employs a limited depth, depth-first, and iterative deepening search procedure, that works surprisingly well in many of our examples.

8.2.2 Correctness

The implementation of the LF theorem prover is 513 lines of SML code, not taking into account the code for unification, and constraint handling. Even though Twelf is programmed with a lot of care, and the central modules are manually verified, the implementation may still contain bugs.

But fortunately, we do not have to rely on the correctness of the implementations of the algorithms used in Twelf. Instead of verifying the correctness of the entire system, we can verify each resulting instance of the theorem prover by type checking! The LF type-checker implemented in Twelf is relatively small, it contains only 206 lines of code, and it can be easily verified. It is autonomous in that it does not depend on other parts of Twelf, such as modules for unification. In fact, the Twelf systems provides an option that forces every object generated by the LF theorem prover to be type checked.

8.2.3 Limitations

The LF theorem prover has one crucial limitation; it implements a straight-forward bottom-up search schema for derivations in a deductive systems. This search technique is advantageous for certain deductive systems, but it is absolutely disastrous for others. In particular systems which define any kind of transitivity suffer extreme hardship because once started the LF theorem prover tries to guess the intermediate object which may be entirely unconstrained, and the run-time of the prover becomes excruciatingly slow.

For certain deductive systems on the other hand, in particular logics, rewrite systems, and programming systems, specialized proof search and rewrite methods have been developed in recent years [DMTV99, Häh99]. We can only outline future directions of research to incorporate these techniques into the LF theorem prover in Section 9.1.4.

8.3 Meta-Theorem Prover

The meta-logic $\mathcal{M}_2^+$ is designed to formalize theorems that express properties *about* formal systems such as logics and programming languages and their proofs. Its main purpose is to encode inductive arguments *about* higher-order encodings of deductive systems — higher-order encodings for which typically no standard induction principles exist. Inductive definitions are at the heart of many theorem provers like Coq, Isabelle, and Lego, and they rely on the closed world assumption. The Twelf system, however, is based on the regular world assumption, which permits the formalization of inductive arguments about higher-order encodings. Besides the standard constant declarations representing inference rules, the regular world assumption permits regular extensions of the world as we have discussed in the previous chapters. In the proof of the reflexivity Lemma 4.3 for example, in particular in the second case, the induction hypothesis is only applicable in a world extended by $\underline{x}, \underline{u}$.

$$
\begin{array}{l}
\mid \ \mathbf{refl}\ (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T.\, E'\ x)) = \\
\quad \mathbf{let} \\
\quad\quad \mathbf{new}\ \underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x} \\
\quad\quad \mathbf{val}\ \ P\ \underline{x}\ \underline{u} = \mathbf{refl}\ (E'\ \underline{x}) \\
\quad \mathbf{in} \\
\quad\quad \mathrm{plam}\ (\lambda x : \mathrm{term}\ T.\, \lambda u : x \stackrel{1}{\Longrightarrow} x.\, P\ x\ u) \\
\quad \mathbf{end}
\end{array}
$$

The regular world assumption guarantees that dynamic extensions can only grow in regular, limited, and well-defined ways. Therefore we can predict their forms and it allows us to reason about them abstractly.

It is this regular world assumption from which the meta-theorem prover in Twelf draws its power. In other theorem provers one has to introduce auxiliary constructions in order to make the natural higher-order encoding artificially first-order; but auxiliary construction hamper efficient proof search since properties about their interactions must be made explicit. Additional substitution lemmas for de Bruijn encodings, weakening lemmas, and exchange lemmas are only few of the examples one encounters when working with artificial first-order encodings.

Therefore the main difference of the meta-theorem prover implemented in the Twelf system and other standard inductive theorem provers is that it provides mechanisms and operations to dynamically reason about the world. All our examples have very natural encodings in LF, the proofs are very elegant — as we have shown in Chapter 4 — and therefore, the meta theorem prover is very efficient when it comes to constructing these kind of proofs automatically. Hence in these special domains, Twelf's meta theorem prover outperforms any other inductive theorem prover. In this section, we describe its basic operations in Section 8.3.1, the treatment of lemmas in Section 8.3.2, and the proof search strategy in Section 8.3.3. Finally we report on the correctness of the implementation in Section 8.3.4, and we describe its limitations in Section 8.3.5.

8.3.1 Basic Operations

The proof search algorithm used for the meta-theorem prover in Twelf is composed of three basic proof search operation: *Filling*, *Splitting*, and *Recursion*. At what point in time to apply which operation is determined by the proof search strategy which we describe in Section 8.3.3. The purpose of this subsection is to motivate the three basic operations.

The meta-theorem prover expects as input the formula to be proven, and the termination order that guides proof search. Once started, it tries to construct a derivation in the proof calculus of $\mathcal{M}_2^+$ described in Chapter 5. In analogy to the description of the LF theorem prover in Section 8.2, we use meta variables (this time ranging over $\mathcal{M}_2^+$-proof terms and *not* over LF objects) which we denote by $\boxed{\mathbf{P}}$. Note the bold type face of the variable inside the box. Formally the search procedure used in the meta-theorem prover is called with a formula F and a context Ψ, and it returns a proof term P or reports failure. We omit the set of meta-level assumptions Δ which is part of the typing judgment of $\mathcal{M}_2^+$. Initially, the meta-theorem prover is called with two more arguments: a termination order and an upper bound for search passed to the underlying LF theorem prover (see Section 8.2). Naturally, the LF signature, and the description of how the world can be extended are fixed before the theorem prover is invoked. For better readability, we write

$$\begin{array}{|l|}\hline \Psi \\ \hline \boxed{\mathbf{P}} \in F \\ \hline \end{array}$$

for *proof goals*. Recall that we only conduct proof search for proofs of Π_2-formulas. That means for the reflexivity lemma, for example, that we would ask the meta-theorem prover for a proof term $\boxed{\mathbf{P}}$ such that

$$\begin{array}{|l|}\hline \cdot \\ \hline \boxed{\mathbf{P}} \in \forall T : \mathrm{tp}. \forall E : \mathrm{term}\ T. \exists P : E \stackrel{1}{\Longrightarrow} E. \top \\ \hline \end{array}$$

After applying the $\forall\mathsf{L}$ twice, the meta-theorem prover arrives at a goal of the following form which we can consider the initial state for the theorem proving process.

$$\begin{array}{|l|}\hline T : \mathrm{tp}, E : \mathrm{term}\ T \\ \hline \boxed{\mathbf{P}'} \in \exists P : E \stackrel{1}{\Longrightarrow} E. \top \\ \hline \end{array}$$

where

$$\boxed{\mathbf{P}} = \Lambda T : \mathrm{tp}. \Lambda E : \mathrm{term}\ T. \boxed{\mathbf{P}'}$$

We consider these kind of goals initial because the domain of problems for the meta-theorem prover is restricted to (possibly empty) conjunctions of Π_2-formulas. In particular $\forall\mathsf{L}$ and $\wedge\mathsf{L}$ can be applied as many times as necessary until the formula to be proven contains only existential quantifiers. Thus, in general the *proof state* of the theorem prover can be described by a set of proof goals to be shown. The formulas $F_1, \ldots, F_n$ are Σ_1-formulas.

$$\begin{array}{|l|}\hline \Psi_1 \\ \hline \boxed{\mathbf{P}_1} \in F_1 \\ \hline \end{array} \quad \ldots \quad \begin{array}{|l|}\hline \Psi_n \\ \hline \boxed{\mathbf{P}_n} \in F_n \\ \hline \end{array}$$

Using the reflexivity lemma as example, we motivate now the three basic operations of the meta-theorem prover of Twelf: splitting, filling, and recursion. The meta-theorem prover is given the following initial state.

$$\begin{array}{|l|}\hline T : \mathrm{tp}, E : \mathrm{term}\ T \\ \hline \boxed{\mathbf{P}'} \in \exists P : E \stackrel{1}{\Longrightarrow} E. \top \\ \hline \end{array}$$

Splitting

Recall, that the original proof proceeds by case analysis. In this setting case analysis simply means to pick an assumption from the context, and to examine all possible cases. The context of this proof goal contains two LF assumptions for which we can analyze cases: T or E. In this situation, the meta-theorem will pick E — how it is determined that E is the right hypothesis to be split is discussed in the Section 8.3.3.

The splitting operation relies crucially on the regular world assumption. By definition E's head can only be a constant declared in the signature, or a global parameter — there are no other options. This is exactly what the regular world assumption expresses. Therefore the meta-theorem prover traverses the entire signature, and by unification it determines that either $E = \text{lam}\ (\lambda x : \text{term}\ T.\, E'\ x)$, or $E = \text{app}\ E_1\ E_2$ as possible shapes, and finally it traverses the context schema, and concludes that $E = \underline{x}$ is a third option. The splitting operation implements one iteration of the coverage algorithm described in Section 7.3. Thus splitting yields a new proof state with three proof goals.

$$\begin{array}{|l|}\hline T : \text{tp}, (\underline{x} : \text{term}\ T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L \\ \hline \boxed{\mathbf{P}''_1} \in \exists P : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x}.\, \top \\ \hline \end{array}$$

$$\begin{array}{|l|}\hline T_1 : \text{tp}, T_2 : \text{tp}, E' : \text{term}\ T_1 \to \text{term}\ T_2 \\ \hline \boxed{\mathbf{P}''_2} \in \exists P : (\text{lam}\ (\lambda x : \text{term}\ T_1.\, E'\ x)) \stackrel{1}{\Longrightarrow} (\text{lam}\ (\lambda x : \text{term}\ T_1.\, E'\ x)).\, \top \\ \hline \end{array}$$

$$\begin{array}{|l|}\hline T_1 : \text{tp}, T_2 : \text{tp}, E_1 : \text{term}\ (T_2\ \text{arrow}\ T_1), E_2 : \text{term}\ T_2 \\ \hline \boxed{\mathbf{P}''_3} \in \exists P : (\text{app}\ E_1\ E_2) \stackrel{1}{\Longrightarrow} (\text{app}\ E_1\ E_2).\, \top \\ \hline \end{array}$$

In addition $\boxed{\mathbf{P}'}$ is instantiated with a case construct, whose list of cases Ω contains three entries. The case bodies are $\boxed{\mathbf{P}''_1}$, $\boxed{\mathbf{P}''_1}$, and $\boxed{\mathbf{P}''_1}$, respectively.

$$\begin{array}{rl} \boxed{\mathbf{P}'} = & \text{case}\ (T/T, E/E; \mathbf{refl}/\mathbf{refl})\ \text{of} \\ & \quad (T : \text{tp}, (\underline{x} : \text{term}\ T, \underline{u} : \underline{x} \stackrel{1}{\Longrightarrow} \underline{x})^L \triangleright T/T, \underline{x}/E \ \mapsto \boxed{\mathbf{P}''_1}), \\ & \quad (T_1 : \text{tp}, T_2 : \text{tp}, E' : \text{term}\ T_1 \to \text{term}\ T_2 \\ & \quad \triangleright (T_1\ \text{arrow}\ T_2)/T, (\text{lam}\ E')/E \ \mapsto \boxed{\mathbf{P}''_2}), \\ & \quad (T_1 : \text{tp}, T_2 : \text{tp}, E_1 : \text{term}\ (T_2\ \text{arrow}\ T_1), E_2 : \text{term}\ T_2 \\ & \quad \triangleright T_1/T, (\text{app}\ E_1\ E_2)/E \ \mapsto \boxed{\mathbf{P}''_3}) \end{array}$$

The current version of the meta-theorem prover computes proof terms only implicitly. In future revisions, the proof terms of $\mathcal{M}_2^+$ will be explicitly generated, and an efficient and independent proof checker will be provided that can verify them.

In summary, the splitting operation is an operation that selects a proof goal from the proof state, it selects a variable declaration (but not a parameter variable) from the proof goal, analyzes its cases, and adds the newly generated proof goals into the proof state.

Filling

The filling operation attempts to close a proof goal by constructing witness objects for the existentially quantified variables. In our example there is only one existential quantifier, but in the general case there might be several. In order to construct witness objects, the meta-theorem prover invokes the underlying LF theorem prover, passes it the list of assumptions, and an upper search bound. The LF theorem prover either returns and reports success or fails. In the case that there are several existentially quantified declarations, the LF theorem prover attempts to find several object simultaneously. The reason is that, that this way the theorem prover can take advantage of the dependencies that constrain the search spaces. Back to the example. Given the proof goal,

$$\begin{array}{|l|}\hline T : \mathrm{tp}, (\underline{x} : \mathrm{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L \\ \hline \boxed{\mathbf{P}''_1} \in \exists P : \underline{x} \overset{1}{\Longrightarrow} \underline{x}.\top \\ \hline \end{array}$$

the meta-theorem prover invokes the LF theorem prover to construct an instantiation for $\boxed{\mathbf{P}''_1}$ from assumptions T, $\underline{x}$, and $\underline{u}$. The LF theorem prover returns success and as solution it reports $\underline{u}$. Already expected by the meta-theorem prover, this solution is embedded in a proof term for $\exists P : \underline{x} \overset{1}{\Longrightarrow} \underline{x}.\top$

$$\boxed{\mathbf{P}''_1} = \langle \underline{u}, \langle\rangle \rangle$$

closing this proof goal. Two goals remain unsolved, but filling alone cannot solve them. In summary, the filling operation employs the underlying LF theorem prover to construct witness objects for the existential objects. If successful, the proof goal is completed and removed from the proof state.

Recursion

The recursion operation eagerly calculates all possible appeals to the induction hypotheses and makes their results available in a proof goal. Consider for example the third goal in the proof state of the meta-theorem prover in our example. Recall, that the original universal variable E is instantiated by 'app E_1 E_2' after the splitting operation.

$$\begin{array}{|l|}\hline T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ (T_2\ \mathrm{arrow}\ T_1), E_2 : \mathrm{term}\ T_2 \\ \hline \boxed{\mathbf{P}''_3} \in \exists P : (\mathrm{app}\ E_1\ E_2) \overset{1}{\Longrightarrow} (\mathrm{app}\ E_1\ E_2).\top \\ \hline \end{array}$$

The meta-theorem prover is invoked with the argument which hypothesis to do induction on: for this theorem it is E. Therefore, in order to guarantee termination, recursive calls can only be applied to subterms of E. Implicitly by splitting, the meta-theorem prover has learned about the form E. In particular it can derive that E_1 and E_2 are subterms of E. As a matter of fact, these are the only two (non-equal) subterms of E whose type matches the one of the induction hypothesis. Thus, there are only two ways of safely applying the induction hypothesis. The first way is to apply it to (T_2 arrow T_1) and to E_1, and the other way is to apply it to T_2 and E_2. It is the recursion operation that calculate all possible outcomes of appeals to the induction hypothesis. In this case the result is:

$$\exists P_1 : E_1 \overset{1}{\Longrightarrow} E_1.\top$$
$$\exists P_2 : E_2 \overset{1}{\Longrightarrow} E_2.\top$$

Because proof search is restricted to the Π_2-fragment of $\mathcal{M}_2^+$, the result of applying an induction hypothesis lies also in the Π_2-fragment. In this particular example on the other hand, the situation is even simpler: both result formulas are existential and lie therefore in the Σ_1-fragment. We postpone the discussion of the more general case until Section 8.3.2.

Next, the recursion operation makes the witness objects of the recursive calls available as assumptions. Logically speaking, it applies the $\exists\mathsf{L}$ rule of $\mathcal{M}_2^+$ to extract the witness objects P_1 and P_2 in this example.

$$\boxed{T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E_1 : \mathrm{term}\ (T_2\ \mathrm{arrow}\ T_1), E_2 : \mathrm{term}\ T_2, P_1 : E_1 \overset{1}{\Longrightarrow} E_1, P_2 : E_2 \overset{1}{\Longrightarrow} E_2}$$
$$\boxed{\boxed{\mathbf{P}_3''} \in \exists P : (\mathrm{app}\ E_1\ E_2) \overset{1}{\Longrightarrow} (\mathrm{app}\ E_1\ E_2).\top}$$

Because of the regular world assumption applying recursion to the second last proof goal is more difficult. In the lam-case, for example, it is not enough to simply calculate all induction hypotheses, but the theorem prover must also consider extensions of the world in order not to miss any.

$$\boxed{T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E' : \mathrm{term}\ T_1 \to \mathrm{term}\ T_2}$$
$$\boxed{\boxed{\mathbf{P}_2''} \in \exists D : (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T_1.\, E'\ x)) \overset{1}{\Longrightarrow} (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T_1.\, E'\ x)).\top}$$

In this situation the original E has been instantiated to (lam $(\lambda x : \mathrm{term}\ T_1.\, E'\ x)$) by the spitting operation. Without extending the current world, there are no possibilities to apply the induction hypothesis at all. On the other hand, it is possible to apply the induction hypothesis to the body of E', assuming that the world has been extended by one new constructor $x : \mathrm{term}\ T_1$. Therefore, the recursion operation takes the context schema into account and considers all possible extensions of the world in order to determine all inductive calls. For this particular proof goal, there is only one way to extend the world

$$(\underline{x} : \mathrm{term}\ T_1, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L$$

and only one possible appeal to the induction hypothesis:

$$\mathbf{\Pi}(\underline{x} : \mathrm{term}\ T_1, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x})^L.\, \exists P : E_1 \overset{1}{\Longrightarrow} E_1.\top$$

The recursion operation interprets this formula as a new hypothesis and inserts it into the proof goal:

$$\boxed{T_1 : \mathrm{tp}, T_2 : \mathrm{tp}, E' : \mathrm{term}\ T_1 \to \mathrm{term}\ T_2, P : \Pi x : \mathrm{term}\ T_1.\, x \overset{1}{\Longrightarrow} x \to (E'\ x) \overset{1}{\Longrightarrow} (E'\ x)}$$
$$\boxed{\boxed{\mathbf{P}_2''} \in \exists D : (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T_1.\, E'\ x)) \overset{1}{\Longrightarrow} (\mathrm{lam}\ (\lambda x : \mathrm{term}\ T_1.\, E'\ x)).\top}$$

In summary, the recursion operation calculates all possible applications of the induction hypothesis, and it adds the new assumptions into the context of the proof goal. Clearly, the main drawback of this approach is that too many applicable induction hypothesis will slow down the underlying LF theorem prover because of a search space explosion. But this is not a problem for this example. Applying two more filling operations to the remaining two proof goals completes the proof of the reflexivity lemma.

8.3.2 Lemmas

In the previous subsection we have described the three basic operations providing the foundation of the meta-theorem prover. But we have postponed one question: How does the prover apply lemmas? Note that there is one fundamental difference between applying an induction hypothesis and applying a lemma. So far we have only considered the *special* case where an appeal to the induction hypothesis instantiates all universally quantified variables according to the induction ordering. The argument to a lemma application on the other is entirely unconstrained. Therefore, the model used for calculating all induction hypothesis in a forward directed manner is not applicable in this setting. There are simply too many possibilities, possibly even infinitely many.

As a matter of fact, a very similar problem occurs already in the *general* case of determining possible appeals to the induction hypothesis. The previously used technique of extracting the LF-level content from a meta-level formula does not work in this setting if only some but not all of the universally quantified variables are constrained by the termination ordering. In these situations, the result of applying the induction hypothesis is typically a formula that is still in the Π_2-fragment. We call these formulas *residual lemmas* and for the purpose of this subsection, they are treated the same way as lemmas are.

The center of the treatment of lemmas stands the idea to exploit the LF theorem prover to execute the search for lemma applications and their appropriate arguments. But how can this be established? Lemmas are meta-level constructs, and the most basic design principle of $\mathcal{M}_2^+$ is to separate the meta-level from the LF level. By design, the LF theorem prover should not be able to access meta-level lemmas.

Fortunately, there is a solution to this dilemma. Using a technique very similar to skolemization, we can encode meta-level lemmas as Skolem constants provided that these constants are only applied to arguments valid in the regular world. We write $\forall$ for Π to make this distinction notationally self-evident. Consider for example the substitution Lemma 4.5 that is required in the proof of the diamond Lemma 4.6. The substitution lemma is made accessible on the LF level by a Skolem constant $\#subst$.

$$
\begin{array}{l}
\#subst : \\
\quad \forall T_1 : \mathrm{tp}. \forall T_2 : \mathrm{tp}. \forall E_1 : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1. \forall E_1' : \mathrm{term}\ T_2 \to \mathrm{term}\ T_1. \\
\qquad \forall E_2 : \mathrm{term}\ T_2. \forall E_2' : \mathrm{term}\ T_2. \\
\qquad\quad \forall D_1 : (\Pi y : \mathrm{term}\ T_2. y \overset{1}{\Longrightarrow} y \to E_1\ y \overset{1}{\Longrightarrow} E_1'\ y). \forall D_2 : E_2 \overset{1}{\Longrightarrow} E_2'. \\
\qquad\quad E_1\ E_2 \overset{1}{\Longrightarrow} E_1'\ E_2'
\end{array}
$$

Skolem constants are only used for proof search by the underlying LF theorem prover and for no other operation. They are different from regular constants and they are neither considered for splitting nor for recursion. One remark about the current implementation: The LF theorem prover is incomplete because it cannot extend the world during proof search.

How are Skolem constants used? In the pbeta/pbeta-case of the diamond lemma, for example, when automatically generated, the filling operation constructs two calls to the lemmas implicitly. (We omit all implicit arguments to $\#subst$). It is a straightforward algorithm to extract the lemma applications from this proof term and replace it by explicit lemma applications, as shown in Figure 4.4.

$$
\begin{array}{l}
\mid \textbf{dia}\ (\text{pbeta}\ (\lambda x:\ \text{term}\ T.\,\lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^l\ x\ u)\ D_2^l) \ = \\
\qquad (\text{pbeta}\ (\lambda x:\ \text{term}\ T.\,\lambda u : x \overset{1}{\Longrightarrow} x.\, D_1^r\ x\ u)\ D_2^r) \\
\quad \textbf{let} \\
\qquad \textbf{new}\ \underline{x} : \text{term}\ T, \underline{u} : \underline{x} \overset{1}{\Longrightarrow} \underline{x} \\
\qquad \textbf{val}\ \ (P_1\ \underline{x}\ \underline{u}, P_2\ \underline{x}\ \underline{u}) = \textbf{dia}\ (D_1^l\ \underline{x}\ \underline{u})\ (D_1^r\ \underline{x}\ \underline{u}) \\
\quad \textbf{in} \\
\qquad \textbf{let} \\
\qquad\quad \textbf{val}\ (Q_1, Q_2) = \textbf{dia}\ D_2^l\ D_2^r \\
\qquad \textbf{in} \\
\qquad\quad (\#subst\ P_1\ Q_1, \#subst\ P_2\ Q_2) \\
\qquad \textbf{end} \\
\quad \textbf{end}
\end{array}
$$

As final example for the treatment of lemmas, consider the formula describing the diamond Lemma 4.6. It is used in the proof of the strip Lemma 4.7.

$$
\begin{array}{l}
\forall T : \text{tp}.\, \forall E : \text{term}\ T.\, \forall E^l : \text{term}\ T.\, \forall E^r : \text{term}\ T. \\
\quad \forall D^l : E \overset{1}{\Longrightarrow} E^l.\, \forall D^r : E \overset{1}{\Longrightarrow} E^r. \\
\quad \exists E' : \text{term}\ T.\, \exists R^l : E^l \overset{1}{\Longrightarrow} E'.\, \exists R^r : E^r \overset{1}{\Longrightarrow} E'.\, \top
\end{array}
$$

Once the meta-theorem prover has successfully completed the proof of this lemma, it emits new Skolem constants to make it accessible for the subsequent theorem. There are three of these constants, each Skolem constant corresponds to one existential quantifiers.

$$
\begin{array}{l}
\#dia_1 : \forall T : \text{tp}.\, \forall E : \text{term}\ T.\, \forall E^l : \text{term}\ T.\, \forall E^r : \text{term}\ T. \\
\qquad\qquad \forall D^l : E \overset{1}{\Longrightarrow} E^l.\, \forall D^r : E \overset{1}{\Longrightarrow} E^r. \\
\qquad\qquad \text{term}\ T
\end{array}
$$

$$
\begin{array}{l}
\#dia_2 : \forall T : \text{tp}.\, \forall E : \text{term}\ T.\, \forall E^l : \text{term}\ T.\, \forall E^r : \text{term}\ T. \\
\qquad\qquad \forall D^l : E \overset{1}{\Longrightarrow} E^l.\, \forall D^r : E \overset{1}{\Longrightarrow} E^r. \\
\qquad\qquad (\#dia_1\ D^l\ D^r) \overset{1}{\Longrightarrow} E'
\end{array}
$$

$$
\begin{array}{l}
\#dia_3 : \forall T : \text{tp}.\, \forall E : \text{term}\ T.\, \forall E^l : \text{term}\ T.\, \forall E^r : \text{term}\ T. \\
\qquad\qquad \forall D^l : E \overset{1}{\Longrightarrow} E^l.\, \forall D^r : E \overset{1}{\Longrightarrow} E^r. \\
\qquad\qquad E^r \overset{1}{\Longrightarrow} (\#dia_1\ D^l\ D^r)
\end{array}
$$

In summary, the meta-theorem prover can efficiently apply lemmas and residual lemmas by encoding them as Skolem constants in LF. In the implementation the LF theorem prover treats them as LF constants applicable only to closed terms valid in the regularly formed world.

8.3.3 Strategy

Filling, splitting, and recursion are the three basic operations underlying the implementation of the meta-theorem prover. Each of the operations has a different output behavior. Filling for example can either succeed (and solves a proof goal) or fail, indicating that further splitting steps

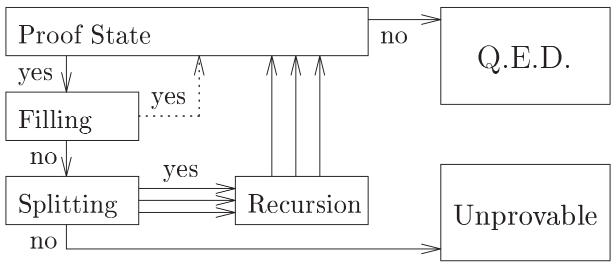

Figure 8.1: Proof strategy

are necessary. Splitting itself is almost always applicable as long as there there are splittable (assumption) variables in the context. On the downside it can be very tricky to predict which assumptions to split. Thus, any implementation of search using these three operations must be *fair* selecting splitting operations, otherwise the search may run into an infinite descent.

The only operation that can be deterministically applied is the recursion operation. It inserts the results of applying *all* induction hypotheses eagerly into the current proof goal, possibly extending the list of residual lemmas. The operation is entirely deterministic and finite, and therefore worth applying to every new proof goal inserted into the proof state by splitting.

These observations lead to the obvious and very straightforward design of a strategy for the theorem prover that is depicted in Figure 8.1. It is this strategy which is implemented in the Twelf system.

Given a proof state consisting of many proof goals, the strategy picks arbitrarily the current proof goal. It then attempts to complete this goal by applying the filling step. There are two possible outcomes. First, the goal has been successfully proven, then it can be safely removed from the proof state, or second the filling step failed and then a splitting operation must be invoked. In general, there are many ways splitting can be applied to a proof state, in the proof of the reflexivity lemma above for example, initially, there are two possible splits on T and E, and in the subsequent lam-case, there are three, and in the app case, there are even four. Splitting typically generates several new proof goals, and each of them is pumped through the recursion operation to compute the result of all inductive calls. Naturally, new assumptions added by recursion may be subject to further splitting steps at later stages of the proof. The new proof goals are added to the proof state.

In the case that neither a filling operation nor a splitting operation can be successfully applied to a proof goal, the meta-theorem prover halts and reports that a proof can not be found. In the case that the filling operation is successful, the meta-theorem simply picks another proof goal from the proof state.

The most difficult decision for Twelf is to select the assumption from the context of a proof goal about which variable to split next. The current implementation employs a very simple and in a few cases unsatisfactory heuristic: for example, it will never split a variable that appears as an index to a type of any other variable, and among the remaining choices it picks a variable that has been part of a splitting operation the least number of times. There are a few other bits of information which influence its choice, such as for example, the position of the variable in the induction order, or the number of cases generated. Concretely, we attach a counter to every splittable variable in the context Ψ of a proof state which is increased and inherited by the children of a the variable affected by a splitting operation. To avoid infinite chains of splitting operations the meta theorem prover is parameterized by an upper bound for the number of splits of one variable. The size of the search space of the meta-theorem prover depends crucially on this bound. As a side effect, it implies fairness of the splitting operation application, since every applicable operations will eventually be applied. Consequently only finitely many splitting operations are applicable.

8.3.4 Correctness

The correctness argument for the implementation of the meta-theorem prover follows the correctness argument of the LF theorem prover. The meta-theorem prover relies on complicated operations that are very difficult to verify, such as splitting, filling, and recursion. Therefore, we should not trust the implementation of the meta-theorem prover. Instead, we should trust an independent proof checker, that verifies the correctness of the proofs generated by the meta-theorem prover.

Proof-checking for $\mathcal{M}_2^+$ is decidable since every proof term constructor uniquely determines the most recently applied rule. Despite this observation an implementation an independent proof checker for $\mathcal{M}_2^+$ is significantly more complicated than a type checker for LF because in addition it also has to verify the termination Condition (5.1), the coverage Condition (5.2), and the strictness Condition (5.3). The decision procedure for the syntactic criterion for the coverage condition is particularly difficult to verify because it relies on the correctness of the unification algorithm that we have defined in Section 7.3.

The current implementation does not provide an independent proof checker for $\mathcal{M}_2^+$, it is still work in progress. A proof-checker for $\mathcal{M}_2^+$ will satisfy the same conditions as the schema-checker for LF was designed to verify [Roh96], namely type preservation, termination, and progress. The main difference between both approaches is that the $\mathcal{M}_2^+$ proof-checker verifies properties about functions in $\mathcal{M}_2^+$, whereas the schema-checker verifies properties about relations represented in LF under a logic programming interpretation. For the purpose of verification, the $\mathcal{M}_2^+$ proof-checker can take full advantage of the type system of $\mathcal{M}_2^+$, all necessary algorithms are described in this thesis. The schema checker on the other hand does not enjoy the luxury of a formal meta-logic, it is merely designed to guarantee termination and coverage properties of logic programs and proofs.

The idea of reducing the problem of correctness away from the tool itself towards the instances the tool generates is not new. Pollack [Pol97] for example distinguishes between the correctness of the method and the correctness of the proofs.

Clearly, the method behind the implementation of the meta-theorem prover in Twelf is in principal correct because it constructs $\mathcal{M}_2^+$ proof-terms, and $\mathcal{M}_2^+$ is sound by Theorem 7.28. To judge if the implementation itself is correct, we propose a small and independent proof-checker that checks each $\mathcal{M}_2^+$ proof term — its design is well-understood, but it is not yet implemented the current version of Twelf. However, a custom made proof checker is not necessary, if we can devise an algorithm that translates Twelf meta-proofs over higher-order encodings, into proofs readable and verifiable by traditional theorem provers. By doing so, the verification problem moves away from $\mathcal{M}_2^+$ into a logic which supports standard induction principles, which relies on the closed world assumption, and for which there are numerous independent implementations. Naturally, after a translation, the proofs explode in size because every appeal to a substitution lemma, weakening lemma, or exchange lemma has to be made explicit.

One such translation technique uses de Bruijn indices [dB72]: variable occurrences are translated into natural numbers. Note, that the correctness of this technique relies on the correctness of the transformation function itself. There are many (more or less) trusted proof checkers that can verify de Bruijn encodings and standard induction principles, such as for example HOL [GM93], LCF/ML [Pau87], Coq [CT95], Lego [Pol94], Isabelle [Pau94], or PVS [OSRSC99].

8.3.5 Limitations

The current implementation of the meta-theorem prover in Twelf is an experimental prototype. Therefore it has several limitations. Some of the limitations are easily generalizable others open entire new research areas. The implementation has one limitation that is due to specialization. In its current form, the meta-theorem prover is restricted to handle only one variable block ρ in the context Ψ of any proof goal. In a situation where more than one variable block is required, the theorem prover fails due to incompleteness. This restriction will be removed in the next release.

A more severe limitation is due to the choice of the splitting variable. Currently, the assumption to be split is chosen by a heuristic, and in some cases it commits to the right choice, but in general it does not. The heuristic implemented in the current prototype is sufficient for many examples and surprisingly effective despite its simplicity, but the general case is not well understood. In particular, failure situations in which no splitting operation makes progress should be be recognized early in the proof but are not in the current implementation. The objective must be to not further explore unpromising branches and provide good feedback to the user of *why* the proof cannot be found.

Therefore, all possible splitting operators that are applicable to a particular proof goal should be ordered in such a way that the "right" splitting operation is among those that rank very highly. Splitting operations, that do not advance the proof should rank very low in this ordering. Only with a better understanding of what constitutes a good splitting operation, the meta-theorem prover stands a chance to formulate helpful error message that may indicate that a lemma is missing or that the current formula to be proven must be further generalized.

The meta-theorem prover works only for the Π_2-fragment of $\mathcal{M}_2^+$. Recall that $\mathcal{M}_2^+$ provides very few connectives for on the level of theorems. In many situations, however, Twelf users would like to formulate and prove theorems that lie outside the Π_2-fragment, but the meta-logic $\mathcal{M}_2^+$ does not support these kind of theorems. In other situations, one may desire to use other connectives than quantification and conjunction, such as, for example, disjunction, implication, or negation (see also the remark on typing continuations in Section 7.1.4). Luckily, for special instances, disjunctions and negations can be encoded directly in LF, and therefore this incompleteness of $\mathcal{M}_2^+$ is not as grave as it looks at first sight.

Yet another connective that is also not provided by $\mathcal{M}_2^+$ but desired by many Twelf users is the ability to express *unique existence*. The reflexivity lemma from above, for example, can be expressed as

$$\forall T : \mathrm{tp}. \forall E : \mathrm{term}\, T. \exists^1 D : E \overset{1}{\Longrightarrow} E. \top$$

where the $\exists^1$ quantifier expresses, that there exists exactly one object of type $E \overset{1}{\Longrightarrow} E$. One remedy to enhance the expressiveness of the meta-logic is to explicitly add equality; if D_1 and D_2 are two objects of type $E \overset{1}{\Longrightarrow} E$ then D_1 *equals* D_2. We postpone any further speculation on how equality can be added to the meta-logic until Section 9.1.3.

Finally, another limitation of the implementation of the meta-theorem prover is that it does not explicitly construct any proof terms yet. Internally, they are there because all the three basic operations such as splitting, recursion, and filling, are directly associated with the recipe of how to construct them; but in the current version Twelf does not export them. Therefore, $\mathcal{M}_2^+$-proofs are currently not verifiable by any other independent and trusted proof checker. This limitation will disappear with the next version.

8.4 A Case Study

In this section we present as case study the entire development of the Church-Rosser example from Chapter 2, and automated versions of the meta-proofs form Chapter 4 in Twelf. We proceed with the presentation in two steps. First, we give a brief overview about Twelf and comment on the concrete syntax implemented in the Twelf system in Section 8.4.1, and then we present the development of the Church-Rosser theorem in Section 8.4.2.

8.4.1 A Brief Overview of Twelf

Twelf implements the logical framework LF; signatures represent all type level and object level constant declarations and are written in regular ASCII files and can be loaded into Twelf. Twelf employs a powerful type reconstruction algorithm that allows the user to be brief and concise. For example, the signatures for the Church-Rosser theorem from Figure 2.2 and Figure 3.1 can be directly loaded into Twelf. It is this elegant correspondence, that makes Twelf an ideal rapid prototyping tool for the design of logics and programming languages. However, this thesis does not account for all details and features that the Twelf system offers. Instead we invite the reader to consult the Twelf manual [PS98] and Pfenning's book [Pfe00] for a complete presentation of of the Twelf system and many more examples.

We begin the discussion with defining lexical conventions before we present the concrete syntax for encoding LF signatures in Twelf. Finally we introduce the syntax of how to express theorems, and proofs in Twelf.

Lexical Conventions

The lexical analysis of Twelf has purposely been kept simple, with few reserved characters and identifiers. As a result one may need to use more whitespace to separate identifiers than in other languages. For example, `A->B` or `A+B` are single identifiers, while `A -> B` and `A + B` both consist of 3 identifiers. During parsing, identifiers are resolved as reserved identifiers, constants, bound variables, or free variables, following the usual rules of static scoping in λ-calculi. Figure 8.2 lists all reserved characters in Twelf.

All printing characters that are not reserved can be included in identifiers, which are separated by whitespace or reserved characters. In particular, `A->B` is an identifier, whereas `A -> B` stands for the type of functions from `A` to `B`. An uppercase identifier is one which begins with an underscore `_` or a letter in the range `A` through `Z`. A lowercase identifier begins with any other character except a reserved one. Numbers also count as lowercase identifiers and are not interpreted specially. Free variables in a declaration must be uppercase, bound variables and constants may be either uppercase or lowercase identifiers.

‘`:`’	colon, constant declaration or ascription
‘`.`’	period, terminates declarations
‘`( )`’	parentheses, for grouping terms
‘`[ ]`’	brackets, for λ-abstraction
‘`{ }`’	braces, for quantification (dependent function types)
‘ ’	whitespace separates identifiers (space, newline, tab, carriage return)
‘`%`’	introduces comments or special keyword declarations
‘`% `’, ‘`%%`’	comment terminated by the end of the line, may contain any characters
‘`%{ }%`’	delimited comment, nested `%{` and `}%` must match
‘`%`*`keyword`*’	various declarations
‘`%.`’	end of input stream
‘`"`’	doublequote, disallowed other printing characters identifier constituents

Figure 8.2: Reserved identifiers

‘`->`’	function type
‘`<-`’	reverse function type
‘`_`’	hole, to be filled by term reconstruction
‘`=`’	definition
‘`type`’	the kind type

Figure 8.3: Reserved identifiers with predefined meaning

Figure 8.3 depicts the five reserved identifiers with a predefined meaning which cannot be changed. These can be constituents of other identifiers which are not interpreted specially. Constants have static scope, which means that they can be shadowed by subsequent declarations. Uppercase identifiers in declarations represent schematic variables.

Syntax for LF

In LF, deductive systems are represented by signatures consisting of constant declarations. Twelf implements declarations in a straightforward way and generalizes signatures by also allowing definitions which are semantically transparent [PS99a]. Twelf currently does not have module-level constructs in the spirit of [HP98] and therefore, for example, signatures cannot be named. Instead, multiple signatures can be manipulated in the programming environment using configurations.

The LF type theory is stratified into three levels: objects, types, and kinds. Twelf does not syntactically distinguish these levels and simply uses one syntactic category of *term*. Similarly, object-level constants and type-level constants as well as variables share one name space of identifiers.

The grammar depicted in Figure 8.4 formalizes the logical framework LF from Section 2.4. It defines the non-terminals *sig*, *decl*, *term* and uses the terminal `id` which stands for identifiers. There are various special declarations `%`*`keyword`* such as `%infix` or `%theorem` with special arguments, such as *ixdecl*, *thdecl*, *pdecl*, or *callpats* which we discuss in detail below. Note, that this

sig	::=		Empty signature
	$\mid$	*decl sig*	Constant declaration
decl	::=	`id :` *term* `.`	$a : K$ or $c : A$
	$\mid$	`id :` *term* `=` *term* `.`	$d : A = M$
	$\mid$	`id =` *term* `.`	$d = M$
	$\mid$	`_ :` *term* `=` *term* `.`	anonymous definition, for type-checking
	$\mid$	`_ =` *term* `.`	anonymous definition, for type-checking
	$\mid$	`%infix` *ixdecl* `.`	operator declaration
	$\mid$	`%name id id.`	name preference declaration
	$\mid$	`%theorem` *thdecl* `.`	theorem declaration
	$\mid$	`%prove` *pdecl* `.`	prove declaration
	$\mid$	`%establish` *pdecl* `.`	prove declaration, don't make available as lemma
	$\mid$	`%assert` *callpats* `.`	assert theorems (only in unsafe mode)

Figure 8.4: Concrete syntax of Twelf

term	::=	`type`	type
	$\mid$	`id`	variable x or constant a, c, or d
	$\mid$	*term* `->` *term*	$A \to B$
	$\mid$	*term* `<-` *term*	$A \leftarrow B$, same as $B \to A$
	$\mid$	`{id :` *term*`}` *term*	$\Pi x : A. K$ or $\Pi x : A. B$
	$\mid$	`[id :` *term*`]` *term*	$\lambda x : A. B$ or $\lambda x : A. M$
	$\mid$	*term term*	$A\ M$ or $M\ N$
	$\mid$	*term* `:` *term*	explicit type ascription
	$\mid$	`_`	hole, to be filled by *term* reconstruction
	$\mid$	`{id}` *term*	same as `{id:_}` *term*
	$\mid$	`[id]` *term*	same as `[id:_]` *term*

Figure 8.5: Syntax for terms

is only a brief description of Twelf, there are many other special declarations that we do not describe here; we restrict this presentation only to the ones that are relevant to the development of the Church-Rosser theorem that we describe in Section 8.4.2.

The syntax for terms is depicted in Figure 8.5. The constructs `{x:U} V` and `[x:U] V` bind the identifier `x` in `V`, which may shadow other constants or bound variables. As usual in type theory, `U -> V` is treated as an abbreviation for `{x:U} V` where `x` does not appear in `V`. However, there is a subtlety in that the latter allows an implicit argument to depend on `x` while the former does not. We shed some light on implicit arguments later in this section.

In the order of precedence, we disambiguate the syntax as follows: Juxtaposition (application) is left associative and has highest precedence. `->` is right and `<-` left associative with equal precedence. `:` is left associative. `{}` and `[]` are weak prefix operators.

New type level and object level constants can be introduced with `id :` *term* `.` Any identifier `x` may be bound by the innermost enclosing binder for `x` of the form `{x:A}` or `[x:A]`. Any identifier

assoc	::=	`none`	not associative
	|	`left`	left associative
	|	`right`	right associative
prec	::=	`nat`	$0 < prec < 10000$
ixdecl	::=	*assoc prec* `id`	

Figure 8.6: User-defined infix operators

which is not explicitly bound may be a declared or defined constant. Any uppercase identifier, that is, identifier starting with _ (underscore) or an upper case letter, may be a free variable. Free variables are interpreted universally and their type is inferred from their occurrences. Any other undeclared identifier is flagged as an error.

Twelf supports notational definitions, currently employing a restriction to allow a simple and efficient internal treatment. Semantically, definitions are completely transparent, that is, both for type checking and the operational semantics definitions may be expanded. Definitions `id :` *term* `=` *term*`.` and `id =` *term*`.` (which is equivalent to `id : _ =` *term*`.`) can only be made on the level of objects, not at the level of type families because the interaction of such definitions with logic programming search has not been fully investigated.

In order to avoid always expanding definitions, Twelf currently only permits strict definitions [PS99a]. A definition of a constant `c` is strict if all arguments to `c` (implicit or explicit) have at least one strict occurrence in the right-hand side of the definition, and the right-hand side contains at least one constant. In practice, most notational definitions are strict.

The user may declare constants to be infix operators. Operator precedence properties are associated with constants, which must therefore already have been declared with a type or kind and a possible definition. It is illegal to shadow an infix operator with a bound variable. We use `nat` for the terminal natural numbers in Figure 8.6. During parsing, ambiguous successive operators of identical precedence such as `a <- b -> c` are flagged as errors. Note that it is not possible to declare an operator with equal or higher precedence than juxtaposition or equal or lower precedence than `->` and `<-`.

During printing, Twelf frequently has to assign names to anonymous variables. In order to improve readability, the user can declare a name preference `%name id id.` for anonymous variables based on their type. Thus name preferences are declared for type family constants. Note that name preferences are not used to disambiguate the types of identifiers during parsing.

Following our same conventions, a name preference declaration has the form `%name a id`, that is, the first identifier must be a type family already declared and the second is the name preference for variables of type `a`. The second identifier must be uppercase, that is, start with a letter from `A` through `Z` or an underscore _. Anonymous variables will then be named `id1`, `id2`, etc.

Representations of deductions in LF typically contain a lot of redundant information. In order to make LF practical, Twelf gives the user the opportunity to omit redundant information in declarations and reconstructs it from context. Unlike for functional languages, this requires recovering objects as well as types, so we refer to this phase as term reconstruction.

There are criteria which guarantee that the term reconstruction problem is decidable, but unfortunately these criteria are either very complicated or still force much redundant information to be supplied. Therefore, the Twelf implementation employs a reconstruction algorithm which always terminates and gives one of three answers:

1. yes, and here is the most general reconstruction;

2. no, and here is the problem; or

3. maybe.

The last characterizes the situations where there is insufficient information to guarantee a most general solution to the term reconstruction problem. Because of the decidable nature of type-checking in LF, the user can always annotate the term further until it falls into one of the definitive categories. For a detailed discussion on many examples related to type reconstruction consult [PS98].

Syntax for $\mathcal{M}_2^+$

There are four special declarations that define the interaction with the meta-theorem prover. The first declaration is `%theorem`, that declares an $\mathcal{M}_2^+$-formula that is to be proven using either `%prove`, `%establish`, or `%assert`. `%prove` and `%establish` take as argument the maximal filling bound, that restricts the size of the search space of the LF theorem prover, an induction ordering, and a call pattern that relates the induction variables to the actual arguments of the theorem. `%assert` on the other hand only expects a call pattern. It allows to assert the correctness of a theorem even if Twelf cannot prove it. Naturally, in a valid proof development no `%assert` is admissible. Hence, in order to take advantage of this feature, the user has to toggle Twelf into unsafe mode.

The syntax for theorems is defined in Figure 8.7. Abstractly, arbitrary quantifier alternations are allowed, but Twelf rejects any formula that lies outside the Π_2-fragment of $\mathcal{M}_2^+$. The `forallG` quantifier binds a context schema that defines a regular extension to the current world described by a context schema for which the theorem is to be proven that is denoted by the non-terminal symbol `regext`. The `some` *decs* `pi` *decs*-blocks describe the individual context blocks in terms of a SOME-block and a BLOCK-block. `forall` and `forall*` are two different notations for the same universal quantifier. The difference between the two is negligible in the current version. If Twelf would generate proof terms, the `forall*`-quantifier defines which universal quantified variables are implicit and need not to be displayed in the proof term. The existential quantifier and $\top$ have the expected meaning.

Example 8.4 (Diamond lemma in Twelf) The diamond Lemma 4.6 can be expressed in Twelf as formula

```
%theorem dia : forallG (some {A:tp} pi {x: term A} {idx : x => x})
               forall* {A:tp}{M:term A}{M':term A}{M'':term A}
               forall  {D1: M => M'} {D2: M => M''}
               exists  {N:term A}{E1: M' => N}{E2 : M'' => N}
               true.
```

dec	::=	{`id`:*term*}	$x : A$
decs	::=	*dec*	singleton block
	|	*dec decs*	block of declarations
regext	::=	`some` *decs* `pi` *decs*	context block
	|	`some` *decs* `pi` *decs* `|` *regext*	context schema
formula	::=	`forallG` *regext formula*	Quantification over regular contexts
	|	`forall*` *decs formula*	implicit universal
	|	`forall` *decs formula*	universal
	|	`exists` *decs formula*	existential
	|	`true`	truth
thdecl	::=	`id : ` *formula*	Assigning a name to a formula

Figure 8.7: Syntax for $\mathcal{M}_2^+$-formulas in Twelf

The argument to the `forallG` quantifier defines the regular extension of the world, the three bound arguments `M`, `M'`, `M''` are implicit; once a proof term is generated (see Figure 4.4) it only expects two arguments `D1` and `D2` and not five. The `forall` quantifier binds `D1` and `D2` and `exists` binds the three returning arguments `N`, `E1`, and `E2`.

More examples of theorems are described below in Section 8.4.2. In summary, Twelf only accepts formulas of the Π_2-fragment, i.e. of the form

`forallG` *regext* `forall*` *decs* `forall` *decs* `exists` *decs* `true.`

After its declaration a theorem is subject to automated proof search. It is initiated by a `%prove` declaration that expects as arguments the maximal filling depth, and an induction order. The induction order associates argument positions of the theorem via call patterns. A call pattern consists of the name of the theorem applied to as many arguments as there are `%forall` and `%exists` quantified declarations (it should be read as a relation that associates input positions with output positions). Each argument can be either named or anonymous. Admissible induction orders include lexicographic and simultaneous extensions of the subterm ordering as explained in Section 7.2. Their syntax in Twelf is depicted in Figure 8.8.

The case of mutually recursive predicates is particularly complex and requires mutually dependent call patterns with mutually related arguments. Their syntax is given in Figure 8.9.

Example 8.5 (Call pattern for diamond lemma) There are several call patterns for the diamond lemma: The most complete is `dia D1 D2 N E1 E2` but in general one typically specifies only those arguments in the call pattern that are needed in an induction ordering `dia D1 D2 _ _ _`.

All variables used to specify an induction order for `%prove` declaration must be upper case, and they must occur in the call patterns. In addition, no variable may be repeated. Furthermore,

ids	$::=$		empty list of arguments
	$\mid$	`id` *ids*	argument name
arg	$::=$	`id`	single argument
	$\mid$	`(`*ids*`)`	mutual arguments
orders	$::=$		empty list of orders
	$\mid$	*order orders*	component order
order	$::=$	*arg*	subterm order
	$\mid$	$\{$*orders*$\}$	lexicographic order
	$\mid$	`[`*orders*`]`	simultaneous order

Figure 8.8: Syntax for induction orders in Twelf

args	$::=$		no argument
	$\mid$	*id args*	named argument
	$\mid$	_ *args*	anonymous argument
callpat	$::=$	`id` *args*	$a\ x_1\ \ldots\ x_n$
callpats	$::=$	`(`*callpat*`)`	single call pattern
	$\mid$	`(`*callpat*`)` *callpats*	mutual call patterns

Figure 8.9: Syntax for call-patterns in Twelf

pdecl	$::=$	`nat` *order callpats*	bound, induction order, theorems

Figure 8.10: Syntax for proof declarations in Twelf

all arguments participating in the termination order must occur in the call patterns in input positions. The argument vector `pdecl` to a `%prove` declaration is depicted in Figure 8.10.

In order to accept a declaration of the form `%prove`, or `%establish`, Twelf activates the meta-theorem prover and attempts to construct a proof. If the meta-theorem prover reports failure, Twelf halts with an error message that a proof could not be found. On the other hand if it finds a proof it applies skolemization and makes the lemma accessible for subsequent proofs. However, adding new Skolem constants may lead to an explosion of the respective search spaces for subsequent theorem proving task. The user can prevent these additions by using `%establish` instead of `%prove`.

The meta-theorem prover implementation has only prototype status. Its proof strategy is simple yet powerful, but in some situations Twelf is not able to find a proof because of search space explosions, due to continuous splits of wrong assumptions or the complexity of elementary reasoning in LF. Twelf offers a way that the user can continue the development by simply asserting that a theorem holds. Obviously, this is a rather dangerous operation, and it requires the user to put Twelf into *unsafe* mode from the Twelf main menu. Different from `%prove`, `%assert` followed by a call pattern asserts a theorem without proving it. This unsafe option of Twelf should only be used with extreme care.

8.4.2 Developing the Church-Rosser Theorem in Twelf

We begin this case study with encoding the LF declarations from Figure 2.2. In essence we replay almost exactly the development from Chapter 2. Here are the declaration of the types `tp` and terms `term`.

```
tp : type.                                  %name tp T.
arrow : tp -> tp -> tp.                     %infix right 10 arrow.

term : tp -> type.                          %name term E.
lam : (term T1 -> term T2) -> term (T1 arrow T2).
app : term (T1 arrow T2) -> term T1 -> term T2.
```

We follow the development in Section 2.5 and introduce the ordinary reduction relation for simply-typed terms. '`-->`' is a type family representing the single step reduction from a term of type `A` to another term of the same type. We declare it as infix operator.

```
--> : term T -> term T -> type.  %infix none 10 -->.
                                 %name --> R.

rbeta :   (app (lam E1) E2) --> E1 E2.

rlam  :    ({x:term T1} E x --> E' x)
      ->            (lam E) --> (lam E').

rapp1 :               E1 --> E1'
      ->    (app E1 E2) --> (app E1' E2).

rapp2 :               E2 --> E2'
      ->    (app E1 E2) --> (app E1 E2').
```

Next, the single step reduction relation is generalized to a multi step reduction relation '`-->*`' by defining its reflexive and transitive closure. '`-->*`' is used as an infix operator. The two inference rules are represented by `rid` and `rstep`.

```
-->* : term T -> term T -> type.  %infix none 10 -->*.
                                  %name -->* R*.

rid   :    E -->* E.

rstep :    E --> E'
      ->  E' -->* E''
      ->   E -->* E''.
```

And finally, the ordinary reduction relation can be generalized to a conversion relation by building the reflexive, transitive, and symmetric closure of the ordinary multi-step reduction relation.

```
<-> : term T -> term T -> type.  %infix none 10 <->.
                                 %name <-> C.

rrefl :    E <-> E.

rred  :    E -->* E'
      ->   E <-> E'.

rsymm :    E <-> E'
      ->  E' <-> E.

rtrans:    E <-> E'
      ->  E' <-> E''
      ->   E <-> E''.
```

We formalize the single-step parallel reduction relation in Twelf, which we generalize to a multi-step parallel reduction, and parallel conversion, as already depicted in Figure 3.1. Note, that declarations in Twelf syntax are in very direct correspondence to the LF declarations given in Chapter 3. It is this elegance, that gives Twelf the expressive power and the meta-theorem prover its deductive power.

```
=> : term T -> term T -> type.  %infix none 10 =>.
                           %name => R.

pbeta : ({x:term T} x => x -> E1 x => E1' x)
      ->                     E2 => E2'
      ->       (app (lam E1) E2) => E1' E2'.

papp  :           E1 => E1'
      ->          E2 => E2'
```

```
      ->  (app E1 E2) => (app E1' E2').

plam  : ({x:term T} x => x -> E x => E' x)
      ->                  lam E => lam E'.
```

As for ordinary reduction, the single step parallel reduction can be generalized to a multi-step parallel reduction, just as discussed in Section 3.2.2. The resulting type family is an infix operator '=>*', and its semantics is expressed by two constants in Twelf in the following way.

```
=>* : term T -> term T -> type.  %infix none 10 =>*.
                                 %name =>* R*.

pid   :           E =>* E.

pstep :           E => E'
      ->         E' =>* E''
      ->          E =>* E''.
```

And again, following a very similar strategy as in the ordinary case, the concept of parallel conversion is the result of closing the parallel multi-step reduction under reflexivity, symmetry, and transitivity.

```
<=> : term T -> term T -> type.  %infix none 10 <=>.
                                 %name <=> C.

pred   :  E  =>* E'
       -> E  <=> E'.

pexp   :  E  =>* E'
       -> E' <=> E.

ptrans :  E  <=> E'
       -> E' <=> E''
       -> E  <=> E''.
```

This concludes the encoding of the simply-typed λ-calculus and its ordinary and parallel reduction semantics in Twelf. Next we tackle the proof of the Church-Rosser theorem itself; and again, the elegance of Twelf allows us to follow directly the development as described in Section 3.2.1 very closely. In order to emphasize this point, we show all theorems from Chapter 3 and their formalizations in Twelf. We also comment on the timing results of each of the proofs.

Lemma 3.1 (Transitivity of $\xrightarrow{*}$) *If $\mathcal{D}_1 :: e \xrightarrow{*} e'$ and $\mathcal{D}_2 :: e' \xrightarrow{*} e''$ then $e \xrightarrow{*} e''$.*

This lemma can be directly formalized in Twelf. The proof goes by simultaneous induction over `D1` and `D2`, and the search space of the underlying LF theorem prover is limited by the bound 4. All experiments with the Twelf meta-theorem prover on which we report in this thesis were conducted on a Pentium II 400, with 192MB of RAM. This proof of the transitivity lemma was found in 0.01 sec.

```
%theorem trans* : forall* {T: tp}{E: term T}{E': term T}{E'': term T}
                  forall  {D1: E -->* E'}{D2: E' -->* E''}
                  exists  {R: E -->* E''}
                  true.
%prove 4 [D1 D2] (trans* D1 D2 _).
```

Following the development of Section 3.2.1, we will now employ Twelf to prove all three parts of Lemma 3.2.

Lemma 3.2 (Admissible rules)

1. *If* $\mathcal{D} :: e \xrightarrow{*} e'$ *then* $\lambda x : \tau.\, e \xrightarrow{*} \lambda x : \tau.\, e'$
2. *If* $\mathcal{D} :: e_1 \xrightarrow{*} e_1'$ *then* $e_1\, e_2 \xrightarrow{*} e_1'\, e_2$
3. *If* $\mathcal{D} :: e_2 \xrightarrow{*} e_2$ *then* $e_1\, e_2 \xrightarrow{*} e_1\, e_2'$

Only because of the inherent similarity of the three properties we have summarized them to one lemma; in fact, they are not mutually dependent on each other. Each of the cases can be formalized and automatically proven in Twelf. The first case rests on the regular world assumption. Twelf derives the admissibility of reductions under the λ-binder in 0.25 sec and the other two parts in 0.17 sec and 0.024 sec, respectively.

```
%theorem lm* : forallG (some {T: tp} pi {x: term T})
               forall* {T1: tp}{T2: tp}
                       {E: term T1 -> term T2}{E': term T1 -> term T2}
               forall  {D: {x: term T1} (E x) -->* (E' x)}
               exists  {R: (lam E) -->* (lam E')}
               true.
%prove 4 D (lm* D _).

%theorem apl1* : forall* {T1: tp}{T2: tp}
                         {E1: term (T1 arrow T2)}{E1': term (T1 arrow T2)}
                         {E2: term T1}
                 forall  {D: E1 -->* E1'}
                 exists  {R: (app E1 E2) -->* (app E1' E2)}
                 true.
%prove 3 D (apl1* D _).

%theorem apr1* : forall* {T1: tp}{T2: tp}
                         {E1: term (T1 arrow T2)}
                         {E2': term T1}{E2: term T1}
                 forall  {D: E2 -->* E2'}
                 exists  {R:  (app E1 E2) -->* (app E1 E2')}
                 true.
%prove 3 D (apr1* D _).
```

The informal development in Section 3.2.3 continues with the presentation of the reflexivity lemma 4.3. In this formal development on the other hand, we postpone its proof until the point where we prove the equivalence of ordinary and parallel reductions. Due to an incompleteness of Twelf, Lemma 3.11 can only be proven simultaneously with Lemma 4.3 even though Lemma 4.3 itself could be proven on its own. Therefore we continue the formal development with the transitivity proof for parallel deduction whose construction takes Twelf merely 0.008 sec.

Lemma 3.5 (Transitivity of $\stackrel{*}{\Longrightarrow}$) *If $\mathcal{D}_1 :: e \stackrel{*}{\Longrightarrow} e'$ and $\mathcal{D}_2 :: e' \stackrel{*}{\Longrightarrow} e''$ are closed then $e \stackrel{*}{\Longrightarrow} e''$.*

```
%theorem trans : forallG (some {T: tp} pi {x: term T}{idx : x => x})
                 forall* {T: tp}{E: term T}{E': term T}{E'': term T}
                 forall  {D1: E =>* E'}{D2: E' =>* E''}
                 exists  {R: E =>* E''}
                 true.
%prove 4 [D1 D2] (trans D1 D2 _).
```

Following the informal development, the substitution lemma is next:

Lemma 3.6 (Substitution lemma) *Consider the situation where a list of the following assumptions is present*

$$x_1 :: \mathit{term}\ \tau_1, u_1 :: x_1 \stackrel{1}{\Longrightarrow} x_1, \ldots, x_n :: \mathit{term}\ \tau_n, u_n :: x_n \stackrel{1}{\Longrightarrow} x_n$$

If

$$\begin{array}{c} \overline{y \stackrel{1}{\Longrightarrow} y}\,v \\ \mathcal{D}_1 \\ e_1 \stackrel{1}{\Longrightarrow} e_1' \end{array}$$

and $\mathcal{D}_2 :: e_2 \stackrel{1}{\Longrightarrow} e_2'$ then exists a reduction $e_1[e_2/y] \stackrel{1}{\Longrightarrow} e_1'[e_2'/y]$.

The formalization of this substitution lemma in Twelf makes the power and elegance of higher-order representation techniques explicit. The assumption `D1` stands for an arbitrary LF function that expect `y:term T` and `v:y => y` as arguments. Thus the formulation of a substitution and an automated proof lie well outside the scope of any other first-order theorem prover. Twelf can prove the substitution lemma in 0.025 sec.

```
%theorem subst : forallG (some {T: tp} pi {x: term T}{idx : x => x})
                 forall* {T1: tp}{T2: tp}
                         {E1: term T1 -> term T2}{E1': term T1 -> term T2}
                         {E2: term T1 }{E2': term T1}
                 forall  {D1: {x: term T1} x => x -> E1 x => E1' x}
                         {D2: E2 => E2'}
                 exists  {R: E1 E2 => E1' E2'}
                 true.
%prove 6 D1 (subst D1 _ _).
```

The diamond lemma from Section 3.2.3 can also be directly formalized in Twelf.

Lemma 3.7 (Diamond lemma) *Let Φ be the list of given assumptions. If $\mathcal{D}^l :: e \stackrel{1}{\Longrightarrow} e^l$ and $\mathcal{D}^r :: e \stackrel{1}{\Longrightarrow} e^r$ then there exists a common reduct e', such that $\mathcal{R}^l :: e^l \stackrel{1}{\Longrightarrow} e'$ and $\mathcal{R}^r :: e^r \stackrel{1}{\Longrightarrow} e'$.*

$$\begin{tikzcd}
& e \arrow[dl, "\mathcal{D}^l"'] \arrow[dr, "\mathcal{D}^r"] & \\
e^l \arrow[dr, dotted, "\mathcal{R}^l"'] & & e^r \arrow[dl, dotted, "\mathcal{R}^r"] \\
& e' &
\end{tikzcd}$$

Its proof is quite involved, as we have shown in Section 3.2.3, since we have to distinguish many cases; nevertheless, Twelf constructs the proof in 8.625 sec.

```
%theorem dia : forallG (some {T: tp} pi {x: term T}{idx : x => x})
               forall* {T: tp}{E: term T}{El: term T}{Er: term T}
               forall  {Dl: E => El}{Dr: E => Er}
               exists  {E': term T}{Rl: El => E'}{Rr : Er => E'}
               true.
%prove 3 [Dl Dr] (dia Dl Dr _ _ _).
```

In order to prove the Church-Rosser theorem for parallel reduction, we generalized the two single-step reduction arrows in the formulation of the diamond lemma in two steps to multi-step reduction arrows. First we proved the strip lemma, and second the confluence lemma.

Lemma 3.8 (Strip lemma) *Let Φ be the dynamic extension of the world. If $\mathcal{D}^l :: e \stackrel{1}{\Longrightarrow} e^l$ and $\mathcal{D}^r :: e \stackrel{*}{\Longrightarrow} e^r$ then there exists a common reduct e', such that $\mathcal{R}_1 :: e^l \stackrel{*}{\Longrightarrow} e'$ and $\mathcal{R}_2 :: e^r \stackrel{1}{\Longrightarrow} e'$.*

$$\begin{tikzcd}
& e \arrow[dl] \arrow[dr, "*"] & \\
e^l \arrow[dr, dotted, "*"'] & & e^r \arrow[dl, dotted] \\
& e' &
\end{tikzcd}$$

The strip lemma is easily formalized in Twelf, but it takes surprising 335.266 sec to prove it. This is a real surprise, considering how simple its proof actually is. Recall from Section 3.2.3, that it follows by a simple induction on the multi-step derivation and compare it to the complexity of the proof of the diamond lemma. We suspect that the slow-down is caused by the transitivity rule in connection with the number of lemmas introduced so far. In particular, the conclusion of the substitution lemma falls outside the pattern fragment causing the underlying LF theorem prover to struggle with constraints; in addition the intermediate term whose existence is postulated by the transitivity rule does not contribute to the solution of those constraints at all.

```
%theorem strip : forallG (some {T: tp} pi {x: term T}{idx : x => x})
                 forall* {T: tp}{E: term T}{El: term T}{Er: term T}
                 forall  {Dl: E => El}{Dr: E =>* Er}
                 exists  {E': term T}{Rl: El =>* E'}{Rr: Er => E'}
                 true.
%prove 4  [Dr] (strip _ Dr _ _ _).
```

By generalizing the remaining single-step reductions of the strip lemma, to multi-step reductions, one obtains the confluence lemma.

Lemma 3.9 (Confluence lemma) *Let Φ be the dynamic extension of the world. If $\mathcal{D}^l :: e \overset{*}{\Longrightarrow} e^l$ and $\mathcal{D}^r :: e \overset{*}{\Longrightarrow} e^r$ then there exists a common reduct e', such that $\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e'$ and $\mathcal{R}_2 :: e^r \overset{*}{\Longrightarrow} e'$.*

$$\begin{tikzcd} & e \arrow[dl, "*"'] \arrow[dr, "*"] & \\ e^l \arrow[dr, dotted, "*"'] & & e^r \arrow[dl, dotted, "*"] \\ & e' & \end{tikzcd}$$

Because of the same effects that slowed down the proof of the strip lemma, the proof of the confluence lemma is significantly slower than the proof of the substitution or the diamond lemma. It takes Twelf 40.989 sec to prove it.

```
%theorem conf : forallG (some {T: tp} pi {x: term T}{idx : x => x})
                forall* {T: tp}{E: term T}{El: term T}{Er: term T}
                forall  {Dl: E =>* El}{Dr: E =>* Er}
                exists  {E': term T}{Rl: El =>* E'}{Rr: Er =>* E'}
                true.
%prove 4 Dl (conf Dl _ _ _ _).
```

Following the development from Section 3.2.3, it is now possible to proof the Church-Rosser theorem for parallel reduction.

Theorem 3.10 (Church-Rosser) *Let Φ be the dynamic extension of the world. If $\mathcal{D} :: e^l \Longleftrightarrow e^r$ then there exists a common reduct e', such that $\mathcal{R}_1 :: e^l \overset{*}{\Longrightarrow} e'$ and $\mathcal{R}_2 :: e^r \overset{*}{\Longrightarrow} e'$.*

$$\begin{tikzcd} e^l \arrow[rr, leftrightarrow] \arrow[dr, dotted, "*"'] & & e^r \arrow[dl, dotted, "*"] \\ & e' & \end{tikzcd}$$

The proof goes by induction on `D`, and it takes Twelf 3.283 sec to construct it.

```
%theorem cr-par : forallG (some {T: tp} pi {x: term T}{idx : x => x})
                  forall* {T: tp}{El: term T}{Er: term T}
                  forall  {D: El <=> Er}
                  exists  {E': term T}{Rl: El =>* E'}{Rr: Er =>* E'}
                  true.
%prove 3 D (cr-par D _ _ _).
```

This concludes the meta-theoretic development of the proof of the Church-Rosser theorem for parallel reduction. The reader should have noticed, that the formal development is extremely close to the informal development. Every informal proof can be formalized an automatically deduced. But more importantly, no additional lemmas arose and needed to be proven! Typically, a development like this in a first-order based system with standard induction principles requires a lot of special infrastructure to encode parametric and hypothetical constructions such as explicit encodings of variables and substitutions. In addition, it requires a lot of extra meta-theoretic reasoning about their properties. This observation clearly justifies the use of higher-order representation techniques in order to support an elegant development of the meta-theory.

We continue with the exposition from Section 3.2.3 and derive the Church-Rosser for ordinary reduction in Twelf. As above, we accurately follow the structure of the development in Section 3.2.3. In particular we begin with the equivalence proof of the single-step correspondence between parallel and ordinary reduction.

Lemma 3.11 (Single-step correspondence)

1. *If* $\mathcal{D} :: e^l \stackrel{1}{\Longrightarrow} e^r$ *then* $e^l \stackrel{*}{\longrightarrow} e^r$.
2. *If* $\mathcal{D} :: e^l \stackrel{1}{\longrightarrow} e^r$ *then* $e^l \stackrel{1}{\Longrightarrow} e^r$.

Recall from the informal proof, that the second half of this theorem depends on the reflexivity Lemma 3.4 whose proof we have postponed so far. Twelf can prove the reflexivity lemma on its own, but because of an incompleteness in the implementation it cannot prove the second half! This artifact is due to the different treatment of induction hypothesis and lemmas. Induction hypothesis are applied by the recursion operation which may extend the regular world Φ. As discussed in Section 8.3.2, lemmas on the other hand can only be applied in form of Skolem constants during the filling operation, and filling cannot extend the world. This incompleteness will be removed in the next released version of the Twelf system.

Lemma 3.4 (Reflexivity of $\stackrel{1}{\Longrightarrow}$) *Consider the situation where a list of the following assumptions is present*

$$x_1 :: \mathit{term}\ \tau_1, u_1 :: x_1 \stackrel{1}{\Longrightarrow} x_1, \ldots, x_n :: \mathit{term}\ \tau_n, u_n :: x_n \stackrel{1}{\Longrightarrow} x_n$$

Then for any well-typed term e*, there exists a derivation of* $e \stackrel{1}{\Longrightarrow} e$.

The first case of the single-step correspondence is proven by Twelf in 0.094 sec.

```
%theorem single1: forallG (some {T: tp} pi {x: term T}{eqx: x => x})
                  forall* {T: tp}{El: term T}{Er: term T}
                  forall  {D: El => Er}
                  exists  {R: El -->* Er}
                  true.
%prove 3 D (single1 D _).
```

And the second case, proven simultaneously with the reflexivity lemma takes only 0.045 sec.

```
%theorem single2: forallG (some {T: tp} pi {x: term T}{eqx: x => x})
                  forall* {T: tp}{El: term T}{Er: term T}
                  forall  {D: El --> Er}
                  exists  {R: El => Er} true.
%theorem refl : forallG (some {T: tp} pi {x: term T}{eq: x => x})
                forall* {T: tp}
                forall  {E: term T}
                exists  {R: E => E} true.

%prove 3 (E D) (refl E _) (single2 D _).
```

This result guarantees that there is a correspondence between single parallel reduction steps, and possibly several ordinary reduction steps. Clearly we can generalize it to a correspondence result about multi-step reductions.

Lemma 3.12 (Multi-step correspondence) $\mathcal{D} :: e^l \xrightarrow{*} e^r$ *iff* $\mathcal{R} :: e^l \overset{*}{\Longrightarrow} e^r$

Twelf proves the sufficient direction of this Lemma in 0.021 sec, and the necessary direction in 1.228 sec.

```
%theorem multi1: forall* {T: tp}{El: term T}{Er: term T}
                 forall  {D: El -->* Er}
                 exists  {R: El =>* Er} true.
%prove 3 D (multi1 D _).
%theorem multi2: forall* {T: tp}{El: term T}{Er: term T}
                 forall  {D: El =>* Er}
                 exists  {R: El -->* Er} true.
%prove 4 D (multi2 D _).
```

The three remaining lemmas analyze the correspondence between parallel conversion and ordinary conversion. Recall that the concept of ordinary conversion is closed under symmetry, differently from parallel conversion. But as we have already shown informally in Section 3.2.3, symmetry is an admissible rule of inference for parallel conversion.

Lemma 3.13 (Symmetry) *If* $\mathcal{D} :: e^l \Longleftrightarrow e^r$ *then* $\mathcal{R} :: e^r \Longleftrightarrow e^l$

The proof goes by induction on `D`, and it takes Twelf 0.006 sec to derive this result.

```
%theorem symm: forall* {T: tp}{El: term T}{Er: term T}
               forall  {D: El <=> Er}
               exists  {R: Er <=> El} true.
%prove 2 D (symm D _).
```

Since symmetry is admissible, there is a correspondence between ordinary conversion and parallel conversion.

Lemma 3.14 (Conversion correspondence)

1. *If* $\mathcal{D} :: e^l \longleftrightarrow e^r$ *then* $e^l \Longleftrightarrow e^r$

2. If $\mathcal{D} :: e^l \Longleftrightarrow e^r$ then $e^l \longleftrightarrow e^r$

Twelf proves the first direction in 0.310 sec, and the second direction in 0.021 sec.

```
%theorem conv1: forall* {T: tp}{El: term T}{Er: term T}
                forall  {D: El <=> Er}
                exists  {R: El <-> Er} true.
%prove 4 D (conv1 D _).
%theorem conv2: forall* {T: tp}{El: term T}{Er: term T}
                forall  {D: El <-> Er}
                exists  {R: El <=> Er} true.
%prove 3 D (conv2 D _).
```

Thus, as partial result Twelf has shown that the Church-Rosser theorem for parallel reduction holds, and that parallel reduction models ordinary reduction and vice versa. Thus, the Church-Rosser theorem for ordinary relation follows directly from applying these two properties. Two well-typed terms that are convertible via ordinary reduction, are also convertible via parallel reduction. Therefore, by the Church-Rosser theorem, there exists a common reduct, and two reduction sequences, reducing each of the terms to the same common reduct. Using the previously proven correspondence theorem, for each of those two parallel reductions there are corresponding ordinary reductions to the same common reduct, and the Church-Rosser theorem is proven.

Theorem 3.15 (Church-Rosser for ordinary reduction) *If $e^l \longleftrightarrow e^r$ then there exists a common reduct e', s.t. $e^l \stackrel{*}{\longrightarrow} e'$ and $e^r \stackrel{*}{\longrightarrow} e'$.*

In order to construct this proof, Twelf delegates the construction of the argument to the underlying LF theorem prover, that attempts to fill the existential quantifier by one appeal the filling operation. Unfortunately, the search space is too big, because many auxiliary lemmas have been proven. In addition, because lemmas are applied during filling, the LF theorem prover has to traverse a search space of at least depth 6 or 7. This search space is huge.

To help Twelf to find this result more quickly, we prove first an intermediate result, namely that ordinary conversion guarantees the existence of two parallel multi-step reductions to the common reduct. The LF theorem prover can prove this fact in 2.657 sec while traversing a search space up to depth 3. Using this intermediate result, the search space for the actual Church-Rosser theorem for ordinary reduction has also reduced to depth 3, and Twelf is able to find the proof quickly in 0.822 sec. Therefore, sometimes we need additional lemmas, if only for performance reasons.

```
%theorem cr-ord': forall* {T: tp}{El: term T}{Er: term T}
                  forall  {D: El <-> Er}
                  exists  {E': term T}{Rl: El =>* E'}{Rr: Er =>* E'} true.
%prove 3 [] (cr-ord' _ _ _ _).
%theorem cr-ord: forall* {T: tp}{El: term T}{Er: term T}
                 forall  {D: El <-> Er}
                 exists  {E': term T}{Rl: El -->* E'}{Rr: Er -->* E'} true.
%prove 3 [] (cr-ord _ _ _ _).
```

This result concludes the presentation of the case study. In summary, Twelf's expressive power allows in this experiment a almost direct formulation of lemmas and theorems needed to proof the Church-Rosser theorem. In particular, all proofs have been generated automatically, from the information presented in this section. In the current version of Twelf, proof terms are not explicitly generated and exported to the user level yet, but if they were, they resemble very much the proofs presented in Section 3.2.3 and in [Pfe93].

The implementation of the Twelf system provides (undocumented) functionality, that allows the user to step through the proof, thus verifying that it works properly. Throughout the entire development of the Church-Rosser example we deviated only in two places from the informal development. First, the single correspondence lemma and the reflexivity lemma had to be made mutually dependent in order to allow for regular extensions of the world when applying a lemma, and second the LF theorem prover is not efficient enough to put all pieces together for the proof of the Church-Rosser theorem. The first restriction will disappear with future releases of Twelf, and the second requires additional research on how to search for objects in LF more efficiently.

8.5 Experimental results

The formal development of the Church-Rosser theorem for the simply typed λ-calculus is only one of many examples, Twelf has been put to work on. Other examples come from the area of programming languages and logics, and in this section we attempt to sketch other experiments we have conducted in Twelf and that we have summarized in Figure 8.11. All timings in this figure are taken on a Pentium II/400 Mhz, 192 MB RAM.

The first two entries in this table describe experiments which involve cut-elimination. Twelf can fully automatically prove cut-elimination for full first-order intuitionistic logic in 6 minutes and 35 seconds. The proof it constructs is very similar to the proof described in [Pfe95]. The main difference is that Twelf has to consider significantly more cases, because it can apply splitting only in a hierarchical manner.

The cut-elimination result [Gen35] is a very important and fundamental result in logic and the area of automated theorem proving. By inspection of the inference rules of a cut-free sequent calculus for either intuitionistic or classical logic for example follows that falsehood is not derivable in this system, therefore warranting the soundness of the calculus and of the logic. The cut-elimination result is very important for the area of automated deduction since it guarantees the subformula property for the cut-free fragment of any sequent calculus.

For intuitionistic and classical logic, the cut rule is an admissible rule of inference rule. This is the basic insight for the cut-elimination theorem and it is not easy to prove. The sequent calculus for intuitionistic logic, for example, contains 18 inference rules; since the cut-rule has two premisses this means that all in all, 324 cases are to be considered in the worst case.

In [Pfe95] the representation of the sequent calculus for classical logic is equally elegant to the one for intuitionistic logic. Nevertheless, the strategy employed in Twelf is not sophisticated enough to prove cut-elimination for this logic. Wrong choices of splitting operations mislead the prover, and a proof cannot be found in tolerable time.

Another experiment that we have conducted in Twelf is the development of a functional programming called Mini-ML [Pfe00]. For a language that contains a simple inductive datatype (the natural numbers), anonymous functions, applications, let binding and fixed points, Twelf can prove automatically properties such as: *the evaluation of an expression yields a value*, or

Experiment	**Theorem**	**Time**
First-order intuitionistic logic	Admissibility of cut-rule	6 min 35 sec
(Sequent calculus)	Cut-elimination	0.28 sec
First-order classical logic	Admissibility of cut-rule	not yet
(Sequent calculus)	Cut-elimination	0.68 sec
Mini-ML	Value soundness	0.13 sec
	Type preservation	0.42 sec
	Evaluation/Reduction	0.66 sec
	Uniqueness of typing	0.25 sec
Compilation	Soundness	not yet
	Completeness	1.13 sec
	Proof equivalence	0.46 sec
Logic programming	Soundness (uniform derivations)	0.31 sec
	Canonical forms (uniform derivations)	0.34 sec
	Completeness (uniform derivations)	0.28 sec
	Soundness (resolution)	1.05 sec
	Completeness (resolution)	0.52 sec
Intuitionistic logic	Deduction theorem	0.11 sec
(Hilbert calculus)	Embedding into natural deduction calculus	0.33 sec
Intuitionistic logic	Natural deduction $\rightarrow$ Sequent calculus	0.11 sec
(implicational fragment)	Sequent calculus $\rightarrow$ Natural deduction	0.12 sec
Cartesian closed categories	Embedding into simply typed λ-calculus	3.39 sec
	Distributivity lemma	no yet
Kolmogorov embedding	Classical logic $\rightarrow$ Intuitionistic logic	9.55 sec
	Intuitionistic logic $\rightarrow$ Classical logic	not yet

Figure 8.11: Experimental results (in CPU seconds)

types are preserved during evaluation, or *the natural meaning of an expression coincides with the one ascribed by a reduction semantics*, or *typing is unique*.

Each of these properties can be verified in less than a second, which makes Twelf an efficient rapid prototyping tool.

Mini-ML's natural semantics is defined by relating the expression to be evaluated and the result of the evaluation. But there are other semantics; we have considered for example another semantics that is defined in terms of execution traces of a compiled expression on an abstract CPM machine [FSDF93]. As a matter of fact, we have used Twelf to verify one direction of the equivalence proofs between the natural and the trace-based semantics. The soundness direction of the proof states, that once the abstract machine has computed a result, it coincides with the natural semantics. The proof of the soundness property requires complete induction, a technique that Twelf does not support in the current version. The completeness direction on the other hand states that each value computed by the abstract machine (upon input of a compiled expression) corresponds to the natural meaning of the expression. This property is very tedious to derive by hand, and Twelf does it in 1.13 sec.

In the same experiment we have used to Twelf to show that every soundness proof for concrete expressions can be transformed into a completeness proof and vice versa. This is a meta-meta result about a relational encoding of the soundness and the completeness proofs as relations in LF.

The third experiment lies in the area of logic programming in the fragment of hereditary Harrop formulas. We have used Twelf to show that the search for uniform derivations and resolution are equivalent.

And finally, there are several small experiments. We could for example show that the Hilbert calculus for intuitionistic logic can be embedded into the natural deduction calculus, and so can the sequent calculus. The reverse also holds, at least for embedding the natural deduction calculus into the sequent calculus. Twelf's underlying LF theorem prover is not efficient enough to prove that any natural deduction derivation can be embedded into the Hilbert calculus.

We have used Twelf to show that Cartesian Closed Categories can be embedded into the simply-typed λ-calculus; objects are interpreted as terms, and morphisms as functions. The distributivity law of (a pair of two morphisms composed with another morphism) could not be proven in Twelf, because the underlying LF theorem prover is not efficient enough, but preliminary experiments with other theorem provers such as SPASS [Wei97] have shown that this is really a hard problem.

The LF theorem prover is also the problem in the proof of the Kolmogorov embedding. Twelf easily proves that classical logic can be embedded into intuitionistic logic via the double negation transformation, but for many cases of the reverse direction, the search space is intractable, and Twelf is unable to find the proof.

8.6 Summary

The Twelf system is a meta-logical framework that is designed to represent deductive systems, and to automate reasoning about them. Its design is two layered. The logical framework LF serves as representation language *for* deductive systems, and the meta-logic $\mathcal{M}_2^+$ serves as a specification language *about* their properties.

In this chapter we have presented the Twelf system with special emphasis on its meta-theorem prover component. The meta-theorem prover uses a sophisticated proof search algorithm to construct proof terms in $\mathcal{M}_2^+$. One novel concept that distinguishes Twelf's meta-theorem prover from others is the ability to reason by induction over higher-order encodings using the regular world assumption. In Twelf inductive definitions are open-ended, they can be dynamically extended, as long as they follow certain a priori specified formation rules, which we have dubbed context schemas. Most other inductive theorem provers however are based on the closed world assumption and employ standard induction principles for reasoning by induction, which disallow higher-order encodings in general since they typically violate the positivity condition associated with standard inductive definitions.

Because of higher-order representation techniques, proofs *about* formal systems enjoy brief and concise formalizations in $\mathcal{M}_2^+$, and Twelf's special purpose meta-theorem prover takes full advantage of their form during search. As case study, we have demonstrated in this chapter how to use Twelf to prove all lemmas in connection to the Church-Rosser theorem from Chapter 2 and Chapter 3. In the special domain of higher-order encodings, Twelf is an ideal rapid prototyping tool for the design of deductive systems and the study of their properties.

Chapter 9

Conclusion

The development of formal systems, such as logics, programming languages and type systems is a task so complex that it benefits greatly from tools that support their design, experimentation, and their verification. To be usable, these tools must allow a formal encoding of the system that is as close as possible to its natural form — only then users are likely to overcome their reservations towards formalization. In addition, the language provided by the tool to express logics, programming languages, and type systems must be as simple and intuitive as possible; otherwise the tool remains accessible to specialists only.

The logical framework LF is an elegant meta-language for the representation of formal systems. It supports higher-order representation techniques, which allow for elegant and natural encodings of inference rules including side conditions, such as for example, freshness conditions for variables and parameters. A user who uses a tool based on LF can employ the context of the logical framework to encode contexts of some object languages given that they share the same properties. By its very definition LF contexts are subject to weakening, contraction, and exchange — the same properties assumptions lists of many logic calculi and typing contexts of many programming languages enjoy.

Thus, LF is a powerful framework to represent formal systems such as logics and programming languages adequately. On the other hand, LF is a type theory, and not a logic per se. It is not designed as a meta-language to represent proofs of correctness, safety, soundness, or completeness conditions, or any other properties a formal system may satisfy. Many of those proofs are inductive; for example, the proof of the diamond lemma requires induction over the reduction derivations and the type preservation proof of a functional programming languages proceeds by induction on the evaluation derivation.

The problem is that for higher-order encodings of formal systems in a logical framework typically standard induction principles do not exist. The closed world assumption that underlies standard induction principles stipulates a positivity condition on inductive definitions — the type defined must only occur in positive positions of its constructor types — and in general, higher-order encodings violate exactly this condition. In fact, the closed world assumption is too restrictive for inductive definition of higher-encodings because inductive arguments are allowed to traverse λ-binders and thus, inductive definitions are open-ended.

In this sense, higher-order representation techniques and inductive reasoning are incompatible. Proof assistant systems such as Isabelle, Coq, and PVS, are based on the closed world assumption and therefore they allow only higher-order encodings, that are compatible with the the positivity condition. However, most of the interesting higher-order encodings we are concerned with, do not satisfy the positivity condition.

In this thesis on the other hand we present an alternative solution: Instead of massaging our representations in such a way that they satisfy the positivity condition, we allow them to be higher-order in the most general sense. One of the contributions of this thesis is, that even though they are not inductive in the standard sense under the closed world assumption, they can be seen as inductive definitions under the regular world assumption. Under the regular world assumption, inductive definitions are open ended, they are permitted to be extended in a regular way when traversing λ-binders.

In our design, regularly formed world extensions possess the same properties as LF contexts, in particular, contraction, weakening, and exchange. Since it is not at all clear which form an induction principle under the regular world assumption should have, this thesis proposes an alternative. We have designed the meta-logic $\mathcal{M}_2^+$ of recursive functions that are defined by cases, and which range over LF objects. In this meta-logic, inductive proofs over higher-order encodings are realized by a total functions.

The main characteristics of our design is that the meta-logic and the logical framework LF are conceptually defined on two different levels. The meta-logic $\mathcal{M}_2^+$ provides a notion of a recursive function to formalize inductive arguments, whereas LF provides a notion of parametric functions, that is used exclusively for the purpose of representation. We have shown that the design of $\mathcal{M}_2^+$ is sound. Thus, $\mathcal{M}_2^+$ is a meta-logical framework based on realizability.

In this thesis we have also developed automated deduction procedures; one that conducts proof search for LF objects of a given LF type. The other searches for recursive functions, which formalize proofs, in the meta-logic $\mathcal{M}_2^+$. Both procedures are implemented in the Twelf system which is publicly available from the Twelf homepage at `http://www.twelf.org`. One is called LF theorem prover, and the other meta-theorem prover. The meta-theorem prover uses the LF theorem during proof search.

Because of the immediacy and the elegance of higher-order encodings of formal systems and because of the direct formalization of meta-theoretic arguments, Twelf's meta-theorem prover outperforms any other theorem prover in this special domain. Twelf has been successfully employed to derive various properties of logics and type systems, such as the consistency of logics, the admissibility of new inference rules, and equivalence of different logic calculi. Other results include automatic proofs of the Church-Rosser theorem, cut-elimination, and type preservation and progress of various operational semantics.

9.1 Future Work

The future research that will follow this thesis is manifold. The overall goal of this research is to devise tools that support the design, the experimentation and the verification of formal systems, such as logics, programming languages, type systems; but the research program does not stop there. Instead, as a next step, we would like to scale this research to engineer real usable tools for security and network protocols designer, for authentication protocol designers, for programming language designers, for system engineers, and possibly even for software engineers. We foresee several possible developments along these lines as described in this section.

First we give an overview over possible application domains for this research in Section 9.1.1. But how good are higher-order encodings for these applications? Are the standard properties associated with LF contexts enough to guarantee adequate and elegant encodings of the formal systems in question? It is very likely, that different applications pose different requirements on the underlying logical framework which we discuss in Section 9.1.2. Another line of future research emerges from the question of how to extend the meta-logic $\mathcal{M}_2^+$ to facilitate the formulation and automatic reasoning about other applications. We discuss possible extensions of the meta-logic in Section 9.1.3. In order for Twelf to be a design and experimentation tool, the prototype implementation of the meta-theorem prover must mature. Possible improvements to the implementation are described in Section 9.1.4. Another direction of future work results from interpreting the recursive functions of $\mathcal{M}_2^+$ as programs of a real programming language which is to be developed. An account of possible research directions is given in Section 9.1.5.

9.1.1 Applications of $\mathcal{M}_2^+$

Twelf owes its tremendous performance in all our experiments partly to design of the meta-logic $\mathcal{M}_2^+$, partly to the representational power of the logical framework LF, but also partly to the cleanliness of the formal systems in question. However, when designing real world programming languages and safe real world systems, there might not be an elegant and direct encoding in the logical framework. Twelf, for example, can show type preservation of the execution of purely functional programs, but it is still an open question, if and how references and exceptions can be added to the encoding in a direct way. Therefore in future work we have to understand what requirements real world systems pose on meta-languages such as LF and we propose to achieve this is by conducting case studies in the area of programming language design, protocol design, and software engineering.

Safe programming languages. In recent years, several techniques have been developed to increase the users confidence in the safety of executable code. The idea of proof carrying code for example [Nec98] suggests to modify compilers to emit not only compiled code but also corresponding safety proofs that a code consumer can use to check a priori safety properties. Typed assembly language is a special instance of this design. Safety proofs are encoded by type information in TAL [MWCG99] following the idea that well-typed programs are safe to be executed. Similarly, more mainstream, Java bytecode [LY96] is subject to verification by a byte code verifier that implements a particular safety policy.

All three ideas are based on the common idea that code should not be executed without checking that it is safe to do so. In each system, safety checking reduces to proof checking, type checking, or bytecode checking, respectively. But note, that all three designs are extremely vulnerable to design mistakes — a logic in which safety proofs are expressed must be consistent (if falsehood is derivable, than any property is derivable), a type system for assembly language must be sound, and so must be the notion of safety attributed to Java bytecode. One possible research direction is to make Twelf a useful development and verification tool. Future experiments in this area will shed some light on the limitations and possible extensions of the Twelf system.

Protocol design. The common practice in the design of network and authentication protocols is *not* to use any formal tools. Important properties are verified only after a design is completed and implemented. Protocols can be modeled in proof assistants such as PVS [ORS92], model checkers, such as SMV [CGL94] and they are examined for different properties, such as liveness, and in the case of authentication protocols for freeness of attacks [MCJ97].

Twelf has not been applied to protocol design yet, but it would be a very instructive experience to do so. We suspect that by using Twelf as a development tool, the design of protocols can be made more secure since a priori specified safety and security properties can be verified and checked throughout the design process. Therefore, design mistakes can be caught early. After a successful design, we foresee Twelf to output the verified code (in a compilable language) that implements protocol stacks, or client/server architectures for authentication systems.

As for the formal development of security protocols, experiments in this domain may reveal shortcomings in the design of Twelf that can give indications for future research.

Software engineering. The functionality of a software module is typically defined through an interface that contains formal descriptions of the computational behavior of functions and procedures provided by the module. The languages used to describe this kind of functionality are typically logics or type theories; the challenge is to design them in such a way that they can capture invariants, while preserving soundness. Twelf is a tool that can help developing these kind of languages.

9.1.2 Adaptation of $\mathcal{M}_2^+$

It is likely that the experiments with real-world systems suggest possible extensions of Twelf such as extensions to the underlying logical framework LF and also extensions to the meta-logic $\mathcal{M}_2^+$. As presented in Section 2.3, LF is a logical framework, which satisfies the requirements for adequate representations of a formal systems such as logics and programming languages. But there are many important extensions of LF, some of them characterized in Barendregt's λ-cube, and other substructural logical frameworks that may be of practical interest.

Polymorphic logical framework. Even though not discussed in this thesis, one can imagine an extension of this work to other logical frameworks. For example, adding polymorphism to LF while preserving canonical forms may be possible but it is certainly challenging, and it is even more challenging to extend the meta-logic $\mathcal{M}_2^+$ discussed in Chapter 5 accordingly. We leave this research to future work together with an extension of Twelf by type constructors.

Linear logical framework. The linear logical framework (LLF) is a substructural logical framework. It is a conservative extension of LF and goes back to work by Cervesato and Pfenning [CP97a]. LLF extends LF by a resource-oriented assumption concept, inspired by linear logic [Gir87]. Linear assumptions are organized in linear contexts which obey only one of the standard structural rules: exchange. Weakening and contraction cannot apply to linear contexts. This gives linear assumptions the flavor of resources: Assumptions can neither disappear nor be duplicated. The advantage of a linear logical framework is, that it allows a concise modeling of resource oriented problems such as for example, the theory of functional programming languages with references. Binding of a value to a reference cell is represented as a resource, and because of properties of the linear logical framework update of reference cells can be modeled directly [CP96].

Ordered logical framework. The ordered logical framework derives from the linear logical framework by dropping the last remaining structural rule: exchange. First case studies by Pfenning and Polakow [PP99] have shown that ordered linear logic is beneficial for the representation of aggregate constructs such as stacks. This framework inherits all properties form the logical framework, and in addition, assumptions can only be consumed *in same order* they have been assumed. Again, in the area of functional programming languages, there are several examples of languages which can be very elegantly represented in an ordered linear framework [DP95].

9.1.3 Extensions of $\mathcal{M}_2^+$

Quantifier Alternations. In the current development, Twelf accepts only Π_2-formulas, i.e. formulas that start with a block of universal followed by a block of existential quantifiers. However, many examples lie outside this fragment. We leave an investigation of this issue to future research.

Adding new logical connectives. From a logical point of view, $\mathcal{M}_2^+$ is relatively impoverished. Not only that it defines only few connectives, but it neither provides nor allows the user to define new predicates. In particular, $\mathcal{M}_2^+$ is missing other logical connectives, such as for example disjunction, implication, and negation; it is also missing mechanisms to express equality of derivations and subterm relations. In many cases, if needed connectives and predicates can be encoded in LF; but in future versions, it might be sensible to extend the meta-logic directly.

Equality is a good candidate for a built-in predicate into $\mathcal{M}_2^+$. It allows the formulation of theorems that express the *unique* existence of a derivation. The drawback of adding equality to the meta-logic is, that the theorem proving aspects will get harder. Unification problems must now be considered modulo equational theories [Sny91]. A different research direction is to investigate how $\mathcal{M}_2^+$ can be extended by new unique existential quantifier $\exists^1$.

Context schema subsumption. In Section 6.3 we have introduced a very simple and direct definition of context schema subsumption. For larger developments it may be important to relax the subsumption condition, for example, by extending context blocks by unrelated parameter declarations. How exactly a refined subsumption criterion could look is an important design question; in addition, it interacts with other design choices such as the design of the modules system or the scope of regular context extensions. These are important questions and should be addressed in future research.

9.1.4 Implementation of $\mathcal{M}_2^+$

Despite its already impressive deductive power, the implementation of the meta-theorem prover in the Twelf system is currently only a prototype. No sophisticated optimizations have been applied so far, and the implementation is incomplete with respect to the theory which has been described in this thesis. For example, many of the techniques developed for traditional inductive theorem provers seem applicable in our setting, but none has actually been adjusted or implemented.

Termination orderings. The prototype implementation of the meta-theorem prover is restricted to proofs by structural induction. The various termination orders defining the proofs of the lemma in this thesis, for example, syntactically restrict the form of the induction hypotheses. In particular, termination orders are lexicographical and simultaneous extensions of the subterm ordering which are expressive enough for many proofs, but not necessarily all. The soundness proof of compiling Mini-ML to the CPM machine, for example, requires as proof principle proof by complete induction. In future research we enrich the notion of termination order by derived reduction information as already implemented in the termination checker for Twelf [PP00].

Integration. Currently, a successful application of Twelf's meta-theorem prover depends crucially on the appropriate choice of the various bounds for filling, splitting, and recursion, and the heuristic that selects the first universally quantified variable to split on. During runtime, a splitting operation is executed upon the failure of the preceding filling operation. Therefore, filling slows the meta-theorem prover down. One possibility to improve the theorem provers performance is to consider filling and splitting operations *simultaneously*. It is left to future research to integrate the different operations of the meta-theorem prover.

Proof Planning. Proof planning was introduced by Alan Bundy et al. [BvHHS91] for inductive theorem proving by a special search heuristic called rippling. This heuristic works particularly well for equational arguments used in proofs by mathematical induction. The question of how rippling scales to the setting of non-standard induction techniques opens a new area of research.

Failure treatment. A very important area of research is the treatment of proof failure. The theorem prover must supply the user with appropriate messages pointing to the problem of design mistakes in the case of failure. In the current prototype implementation, the prover is too eager to continue; it will continue to apply splitting operations that do not advance the search for a proof. How can the prover distinguish between promising an non-promising splitting operation? How can it return information to the user such as, for example, that a particular inference rule renders a logic design unsound, or that a proof does not go through because the world extension was assumed to be closed? If meaning could be assigned to failure, intelligent error messages could be generated and system design cycles would shrink tremendously.

Optimizations. Among the many restrictions and prototype features of the Twelf system, there is one that is particularly important; many decisions about which operations to apply next depend on the filling operations. Most of the time spend by the theorem prover in Figure 8.11, for example, is due to filling.

Currently not employing any kind of advanced implementation techniques, the LF theorem prover uses straightforward, depth first, iterative deepening search that is limited only by a filling bound. We believe that the efficiency of the theorem prover could be tremendously improved by other techniques such as the inverse method [DMTV99], the tableaux method [Häh99], in connection with special indexing techniques [RSV99].

Proof translation. Trusting a proof means to verify it. One of the shortcomings of the current prototype implementation of Twelf is that it does not provide an independent meta-proof checker. Even though we hope that it is small, and verifiable correct, its design is significantly more complicated than that of the standard LF type checker because it relies on a correct implementation of pattern unification for coverage analysis. As alternative, in another line of research we want to investigate how to convert higher-order encodings and their meta-theory into the language of standard inductive definitions, interpretable and verifiable by trusted and well-understood theorem provers, such as Coq [DFH+93], Lego [LP92], Gandalf[Tam97], Spass [Wei97], TPS [AINP90] and others.

Tactics. Independent experiments with the meta-theorem prover have shown that its current strategy is not powerful enough to reach satisfactory results in certain application areas. The main drawback of the implementation is that it has a fixed heuristic which selects the assumption the system will splits next. In addition, the meta-theorem prover does not implement back tracking. On the contrary, whenever an operation is applied, Twelf commits to it once and for all.

Twelf's built in heuristic is unsatisfactory because it is programmed in such a way that it never splits assumptions that occur in the type of another. These assumptions are called *index assumptions.* In most of our experiments, this design decision drastically cuts down the size of the search space, but unfortunately, in other situations a successful proof relies on the ability to split index assumptions.

Therefore, another very challenging research direction is the design of good heuristics, better search strategies, and user-defined tactics to guide proof search and the selection of assumption to be split.

9.1.5 Functional Programming in $\mathcal{M}_2^+$

The proof term calculus of $\mathcal{M}_2^+$ bears the basic elements of a programming language, such as a notion of a recursive function, application, and definition by cases. Datatypes are expressed in the logical framework LF in form of LF signatures. By omitting side condition (5.1) that ensures termination, and side condition (5.2) that ensures coverage, we obtain a simple functional programming language whose functions range over LF-objects. In future research we will investigate how to turn $\mathcal{M}_2^+$ into a programming language by adding references, exceptions and a module system. This research extends into the areas of compiler and garbage collector design.

9.2 Summary

The contributions of this thesis are manifold. We have presented a meta-logic $\mathcal{M}_2^+$ whose quantifiers range over LF objects. The meta-logic is designed to formalize inductive arguments about higher-order encodings of formal systems in LF. Therefore, one of the main contributions of this thesis is a solution to the problem of how to bring together inductive reasoning and higher-order representation techniques.

In several experiments we have shown that $\mathcal{M}_2^+$ is expressive enough to formalize proofs of many important properties about logics, programming languages, and type systems. Those formalizations are so elegant, that they can be automatically constructed by the meta-theorem prover that is implemented, as a prototype, in the Twelf system.

Twelf has been used to develop and prove several fundamental results of computer science and logic with a high degree of automation. Among the examples, are the Church-Rosser property of the simply typed λ-calculus, which we have discussed in this thesis in depth, and the cut-elimination theorem for intuitionistic first-order logic.

Appendix A

Inference rules

A.1 Meta-Logic $\mathcal{M}_2^+$

Judgments:

Provability of general formulas:	$\Xi \vdash_\Sigma Q \in G$
Provability of formulas:	$\Psi; \Delta; \Xi \vdash_{\Sigma;S} P \in F$
Inference:	$\Psi; \Delta; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta'$

Rules

$$\frac{\cdot; \cdot; \Xi \vdash_{\Sigma;S} P \in F}{\Xi \vdash_\Sigma \mathbf{box}\, S.\, P \in \Box S.\, F}\,\mathsf{generalR}$$

$$(\prec_S) \subset (\prec_\Sigma) \tag{6.1}$$

$$\frac{\mathbf{x} \in G \in \Xi}{\Xi \vdash_\Sigma \mathbf{x} \in G}\,\mathsf{mhyp}$$

$$\frac{(\mathbf{x} \in F) \in \Delta}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \mathbf{x} \in F}\,\mathsf{axvar} \qquad \frac{}{\Psi; \Delta; \Xi \vdash \langle\rangle \in \top}\,\mathsf{R}\top$$

$$\frac{\Psi, x : A; \Delta; \Xi \vdash_{\Sigma;S} P \in F}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \Lambda x : A.\, P \in \forall x : A.\, F}\,\mathsf{R}\forall \qquad \frac{\Psi, \rho^L; \Delta; \Xi \vdash_{\Sigma;S} P \in F}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \lambda\rho^L.\, P \in \mathbf{\Pi}\rho^L.\, F}\,\mathsf{R\Pi}$$

$$\frac{\Psi \vdash_{\Sigma;S} M : A \qquad \Psi; \Delta; \Xi \vdash_{\Sigma;S} P \in F[M/x]}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \langle M, P\rangle \in \exists x : A.\, F}\,\mathsf{R}\exists \qquad \frac{\Psi; \Delta; \Xi \vdash_{\Sigma;S} P_1 \in F_1 \qquad \Psi; \Delta; \Xi \vdash_{\Sigma;S} P_2 \in F_2}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \langle P_1, P_2\rangle \in F_1 \wedge F_2}\,\mathsf{R}\wedge$$

$$\frac{\Psi; \Delta; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta' \qquad \Psi, \Psi'; \Delta, \Delta'; \Xi \vdash_{\Sigma;S} P \in F}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \mathbf{let}\ D\ \mathbf{in}\ P \in F}\,\mathsf{sel}$$

$$\frac{}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \cdot \in \cdot ; \cdot}\,\mathsf{Ldone}$$

$$\frac{S(L) = \text{SOME } C_1.\, \text{BLOCK } C_2 \quad \Psi \vdash_{\Sigma;S} \sigma : C_1 \quad \Psi \vdash_{\Sigma;S} \rho \equiv [\sigma]C_2 \quad \Psi, \rho^L; \Delta; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta'}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \nu\, \rho^L.\, D \in \Pi\rho^L.\, (\Psi'; \Delta')}\,\mathsf{Lnew}$$

$$\frac{\Psi; \Delta; \Xi \vdash_{\Sigma;S} P \in \forall x : A.\, F \quad \Psi \vdash_{\Sigma;S} M : A \quad \Psi; \Delta, \mathbf{y} \in F[M/x]; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta'}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \mathbf{y} \in F[M/x] = P\ M, D \in \Psi'; \mathbf{y} \in F[M/x], \Delta'}\,\mathsf{L}\forall$$

$$\frac{\Psi; \Delta; \Xi \vdash_{\Sigma;S} P \in \mathbf{\Pi}\rho^L.\, F \quad \rho'^L \in \Psi \quad \Psi \vdash_{\Sigma;S} \rho' \equiv \rho \quad \Psi; \Delta, \mathbf{y} \in F[\rho'/\rho]; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta'}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \mathbf{y} \in F[\rho'/\rho] = P\ \rho', D \in \Psi'; \mathbf{y} \in F[\rho'/\rho], \Delta'}\,\mathsf{L}\Pi$$

$$\frac{\Xi \vdash_{\Sigma} Q \in \Box S'.\, F \quad \Psi; \Delta, \mathbf{y} \in F; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta'}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \mathbf{y} \in F = \text{lemma } Q, D \in \Psi'; \mathbf{y} \in F, \Delta'}\,\mathsf{L}\Xi$$

$$S' \text{ subsumes } S \tag{5.4}$$

$$\frac{\Psi; \Delta; \Xi \vdash_{\Sigma;S} P \in \exists x : A.\, F \quad \Psi, x : A; \Delta, \mathbf{y} \in F; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta'}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \langle x : A, \mathbf{y} \in F \rangle = P, D \in x : A, \Psi'; \mathbf{y} \in F, \Delta'; \Xi}\,\mathsf{L}\exists$$

$$\frac{\Psi; \Delta; \Xi \vdash_{\Sigma;S} P \in F_1 \wedge F_2 \quad \Psi; \Delta, \mathbf{x} \in F_1; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta'}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \mathbf{x} \in F_1 = \pi_1\ P, D \in \Psi'; \mathbf{x} \in F_1, \Delta'}\,\mathsf{L}\wedge_1$$

$$\frac{\Psi; \Delta; \Xi \vdash_{\Sigma;S} P \in F_1 \wedge F_2 \quad \Psi; \Delta, \mathbf{x} \in F_2; \Xi \vdash_{\Sigma;S} D \in \Psi'; \Delta'}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \mathbf{x} \in F_2 = \pi_2\ P, D \in \Psi'; \mathbf{x} \in F_2, \Delta'}\,\mathsf{L}\wedge_2$$

$$\frac{\Psi; \Delta, \mathbf{x} \in F; \Xi \vdash_{\Sigma;S} P \in F}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \mu\mathbf{x} \in F.\, P \in F}\,\mathsf{Rctx}$$

$$P \text{ terminates in } \mathbf{x} \tag{5.1}$$

$$\frac{\Psi; \Delta \vdash_{\Sigma;S} \psi; \delta \in \Psi'; \Delta' \quad \Psi'; \Delta'; \Xi \vdash_{\Sigma;S} \Omega \in F}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \text{case } (\psi; \delta) \text{ of } \Omega \in F[\psi]}\,\mathsf{case}$$

$$\Omega \text{ is a complete case cover} \tag{5.2}$$

$$\frac{}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \cdot \in F}\,\mathsf{base}$$

$$\frac{\Psi' \vdash_{\Sigma;S} \psi \in \Psi \quad \Psi; \Delta; \Xi \vdash_{\Sigma;S} \Omega \in F \quad \Psi'; [\psi]\Delta; \Xi \vdash_{\Sigma;S} P \in F[\psi]}{\Psi; \Delta; \Xi \vdash_{\Sigma;S} \Omega, (\Psi' \triangleright \psi \mapsto P) \in F}\,\mathsf{alt}$$

$$\Psi' \text{ is minimal} \tag{5.3}$$

A.2 Operational Big-Step Semantics

Judgments

Evaluation	$\Phi \vdash P \hookrightarrow V$
Assumption	$\Phi \vdash D \hookrightarrow \eta; \delta$
Selection	$\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V$

Rules

$$\frac{}{\Phi \vdash \Lambda x : A.\, P \hookrightarrow \Lambda x : A.\, P}\ \textsf{ev\_Lam} \qquad \frac{}{\Phi \vdash \lambda\rho^L.\, P \hookrightarrow \lambda\rho^L.\, P}\ \textsf{ev\_lam}$$

$$\frac{\Phi \vdash P \hookrightarrow V}{\Phi \vdash \langle M, P\rangle \hookrightarrow \langle M, V\rangle}\ \textsf{ev\_inx} \qquad \frac{\Phi \vdash P_1 \hookrightarrow V_1 \quad \Phi \vdash P_2 \hookrightarrow V_2}{\Phi \vdash \langle P_1, P_2\rangle \hookrightarrow \langle V_1, V_2\rangle}\ \textsf{ev\_pair} \qquad \frac{}{\Phi \vdash \langle\rangle \hookrightarrow \langle\rangle}\ \textsf{ev\_unit}$$

$$\frac{\Phi \vdash D \hookrightarrow \psi; \delta \quad \Phi \vdash P[\mathrm{id}_\Phi, \psi; \delta] \hookrightarrow V}{\Phi \vdash \textbf{let}\ D\ \textbf{in}\ P \hookrightarrow V}\ \textsf{ev\_let}$$

$$\frac{\Phi \vdash P[\mu\mathbf{x} \in F.\, P/\mathbf{x}] \hookrightarrow V}{\Phi \vdash \mu\mathbf{x} \in F.\, P \hookrightarrow V}\ \textsf{ev\_rec} \qquad \frac{\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V}{\Phi \vdash \mathrm{case}\ (\psi; \delta)\ \mathrm{of}\ \Omega \hookrightarrow V}\ \textsf{ev\_case}$$

$$\frac{}{\Phi \vdash \cdot \hookrightarrow \cdot; \cdot}\ \textsf{ev\_empty}$$

$$\frac{\Phi \vdash P \hookrightarrow \langle M, V\rangle \quad \Phi \vdash D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}] \hookrightarrow \psi'; \delta'}{\Phi \vdash \langle x : A, \mathbf{y} \in F\rangle = P, D \hookrightarrow M/x, \psi'; V/\mathbf{y}, \delta'}\ \textsf{ev\_split}$$

$$\frac{\Phi \vdash P \hookrightarrow \Lambda x : A.\, P' \quad \Phi \vdash P'[\mathrm{id}_\Phi, M/x] \hookrightarrow V \quad \Phi \vdash D[V/\mathbf{y}] \hookrightarrow \psi'; \delta'}{\Phi \vdash \mathbf{y} \in F = P\ M, D \hookrightarrow \psi'; V/\mathbf{y}, \delta'}\ \textsf{ev\_App}$$

$$\frac{\Phi \vdash P \hookrightarrow \lambda\rho'^L.\, P' \quad \Phi \vdash P'[\mathrm{id}_\Phi, \rho/\rho'] \hookrightarrow V \quad \Phi \vdash D[V/\mathbf{y}] \hookrightarrow \psi'; \delta'}{\Phi \vdash \mathbf{y} \in F = P\ \rho, D \hookrightarrow \psi'; V/\mathbf{y}, \delta'}\ \textsf{ev\_app}$$

$$\frac{\Phi, \rho^L \vdash D \hookrightarrow \psi'; \delta'}{\Phi \vdash \nu\ \rho^L.\, D \hookrightarrow (\lambda\rho^L.\, (\psi'; \delta'))}\ \textsf{ev\_new}$$

$$\frac{\Phi \vdash P \hookrightarrow \langle V_1, V_2\rangle \quad \Phi \vdash D[V_1/\mathbf{x}] \hookrightarrow \psi'; \delta'}{\Phi \vdash \mathbf{x} \in F_1 = \pi_1\ P, D \hookrightarrow \psi'; V_1/\mathbf{x}, \delta'}\ \textsf{ev\_fst}$$

$$\frac{\Phi \vdash P \hookrightarrow \langle V_1, V_2 \rangle \qquad \Phi \vdash D[V_2/\mathbf{x}] \hookrightarrow \psi'; \delta'}{\Phi \vdash \mathbf{x} \in F_2 = \pi_2\ P, D \hookrightarrow \psi'; V_2/\mathbf{x}, \delta'}\ \mathsf{ev\_snd}$$

$$\frac{\Phi \vdash P[\psi''; \delta] \hookrightarrow V}{\Phi \vdash (\psi; \delta) \sim (\Omega, (\Psi \triangleright \psi' \mapsto P)) \hookrightarrow V}\ \mathsf{ev\_yes}$$

if there exists a ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$

$$\frac{\Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V}{\Phi \vdash (\psi; \delta) \sim (\Omega, (\Psi \triangleright \psi' \mapsto P)) \hookrightarrow V}\ \mathsf{ev\_no}$$

A.3 Operational Small-Step Semantics

Judgment

One-step reduction	$\Phi \vdash S_1 \Longrightarrow S_2$
Multi-step reduction	$\Phi \vdash S_1 \overset{*}{\Longrightarrow} S_2$

Rules

trlet	$:: \Phi; C \triangleright \mathbf{let}\ D\ \mathbf{in}\ P \implies \Phi; C, \mathbf{let} \bullet \mathbf{in}\ P \triangleright D$
trletC	$:: \Phi; C, \mathbf{let} \bullet \mathbf{in}\ P \triangleright (\psi; \delta) \implies \Phi; C \triangleright P[\mathrm{id}_\Phi, \psi; \delta]$
trpair	$:: \Phi; C \triangleright \langle P_1, P_2\rangle \implies \Phi; C, \langle \bullet, P_2\rangle \triangleright P_1$
trpairC	$:: \Phi; C, \langle \bullet, P_2\rangle \triangleright V \implies \Phi; C \triangleright \langle V, P_2\rangle$
trmix	$:: \Phi; C \triangleright \langle V_1, P_2\rangle \implies \Phi; C, \langle V_1, \bullet\rangle \triangleright P_2$
trmixC	$:: \Phi; C, \langle V_1, \bullet\rangle \triangleright V \implies \Phi; C \triangleright \langle V_1, V\rangle$
trfst	$:: \Phi; C \triangleright \mathbf{x} \in F = \pi_1\ P, D \implies \Phi; C, \mathbf{x} \in F = \pi_1\ \bullet, D \triangleright P$
trfstC	$:: \Phi; C, \mathbf{x} \in F = \pi_1\ \bullet, D \triangleright \langle V_1, V_2\rangle \implies \Phi; C, (\bullet; V_1/\mathbf{x}, \bullet) \triangleright D[V_1/\mathbf{x}]$
trsnd	$:: \Phi; C \triangleright \mathbf{x} \in F = \pi_2\ P, D \implies \Phi; C, \mathbf{x} \in F = \pi_2\ \bullet, D \triangleright P$
trsndC	$:: \Phi; C, \mathbf{x} \in F = \pi_2\ \bullet, D \triangleright \langle V_1, V_2\rangle \implies \Phi; C, (\bullet; V_2/\mathbf{x}, \bullet) \triangleright D[V_2/\mathbf{x}]$
trinx	$:: \Phi; C \triangleright \langle M, P\rangle \implies \Phi; C, \langle M, \bullet\rangle \triangleright P$
trinxC	$:: \Phi; C, \langle M, \bullet\rangle \triangleright V \implies \Phi; C \triangleright \langle M, V\rangle$
trsplit	$:: \Phi; C \triangleright \langle x : A, \mathbf{y} \in F\rangle = P, D \implies \Phi; C, \langle x : A, \mathbf{y} \in F\rangle = \bullet, D \triangleright P$
trsplitC	$:: \Phi; C, \langle x : A, \mathbf{y} \in F\rangle = \bullet, D \triangleright \langle M, V\rangle \implies \Phi; C, (M/x, \bullet; V/\mathbf{y}, \bullet) \triangleright D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}]$
trsubst	$:: \Phi; C, (M/x, \bullet; V/\mathbf{y}, \bullet) \triangleright (\psi; \delta) \implies \Phi; C \triangleright (M/x, \psi; V/\mathbf{y}, \delta)$
trrec	$:: \Phi; C \triangleright \mu\mathbf{x} \in F. P \implies \Phi; C \triangleright P[\mu\mathbf{x} \in F. P/\mathbf{x}]$
trempty	$:: \Phi; C \triangleright \cdot \implies \Phi; C \triangleright \cdot; \cdot$
trApp	$:: \Phi; C \triangleright \mathbf{x} \in F = P\ M, D \implies \Phi; C, \mathbf{x} \in F = \bullet\ M, D \triangleright P$
trAppC	$:: \Phi; C, \mathbf{x} \in F = \bullet\ M, D \triangleright \Lambda x : A. P \implies \Phi; C, \mathbf{x} \in F = \bullet, D \triangleright P[\mathrm{id}_\Phi, M/x]$
trapp	$:: \Phi; C \triangleright \mathbf{x} \in F = P\ \rho, D \implies \Phi; C, \mathbf{x} \in F = \bullet\ \rho, D \triangleright P$
trappC	$:: \Phi; C, \mathbf{x} \in F = \bullet\ \rho', D \triangleright \lambda\rho^L. P \implies \Phi; C, \mathbf{x} \in F = \bullet, D \triangleright P[\mathrm{id}_\Phi, \rho'/\rho]$
trassign	$:: \Phi; C, \mathbf{x} \in F = \bullet, D \triangleright V \implies \Phi; C, (\bullet; V/\mathbf{x}, \bullet) \triangleright D[V/\mathbf{x}]$
trmeta	$:: \Phi; C, (\bullet; V/\mathbf{x}, \bullet) \triangleright (\psi; \delta) \implies \Phi; C \triangleright (\psi; V/\mathbf{x}, \delta)$
trnew	$:: \Phi; C \triangleright \nu\ \rho^L. D \implies \Phi, \rho^L; C, (\lambda\rho^L. (\bullet; \bullet)) \triangleright D$
trnewC	$:: \Phi, \rho^L; C, (\lambda\rho^L. (\bullet; \bullet)) \triangleright \psi; \delta \implies \Phi; C \triangleright \lambda\rho^L. (\psi; \delta)$
trcase	$:: \Phi; C \triangleright \mathrm{case}\ (\psi; \delta)\ \mathrm{of}\ \Omega \implies \Phi; C \triangleright (\psi; \delta) \sim \Omega$
tryes	$:: \Phi; C \triangleright (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P)) \implies \Phi; C \triangleright P[\psi''; \delta]$ if there exists a ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$
trno	$:: \Phi; C \triangleright (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P)) \implies \Phi; C \triangleright (\psi; \delta) \sim \Omega$ if there is no ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$

$$\frac{}{S \overset{*}{\implies} S}\ \mathsf{trid}$$

$$\frac{S_1 \implies S_2 \qquad S_2 \overset{*}{\implies} S_3}{S_1 \overset{*}{\implies} S_3}\ \mathsf{trstep}$$

A.4 Typing Rules for Continuations

Judgments

Valid continuations: $\Phi \vdash C \in T$

Rules

$$\frac{}{\Phi \vdash \star \in F \Rightarrow F}\,\mathsf{tcdone}$$

$$\frac{\Phi \vdash C \in F_1 \Rightarrow F \qquad \Phi, \Psi; \Delta \vdash P \in F_1}{\Phi \vdash C, \mathbf{let} \bullet \mathbf{in}\ P \in (\Psi; \Delta) \Rightarrow F}\,\mathsf{tclet}$$

$$\frac{\Phi \vdash C \in F_1 \wedge F_2 \Rightarrow F \qquad \Phi; \cdot \vdash P \in F_2}{\Phi \vdash C, \langle \bullet, P \rangle \in F_1 \Rightarrow F}\,\mathsf{tcpair}$$

$$\frac{\Phi \vdash C \in F_1 \wedge F_2 \Rightarrow F \qquad \Phi; \cdot \vdash V \in F_1}{\Phi \vdash C, \langle V, \bullet \rangle \in F_2 \Rightarrow F}\,\mathsf{tcmix}$$

$$\frac{\Phi \vdash C \in \exists x : A.\, F_1 \Rightarrow F \qquad [\![\Phi]\!] \vdash M : A}{\Phi \vdash C, \langle M, \bullet \rangle \in F_1[\mathrm{id}_\Phi, M/x] \Rightarrow F}\,\mathsf{tcinx}$$

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F \qquad \Phi; \mathbf{x} \in F_1 \vdash D : \Psi; \Delta}{\Phi \vdash C, (\mathbf{x} \in F_1 = \pi_1 \bullet, D) \in F_1 \wedge F_2 \Rightarrow F}\,\mathsf{tcfst}$$

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_2, \Delta) \Rightarrow F \qquad \Phi; \mathbf{x} \in F_2 \vdash D : \Psi; \Delta}{\Phi \vdash C, (\mathbf{x} \in F_2 = \pi_2 \bullet, D) \in F_1 \wedge F_2 \Rightarrow F}\,\mathsf{tcsnd}$$

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F \qquad \Phi; \mathbf{x} \in F_1 \vdash D : \Psi; \Delta}{\Phi \vdash C, (\mathbf{x} \in F_1 = \bullet, D) \in F_1 \Rightarrow F}\,\mathsf{tcassign}$$

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F \qquad \Phi; \cdot \vdash V \in F_1}{\Phi \vdash C, (\bullet; V/\mathbf{x}, \bullet) \in (\Psi; \Delta) \Rightarrow F}\,\mathsf{tcmeta}$$

$$\frac{\Phi \vdash C \in (x : A, \Psi; \mathbf{y} \in F_1, \Delta) \Rightarrow F \qquad \Phi, x : A; \mathbf{y} \in F_1 \vdash D \in \Psi; \Delta}{\Phi \vdash C, (\langle x : A, \mathbf{y} \in F_1 \rangle = \bullet, D) \in \exists x : A.\, F_1 \Rightarrow F}\,\mathsf{tcsplit}$$

$$\frac{\Phi \vdash C \in (x : A, \Psi; \mathbf{y} \in F_1, \Delta) \Rightarrow F \qquad [\![\Phi]\!] \vdash M : A \qquad \Phi; \cdot \vdash V \in F_1[\mathrm{id}_\Phi, M/x]}{\Phi \vdash C, (M/x, \bullet; V/\mathbf{y}, \bullet) \in [\mathrm{id}_\Phi, M/x](\Psi; \Delta) \Rightarrow F}\,\mathsf{tcsubst}$$

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x], \Delta) \Rightarrow F \qquad [\![\Phi]\!] \vdash M : A \qquad \Phi; \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] \vdash D \in \Psi; \Delta}{\Phi \vdash C, (\mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D) \in \forall x : A.\, F_1 \Rightarrow F}\,\mathsf{tcApp}$$

$$\frac{\Phi \vdash C \in (\Psi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho], \Delta) \Rightarrow F \qquad [\![\Phi]\!] \vdash \rho \equiv \rho' \qquad \Phi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] \vdash D \in \Psi; \Delta}{\Phi \vdash C, (\mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D) \in \mathbf{\Pi}\rho^L.\, F_1 \Rightarrow F}\,\mathsf{tcapp}$$

$$\frac{\Phi \vdash C \in \Pi\rho^L.\, (\Psi; \Delta) \Rightarrow F}{\Phi, \rho^L \vdash C, (\lambda\rho^L.\, (\bullet; \bullet)) \in (\Psi; \Delta) \Rightarrow F}\,\mathsf{tcnew}$$

Judgments:

$$\textit{Valid states:}\ \ \vdash S \in F$$

Rules:

$$\frac{\Phi \vdash C \in F_1 \Rightarrow F \qquad \Phi; \cdot \vdash P \in F_1}{\vdash (\Phi; C \triangleright P) \in F}\,\textsf{tsprg}$$

$$\frac{\Phi \vdash C \in (\Psi; \Delta) \Rightarrow F \qquad \Phi; \cdot \vdash D \in \Psi; \Delta}{\vdash (\Phi; C \triangleright D) \in F}\,\textsf{tsdec}$$

$$\frac{\Phi \vdash C \in F_1[\psi] \Rightarrow F \qquad \Phi; \cdot \vdash \psi; \delta : \Psi; \Delta \qquad \Psi; \Delta \vdash \Omega \in F_1}{\vdash (\Phi; C \triangleright (\psi; \delta) \sim \Omega) \in F}\,\textsf{tscase}$$

$$\frac{\Phi \vdash C \in (\Psi; \Delta) \Rightarrow F \qquad \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi; \Delta}{\vdash (\Phi; C \triangleright \psi; \delta) \in F}\,\textsf{tssub}$$

Appendix B

Operational Semantics

B.1 Preliminiaries

B.1.1 Abstraction

Lemma 6.5 (Well-definedness of abstraction)

1. *For all contexts* Γ_1
   if $\Gamma_1, \Gamma_2 \vdash A : type$
   then $\Gamma_1 \vdash \Pi\Gamma_2.\, A : type$

2. *For all contexts* Γ_1
   if $\Gamma_1, \Gamma_2 \vdash M : A$
   then $\Gamma_1 \vdash \lambda\Gamma_2.\, M : \Pi\Gamma_2.\, A$

Proof: , using Lemma 6.3. A detailed proof can be found in Appendix B.1.1. □

Proof: by induction over $\Gamma_2(1)$ and $\Gamma_2(2)$

1. **Case:** $\Gamma_2 = \cdot$:

$\Gamma_1 \vdash A : \text{type}$ — by assumption
$\Gamma_1 \vdash \Pi\cdot.\, A : \text{type}$ — by Definition 6.4

Case: $\Gamma_2 = x : A', \Gamma_2'$:

Case: $A' \not\prec A$:

$\Gamma_1, x : A', \Gamma_2' \vdash A : \text{type}$ — by assumption
$\mathcal{D} :: \Gamma_1, \Gamma_2' \vdash A : \text{type}$ — by Lemma 6.3 (2)
$\Gamma_1 \vdash \Pi\Gamma_2'.\, A : \text{type}$ — by i.h. on $\mathcal{D}$
$\Gamma_1 \vdash \Pi x : A', \Gamma_2'.\, A : \text{type}$ — by Definition 6.4

Case: $A' \prec A$:

$\mathcal{D} :: \Gamma_1, x : A', \Gamma_2' \vdash A : \text{type}$ — by assumption
$\Gamma_1, x : A' \vdash \Pi\Gamma_2'.\, A : \text{type}$ — by i.h. on $\mathcal{D}$
$\Gamma_1 \vdash \Pi x : A'.\, (\Pi\Gamma_2'.\, A) : \text{type}$ — by rule fampi

$\Gamma_1 \vdash \Pi(x : A', \Gamma_2'). A : \text{type}$ by Definition 6.4

2. **Case:** $\Gamma_2 = \cdot$:

$\Gamma_1 \vdash M : A$ by assumption
$\Gamma_1 \vdash \lambda\cdot. M : \Pi\cdot. A$ by Definition 6.4

Case: $\Gamma_2 = x : A', \Gamma_2'$:

Case: $A' \not\prec A$:

$\Gamma_1, x : A', \Gamma_2' \vdash M : A$ by assumption
$\mathcal{D} :: \Gamma_1, \Gamma_2' \vdash M : A$ by Lemma 6.3 (2)
$\Gamma_1 \vdash \lambda\Gamma_2'. M : \Pi\Gamma_2'. A$ by i.h. on $\mathcal{D}$
$\Gamma_1 \vdash \lambda(x : A', \Gamma_2'). M : \Pi(x : A', \Gamma_2'). A$ by Definition 6.4

Case: $A' \prec A$:

$\mathcal{D} :: \Gamma_1, x : A', \Gamma_2' \vdash M : A$ by assumption
$\Gamma_1, x : A' \vdash \lambda\Gamma_2'. M : \Pi\Gamma_2'. A$ by i.h. on $\mathcal{D}$
$\Gamma_1 \vdash \lambda x : A'. \lambda\Gamma_2'. M : \Pi x : A'. (\Pi\Gamma_2'. A)$ by rule objlam
$\Gamma_1 \vdash \lambda(x : A', \Gamma_2'). M : \Pi(x : A', \Gamma_2'). A$ by Definition 6.4

□

Lemma 6.7 (Abstraction)

1. *If* $\mathcal{E} :: \Psi_0, \rho^L; \cdot \vdash \psi_1, \rho/\rho, \psi; \delta \in \Psi_1, \rho^L, \Psi; \Delta$
and $\mathcal{D} :: \Psi_0; \cdot \vdash \psi_1; \cdot \in \Psi_1; \cdot$
then $\Psi_0; \cdot \vdash \psi_1, \psi'; \delta' \in \Psi_1, \Psi'; \Delta'$
and $\psi'; \delta' = \lambda\rho^L. (\psi; \delta)$
and $\Psi'; \Delta' = \Pi\rho^L. (\Psi; \Delta)$

2. *If* $\Psi_0, \rho^L; \cdot \vdash \psi_1, \rho/\rho; \delta \in \Psi_1, \rho^L; \Delta$
and $\mathcal{D} :: \Psi_0; \cdot \vdash \psi_1; \cdot \in \Psi_1; \cdot$
then $\Psi_0; \cdot \vdash \psi_1; \delta' \in \Psi_1; \Delta'$
and $\cdot; \delta' = \lambda\rho^L. (\cdot; \delta)$
and $\cdot; \Delta' = \Pi\rho^L. (\cdot; \Delta)$

Proof:

1. by induction on Ψ:

Case: $\Psi = \cdot$

$\mathcal{E} :: \Psi_0, \rho^L; \cdot \vdash \psi_1, \rho/\rho; \delta \in \Psi_1, \rho^L; \Delta$ by assumption
$\mathcal{Q}_1 :: \Psi_0; \cdot \vdash \psi_1; \delta' \in \Psi_1; \Delta'$ by i.h.(2) on $\mathcal{E}, \mathcal{D}$
$\mathcal{Q}_2 :: \cdot; \delta' = \lambda\rho^L. (\cdot; \delta)$ by i.h.(2) on $\mathcal{E}, \mathcal{D}$
$\mathcal{Q}_3 :: \cdot; \Delta' = \Pi\rho^L. (\cdot; \Delta)$ by i.h.(2) on $\mathcal{E}, \mathcal{D}$

Case: $\Psi = x : A, \Psi'$

$\mathcal{E} :: \Psi_0, \rho^L; \cdot \vdash \psi_1, \rho/\rho, M/x, \psi'; \delta' \in \Psi_1, \rho^L, x : A, \Psi'; \Delta'$	by assumption
$\mathcal{E}_1 :: \Psi_0, \rho^L \vdash M : A[\psi_1, \rho/\rho]$	by several inversion steps
$\mathcal{E}_2 :: \Psi_0 \vdash \lambda\rho. M : \Pi\rho. A[\psi_1, \rho/\rho]$	Lemma 6.5(2) on $\mathcal{E}_1$
$\mathcal{E}_2 :: \Psi_0 \vdash \lambda\rho. M : (\Pi\rho. A)[\psi_1]$	Definition LF substitution
$\mathcal{E}_3 :: \Psi_0, \rho^L; \cdot \vdash \psi_1, \lambda\rho. M/x, \rho/\rho, \psi'; \delta' \in \Psi_1, x : \Pi\rho. A, \rho^L, \Psi''; \Delta''$	by limited LF exchange property
$\mathcal{E}_4 :: \Psi''; \Delta'' = [(x\ \rho)/x](\Psi'; \Delta')$	trivial
$\mathcal{D}_1 :: \Psi_0; \cdot \vdash \psi_1, \lambda\rho. M/x; \cdot \in \Psi_1, x : \Pi\rho. A; \cdot$	by sass on $\mathcal{D}$,$\mathcal{E}_2$
$\mathcal{Q}_1 :: \Psi_0; \cdot \vdash \psi_1, \lambda\rho. M/x, \psi''; \delta'' \in \Psi_1, x : \Pi\rho. A, \Psi'''; \Delta'''$	by i.h.(1) on $\mathcal{E}_4, \mathcal{D}_1$
$\mathcal{P}_2 :: \psi''; \delta'' = \lambda\rho^L. (\psi'; \delta')$	by i.h.(1) on $\mathcal{E}_4, \mathcal{D}_1$
$\mathcal{P}_3 :: \Psi'''; \Delta''' = \Pi\rho^L. (\Psi''; \Delta'')$	by i.h.(1) on $\mathcal{E}_4, \mathcal{D}_1$
$\mathcal{Q}_2 :: \lambda\rho^L. (M/x, \psi'; \delta') = \lambda\rho. M/x, \psi''; \delta''$	by rpass and $\mathcal{P}_2$
$\mathcal{Q}_3 :: \Pi\rho^L. (x : A, \Psi'; \Delta') = x : \Pi\rho^L. A, \Psi^{(4)}; \Delta^{(4)}$	by rass
$\mathcal{R}_1 :: \Psi^{(4)}; \Delta^{(4)} = \Pi\rho^L. ([(x\ \rho)/x](\Psi'; \Delta')) = \Pi\rho^L. (\Psi''; \Delta'') = \Psi'''; \Delta'''$	by rass

2. by induction on Δ:

Case: $\Delta = \cdot$

$\Psi_0; \cdot \vdash \psi_1; \cdot \in \Psi_1; \cdot$	by assumption $\mathcal{D}$

Case: $\Delta = \mathbf{x} \in F, \Delta'$

$\mathcal{E} :: \Psi_0, \rho^L; \cdot \vdash \psi_1, \rho/\rho; P/\mathbf{x}, \delta' \in \Psi_1, \rho^L; \mathbf{x} \in F, \Delta'$	by assumption
$\mathcal{E}_1 :: \Psi_0, \rho^L; \cdot \vdash \psi_1, \rho/\rho; \delta' \in \Psi_1, \rho^L; \Delta'$	trivial
$\mathcal{P}_1 :: \Psi_0; \cdot \vdash \psi_1; \delta'' \in \Psi_1; \Delta''$	by i.h.(2) on $\mathcal{E}_1, \mathcal{D}$
$\mathcal{P}_2 :: \cdot; \delta'' = \lambda\rho^L. (\cdot; \delta')$	by i.h.(2) on $\mathcal{E}_1, \mathcal{D}$
$\mathcal{P}_3 :: \cdot; \Delta'' = \Pi\rho^L. (\cdot; \Delta')$	by i.h.(2) on $\mathcal{E}_1, \mathcal{D}$
$\mathcal{Q}_1 :: \Psi_0; \cdot \vdash \psi_1; \lambda\rho^L. P/\mathbf{x}, \delta'' \in \Psi_1; \mathbf{x} \in \boldsymbol{\Pi}\rho^L. F, \Delta''$	trivial
$\mathcal{Q}_2 :: \lambda\rho^L. (\cdot; P/\mathbf{x}, \delta') = \cdot; \lambda\rho^L. P/\mathbf{x}, \delta''$	by rpmeta on $\mathcal{P}_2$
$\mathcal{Q}_3 :: \Pi\rho^L. (\cdot; \mathbf{x} \in F, \Delta') = \cdot; \mathbf{x} \in \boldsymbol{\Pi}\rho^L. F, \Delta''$	by rmeta on $\mathcal{P}_3$

□

B.1.2 Substitution

Lemma 6.20 (Substitution lemma for meta-substitutions)

1. *If* $\mathcal{D} :: \Psi; \Delta \vdash P \in F$
   and $\mathcal{P} :: \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$
   then $\Psi'; \Delta' \vdash P[\psi; \delta] \in F[\psi]$.

2. *If* $\mathcal{D} :: \Psi; \Delta \vdash D \in \Psi''; \Delta''$
   and $\mathcal{P} :: \Psi'; \Delta' \vdash \psi; \delta \in \Psi; \Delta$
   then $\Psi'; \Delta' \vdash D[\psi; \delta] \in [\psi]\Psi''; [\psi, id_{\Psi''}]\Delta''$.

3. *If* $\mathcal{D}_1 :: \Psi_2; \Delta_2 \vdash \psi_1; \delta_1 \in \Psi_1; \Delta_1$
and $\mathcal{D}_2 :: \Psi_3; \Delta_3 \vdash \psi_2; \delta_2 \in \Psi_2; \Delta_2$
then $\Psi_3; \Delta_3 \vdash (\psi_1; \delta_1) \circ (\psi_2; \delta_2) \in \Psi_1; \Delta_1$
and $(\psi_1; \delta_1) \circ (\psi_2; \delta_2) = (\psi_1 \circ \psi_2, \delta')$ *for some meta-subsitution* δ'

Proof: by simultaneous induction over $\mathcal{D}(1)$, $\mathcal{D}(2)$, and $\mathcal{D}_1(3)$.

1. **Case:** $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ (\mathbf{x} \in F) \in \Delta\end{matrix}}{\Psi; \Delta \vdash \mathbf{x} \in F}\,\mathsf{axvar}$

$\mathcal{E} :: \mathbf{x}[\psi; \delta] = \delta(\mathbf{x})$ by inversion on $\mathcal{E}$
$\Delta'; \Psi' \vdash \delta(\mathbf{x}) \in F[\psi]$ by Lemma 6.19 (1) on $\mathcal{P}, \mathcal{D}_1$

Case: $\mathcal{D} = \dfrac{}{\Psi; \Delta \vdash \langle\rangle \in \top}\,\mathsf{R}\top$

$\mathcal{E} :: \langle\rangle[\psi; \delta] = \langle\rangle$ by assumption
$\mathcal{Q} :: \Psi'; \Delta' \vdash \langle\rangle \in \top$ by $\mathsf{R}\top$

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Psi, x : A; \Delta \vdash P \in F\end{matrix}}{\Psi; \Delta \vdash \Lambda x : A.\, P \in \forall x : A.\, F}\,\mathsf{R}\forall$

$\mathcal{P}_1 :: \Psi', x : A[\psi]; \Delta' \vdash \psi, x/x; \delta : \Psi, x : A; \Delta$ by Lemma 6.16 (2) on $\mathcal{P}$
$\mathcal{E} :: (\Lambda x : A.\, P)[\psi; \delta] = \Lambda x : A[\psi].\, P'$ by assumption
$\mathcal{E}_1 :: P[\psi, x/x; \delta] = P'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi', x : A[\psi]; \Delta' \vdash P' \in F[\psi, x/x]$ by i.h.(1) on $\mathcal{P}_1, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{Q} :: \Psi'; \Delta' \vdash \Lambda x : A[\psi].\, P' \in \forall x : A[\psi].\, F[\psi, x/x]$ by $\mathsf{R}\forall$ on $\mathcal{Q}_1$
$\mathcal{Q} :: \Psi'; \Delta' \vdash \Lambda x : A[\psi].\, P' \in (\forall x : A.\, F)[\psi]$ by sAll

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Psi, \rho^L; \Delta \vdash P \in F\end{matrix}}{\Psi; \Delta \vdash \lambda\rho^L.\, P \in \mathbf{\Pi}\rho^L.\, F}\,\mathsf{R}\Pi$

$\mathcal{P}_1 :: \Psi', ([\psi]\rho)^L; \Delta' \vdash \psi, [\psi]\rho/\rho; \delta : \Psi, \rho^L; \Delta$ by Lemma 6.16 (2) on $\mathcal{P}$
$\mathcal{E} :: (\lambda\rho^L.\, P)[\psi; \delta] = \lambda([\psi]\rho)^L.\, P'$ by assumption
$\mathcal{E}_1 :: P[\psi, [\psi]\rho/\rho; \delta] = P'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi', ([\psi]\rho)^L; \Delta' \vdash P' \in F[\psi, [\psi]\rho/\rho]$ by i.h.(1) on $\mathcal{P}_1, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{Q} :: \Psi'; \Delta' \vdash \lambda([\psi]\rho)^L.\, P' \in \mathbf{\Pi}([\psi]\rho)^L.\, F[\psi, [\psi]\rho/\rho]$ by $\mathsf{R}\Pi$ on $\mathcal{Q}_1$
$\mathcal{Q} :: \Psi'; \Delta' \vdash \lambda([\psi]\rho)^L.\, P' \in (\mathbf{\Pi}\rho^L.\, F)[\psi]$ by sAllP

Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 & & \mathcal{D}_2 \\ \Psi \vdash M : A & & \Psi; \Delta \vdash P \in F[M/x]\end{matrix}}{\Psi; \Delta \vdash \langle M, P\rangle \in \exists x : A.\, F}\,\mathsf{R}\exists$

$\mathcal{P}' :: \Psi' \vdash \psi \in \Psi$ by Lemma 5.21 on $\mathcal{P}$
$\mathcal{Q}_1 :: \Psi' \vdash M[\psi] : A[\psi]$ by Lemma 6.2 on $\mathcal{P}'$ and $\mathcal{D}_1$
$\mathcal{E} :: \langle M, P\rangle[\psi;\delta] = \langle M[\psi], P'\rangle$ by assumption
$\mathcal{E}_1 :: P[\psi;\delta] = P'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_2 :: \Psi';\Delta' \vdash P' \in F[M/x][\psi]$ by i.h.(1) on $\mathcal{P}, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{Q}_2 :: \Psi';\Delta' \vdash P' \in F[\psi, x/x][M[\psi]/x]$ by Lemma 6.14 (1)
$\mathcal{Q} :: \Psi';\Delta' \vdash \langle M[\psi], P'\rangle \in \exists x : A[\psi].\, F[\psi, x/x]$ by $\mathsf{R}\exists$ on $\mathcal{Q}_1, \mathcal{Q}_2$
$\mathcal{Q} :: \Psi';\Delta' \vdash \langle M[\psi], P'\rangle \in (\exists x : A.\, F)[\psi]$ by sEx

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Psi;\Delta \vdash P_1 \in F_1 & \Psi;\Delta \vdash P_2 \in F_2\end{array}}{\Psi;\Delta \vdash \langle P_1, P_2\rangle \in F_1 \wedge F_2}\,\mathsf{R}\wedge$

$\mathcal{E} :: \langle P_1, P_2\rangle[\psi;\delta] = \langle P_1', P_2'\rangle$ by assumption
$\mathcal{E}_1 :: P_1[\psi;\delta] = P_1'$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: P_2[\psi;\delta] = P_2'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi';\Delta' \vdash P_1' \in F_1[\psi]$ by i.h.(1) on $\mathcal{P}, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{Q}_2 :: \Psi';\Delta' \vdash P_2' \in F_2[\psi]$ by i.h.(1) on $\mathcal{P}, \mathcal{D}_2$ and $\mathcal{E}_2$
$\mathcal{Q} :: \Psi';\Delta' \vdash \langle P_1', P_2'\rangle \in F_1[\psi] \wedge F_2[\psi]$ by $\mathsf{R}\wedge$ on $\mathcal{Q}_1$, $\mathcal{Q}_2$
$\mathcal{Q} :: \Psi';\Delta' \vdash \langle P_1', P_2'\rangle \in (F_1 \wedge F_2)[\psi]$ by sAnd

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Psi;\Delta \vdash D \in \Psi'';\Delta'' & \Psi,\Psi'';\Delta,\Delta'' \vdash P \in F\end{array}}{\Psi;\Delta \vdash \textbf{let } D \textbf{ in } P \in F}\,\mathsf{sel}$

$\mathcal{E} :: (\textbf{let } D \textbf{ in } P)[\psi;\delta] = \textbf{let } D' \textbf{ in } P'$ by assumption
$\mathcal{E}_1 :: D[\psi;\delta] = D'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi';\Delta' \vdash D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}]\Delta''$ by i.h.(2) on $\mathcal{P}, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{P}_2 :: \Psi', [\psi]\Psi''; \Delta', [\psi, \mathrm{id}_{\Psi''}]\Delta'' \vdash \psi, \mathrm{id}_{\Psi''}; \delta, \mathrm{id}_{\Delta''} : \Psi, \Psi''; \Delta, \Delta''$
by Lemma 6.16 (3) on $\mathcal{P}_1$
$\mathcal{E}_2 :: P[\psi, \mathrm{id}_{\Psi''}; \delta, \mathrm{id}_{\Delta''}] = P'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_2 :: \Psi, [\psi]\Psi''; \Delta, [\psi, \mathrm{id}_{\Psi''}]\Delta'' \vdash P' \in F[\psi, \mathrm{id}_{\Psi''}]$ by i.h.(1) on $\mathcal{P}_2, \mathcal{D}_2$ and $\mathcal{E}_2$
$\mathcal{Q}_2 :: \Psi, [\psi]\Psi''; \Delta, [\psi, \mathrm{id}_{\Psi''}]\Delta'' \vdash P' \in F[\psi]$ trivial
$\mathcal{Q} :: \Psi';\Delta' \vdash \textbf{let } D' \textbf{ in } P' \in F[\psi]$ by sel on $\mathcal{Q}_1, \mathcal{Q}_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi;\Delta, \mathbf{x} \in F \vdash P \in F\end{array}}{\Psi;\Delta \vdash \mu\mathbf{x} \in F.\, P \in F}\,\mathsf{Rctx}$

$\mathcal{P}_1 :: \Psi';\Delta', \mathbf{x} \in F[\psi] \vdash \psi; \delta, \mathbf{x}/\mathbf{x} : \Psi; \Delta, \mathbf{x} \in F$ by Lemma 6.16 (3) on $\mathcal{P}$
$\mathcal{E} :: (\mu\mathbf{x} \in F.\, P)[\psi;\delta] = \mu\mathbf{x} \in F[\psi].\, P'$ by assumption
$\mathcal{E}_1 :: P[\psi; \delta, \mathbf{x}/\mathbf{x}] = P'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi';\Delta', \mathbf{x} \in F[\psi] \vdash P' \in F[\psi]$ by i.h.(1) on $\mathcal{P}_1, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{Q} :: \Psi';\Delta' \vdash \mu\mathbf{x} \in F[\psi].\, P' \in F[\psi]$ by Rctx on $\mathcal{Q}_1$

$$\textbf{Case: } \mathcal{D} = \cfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Psi; \Delta \vdash \psi''; \delta'' : \Psi''; \Delta'' & \Psi''; \Delta'' \vdash \Omega \in F \end{array}}{\Psi; \Delta \vdash \text{case } (\psi''; \delta'') \text{ of } \Omega \in F[\psi'']} \,\mathsf{case}$$

$\mathcal{E} :: (\text{case } (\psi''; \delta'') \text{ of } \Omega)[\psi; \delta] = \text{case } (\psi''; \delta'') \circ (\psi; \delta) \text{ of } \Omega$ by assumption
$\mathcal{F} :: (\psi'; \delta') = (\psi''; \delta'') \circ (\psi; \delta)$ by i.h.(3) on $\mathcal{P}, \mathcal{D}_1$
$\mathcal{F}_1 :: \Psi'; \Delta' \vdash \psi'; \delta' : \Psi''; \Delta''$ by i.h.(3) on $\mathcal{P}, \mathcal{D}_1$
$\mathcal{F}_2 :: \psi' = \psi'' \circ \psi$ by i.h.(3) on $\mathcal{P}, \mathcal{D}_1$
$\mathcal{Q} :: \Psi'; \Delta' \vdash \text{case } (\psi''; \delta'') \circ (\psi; \delta) \text{ of } \Omega \in F[\psi'' \circ \psi]$ by case on $\mathcal{F}_1, \mathcal{D}_2$
$\mathcal{Q} :: \Psi'; \Delta' \vdash \text{case } (\psi''; \delta'') \circ (\psi; \delta) \text{ of } \Omega \in F[\psi''][\psi]$ by definition

2. $$\textbf{Case: } \mathcal{D} = \cfrac{}{\Psi; \Delta \vdash \cdot \in \cdot; \cdot} \,\mathsf{Ldone}$$

$\mathcal{E} :: [\psi] \cdot = \cdot$ by assumption
$\mathcal{Q} :: \Psi'; \Delta' \vdash \cdot \in \cdot; \cdot$ by Ldone
$\mathcal{Q} :: \Psi'; \Delta' \vdash \cdot \in [\psi]\cdot; [\psi]\cdot$ by def. substitution

$$\textbf{Case: } \mathcal{D} = \cfrac{\begin{array}{lcl} \mathcal{D}_1 & :: & S(L) = \text{SOME } C_1.\ \text{BLOCK } C_2 \\ \mathcal{D}_2 & :: & \Psi \vdash \sigma : C_1 \\ \mathcal{D}_3 & :: & \Psi \vdash \rho \equiv [\sigma] C_2 \\ \mathcal{D}_4 & :: & \Psi, \rho^L; \Delta \vdash D \in \Psi''; \Delta'' \end{array}}{\Psi; \Delta \vdash \nu\ \rho^L.\, D \in \Pi \rho^L.\, (\Psi''; \Delta'')} \,\mathsf{Lnew}$$

$\mathcal{E} :: (\nu\ \rho^L.\, D)[\psi; \delta] = \nu\ ([\psi]\rho)^L.\, D'$ by assumption
$\mathcal{E}_1 :: D' = D[\psi, [\psi]\rho/\rho; \delta]$ by inversion on $\mathcal{E}$
$\mathcal{P}_1 :: \Psi', ([\psi]\rho)^L; \Delta' \vdash \psi, [\psi]\rho/\rho; \delta : \Psi, \rho^L; \Delta$ by Lemma 6.16 (2) on $\mathcal{P}$
$\mathcal{Q}_4 :: \Psi', ([\psi]\rho)^L; \Delta' \vdash D' \in [\psi, [\psi]\rho/\rho]\Psi''; [\psi, [\psi]\rho/\rho, \mathrm{id}_{\Psi''}]\Delta''$
by i.h.(2) on $\mathcal{P}_1, \mathcal{D}_1$ and $\mathcal{E}_4$
$\mathcal{Q}_2 :: \Psi' \vdash \sigma \circ \psi : C_1$ by Lemma 6.12 (1)
$\mathcal{Q}_3 :: \Psi' \vdash [\psi]\rho \equiv C_2[\sigma \circ \psi]$ by Lemma 6.12 (2)
$\mathcal{Q} :: \Psi'; \Delta' \vdash \nu\ ([\psi]\rho)^L.\, D' \in \mathbf{\Pi}([\psi]\rho)^L.\, ([\psi, [\psi]\rho/\rho]\Psi''; [\psi, [\psi]\rho/\rho, \mathrm{id}_{\Psi''}]\Delta'')$
by Lnew on $\mathcal{D}_1, \mathcal{Q}_2, \mathcal{Q}_3$ and $\mathcal{Q}_4$
$\mathcal{Q} :: \Psi'; \Delta' \vdash \nu\ ([\psi]\rho)^L.\, D' \in (\mathbf{\Pi}\rho^L.\, (\Psi''; \Delta''))[\psi]$ by Lemma 6.17

$$\textbf{Case: } \mathcal{D} = \cfrac{\begin{array}{l} \mathcal{D}_1 :: \Psi; \Delta \vdash P \in \forall x : A.\, F \\ \mathcal{D}_2 :: \Psi \vdash M : A \\ \mathcal{D}_3 :: \Psi; \Delta, \mathbf{y} \in F[M/x] \vdash D \in \Psi''; \Delta'' \end{array}}{\Psi; \Delta \vdash \mathbf{y} \in F[M/x] = P\ M, D \in \Psi''; \mathbf{y} \in F[M/x], \Delta''} \,\mathsf{L}\forall$$

$\mathcal{E} :: (\mathbf{y} \in F[M/x] = P\ M, D)[\psi; \delta] = (\mathbf{y} \in F[M/x][\psi] = P'\ M[\psi], D')$
by assumption

$\mathcal{E}_1 :: P[\psi;\delta] = P'$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: D[\psi;\delta,\mathbf{y}/\mathbf{y}] = D'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi';\Delta' \vdash P' \in \forall x : A[\psi].\, F[\psi, x/x]$ by i.h.(1) on $\mathcal{P}, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{Q}_2 :: \Psi' \vdash M[\psi] : A[\psi]$ by Lemma 6.2
$\mathcal{P}_1 :: \Psi';\Delta', \mathbf{y} \in F[M/x][\psi] \vdash \psi;\delta,\mathbf{y}/\mathbf{y} : \Psi;\Delta,\mathbf{y} \in F[M/x]$
by Lemma 6.16 (3) on $\mathcal{P}$
$\mathcal{Q}_3 :: \Psi';\Delta', \mathbf{y} \in F[M/x][\psi] \vdash D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}]\Delta'')$ by i.h.(2) on $\mathcal{P}_1, \mathcal{D}_3$ and $\mathcal{E}_2$
$\mathcal{Q}_3 :: \Psi';\Delta', \mathbf{y} \in F[\psi, x/x][M[\psi]/x] \vdash D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}]\Delta''$ by Lemma 6.14 (1)
$\mathcal{Q} :: \Psi';\Delta' \vdash \mathbf{y} \in F[\psi, x/x][M[\psi]/x] = P'\ M[\psi], D'$
$\in [\psi]\Psi''; \mathbf{y} \in F[\psi, x/x][M[\psi]/x], [\psi, \mathrm{id}_{\Psi''}]\Delta''$
by $\mathsf{L}\forall$ on $\mathcal{Q}_1, \mathcal{Q}_2$, and $\mathcal{Q}_3$
$\mathcal{Q} :: \Psi';\Delta' \vdash \mathbf{y} \in F[M/x][\psi] = P'\ M[\psi], D'$
$\in [\psi]\Psi''; \mathbf{y} \in F[M/x][\psi], [\psi, \mathrm{id}_{\Psi''}]\Delta''$ by Lemma 6.14 (1)
$\mathcal{Q} :: \Psi';\Delta' \vdash \mathbf{y} \in F[M/x][\psi] = P'\ M[\psi], D'$
$\in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}](\mathbf{y} \in F[M/x], \Delta'')$ trivial

Case: $$\mathcal{D} = \cfrac{\begin{array}{l}\mathcal{D}_1 :: \Psi;\Delta \vdash P \in \mathbf{\Pi}\rho^L.\, F\\ \mathcal{D}_2 :: \rho'^L \in \Psi\\ \mathcal{D}_3 :: \Psi \vdash \rho' \equiv \rho\\ \mathcal{D}_4 :: \Psi;\Delta, \mathbf{y} \in F[\rho'/\rho] \vdash D \in \Psi'';\Delta''\end{array}}{\Psi;\Delta \vdash \mathbf{y} \in F[\rho'/\rho] = P\ \rho', D \in \Psi''; \mathbf{y} \in F[\rho'/\rho], \Delta''}\ \mathsf{L}\Pi$$

$\mathcal{E} :: (\mathbf{y} \in F[\rho'/\rho] = P\ \rho', D)[\psi;\delta] = (\mathbf{y} \in F[\rho'/\rho][\psi] = P'\ [\psi]\rho', D')$ by assumption
$\mathcal{E}_1 :: P[\psi;\delta] = P'$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: D[\psi;\delta,\mathbf{y}/\mathbf{y}] = D'$ by inversion on $\mathcal{E}$
$\mathcal{P}' :: \Psi' \vdash \psi \in \Psi$ by Lemma 5.21
$\mathcal{Q}_1 :: \Psi';\Delta' \vdash P' \in \mathbf{\Pi}([\psi]\rho)^L.\, F[\psi, \rho/\rho]$ by i.h.(1) on $\mathcal{P}, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{Q}_2 :: ([\psi]\rho')^L \in \Psi'$ by Lemma 6.19 (2)
$\mathcal{Q}_3 :: \Psi' \vdash [\psi]\rho' \equiv [\psi]\rho$ by Lemma 6.23 on $\mathcal{P}'$ and $\mathcal{D}_2$
$\mathcal{P}_1 :: \Psi';\Delta', \mathbf{y} \in F[\rho'/\rho][\psi] \vdash \psi;\delta,\mathbf{y}/\mathbf{y} : \Psi;\Delta,\mathbf{y} \in F[\rho'/\rho]$ by Lemma 6.16 (3) on $\mathcal{P}$
$\mathcal{Q}_4 :: \Psi';\Delta', \mathbf{y} \in F[\rho'/\rho][\psi] \vdash D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}]\Delta''$ by i.h.(2) on $\mathcal{P}_1, \mathcal{D}_4$ and $\mathcal{E}_2$
$\mathcal{Q}_4 :: \Psi';\Delta', \mathbf{y} \in F[\psi, \rho/\rho][[\psi]\rho'/\rho] \vdash D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}]\Delta''$ by Lemma 6.14 (2)
$\mathcal{Q} :: \Psi';\Delta' \vdash \mathbf{y} \in F[\psi, \rho/\rho][[\psi]\rho'/\rho] = P'\ [\psi]\rho', D'$
$\in [\psi]\Psi''; \mathbf{y} \in F[\psi, \rho/\rho][[\psi]\rho'/\rho], [\psi, \mathrm{id}_{\Psi''}]\Delta''$
by $\mathsf{L}\Pi$ on $\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4$
$\mathcal{Q} :: \Psi';\Delta' \vdash \mathbf{y} \in F[\rho'/\rho][\psi] = P'\ [\psi]\rho', D' \in [\psi]\Psi''; \mathbf{y} \in F[\rho'/\rho][\psi], [\psi, \mathrm{id}_{\Psi''}]\Delta''$
by Lemma 6.14 (2)
$\mathcal{Q} :: \Psi';\Delta' \vdash \mathbf{y} \in F[\rho'/\rho][\psi] = P'\ [\psi]\rho', D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}](\mathbf{y} \in F[\rho'/\rho], \Delta'')$ trivial

Case: $$\mathcal{D} = \cfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \cdot \vdash_\Sigma Q \in \Box S'.\, F & \Psi;\Delta, \mathbf{y} \in F \vdash_{\Sigma;S} D \in \Psi'';\Delta''\end{array}}{\Psi;\Delta \vdash_{\Sigma;S} \mathbf{y} \in F = \text{lemma } Q, D \in \Psi''; \mathbf{y} \in F, \Delta''}\ \mathsf{L}\Xi$$

$\mathcal{E} :: (\mathbf{y} \in F = \text{lemma } Q, D)[\psi;\delta] = (\mathbf{y} \in F = \text{lemma } Q, D')$ by assumption
$\mathcal{E}_1 :: D[\psi;\delta,\mathbf{y}/\mathbf{y}] = D'$ by inversion on $\mathcal{E}$

$F[\psi] = F$ F is closed
$\mathcal{P}_1 :: \Psi'; \Delta', \mathbf{y} \in F \vdash \psi; \delta, \mathbf{y}/\mathbf{y} : \Psi; \Delta, \mathbf{y} \in F$ by Lemma 6.16 (3) on $\mathcal{P}$
$\mathcal{Q}_1 :: \Psi', \Delta', \mathbf{y} \in F \vdash D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}]\Delta''$ by i.h.(2) on $\mathcal{P}_1, \mathcal{D}_2, \mathcal{E}_1$
$\mathcal{Q} :: \Psi', \Delta' \vdash \mathbf{y} \in F = \text{lemma } Q, D' \in [\psi]\Psi''; \mathbf{y} \in F, [\psi, \mathrm{id}_{\Psi''}]\Delta''$ by $\mathsf{L}\Xi$ on $\mathcal{D}_1, \mathcal{Q}_1$
$\mathcal{Q} :: \Psi', \Delta' \vdash \mathbf{y} \in F = \text{lemma } Q, D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}](\mathbf{y} \in F, \Delta'')$ trivial

Case: $$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Psi; \Delta \vdash P \in \exists x : A.\, F\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Psi, x : A; \Delta, \mathbf{y} \in F \vdash D \in \Psi''; \Delta''\end{matrix}}{\Psi; \Delta \vdash \langle x : A, \mathbf{y} \in F\rangle = P, D \in x : A, \Psi''; \mathbf{y} \in F, \Delta''}\ \mathsf{L}\exists$$

$\mathcal{E} :: (\langle x : A, \mathbf{y} \in F\rangle = P, D)[\psi; \delta] = (\langle x : A[\psi], \mathbf{y} \in F[\psi, x/x]\rangle = P', D')$
by assumption
$\mathcal{E}_1 :: P[\psi; \delta] = P'$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: D[\psi, x/x; \delta, \mathbf{y}/\mathbf{y}] = D'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi'; \Delta' \vdash P' \in (\exists x : A[\psi].\, F[\psi, x/x])$ by i.h.(1) on $\mathcal{P}, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{P}_1 :: \Psi', x : A[\psi]; \Delta', \mathbf{y} \in F[\psi, x/x] \vdash \psi, x/x; \delta, \mathbf{y}/\mathbf{y} : \Psi, x : A; \Delta, \mathbf{y} \in F$
by Lemma 6.16 (3) on $\mathcal{P}$
$\mathcal{Q}_2 :: \Psi', x : A[\psi]; \Delta', \mathbf{y} \in F[\psi, x/x] \vdash D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}]\Delta''$
by i.h.(2) on $\mathcal{P}_1, \mathcal{D}_2$ and $\mathcal{E}_2$
$\mathcal{Q} :: \Psi'; \Delta' \vdash (\langle x : A[\psi], \mathbf{y} \in F[\psi, x/x]\rangle = P', D')$
$\in x : A[\psi], [\psi]\Psi''; \mathbf{y} \in F[\psi, x/x], [\psi, \mathrm{id}_{\Psi''}]\Delta'''$
by $\mathsf{L}\exists$ on $\mathcal{Q}_1$ and $\mathcal{Q}_2$
$\mathcal{Q} :: \Psi'; \Delta' \vdash (\langle x : A[\psi], \mathbf{y} \in F[\psi, x/x]\rangle = P', D') \in [\psi](x : A, \Psi''); [\psi, \mathrm{id}_{\Psi''}](\mathbf{y} \in F, \Delta'')$
trivial

Case: $$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Psi; \Delta \vdash P \in F_1 \wedge F_2\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Psi; \Delta, \mathbf{x} \in F_1 \vdash D \in \Psi''; \Delta''\end{matrix}}{\Psi; \Delta \vdash \mathbf{x} \in F_1 = \pi_1\ P, D \in \Psi''; \mathbf{x} \in F_1, \Delta''}\ \mathsf{L}\wedge_1$$

$\mathcal{E} :: (\mathbf{x} \in F_1 = \pi_1\ P, D)[\psi; \delta] = (\mathbf{x} \in F_1[\psi] = \pi_1\ P', D')$ by assumption
$\mathcal{E}_1 :: P[\psi; \delta] = P'$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: D[\psi; \delta, \mathbf{x}/\mathbf{x}] = D'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi'; \Delta' \vdash P' \in F_1[\psi] \wedge F_2[\psi]$ by i.h.(1) on $\mathcal{P}, \mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{P}_1 :: \Psi'; \Delta', \mathbf{x} \in F_1[\psi] \vdash \psi; \delta, \mathbf{x}/\mathbf{x} : \Psi; \Delta, \mathbf{x} \in F_1$ by Lemma 6.16 (3) on $\mathcal{P}$
$\mathcal{Q}_2 :: \Psi'; \Delta', \mathbf{x} \in F_1[\psi] \vdash D' \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}]\Delta''$ by i.h.(2) on $\mathcal{P}_1, \mathcal{D}_2$ and $\mathcal{E}_2$
$\mathcal{Q} :: \Psi'; \Delta' \vdash (\mathbf{x} \in F_1[\psi] = \pi_1\ P', D') \in [\psi]\Psi''; \mathbf{x} \in F_1[\psi], [\psi, \mathrm{id}_{\Psi''}]\Delta''$
by $\mathsf{L}\wedge_1$ on $\mathcal{Q}_1$ and $\mathcal{Q}_2$
$\mathcal{Q} :: \Psi'; \Delta' \vdash (\mathbf{x} \in F_1[\psi] = \pi_1\ P', D') \in [\psi]\Psi''; [\psi, \mathrm{id}_{\Psi''}](\mathbf{y} \in F_1, \Delta'')$ trivial

Case: $$\mathcal{D} = \cfrac{\begin{matrix}\mathcal{D}_1 \\ \Psi; \Delta \vdash P \in F_1 \wedge F_2\end{matrix} \qquad \begin{matrix}\mathcal{D}_1 \\ \Psi; \Delta, \mathbf{x} \in F_2 \vdash D \in \Psi''; \Delta''\end{matrix}}{\Psi; \Delta \vdash \mathbf{x} \in F_2 = \pi_2\ P, D \in \Psi''; \mathbf{x} \in F_2, \Delta''}\ \mathsf{L}\wedge_2$$

$\mathcal{E} :: (\mathbf{x} \in F_2 = \pi_2\ P, D)[\psi; \delta] = (\mathbf{x} \in F_2[\psi] = \pi_2\ P', D')$ by assumption

$\mathcal{E}_1 :: P[\psi;\delta] = P'$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: D[\psi;\delta,\mathbf{x}/\mathbf{x}] = D'$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Psi';\Delta' \vdash P' \in F_1[\psi] \wedge F_2[\psi]$ by i.h.(1) on $\mathcal{P},\mathcal{D}_1$ and $\mathcal{E}_1$
$\mathcal{P}_1 :: \Psi';\Delta',\mathbf{x} \in F_2[\psi] \vdash \psi;\delta,\mathbf{x}/\mathbf{x} : \Psi;\Delta,\mathbf{x} \in F_2$ by Lemma 6.16 (3) on $\mathcal{P}$
$\mathcal{Q}_2 :: \Psi';\Delta',\mathbf{x} \in F_2[\psi] \vdash D' \in [\psi]\Psi'';[\psi,\mathrm{id}_{\Psi''}]\Delta''$ by i.h.(2) on $\mathcal{P}_1,\mathcal{D}_2$ and $\mathcal{E}_2$
$\mathcal{Q} :: \Psi';\Delta' \vdash (\mathbf{x} \in F_2[\psi] = \pi_2\ P', D') \in [\psi]\Psi'';\mathbf{x} \in F_2[\psi],[\psi,\mathrm{id}_{\Psi''}]\Delta''$
by $\mathsf{L}\wedge_1$ on $\mathcal{Q}_1$ and $\mathcal{Q}_2$
$\mathcal{Q} :: \Psi';\Delta' \vdash (\mathbf{x} \in F_2[\psi] = \pi_2\ P', D') \in [\psi]\Psi'';[\psi,\mathrm{id}_{\Psi''}](\mathbf{y} \in F_2,\Delta'')$ trivial

3. **Case:** $$\mathcal{D}_1 = \cfrac{\begin{matrix}\mathcal{D}_1' \\ \Psi_2 \vdash \psi_1 \in \Psi_1\end{matrix}}{\Psi_2;\Delta_2 \vdash \psi_1;\cdot \in \Psi_1;\cdot}\ \mathsf{sabstract}$$

$\mathcal{Q}_1 :: \Psi_3 \vdash \psi_2 \in \Psi_2$ by Lemma 5.21 on $\mathcal{D}_2$
$\mathcal{Q}_2 :: \Psi_3 \vdash \psi_1 \circ \psi_2 \in \Psi_1$ by Lemma 5.18 on $\mathcal{D}_1',\mathcal{Q}_1$
$\mathcal{R}_1 :: \Psi_3;\Delta_3 \vdash (\psi_1 \circ \psi_2);\cdot \in \Psi_1;\cdot$ by $\mathsf{sabstract}$ on $\mathcal{Q}_2$
$\mathcal{R}_2 :: \Psi_3;\Delta_3 \vdash (\psi_1;\cdot) \circ (\psi_2;\delta_2) \in \Psi_1;\cdot$ by Definition 5.19 (cempty)

Case: $$\mathcal{D}_1 = \cfrac{\begin{matrix}\mathcal{D}_1' \\ \Psi_2;\Delta_2 \vdash P \in F[\psi_1]\end{matrix} \qquad \begin{matrix}\mathcal{D}_1'' \\ \Psi_2;\Delta_2 \vdash \psi_1;\delta_1 \in \Psi_1;\Delta_1\end{matrix}}{\Psi_2;\Delta_2 \vdash \psi_1;\delta_1,P/\mathbf{x} \in \Psi_1;\Delta_1,\mathbf{x} \in F}\ \mathsf{smeta}$$

$\mathcal{Q}_1 :: \Psi_3;\Delta_3 \vdash P[\psi_2;\delta_2] \in F[\psi_1][\psi_2]$ by i.h.(1) on $\mathcal{D}_1',\mathcal{D}_2$
$\mathcal{Q}_1 :: \Psi_3;\Delta_3 \vdash P[\psi_2;\delta_2] \in F[\psi_1 \circ \psi_2]$ trivial
$\mathcal{Q}_2 :: \Psi_3;\Delta_3 \vdash (\psi_1;\delta_1) \circ (\psi_2;\delta_2) \in \Psi_1;\Delta_1$ by i.h.(3) on $\mathcal{D}_1'',\mathcal{D}_2$
$\mathcal{Q}_3 :: (\psi_1;\delta_1) \circ (\psi_2;\delta_2) = (\psi_1 \circ \psi_2,\delta')$ by i.h.(3) on $\mathcal{D}_1'',\mathcal{D}_2$
$\mathcal{R}_1 :: \Psi_3;\Delta_3 \vdash (\psi_1 \circ \psi_2,\delta',P[\psi_2;\delta_2]/\mathbf{x}) \in \Psi_1;\Delta_1,\mathbf{x} \in F$ by smeta on $\mathcal{Q}_1,\mathcal{Q}_2$
$\mathcal{R}_2 :: \Psi_3;\Delta_3 \vdash (\psi_1;\delta_1,P/\mathbf{x}) \circ (\psi_2;\delta_2) \in \Psi_1;\Delta_1,\mathbf{x} \in F$ by Definition 5.19 (cmeta)

□

B.2 Strictness

Lemma 6.30 (Soundness)
If $\mathcal{D} ::\vdash \Phi \triangleright \exists\Psi'.\,\psi \approx \eta\{\top\}$ *matchable*
then there exists a (unique) η', $\Phi \vdash \eta' \in \Psi'$ *and* $\psi \circ \eta' = \eta$

Proof: direct.

$\mathcal{D}_1 :: \Phi \triangleright \exists\Psi'.\,\psi \approx \eta\{\top\} \overset{*}{\Longrightarrow} \Phi \triangleright \top\{U\}$ for some U by inversion
$\mathcal{D}_2 :: \Phi \triangleright U\{\top\} \overset{*}{\Longrightarrow} \Phi \triangleright \top\{\top\}$ by inversion
$\mathcal{E} :: \cdot$ is solution for $\top\{\top\}$ by Lemma 6.29
there exists an $\eta = \cdot\ (\Phi \vdash \eta \in \cdot)$
s.t. η is a solution of $U\{\top\}$ by Lemma 6.28(2) on $\mathcal{D}_2$

η is solution for $U_1 \wedge \top$ by Definition 6.25
η is solution for U_1 by Definition 6.25
η is solution for $\top \wedge U$ by Definition 6.25
η is solution for $\top\{U\}$ by Definition 6.25
there exists an η' $(\Phi \vdash \eta' \in \Psi')$
s.t. η' is a solution of $\exists\Psi'.\,\psi \approx \eta'\{\top\}$ by Lemma 6.28(2) on $\mathcal{D}_1$
$\psi \circ \eta' = \eta$ by Definition 6.25

□

Lemma 6.32 (Completeness I)

1. *If* $U_1 \neq \top$
and $\Phi \triangleright \exists\Psi.\,U_1\{U_2\}$ *is given*
and η $(\Phi \vdash \eta \in \Psi)$ *is a solution of* $\exists\Psi.\,U_1\{U_2\}$
and $\Psi \vdash U_1$ *strict*
then $\Phi \triangleright \exists\Psi.\,U_1\{U_2\} \implies \Phi \triangleright \exists\Psi'.\,U_1'\{U_2'\}$
and there exists an η' $(\Phi \vdash \eta' \in \Psi')$ *which is a solution of* $\exists\Psi'.\,U_1'\{U_2'\}$
and $\Psi' \vdash U_1'$ *strict*
and $(|\Psi'|, |U_1'|) <_{lex} (|\Psi|, |U_1|)$.

2. *If* $T = \Phi \triangleright \exists\Psi.\,U_1\{U_2\}$ *is given matching state*
then $T \overset{*}{\implies} \Phi \triangleright \top\{U\}$ *for some* U.

Proof: 1) by inspection of the rules, 2) by induction on $(|\Psi|, |U_1|)$.

Case: $(|\Psi_1|, |U_1|) = (0, 0)$.

$T = \Phi \triangleright \top\{U_2\}$ by Lemma 6.31 (1)
$\Phi \triangleright \top\{U_2\} \overset{*}{\implies} \Phi \triangleright \top\{U_2\}$ by mrefl

Case: $(|\Psi_1|, |U_1|) \neq (0, 0)$.

$T = \Phi \triangleright \exists\Psi.\,U_1\{U_2\}$ by assumption
$\Phi \triangleright \exists\Psi.\,U_1\{U_2\} \implies \Phi \triangleright \exists\Psi'.\,U_1'\{U_2'\}$ by i.h.(1)
$(|\Psi'|, |U_1'|) <_{\text{lex}} (|\Psi|, |U_1|)$ by i.h.(1)
$\Phi \triangleright \exists\Psi'.\,U_1'\{U_2'\} \overset{*}{\implies} \Phi \triangleright \top\{U\}$ for some U. by i.h.(2)
$\Phi \triangleright \exists\Psi.\,U_1\{U_2\} \overset{*}{\implies} \Phi \triangleright \top\{U\}$ for some U. by mtrans

□

Lemma 6.33 (Completeness II)

1. *If* $U \neq \top$
and $\Phi \triangleright U\{\top\}$ *is given*
and $\cdot$ $(\Phi \vdash \cdot \in \cdot)$ *is a trival solution for* $U\{\top\}$
then $\Phi \triangleright U\{\top\} \implies \Phi \triangleright U'\{\top\}$
and $\cdot$ $(\Phi \vdash \cdot \in \cdot)$ *is a trival solution for* $U'\{\top\}$
and $|U'| < |U|$.

2. *If* $T = \Phi \triangleright U\{\top\}$ *is given matching state then* $\Phi \triangleright U\{\top\} \overset{*}{\Longrightarrow} \Phi \triangleright \top\{\top\}$

Proof: 1) by inspection of the rules, 2) by by induction on $|U|$.

Case: $|U| = 0$.

$T = \Phi \triangleright \top\{\top\}$	by Lemma 6.31 (1)
$\Phi \triangleright \top\{\top\} \overset{*}{\Longrightarrow} \Phi \triangleright \top\{\top\}$	by mrefl

Case: $|U| \neq 0$.

$T = \Phi \triangleright U\{\top\}$	by assumption
$\Phi \triangleright U\{\top\} \Longrightarrow \Phi \triangleright U'\{\top\}$	by i.h.(1)
$\lvert U'\rvert < \lvert U\rvert$	by i.h.(1)
$\Phi \triangleright U'\{\top\} \overset{*}{\Longrightarrow} \Phi \triangleright \top\{\top\}$	by i.h.(2)
$\Phi \triangleright U\{\top\} \overset{*}{\Longrightarrow} \Phi \triangleright \top\{\top\}$	by mtrans

□

Theorem 6.34 (Completeness)
If $T = \Phi \triangleright \exists\Psi.\, U_1\{U_2\}$
and $\mathcal{D} :: \Psi \vdash U_1$ *strict*
and $\mathcal{E} :: \eta$ $(\Phi \vdash \eta \in \Psi)$ *is a solution of* $\exists\Psi.\, U_1\{U_2\}$
then $\vdash T$ *matchable*

Proof: direct

$T \overset{*}{\Longrightarrow} \Phi \triangleright \top\{U\}$ for some U	by Lemma 6.32(2) on $\mathcal{D}$ and $\mathcal{E}$
$\Phi \triangleright U\{\top\} \overset{*}{\Longrightarrow} \Phi \triangleright \top\{\top\}$	by Lemma 6.33(2)
$\vdash T$ matchable	by msuccess

□

Theorem 6.36 (Determinacy)
If $\mathcal{D} :: \Psi' \vdash \psi \in \Psi$
and $\mathcal{E} :: \Psi' \vdash \psi$ *strict*
and $\mathcal{F} :: \Phi \vdash \eta \in \Psi$
then there exists a (unique) η' $(\Phi \vdash \eta' \in \Psi')$ *s.t.* $\psi \circ \eta' = \eta$
or not.

Proof:

$\mathcal{E}' :: \Psi' \vdash (\psi \approx \eta \wedge True)$ strict	by Lemma 6.27 on $\mathcal{E}$

Case: $\vdash \Phi \triangleright \exists \Psi'.\, \psi \approx \eta$ matchable

There exists an η' $(\Phi \vdash \eta' \in \Psi')$ s.t. $\psi \circ \eta' = \eta$ — by Theorem 6.30 on $\mathcal{E}', \mathcal{F}$

Case: $\not\vdash \Phi \triangleright \exists \Psi'.\, \psi \approx \eta$ matchable

There exists no η' $(\Phi \vdash \eta' \in \Psi')$ s.t. $\psi \circ \eta' = \eta$ — by Theorem 6.34 on $\mathcal{E}', \mathcal{F}$

□

B.3 Big-Step Semantics

Lemma 6.37 (Context)

1. If $\mathcal{D} :: \Phi; \cdot \vdash id_\Phi, \psi; \delta \in \Phi, [id_\Phi, M/x]\Psi; [id_\Phi, M/x, id_\Psi]\Delta$
and $\mathcal{E} :: [\![\Phi]\!] \vdash M : A$
and $\mathcal{P} :: \Phi; \cdot \vdash V \in F[id_\Phi, M/x]$
then $\Phi; \cdot \vdash (id_\Phi, M/x, \psi; V/\mathbf{y}, \delta) \in (\Phi, x : A, \Psi; \mathbf{y} \in F, \Delta)$

2. If $\mathcal{D} :: \Phi; \cdot \vdash id_\Phi, \psi; \delta \in \Phi, \Psi; \Delta$
and $\mathcal{P} :: \Phi; \cdot \vdash V \in F$
then $\Phi; \cdot \vdash (id_\Phi; \psi; V/\mathbf{y}, \delta) \in (\Phi, \Psi; \mathbf{y} \in F, \Delta)$

Proof: direct in both cases.

1. Let $\psi'; \delta' = \mathrm{id}_\Phi, M/x, \mathrm{id}_\Psi; V/\mathbf{y}, \mathrm{id}_\Delta$
$\Phi, [\mathrm{id}_\Phi, M/x]\Psi; [\mathrm{id}_\Phi, M/x, \mathrm{id}_\Psi]\Delta \vdash \psi'; \delta' \in \Phi, x : A, \Psi; y \in F, \Delta$ — by definition substitution
Let $\psi''; \delta'' = (\psi'; \delta') \circ (\mathrm{id}_\Phi, \psi; \delta) = \mathrm{id}_\Phi, M/x, \psi; V/\mathbf{y}, \delta$
$\Phi; \cdot \vdash \psi''; \delta'' \in \Phi, x : A, \Psi; \mathbf{y} \in F, \Delta$ — by Corollaray 6.21

2. Let $\psi'; \delta' = \mathrm{id}_\Phi, \mathrm{id}_\Psi; V/\mathbf{y}, \mathrm{id}_\Delta$
$\Phi, \Psi; \Delta \vdash \psi'; \delta' \in \Phi, \Psi; y \in F, \Delta$ — by definition substitution
Let $\psi''; \delta'' = (\psi'; \delta') \circ (\mathrm{id}_\Phi, \psi; \delta) = \mathrm{id}_\Phi, \psi; V/\mathbf{y}, \delta$
$\Phi; \cdot \vdash \psi''; \delta'' \in \Phi, \Psi; \mathbf{y} \in F, \Delta$ — by Corollaray 6.21

□

Theorem 6.38 (Type-preservation)

1. If $\mathcal{D} :: \Phi \vdash P \hookrightarrow V$
and $\mathcal{E} :: \Phi; \cdot \vdash P \in F$
then $\Phi; \cdot \vdash V \in F$

2. *If* $\mathcal{D} :: \Phi \vdash D \hookrightarrow \psi; \delta$
   and $\mathcal{E} :: \Phi; \cdot \vdash D \in \Psi; \Delta$
   then $\Phi; \cdot \vdash (id_\Phi, \psi; \delta) \in (\Phi, \Psi; \Delta)$
   which extends $\Phi; \cdot \vdash (id_\Phi; \cdot) \in (\Phi; \cdot)$

3. *If* $\mathcal{D} :: \Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V$
   and $\mathcal{F} :: \Phi; \cdot \vdash \psi; \delta \in \Psi; \Delta$
   and $\mathcal{E} :: \Psi; \Delta \vdash \Omega \in F$
   then $\Phi; \cdot \vdash V \in F[\psi]$

Proof: by simultaneous induction over $\mathcal{D}(1), \mathcal{D}(2), \mathcal{D}(3)$.

1. **Case:** $\mathcal{D} = \dfrac{}{\Phi \vdash \Lambda x : A.\, P \hookrightarrow \Lambda x : A.\, P}\ \mathsf{ev\_Lam}$

   $\mathcal{E} = \Phi; \cdot \vdash \Lambda x : A.\, P \in F$ — by assumption

   Case: $\mathcal{D} = \dfrac{}{\Phi \vdash \lambda \rho^L.\, P \hookrightarrow \lambda \rho^L.\, P}\ \mathsf{ev\_lam}$

   $\mathcal{E} = \Phi; \cdot \vdash \lambda \rho^L.\, P \in F$ — by assumption

   Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Phi \vdash P \hookrightarrow V\end{matrix}}{\Phi \vdash \langle M, P \rangle \hookrightarrow \langle M, V \rangle}\ \mathsf{ev\_inx}$

$\mathcal{E} :: \Phi; \cdot \vdash \langle M, P \rangle \in \exists x : A.\, F$	by assumption
$\mathcal{E}_1 :: [\![\Phi]\!] \vdash M : A$	by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \Phi; \cdot \vdash P \in F[M/x]$	by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash V \in F[M/x]$	by i.h.(1) on $\mathcal{D}_1, \mathcal{E}_2$
$\mathcal{Q}_2 :: \Phi; \cdot \vdash \langle M, V \rangle \in \exists x : A.\, F$	by $\mathsf{R}\exists$ on $\mathcal{E}_1$ and $\mathcal{Q}_1$

   Case: $\mathcal{D} = \dfrac{}{\Phi \vdash \langle\rangle \hookrightarrow \langle\rangle}\ \mathsf{ev\_unit}$

   $\mathcal{E} = \Phi; \cdot \vdash \langle\rangle \in F$ — by assumption

   Case: $\mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Phi \vdash D \hookrightarrow \psi; \delta\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Phi \vdash P[\mathrm{id}_\Phi, \psi; \delta] \hookrightarrow V\end{matrix}}{\Phi \vdash \mathbf{let}\ D\ \mathbf{in}\ P \hookrightarrow V}\ \mathsf{ev\_let}$

$\mathcal{E} :: \Phi; \cdot \vdash \mathbf{let}\ D\ \mathbf{in}\ P \in F$	by assumption
$\mathcal{E}_1 :: \Phi; \cdot \vdash D \in \Psi; \Delta$	by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \Phi, \Psi; \Delta \vdash P \in F$	by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta : \Phi, \Psi; \Delta$	by i.h.(2) on $\mathcal{D}_1, \mathcal{E}_1$
$\mathcal{D}_3 :: \Phi; \cdot \vdash P[\mathrm{id}_\Phi, \psi; \delta] = P'$	by definition of $\mathcal{D}_2$
$\mathcal{D}_4 :: \Phi; \cdot \vdash P' \hookrightarrow V$	by definition of $\mathcal{D}_2$

$\mathcal{Q}_2 :: \Phi; \cdot \vdash P' \in F[\mathrm{id}_\Phi, \psi]$ by Lemma 6.20(2) on $\mathcal{E}_2, \mathcal{D}_3$
$\mathcal{Q}_2 :: \Phi; \cdot \vdash P' \in F$ F closed on Ψ
$\mathcal{Q} :: \Phi; \cdot \vdash V \in F$ by i.h.(1) on $\mathcal{D}_4, \mathcal{Q}_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi \vdash P[\mu\mathbf{x} \in F.\, P/\mathbf{x}] \hookrightarrow V\end{array}}{\Phi \vdash \mu\mathbf{x} \in F.\, P \hookrightarrow V}\,\mathsf{ev\_rec}$

$\mathcal{E} :: \Phi; \cdot \vdash \mu\mathbf{x} \in F.\, P \in F$ by assumption
$\mathcal{E}_1 :: \Phi; \mathbf{x} \in F \vdash P \in F$ by inversion on $\mathcal{E}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash P[\mu\mathbf{x} \in F.\, P/\mathbf{x}] = P'$ by definition of $\mathcal{D}_1$
$\mathcal{D}_3 :: \Phi \vdash P' \hookrightarrow V$ by definition of $\mathcal{D}_1$
$\mathcal{P} :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot : \Phi; \cdot$ by Lemma 6.22
$\mathcal{P}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \mu\mathbf{x} \in F.\, P/\mathbf{x} : \Phi; \mathbf{x} \in F$ by smeta on $\mathcal{E}, \mathcal{P}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash P' \in F$ by Lemma 6.20(1) on $\mathcal{D}_1, \mathcal{P}_1$
$\mathcal{Q} :: \Phi; \cdot \vdash V \in F$ by i.h.(1) on $\mathcal{D}_3, \mathcal{Q}_1$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi \vdash P_1 \hookrightarrow V_1\end{array} \qquad \begin{array}{c}\mathcal{D}_2 \\ \Phi \vdash P_2 \hookrightarrow V_2\end{array}}{\Phi \vdash \langle P_1, P_2\rangle \hookrightarrow \langle V_1, V_2\rangle}\,\mathsf{ev\_pair}$

$\mathcal{E} :: \Phi; \cdot \vdash \langle P_1, P_2\rangle \in F_1 \wedge F_2$ by assumption
$\mathcal{E}_1 :: \Phi; \cdot \vdash P_1 \in F_1$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \Phi; \cdot \vdash P_2 \in F_2$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash V_1 \in F_1$ by i.h.(1) on $\mathcal{D}_1, \mathcal{E}_1$
$\mathcal{Q}_2 :: \Phi; \cdot \vdash V_2 \in F_2$ by i.h.(1) on $\mathcal{D}_2, \mathcal{E}_2$
$\mathcal{Q} :: \Phi; \cdot \vdash \langle V_1, V_2\rangle \in F_1 \wedge F_2$ by $\mathsf{R}\wedge$ on $\mathcal{Q}_1, \mathcal{Q}_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V\end{array}}{\Phi \vdash \text{case } (\psi; \delta) \text{ of } \Omega \hookrightarrow V}\,\mathsf{ev\_case}$

$\mathcal{E} :: \Phi; \cdot \vdash \text{case } (\psi; \delta) \text{ of } \Omega \in F[\psi]$ by assumption
$\mathcal{E}_1 :: \Phi; \cdot \vdash \psi; \delta : \Psi; \Delta$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \Psi; \Delta \vdash \Omega \in F$ by inversion on $\mathcal{E}$
$\mathcal{Q} :: \Phi; \cdot \vdash V \in F[\psi]$ by i.h.(3) on $\mathcal{D}_1, \mathcal{E}_1, \mathcal{E}_2$

2. **Case:** $\mathcal{D} = \dfrac{}{\Phi \vdash \cdot \hookrightarrow \cdot; \cdot}\,\mathsf{ev\_empty}$

$\Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot \in \Phi; \cdot$ by Lemma 6.22

$$\textbf{Case: } \mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Phi \vdash P \hookrightarrow \langle M, V\rangle & \Phi \vdash D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}] \hookrightarrow \psi; \delta\end{array}}{\Phi \vdash \langle x : A, \mathbf{y} \in F\rangle = P, D \hookrightarrow M/x, \psi; V/\mathbf{y}, \delta}\ \mathsf{ev\_split}$$

$\mathcal{E} :: \Phi; \cdot \vdash \langle x : A, \mathbf{y} \in F\rangle = P, D \in x : A, \Psi; \mathbf{y} \in F, \Delta$	by assumption
$\mathcal{E}_1 :: \Phi; \cdot \vdash P \in \exists x : A.\, F$	by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \Phi, x : A; \mathbf{y} \in F \vdash D \in \Psi; \Delta$	by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash \langle M, V\rangle \in \exists x : A.\, F$	by i.h.(1) on $\mathcal{D}_1, \mathcal{E}_1$
$\mathcal{Q}_2 :: [\![\Phi]\!] \vdash M : A$	by inversion on $\mathcal{Q}_1$
$\mathcal{Q}_3 :: \Phi; \cdot \vdash V \in F[\mathrm{id}_\Phi, M/x]$	by inversion on $\mathcal{Q}_1$
$\mathcal{D}_3 :: \Phi; \cdot \vdash D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}] = D'$	by definition of $\mathcal{D}_2$
$\mathcal{D}_4 :: \Phi \vdash D' \hookrightarrow \psi; \delta$	by definition of $\mathcal{D}_2$
$\mathcal{P}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot : \Phi; \cdot$	by Lemma 6.22
$\mathcal{P}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, M/x; \cdot \in \Phi, x : A; \cdot$	by sass on $\mathcal{Q}_2, \mathcal{P}_1$
$\mathcal{P} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, M/x; V/\mathbf{y} \in \Phi, x : A; \mathbf{y} \in F$	by smeta on $\mathcal{Q}_3, \mathcal{P}_2$
$\mathcal{R}_1 :: \Phi; \cdot \vdash D' \in [\mathrm{id}_\Phi, M/x]\Psi; [\mathrm{id}_\Phi, M/x]\Delta$	by Lemma 6.20(2) on $\mathcal{P}, \mathcal{D}_3, \mathcal{E}_2$
$\mathcal{R}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, [\mathrm{id}_\Phi, M/x]\Psi; [\mathrm{id}_\Phi, M/x]\Delta$	by i.h.(2) on $\mathcal{R}_1, \mathcal{D}_4$
$\mathcal{R} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, M/x, \psi; V/\mathbf{y}, \delta \in \Phi, x : A, \Psi; \mathbf{y} \in F, \Delta$	by Lemma 6.37(1) on $\mathcal{R}_2, \mathcal{Q}_2, \mathcal{Q}_3$

$$\textbf{Case: } \mathcal{D} = \dfrac{\begin{array}{ccc}\mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ \Phi \vdash P \hookrightarrow \Lambda x : A.\, P' & \Phi \vdash P'[\mathrm{id}_\Phi, M/x] \hookrightarrow V & \Phi \vdash D[V/\mathbf{y}] \hookrightarrow \psi; \delta\end{array}}{\Phi \vdash \mathbf{y} \in F[\mathrm{id}_\Phi, M/x] = P\ M, D \hookrightarrow \psi; V/\mathbf{y}, \delta}\ \mathsf{ev\_App}$$

$\mathcal{E} :: \Phi; \cdot \vdash \mathbf{y} \in F[\mathrm{id}_\Phi, M/x] = P\ M, D \in \Psi; \mathbf{y} \in F[\mathrm{id}_\Phi, M/x], \Delta$	by assumption
$\mathcal{E}_1 :: \Phi; \cdot \vdash P \in \forall x : A.\, F$	by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: [\![\Phi]\!] \vdash M : A$	by inversion on $\mathcal{E}$
$\mathcal{E}_3 :: \Phi; \mathbf{y} \in F[\mathrm{id}_\Phi, M/x] \vdash D \in \Psi; \Delta$	by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash \Lambda x : A.\, P' \in \forall x : A.\, F$	by i.h.(1) on $\mathcal{D}_1, \mathcal{E}_1$
$\mathcal{Q}_2 :: \Phi, x : A; \cdot \vdash P' \in F$	by inversion on $\mathcal{Q}_1$
$\mathcal{F}_1 :: \Phi; \cdot \vdash P'[\mathrm{id}_\Phi, M/x] = P''$	by definition of $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi \vdash P'' \hookrightarrow V$	by definition of $\mathcal{D}_2$
$\mathcal{P}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot \Phi; \cdot$	by Lemma 6.22
$\mathcal{P} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, M/x; \cdot \in \Phi, x : A; \cdot$	by sass on $\mathcal{P}_1, \mathcal{E}_2$
$\mathcal{R}_1 :: \Phi; \cdot \vdash P'' \in F[\mathrm{id}_\Phi, M/x]$	by Lemma 6.20(1) on $\mathcal{P}, \mathcal{F}_1, \mathcal{Q}_2$
$\mathcal{R}_2 :: \Phi; \cdot \vdash V \in F[\mathrm{id}_\Phi, M/x]$	by ih.(1) on $\mathcal{F}_2, \mathcal{R}_1$
$\mathcal{F}_3 :: \Phi; \cdot \vdash D[V/\mathbf{y}] = D'/\Psi; \Delta$	by definition of $\mathcal{D}_3$
$\mathcal{F}_4 :: \Phi \vdash D' \hookrightarrow \psi; \delta$	by definition of $\mathcal{D}_3$
$\mathcal{P}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; V/\mathbf{y} \in \Phi; \mathbf{y} \in F[\mathrm{id}_\Phi, M/x]$	by smeta on $\mathcal{P}_1, \mathcal{R}_2$
$\mathcal{R}_3 :: \Phi; \cdot \vdash D' \in \Psi; \Delta$	by Lemma 6.20(2) on $\mathcal{P}_1, \mathcal{F}_3, \mathcal{E}_3$
$\mathcal{R}_4 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi; \Delta$	by ih.(2) on $\mathcal{F}_4, \mathcal{R}_3$
$\mathcal{R} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; V/\mathbf{y}, \delta \in \Phi, \Psi; \mathbf{y} \in F[\mathrm{id}_\Phi, M/x], \Delta$	by Lemma 6.37(2) on $\mathcal{R}_4, \mathcal{R}_2$

$$\textbf{Case: } \mathcal{D} = \dfrac{\begin{array}{ccc}\mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ \Phi \vdash P \hookrightarrow \lambda \rho^L.\, P' & \Phi \vdash P'[\mathrm{id}_\Phi, \rho'/\rho] \hookrightarrow V & \Phi \vdash D[V/\mathbf{y}] \hookrightarrow \psi; \delta\end{array}}{\Phi \vdash \mathbf{y} \in F[\mathrm{id}_\Phi, \rho'/\rho] = P\ \rho, D \hookrightarrow \psi; V/\mathbf{y}, \delta}\ \mathsf{ev\_app}$$

$\mathcal{E} :: \Phi; \cdot \vdash \mathbf{y} \in F[\mathrm{id}_\Phi, \rho'/\rho] = P\ \rho, D \in \Psi; \mathbf{y} \in F[\mathrm{id}_\Phi, \rho'/\rho], \Delta$ by assumption
$\mathcal{E}_1 :: \Phi; \cdot \vdash P \in \mathbf{\Pi}\rho^L. F$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \rho'^L \in \Phi$ by inversion on $\mathcal{E}$
$\mathcal{E}_3 :: [\![\Phi]\!] \vdash \rho' \equiv \rho$ by inversion on $\mathcal{E}$
$\mathcal{E}_4 :: \Phi; \mathbf{y} \in F[\mathrm{id}_\Phi, \rho'/\rho] \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash \lambda\rho^L. P' \in \mathbf{\Pi}\rho^L. F$ by i.h.(1) on $\mathcal{D}_1, \mathcal{E}_1$
$\mathcal{Q}_2 :: \Phi, \rho^L; \cdot \vdash P' \in F$ by inversion on $\mathcal{Q}_1$
$\mathcal{F}_1 :: \Phi; \cdot \vdash P'[\mathrm{id}_\Phi, \rho'/\rho] = P''$ by definition of $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi \vdash P'' \hookrightarrow V$ by definition of $\mathcal{D}_2$
$\mathcal{P}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \cdot : \Phi; \cdot$ by Lemma 6.22
$\mathcal{P} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \rho'/\rho; \cdot \in \Phi, \rho^L; \cdot$ by sblock on $\mathcal{E}_2, \mathcal{E}_3, \mathcal{P}_1$
$\mathcal{R}_1 :: \Phi; \cdot \vdash P'' \in F[\mathrm{id}_\Phi, \rho'/\rho]$ by Lemma 6.20(1) on $\mathcal{P}, \mathcal{F}_1, \mathcal{Q}_2$
$\mathcal{R}_2 :: \Phi; \cdot \vdash V \in F[\mathrm{id}_\Phi, \rho'/\rho]$ by ih.(1) on $\mathcal{F}_2, \mathcal{R}_1$
$\mathcal{F}_3 :: \Phi; \cdot \vdash D[V/\mathbf{y}] = D'/\Psi; \Delta$ by definition of $\mathcal{D}_3$
$\mathcal{F}_4 :: \Phi \vdash D' \hookrightarrow \psi; \delta$ by definition of $\mathcal{D}_3$
$\mathcal{P}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; V/\mathbf{y} \in \Phi; \mathbf{y} \in F[\mathrm{id}_\Phi, \rho'/\rho]$ by smeta on $\mathcal{R}_2, \mathcal{P}_1$
$\mathcal{R}_3 :: \Phi; \cdot \vdash D' \in \Psi; \Delta$ by Lemma 6.20(2) on $\mathcal{P}_2, \mathcal{F}_3, \mathcal{E}_4$
$\mathcal{R}_4 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi; \Delta$ by ih.(2) on $\mathcal{F}_4, \mathcal{R}_3$
$\mathcal{R} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; V/\mathbf{y}, \delta \in \Phi, \Psi; \mathbf{y} \in F[\mathrm{id}_\Phi, \rho'/\rho], \Delta$ by Lemma 6.37(2) on $\mathcal{R}_4, \mathcal{R}_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Phi, \rho^L \vdash D \hookrightarrow \psi; \delta\end{array}}{\Phi \vdash \nu\ \rho^L. D \hookrightarrow \lambda\rho. \psi; \lambda\rho^L. \delta}\ \mathsf{ev\_new}$

$\mathcal{E} :: \Phi; \cdot \vdash \nu\ \rho^L. D \in \Pi\rho^L. (\Psi; \Delta)$ by assumption
$\mathcal{E}_1 :: \Phi, \rho^L; \cdot \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{E}$
$\mathcal{R}_1 :: \Phi, \rho^L; \cdot \vdash \mathrm{id}_\Phi, \rho/\rho, \psi; \delta \in \Phi, \rho^L, \Psi; \Delta$ by i.h.(2) on $\mathcal{D}_1, \mathcal{E}_1$
$\mathcal{R}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot \Phi; \cdot$ by Lemma 6.11 (2)
$\mathcal{Q}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi'; \delta' \in \Phi, \Psi'; \Delta'$ by Lemma 6.7(1) on $\mathcal{R}_1, \mathcal{R}_2$
$\mathcal{Q}_2 :: \psi'; \delta' = \lambda\rho^L. (\psi; \delta)$ by Lemma 6.7(1) on $\mathcal{R}_1, \mathcal{R}_2$
$\mathcal{Q}_3 :: \Psi'; \Delta' = \Pi\rho^L. (\Psi; \Delta)$ by Lemma 6.7(1) on $\mathcal{R}_1, \mathcal{R}_2$

Case: $\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Phi \vdash P \hookrightarrow \langle V_1, V_2\rangle & \Phi \vdash D[V_1/\mathbf{x}] \hookrightarrow \psi; \delta\end{array}}{\Phi \vdash \mathbf{x} = \pi_1\ P, D \hookrightarrow \psi; V_1/\mathbf{x}, \delta}\ \mathsf{ev\_fst}$

$\mathcal{E} :: \Phi; \cdot \vdash \mathbf{x} = \pi_1\ P, D \in \Psi; \mathbf{x} \in F_1, \Delta$ by assumption
$\mathcal{E}_1 :: \Phi; \cdot \vdash P \in F_1 \wedge F_2$ by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \Phi; \mathbf{x} \in F_1 \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash \langle V_1, V_2\rangle \in F_1 \wedge F_2$ by i.h.(1) on $\mathcal{D}_1, \mathcal{E}_1$
$\mathcal{Q}_2 :: \Phi; \cdot \vdash V_1 \in F_1$ by inversion on $\mathcal{Q}_1$
$\mathcal{Q}_3 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot : \Phi; \cdot$ by Lemma 6.22
$\mathcal{Q}_4 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; V/\mathbf{x} : \Phi; \mathbf{x} \in F_1$ by smeta on $\mathcal{Q}_3$
$\mathcal{R}_1 :: \Phi; \cdot \vdash D[V_1/\mathbf{x}] = D'$ by definition of $\mathcal{D}_2$

$\mathcal{R}_2 :: \Phi; \cdot \vdash D' \hookrightarrow \psi; \delta$ — by definition of $\mathcal{D}_2$
$\mathcal{P}_1 :: \Phi; \cdot \vdash D' \in \Psi; \Delta$ — by Lemma 6.20(2) on $\mathcal{R}_1, \mathcal{Q}_4, \mathcal{E}_2$
$\mathcal{P}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi; \Delta$ — by i.h.(2) on $\mathcal{P}_1, \mathcal{R}_2$
$\mathcal{R} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; V_1/\mathbf{x}, \delta \in \Phi, \Psi; \mathbf{x} \in F_1, \Delta$ — by Lemma 6.37(2) on $\mathcal{Q}_2, \mathcal{P}_2$

Case: $$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1\\ \Phi \vdash P \hookrightarrow \langle V_1, V_2\rangle\end{array} \qquad \begin{array}{c}\mathcal{D}_2\\ \Phi \vdash D[V_2/\mathbf{x}] \hookrightarrow \psi'; \delta'\end{array}}{\Phi \vdash \mathbf{x} = \pi_2\ P, D \hookrightarrow \psi'; V_2/\mathbf{x}, \delta'}\ \mathsf{ev\_snd}$$

$\mathcal{E} :: \Phi; \cdot \vdash \mathbf{x} = \pi_2\ P, D \in \Psi; \mathbf{x} \in F_2, \Delta$ — by assumption
$\mathcal{E}_1 :: \Phi; \cdot \vdash P \in F_1 \wedge F_2$ — by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \Phi; \mathbf{x} \in F_2 \vdash D \in \Psi; \Delta$ — by inversion on $\mathcal{E}$
$\mathcal{Q}_1 :: \Phi; \cdot \vdash \langle V_1, V_2\rangle \in F_1 \wedge F_2$ — by i.h.(1) on $\mathcal{D}_1, \mathcal{E}_1$
$\mathcal{Q}_2 :: \Phi; \cdot \vdash V_2 \in F_2$ — by inversion on $\mathcal{Q}_1$
$\mathcal{Q}_3 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot : \Phi; \cdot$ — by Lemma 6.22
$\mathcal{Q}_4 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; V/\mathbf{x} : \Phi; \mathbf{x} \in F_2$ — by smeta on $\mathcal{Q}_3$
$\mathcal{R}_1 :: \Phi; \cdot \vdash D[V_2/\mathbf{x}] = D'$ — by definition of $\mathcal{D}_2$
$\mathcal{R}_2 :: \Phi; \cdot \vdash D' \hookrightarrow \psi; \delta$ — by definition of $\mathcal{D}_2$
$\mathcal{P}_1 :: \Phi; \cdot \vdash D' \in \Psi; \Delta$ — by Lemma 6.20(2) on $\mathcal{R}_1, \mathcal{Q}_4, \mathcal{E}_2$
$\mathcal{P}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi; \Delta$ — by i.h.(2) on $\mathcal{P}_1, \mathcal{R}_2$
$\mathcal{R} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; V_2/\mathbf{x}, \delta \in \Phi, \Psi; \mathbf{x} \in F_2, \Delta$ — by Lemma 6.37(2) on $\mathcal{Q}_2, \mathcal{P}_2$

3. **Case:** $$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1\\ \Phi \vdash P[\psi''; \delta] \hookrightarrow V\end{array}}{\Phi \vdash (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P)) \hookrightarrow V}\ \mathsf{ev\_yes}$$

There exists a ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$ — by side condition
$\mathcal{F} :: \Phi; \cdot \vdash \psi; \delta : \Psi; \Delta$ — by assumption
$\mathcal{E} :: \Psi; \Delta \vdash \Omega, (\Psi' \triangleright \psi' \mapsto P) \in F$ — by assumption
$\mathcal{E}_1 :: \Psi'; [\psi']\Delta \vdash \psi'; \mathrm{id}_\Delta : \Psi; \Delta$ — by inversion on $\mathcal{E}$
$\mathcal{E}_2 :: \Psi'; [\psi']\Delta \vdash P \in F[\psi']$ — by inversion on $\mathcal{E}$
$\mathcal{G}_1 :: \Phi; \cdot \vdash \psi''; \delta : \Psi'; [\psi']\Delta$ — by type correctness of side condition
$\mathcal{P}_1 :: \Phi; \cdot \vdash P[\psi''; \delta] \in F[\psi'][\psi'']$ — by Lemma 6.20(1) on $\mathcal{E}_2, \mathcal{G}_1$
$\mathcal{P}_1 :: \Phi; \cdot \vdash P[\psi''; \delta] \in F[\psi]$ — by Definition of ψ''
$\mathcal{R} :: \Phi; \cdot \vdash V \in F[\psi]$ — by i.h.(1) on $\mathcal{D}_1$, $\mathcal{P}_1$

Case: $$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}\\ \Phi \vdash (\psi; \delta) \sim \Omega \hookrightarrow V\end{array}}{\Phi \vdash (\psi; \delta) \sim (\Omega, (\Psi \triangleright \psi' \mapsto P)) \hookrightarrow V}\ \mathsf{ev\_no}$$

There is no ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$ — by side condition
$\mathcal{F} :: \Phi; \cdot \vdash \psi; \delta : \Psi; \Delta$ — by assumption
$\mathcal{E} :: \Psi; \Delta \vdash \Omega, (\Psi \triangleright \psi' \mapsto P) \in F$ — by assumption

$\mathcal{E}_1 :: \Psi; \Delta \vdash \Omega \in F$ by inversion on $\mathcal{E}$
$\mathcal{R} :: \Phi; \cdot \vdash V \in F[\psi]$ by i.h.(3) on $\mathcal{D}_1$, $\mathcal{F}$, $\mathcal{E}_1$

$\square$

Appendix C

Realizability

Theorem 7.3 *(Local type preservation for small-step semantics)*

If $\mathcal{D} :: \vdash S \in F$
and $\mathcal{E} :: S \implies S'$
then $\vdash S' \in F$*.*

Proof: by case analysis on $\mathcal{E}$

Case: $\mathcal{E} = \mathsf{trlet} :: \Phi; C \triangleright \textbf{let } D \textbf{ in } P \implies \Phi; C, \textbf{let } \bullet \textbf{ in } P \triangleright D$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \textbf{let } D \textbf{ in } P) \in F$	by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in F_1 \Rightarrow F$	by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \textbf{let } D \textbf{ in } P \in F_1$	by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \cdot \vdash D \in \Psi; \Delta$	by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi, \Psi; \Delta \vdash P \in F_1$	by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, \textbf{let } \bullet \textbf{ in } P \in (\Psi; \Delta) \Rightarrow F$	by tclet on $\mathcal{D}_1, \mathcal{F}_2$
$\mathcal{Q} :: \vdash (\Phi; C, \textbf{let } \bullet \textbf{ in } P \triangleright D) \in F$	by tsdec on $\mathcal{P}, \mathcal{F}_1$

Case: $\mathcal{E} = \mathsf{trletC} :: \Phi; C, \textbf{let } \bullet \textbf{ in } P \triangleright (\psi; \delta) \implies \Phi; C \triangleright P[\mathrm{id}_\Phi, \psi; \delta]$

$\mathcal{D} :: \vdash (\Phi; C, \textbf{let } \bullet \textbf{ in } P \triangleright (\psi; \delta)) \in F$	by assumption
$\mathcal{D}_1 :: \Phi \vdash C, \textbf{let } \bullet \textbf{ in } P \in (\Psi; \Delta) \Rightarrow F$	by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi; \Delta$	by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in F_1 \Rightarrow F$	by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi, \Psi; \Delta \vdash P \in F_1$	by inversion on $\mathcal{D}_1$
$\mathcal{P} :: \Phi; \cdot \vdash P[\mathrm{id}_\Phi, \psi; \delta] \in F_1[\mathrm{id}_\Phi, \psi]$	by Lemma 6.20(1) on $\mathcal{D}_2$, $\mathcal{F}_2$
$\mathcal{P} :: \Phi; \cdot \vdash P[\mathrm{id}_\Phi, \psi; \delta] \in F_1$	since $\Phi; \cdot \vdash F$ formula
$\mathcal{Q} :: \vdash (\Phi; C \triangleright P[\mathrm{id}_\Phi, \psi; \delta]) \in F$	by tsprg on $\mathcal{F}_1, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trpair} :: \Phi; C \triangleright \langle P_1, P_2 \rangle \implies \Phi; C, \langle \bullet, P_2 \rangle \triangleright P_1$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \langle P_1, P_2 \rangle) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in F_1 \wedge F_2 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \langle P_1, P_2 \rangle \in F_1 \wedge F_2$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \cdot \vdash P_1 \in F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi; \cdot \vdash P_2 \in F_2$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, \langle \bullet, P_2 \rangle \in F_1 \Rightarrow F$ by tcpair on $\mathcal{D}_1, \mathcal{F}_2$
$\mathcal{Q} :: \vdash (\Phi; C, \langle \bullet, P_2 \rangle \triangleright P_1) \in F$ by tsprg on $\mathcal{P}$, $\mathcal{F}_1$

Case: $\mathcal{E} = \mathsf{trpairC} :: \Phi; C, \langle \bullet, P_2 \rangle \triangleright V \implies \Phi; C \triangleright \langle V, P_2 \rangle$

$\mathcal{D} :: \vdash (\Phi; C, \langle \bullet, P_2 \rangle \triangleright V) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, \langle \bullet, P_2 \rangle \in F_1 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash V \in F_1$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in F_1 \wedge F_2 \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi; \cdot \vdash P_2 \in F_2$ by inversion on $\mathcal{D}_1$
$\mathcal{P} :: \Phi; \cdot \vdash \langle V, P_2 \rangle \in F_1 \wedge F_2$ by $\mathsf{R}\wedge$ on $\mathcal{D}_2, \mathcal{F}_2$
$\mathcal{Q} :: \vdash (\Phi; C \triangleright \langle V, P_2 \rangle) \in F$ by tsprg on $\mathcal{F}_1, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trmix} :: \Phi; C \triangleright \langle V_1, P_2 \rangle \implies \Phi; C, \langle V_1, \bullet \rangle \triangleright P_2$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \langle V_1, P_2 \rangle) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in F_1 \wedge F_2 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \langle V_1, P_2 \rangle \in F_1 \wedge F_2$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \cdot \vdash V_1 \in F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi; \cdot \vdash P_2 \in F_2$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, \langle V_1, \bullet \rangle \in F_2 \Rightarrow F$ by tcmix on $\mathcal{D}_1, \mathcal{F}_1$
$\mathcal{Q} :: \vdash (\Phi; C, \langle V_1, \bullet \rangle \triangleright P_2) \in F$ by tsprg on $\mathcal{P}, \mathcal{F}_2$

Case: $\mathcal{E} = \mathsf{trmixC} :: \Phi; C, \langle V_1, \bullet \rangle \triangleright V \implies \Phi; C \triangleright \langle V_1, V \rangle$

$\mathcal{D} :: \vdash (\Phi; C, \langle V_1, \bullet \rangle \triangleright V) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, \langle V_1, \bullet \rangle \in F_2 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash V \in F_2$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in F_1 \wedge F_2 \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi; \cdot \vdash V_1 \in F_1$ by inversion on $\mathcal{D}_1$
$\mathcal{P} :: \Phi; \cdot \vdash \langle V_1, V \rangle \in F_1 \wedge F_2$ by $\mathsf{R}\forall$ on $\mathcal{F}_2, \mathcal{D}_2$
$\mathcal{Q} :: \vdash (\Phi, C \triangleright \langle V_1, V \rangle) \in F$ by tsprg on $\mathcal{F}_1, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trfst} :: \Phi; C \triangleright \mathbf{x} \in F_1 = \pi_1\ P, D \implies \Phi; C, \mathbf{x} \in F_1 = \pi_1\ \bullet, D \triangleright P$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \mathbf{x} \in F_1 = \pi_1\ P, D) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}$

$\mathcal{D}_2 :: \Phi; \cdot \vdash \mathbf{x} \in F_1 = \pi_1\ P, D \in \Psi; \mathbf{x} \in F_1, \Delta$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \cdot \vdash P \in F_1 \wedge F_2$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi; \mathbf{x} \in F_1 \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, (\mathbf{x} \in F_1 = \pi_1\ \bullet, D) \in F_1 \wedge F_2 \Rightarrow F$ by tcfst on $\mathcal{D}_1, \mathcal{F}_2$
$\mathcal{Q} :: \vdash (\Phi; C, (\mathbf{x} \in F_1 = \pi_1\ \bullet, D) \triangleright P) \in F$ by tsdec on $\mathcal{P}, \mathcal{F}_1$

Case: $\mathcal{E} = \mathsf{trfstC} :: \Phi; C, \mathbf{x} \in F_1 = \pi_1\ \bullet, D \triangleright \langle V_1, V_2 \rangle \implies \Phi; C, (\bullet; V_1/\mathbf{x}, \bullet) \triangleright D[V_1/\mathbf{x}]$

$\mathcal{D} :: \vdash (\Phi; C, \mathbf{x} \in F_1 = \pi_1\ \bullet, D \triangleright \langle V_1, V_2 \rangle) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, \mathbf{x} \in F_1 = \pi_1\ \bullet, D \in F_1 \wedge F_2 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \langle V_1, V_2 \rangle \in F_1 \wedge F_2$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi; \mathbf{x} \in F_1 \vdash D : \Psi; \Delta$ by inversion on $\mathcal{D}_1$
$\mathcal{G}_1 :: \Phi; \cdot \vdash V_1 \in F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{P}_1 :: \Phi \vdash C, (\bullet; V_1/\mathbf{x}, \bullet) \in (\Psi; \Delta) \Rightarrow F$ by tcmeta on $\mathcal{F}_1, \mathcal{G}_1$
$\mathcal{P}_2 :: \Phi; \cdot \vdash D[V_1/\mathbf{x}] : \Psi; \Delta$ by Lemma 6.20 (2) on $\mathcal{F}_2$
$\mathcal{Q} :: \vdash (\Phi; C, (\bullet; V_1/\mathbf{x}, \bullet) \triangleright D[V_1/\mathbf{x}]) \in F$ by tsdec on $\mathcal{P}_1, \mathcal{P}_2$

Case: $\mathcal{E} = \mathsf{trsnd} :: \Phi; C \triangleright \mathbf{x} \in F_2 = \pi_2\ P, D \implies \Phi; C, \mathbf{x} \in F_2 = \pi_2\ \bullet, D \triangleright P$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \mathbf{x} \in F_2 = \pi_2\ P, D) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in (\Psi; \mathbf{x} \in F_2, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \mathbf{x} \in F_2 = \pi_2\ P, D \in \Psi; \mathbf{x} \in F_2, \Delta$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \cdot \vdash P \in F_1 \wedge F_2$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi; \mathbf{x} \in F_2 \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, (\mathbf{x} \in F_2 = \pi_2\ \bullet, D) \in F_1 \wedge F_2 \Rightarrow F$ by tcsnd on $\mathcal{D}_1, \mathcal{F}_2$
$\mathcal{Q} :: \vdash (\Phi; C, (\mathbf{x} \in F_2 = \pi_2\ \bullet, D) \triangleright P) \in F$ by tsdec on $\mathcal{P}, \mathcal{F}_1$

Case: $\mathcal{E} = \mathsf{trsndC} :: \Phi; C, \mathbf{x} \in F_2 = \pi_2\ \bullet, D \triangleright \langle V_1, V_2 \rangle \implies \Phi; C, (\bullet; V_2/\mathbf{x}, \bullet) \triangleright D[V_2/\mathbf{x}]$

$\mathcal{D} :: \vdash (\Phi; C, \mathbf{x} \in F_2 = \pi_2\ \bullet, D \triangleright \langle V_1, V_2 \rangle) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, \mathbf{x} \in F_2 = \pi_2\ \bullet, D \in F_1 \wedge F_2 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \langle V_1, V_2 \rangle \in F_1 \wedge F_2$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in (\Psi; \mathbf{x} \in F_2, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi; \mathbf{x} \in F_2 \vdash D : \Psi; \Delta$ by inversion on $\mathcal{D}_1$
$\mathcal{G}_1 :: \Phi; \cdot \vdash V_2 \in F_2$ by inversion on $\mathcal{D}_2$
$\mathcal{P}_1 :: \Phi \vdash C, (\bullet; V_2/\mathbf{x}, \bullet) \in (\Psi; \Delta) \Rightarrow F$ by tcmeta on $\mathcal{F}_1, \mathcal{G}_1$
$\mathcal{P}_2 :: \Phi; \cdot \vdash D[V_2/\mathbf{x}] : \Psi; \Delta$ by Lemma 6.20 (2) on $\mathcal{F}_2$
$\mathcal{Q} :: \vdash (\Phi; C, (\bullet; V_2/\mathbf{x}, \bullet) \triangleright D[V_2/\mathbf{x}]) \in F$ by tsdec on $\mathcal{P}_1, \mathcal{P}_2$

Case: $\mathcal{E} = \mathsf{trinx} :: \Phi; C \triangleright \langle M, P \rangle \implies \Phi; C, \langle M, \bullet \rangle \triangleright P$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \langle M, P \rangle) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in \exists x : A.\, F_1 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \langle M, P \rangle \in \exists x : A.\, F_1$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: [\![\Phi]\!] \vdash M : A$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi; \cdot \vdash P \in F_1[\mathrm{id}_\Phi, M/x]$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, \langle M, \bullet \rangle \in F_1[\mathrm{id}_\Phi, M/x] \Rightarrow F$ by tcinx on $\mathcal{D}_1, \mathcal{F}_1$
$\mathcal{Q} :: \vdash (\Phi; C, \langle M, \bullet \rangle \triangleright P) \in F$ by tsprg on $\mathcal{P}, \mathcal{F}_2$

Case: $\mathcal{E} = \mathsf{trinxC} :: \Phi; C, \langle M, \bullet \rangle \triangleright V \implies \Phi; C \triangleright \langle M, V \rangle$

$\mathcal{D} :: \vdash (\Phi; C, \langle M, \bullet \rangle \triangleright V) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, \langle M, \bullet \rangle \in F_1[\mathrm{id}_\Phi, M/x] \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash V \in F_1[\mathrm{id}_\Phi, M/x]$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in \exists x : A.\, F_1 \Rightarrow F$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: [\![\Phi]\!] \vdash M : A$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi; \cdot \vdash \langle M, V \rangle \in \exists x : A.\, F_1$ by $\mathsf{R}\exists$ on $\mathcal{F}_2, \mathcal{D}_2$
$\mathcal{Q} :: \vdash (\Phi; C \triangleright \langle M, V \rangle) \in F$ by tsprg on $\mathcal{F}_1, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trsplit} :: \Phi; C \triangleright \langle x : A, \mathbf{y} \in F_1 \rangle = P, D \implies \Phi; C, \langle x : A, \mathbf{y} \in F_1 \rangle = \bullet, D \triangleright P$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \langle x : A, \mathbf{y} \in F_1 \rangle = P, D) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in (x : A, \Psi; \mathbf{y} \in F_1, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash (\langle x : A, \mathbf{y} \in F_1 \rangle = P, D) \in x : A, \Psi; \mathbf{y} \in F_1, \Delta$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \cdot \vdash P \in \exists x : A.\, F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \Phi, x : A; \mathbf{y} \in F_1 \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, (\langle x : A, \mathbf{y} \in F_1 \rangle = \bullet, D) \in \exists x : A.\, F_1 \Rightarrow F$ by $\mathsf{tcsplit}$ on $\mathcal{D}_1, \mathcal{F}_2$
$\mathcal{Q} :: \vdash (\Phi; C, (\langle x : A, \mathbf{y} \in F_1 \rangle = \bullet, D) \triangleright P) \in F$ by tsprg on $\mathcal{P}, \mathcal{F}_1$

Case: $\mathcal{E} = \mathsf{trsplitC} :: \Phi; C, \langle x : A, \mathbf{y} \in F \rangle = \bullet, D \triangleright \langle M, V \rangle \implies \Phi; C, (M/x, \bullet; V/\mathbf{y}, \bullet) \triangleright D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}]$

$\mathcal{D} :: \vdash (\Phi; C, \langle x : A, \mathbf{y} \in F \rangle = \bullet, D \triangleright \langle M, V \rangle) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, (\langle x : A, \mathbf{y} \in F_1 \rangle = \bullet, D) \in \exists x : A.\, F_1 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \langle M, V \rangle \in \exists x : A.\, F_1$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in (x : A, \Psi; \mathbf{y} \in F_1, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi, x : A; \mathbf{y} \in F_1 \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{D}_1$
$\mathcal{G}_1 :: [\![\Phi]\!] \vdash M : A$ by inversion on $\mathcal{D}_2$
$\mathcal{G}_2 :: \Phi; \cdot \vdash V \in F[\mathrm{id}_\Phi, M/x]$ by inversion on $\mathcal{D}_2$
$\mathcal{P}_1 :: \Phi \vdash C, (M/x, \bullet; V/\mathbf{y}, \bullet) \in [\mathrm{id}_\Phi, M/x](\Psi; \Delta) \Rightarrow F$ by $\mathsf{tcsubst}$ on $\mathcal{F}_1, \mathcal{G}_1, \mathcal{G}_2$
$\mathcal{P}_2 :: \Phi; \cdot \vdash D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}] \in [\mathrm{id}_\Phi, M/x](\Psi; \Delta)$ by Lemma 6.20(2) on $\mathcal{F}_2$
$\vdash (\Phi; C, (M/x, \bullet; V/\mathbf{y}, \bullet) \triangleright D[\mathrm{id}_\Phi, M/x; V/\mathbf{y}]) \in F$ by tsdec on $\mathcal{P}_1, \mathcal{P}_2$

Case: $\mathcal{E} = \mathsf{trsubst} :: \Phi; C, (M/x, \bullet; V/\mathbf{y}, \bullet) \triangleright (\psi; \delta) \implies \Phi; C \triangleright (M/x, \psi; V/\mathbf{y}, \delta)$

$\mathcal{D} :: \vdash (\Phi; C, (M/x, \bullet; V/\mathbf{y}, \bullet) \triangleright (\psi; \delta)) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, (M/x, \bullet; V/\mathbf{y}, \bullet) \in (\Psi'; \Delta') \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi'; \Delta'$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Psi'; \Delta' = [\mathrm{id}_\Phi, M/x](\Psi; \Delta)$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi \vdash C \in (x : A, \Psi; \mathbf{y} \in F_1, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_3 :: [\![\Phi]\!] \vdash M : A$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_4 :: \Phi; \cdot \vdash V \in F_1[\mathrm{id}_\Phi, M/x]$ by inversion on $\mathcal{D}_1$
$\mathcal{P} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, M/x, \psi; V/\mathbf{y}, \delta \in \Phi, x : A, \Psi; \mathbf{y} \in F_1, \Delta$ by Lemma 6.37(1) on $\mathcal{D}_2, \mathcal{F}_3, \mathcal{F}_4$
$\mathcal{Q} :: \vdash (\Phi; C \triangleright M/x, \psi; V/\mathbf{y}, \delta) \in F$ by tssub on $\mathcal{F}_2, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trrec} :: \Phi; C \triangleright \mu\mathbf{x} \in F.\, P \implies \Phi; C \triangleright P[\mu\mathbf{x} \in F.\, P/\mathbf{x}]$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \mu\mathbf{x} \in F_1.\, P) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in F_1 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \mu\mathbf{x} \in F_1.\, P \in F_1$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \mathbf{x} \in F_1 \vdash P \in F_1$ by inversion on Rctx
$\mathcal{P} :: \Phi; \cdot \vdash P[\mu\mathbf{x} \in F_1.\, P/\mathbf{x}] \in F_1$ by Lemma 6.20(1) on $\mathcal{F}_1$
$\mathcal{Q} :: \vdash (\Phi; C \triangleright P[\mu\mathbf{x} \in F_1.\, P/\mathbf{x}]) \in F$ by tsprg on $\mathcal{D}_1, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trempty} :: \Phi; C \triangleright \cdot \implies \Phi; C \triangleright \cdot; \cdot$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \cdot) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in (\cdot; \cdot) \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \cdot \in \cdot; \cdot$ by inversion on $\mathcal{D}$
$\mathcal{P} :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot : \Phi; \cdot$ by Lemma 6.22
$\mathcal{Q} :: \vdash (\Phi; C \triangleright \cdot; \cdot) \in \cdot; \cdot$ by tssub on $\mathcal{D}_1, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trApp} :: \Phi; C \triangleright \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = P\ M, D \implies \Phi; C, \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D \triangleright P$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = P\ M, D) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in (\Psi; \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x], \Delta) \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = P\ M, D \in \Psi; \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x], \Delta$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \cdot \vdash P \in \forall x : A.\, F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: [\![\Phi]\!] \vdash M : A$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_3 :: \Phi; \mathbf{y} \in F_1[\mathrm{id}_\Psi, M/x] \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, (\mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D) \in \forall x : A.\, F_1 \Rightarrow F$ by tcApp on $\mathcal{D}_1, \mathcal{F}_2, \mathcal{F}_3$
$\mathcal{Q} :: \vdash (\Phi; C, (\mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D) \triangleright P) \in F$ by tsprg on $\mathcal{P}, \mathcal{F}_1$

Case: $\mathcal{E} = \mathsf{trAppC} :: \Phi; C, \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D \triangleright \Lambda x : A.\, P \implies \Phi; C, \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet, D \triangleright P[\mathrm{id}_\Phi, M/x]$

$\mathcal{D} :: \vdash (\Phi; C, \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D \triangleright \Lambda x : A.\, P) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, (\mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet\ M, D) \in \forall x : A.\, F_1 \Rightarrow F$ by inversion on $\mathcal{D}$

$\mathcal{D}_2 :: \Phi; \cdot \vdash \Lambda x : A.\, P \in \forall x : A.\, F_1$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in (\Psi; \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x], \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: [\![\Phi]\!] \vdash M : A$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_3 :: \Phi; \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{D}_1$
$\mathcal{G}_1 :: \Phi, x : A; \cdot \vdash P \in F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{G}_2 :: \Phi; \cdot \vdash P[\mathrm{id}_\Phi, M/x] \in F_1[\mathrm{id}_\Phi, M/x]$ by Lemma 6.20(1) on $\mathcal{G}_1$
$\mathcal{P} :: \Phi \vdash C, \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet, D \in F_1[\mathrm{id}_\Phi, M/x] \Rightarrow F$ by $\mathsf{tcassign}$ on $\mathcal{F}_1, \mathcal{F}_3$
$\mathcal{Q} :: \vdash (\Phi; C, \mathbf{y} \in F_1[\mathrm{id}_\Phi, M/x] = \bullet, D \triangleright P[\mathrm{id}_\Phi, M/x]) \in F$ by tsprg on $\mathcal{P}, \mathcal{G}_2$

Case: $\mathcal{E} = \mathsf{trapp} :: \Phi; C \triangleright \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = P\ \rho', D \implies \Phi; C, \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D \triangleright P$

$\mathcal{D} :: \vdash (\Phi; C \triangleright \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = P\ \rho', D) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in (\Psi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho], \Delta) \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = P\ \rho', D \in \Psi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho], \Delta$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi; \cdot \vdash P \in \mathbf{\Pi}\rho^L.\, F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \rho'^L \in \Phi$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_3 :: [\![\Phi]\!] \vdash \rho' \equiv \rho$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_4 :: \Phi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi \vdash C, (\mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D) \in \mathbf{\Pi}\rho^L.\, F_1 \Rightarrow F$ by tcapp on $\mathcal{D}_2, \mathcal{F}_3, \mathcal{F}_4$
$\mathcal{Q} :: \vdash (\Phi; C, (\mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D) \triangleright P) \in F$ by tsprg on $\mathcal{P}, \mathcal{F}_1$

Case: $\mathcal{E} = \mathsf{trappC} :: \Phi; C, \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D \triangleright \lambda\rho^L.\, P \implies$
$\Phi; C, (\mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet, D) \triangleright P[\mathrm{id}_\Phi, \rho'/\rho]$

$\mathcal{D} :: \vdash (\Phi; C, \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D \triangleright \lambda\rho^L.\, P) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet\ \rho', D \in \mathbf{\Pi}\rho^L.\, F_1 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \lambda\rho^L.\, P \in \mathbf{\Pi}\rho^L.\, F_1$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in (\Psi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho], \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: [\![\Phi]\!] \vdash \rho \equiv \rho'$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_3 :: \Phi; \mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] \vdash D \in \Psi; \Delta$ by inversion on $\mathcal{D}_1$
$\mathcal{G}_1 :: \Phi, \rho^L; \cdot \vdash P \in F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{G}_2 :: \Phi; \cdot \vdash P[\mathrm{id}_\Phi, \rho'/\rho] \in F_1[\mathrm{id}_\Phi, \rho'/\rho]$ by Lemma 6.20(1) on $\mathcal{G}_1$
$\mathcal{P} :: \Phi \vdash C, (\mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet, D) \in F_1[\mathrm{id}_\Phi, \rho'/\rho] \Rightarrow F$ by $\mathsf{tcassign}$ on $\mathcal{F}_1, \mathcal{F}_3$
$\mathcal{Q} :: \vdash (\Phi; C, (\mathbf{x} \in F_1[\mathrm{id}_\Phi, \rho'/\rho] = \bullet, D) \triangleright P[\mathrm{id}_\Phi, \rho'/\rho]) \in F$ by tsprg on $\mathcal{P}, \mathcal{G}_2$

Case: $\mathcal{E} = \mathsf{trassign} :: \Phi; C, \mathbf{x} \in F_1 = \bullet, D \triangleright V \implies \Phi; C, (\bullet; V/\mathbf{x}, \bullet) \triangleright D[V/\mathbf{x}]$

$\mathcal{D} :: \vdash (\Phi; C, \mathbf{x} \in F_1 = \bullet, D \triangleright V) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, (\mathbf{x} \in F_1 = \bullet, D) \in F_1 \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash V \in F_1$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi; \mathbf{x} \in F_1 \vdash D : \Psi; \Delta$ by inversion on $\mathcal{D}_2$
$\mathcal{G} :: \Phi; \cdot \vdash D[V/\mathbf{x}] : \Psi; \Delta$ by Lemma 6.20(2) on $\mathcal{F}_2$

$\mathcal{P} :: \Phi \vdash C, (\bullet; V/\mathbf{x}, \bullet) \in (\Psi; \Delta) \Rightarrow F$ by tcmeta on $\mathcal{F}_1, \mathcal{D}_2$
$\mathcal{Q} ::\vdash (\Phi; C, (\bullet; V/\mathbf{x}, \bullet) \triangleright D[V/\mathbf{x}]) \in F$ by tsdec on $\mathcal{P}, \mathcal{G}$

Case: $\mathcal{E} = \mathsf{trmeta} :: \Phi; C, (\bullet; V/\mathbf{x}, \bullet) \triangleright (\psi; \delta) \implies \Phi; C \triangleright (\psi; V/\mathbf{x}, \delta)$

$\mathcal{D} ::\vdash (\Phi; C, (\bullet; V/\mathbf{x}, \bullet) \triangleright (\psi; \delta)) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C, (\bullet; V/\mathbf{x}, \bullet) \in (\Psi; \Delta) \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; \delta \in \Phi, \Psi; \Delta$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in (\Psi; \mathbf{x} \in F_1, \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{F}_2 :: \Phi; \cdot \vdash V \in F_1$ by inversion on $\mathcal{D}_1$
$\mathcal{P} :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi; V/\mathbf{x}, \delta \in \Phi, \Psi; \mathbf{x} \in F_1, \Delta$ by Lemma 6.37(2) on $\mathcal{D}_2, \mathcal{F}_2$
$\mathcal{Q} ::\vdash (\Phi; C \triangleright (\psi; V/\mathbf{x}, \delta) \in \Psi; \mathbf{x} \in F_1, \Delta) \in F$ by tssub on $\mathcal{F}_1, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trnew} :: \Phi; C \triangleright \nu\ \rho^L. D \implies \Phi, \rho^L; C, (\lambda\rho^L. (\bullet; \bullet)) \triangleright D$

$\mathcal{D} ::\vdash (\Phi; C \triangleright \nu\ \rho^L. D) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in (\Psi; \Delta) \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \nu\ \rho^L. D \in \Psi; \Delta$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi, \rho^L; \cdot \vdash D \in \Psi'; \Delta'$ by inversion on $\mathcal{D}_2$
$\mathcal{F}_2 :: \Psi; \Delta = \Pi\rho^L. (\Psi'; \Delta')$ by inversion on $\mathcal{D}_2$
$\mathcal{P} :: \Phi, \rho^L \vdash C, (\lambda\rho^L. (\bullet; \bullet)) \in \Psi'; \Delta' \Rightarrow F$ by tcnew on $\mathcal{D}_1$
$\mathcal{Q} ::\vdash (\Phi, \rho^L; C, (\lambda\rho^L. (\bullet; \bullet)) \triangleright D) \in F$ by tsdec on $\mathcal{P}, \mathcal{F}_1$

Case: $\mathcal{E} = \mathsf{trnewC} :: \Phi, \rho^L; C, (\lambda\rho^L. (\bullet; \bullet)) \triangleright \psi; \delta \implies \Phi; C \triangleright \lambda\rho^L. (\psi; \delta)$

$\mathcal{D} ::\vdash (\Phi, \rho^L; C, (\lambda\rho^L. (\bullet; \bullet)) \triangleright \psi; \delta) \in F$ by assumption
$\mathcal{D}_1 :: \Phi, \rho^L \vdash C, (\lambda\rho^L. (\bullet; \bullet)) \in (\Psi; \Delta) \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi, \rho^L; \cdot \vdash \mathrm{id}_\Phi, \rho/\rho, \psi; \delta : \Phi, \rho^L, \Psi; \Delta$ by inversion on $\mathcal{D}$
$\mathcal{F}_1 :: \Phi \vdash C \in \Pi\rho^L. (\Psi; \Delta) \Rightarrow F$ by inversion on $\mathcal{D}_1$
$\mathcal{G} :: \Phi \vdash \mathrm{id}_\Phi; \cdot \in \Phi; \cdot$ by Lemma 6.22
$\mathcal{G}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \psi'; \delta' : \Phi, \Psi'; \Delta'$ by Lemma 6.7(1)
$\mathcal{G}_2 :: \psi'; \delta' = \lambda\rho^L. (\psi; \delta)$ by Lemma 6.7(1)
$\mathcal{G}_3 :: \Psi'; \Delta' = \Pi\rho^L. (\Psi; \Delta)$ by Lemma 6.7(1)
$\mathcal{P}_1 :: \Phi \vdash C \in (\Psi'; \Delta') \Rightarrow F$ by using $\mathcal{G}_3$ on $\mathcal{P}_1$
$\mathcal{P}_2 ::\vdash (\Phi; C \triangleright \psi'; \delta') \in F$ by tssub on $\mathcal{F}_1, \mathcal{P}_1$
$\mathcal{Q} ::\vdash (\Phi; C \triangleright \lambda\rho^L. (\psi; \delta)) \in F$ by using $\mathcal{G}_2$ on $\mathcal{P}_2$

Case: $\mathcal{E} = \mathsf{trcase} :: \Phi; C \triangleright \mathrm{case}\ (\psi; \delta)\ \mathrm{of}\ \Omega \implies \Phi; C \triangleright (\psi; \delta) \sim \Omega$

$\mathcal{D} ::\vdash (\Phi; C \triangleright \mathrm{case}\ (\psi; \delta)\ \mathrm{of}\ \Omega) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in F_1[\psi] \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \mathrm{case}\ (\psi; \delta)\ \mathrm{of}\ \Omega \in F_1[\psi]$ by inversion on $\mathcal{D}$

$\mathcal{E}_1 :: \Phi; \cdot \vdash \psi; \delta : \Psi; \Delta$ by inversion on $\mathcal{D}_2$
$\mathcal{E}_2 :: \Psi; \Delta \vdash \Omega \in F_1$ by inversion on $\mathcal{D}_2$
$\mathcal{Q} :: \vdash (\Phi; C \triangleright (\psi; \delta) \sim \Omega) \in F$ by tscase on $\mathcal{D}_2, \mathcal{E}_1, \mathcal{E}_2$

Case: $\mathcal{E} = \mathsf{tryes} :: \Phi; C \triangleright (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P)) \implies \Phi; C \triangleright P[\psi''; \delta]$

There exists a ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$ by side condition
$\mathcal{D} :: \vdash (\Phi; C \triangleright (\psi; \delta) \sim \Omega, (\Psi' \triangleright \psi' \mapsto P)) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in F_1[\psi] \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \psi; \delta : \Psi; \Delta$ by inversion on $\mathcal{D}$
$\mathcal{D}_3 :: \Psi; \Delta \vdash \Omega, (\Psi' \triangleright \psi' \mapsto P) \in F_1$ by inversion on $\mathcal{D}$
$\mathcal{E}_1 :: \Psi'; [\psi']\Delta \vdash \psi'; \mathrm{id}_\Delta : \Psi; \Delta$ by inversion on $\mathcal{D}_3$
$\mathcal{E}_2 :: \Psi; \Delta \vdash \Omega \in F_1$ by inversion on $\mathcal{D}_3$
$\mathcal{E}_3 :: \Psi'; [\psi']\Delta \vdash P \in F_1[\psi']$ by inversion on $\mathcal{D}_3$
$\mathcal{F}_1 :: \Phi; \cdot \vdash \psi''; \delta : \Psi'; [\psi']\Delta$ by type correctness of side condition
$\mathcal{P} :: \Phi; \cdot \vdash P[\psi''; \delta] \in F_1[\psi'][\psi'']$ by Lemma 6.20(1) on $\mathcal{E}_3, \mathcal{F}_1$
$\mathcal{P} :: \Phi; \cdot \vdash P[\psi''; \delta] \in F_1[\psi]$ by Definition of ψ''
$\mathcal{Q} :: \vdash (\Phi; C \triangleright P[\psi''; \delta]) \in F$ by tsprg on $\mathcal{D}_1, \mathcal{P}$

Case: $\mathcal{E} = \mathsf{trno} :: \Phi; C \triangleright (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P)) \implies \Phi; C \triangleright (\psi; \delta) \sim \Omega$

$\mathcal{D} :: \vdash (\Phi; C \triangleright (\psi; \delta) \sim \Omega, (\Psi' \triangleright \psi' \mapsto P)) \in F$ by assumption
$\mathcal{D}_1 :: \Phi \vdash C \in F_1[\psi] \Rightarrow F$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash \psi; \delta : \Psi; \Delta$ by inversion on $\mathcal{D}$
$\mathcal{D}_3 :: \Psi; \Delta \vdash \Omega, (\Psi' \triangleright \psi' \mapsto P) \in F_1$ by inversion on $\mathcal{D}$
$\mathcal{P} :: \Psi; \Delta \vdash (\psi; \delta) \sim \Omega \in F_1$ by inversion on $\mathcal{D}_3$
$\mathcal{Q} :: \vdash (\Phi; C \triangleright (\psi; \delta) \sim \Omega) \in F$ by tscase on $\mathcal{D}_1, \mathcal{D}_2, \mathcal{P}$

□

Theorem 7.4 *(Type preservation for small-step semantics)*

If $\mathcal{D} :: S \stackrel{*}{\Longrightarrow} S'$
and $\mathcal{E} :: \vdash S \in F$
then $\vdash S' \in F$.

Proof: by induction on $\mathcal{D}$:

Case: $\mathcal{D} = \dfrac{}{S \stackrel{*}{\Longrightarrow} S}\,\mathsf{trid}$

$\mathcal{E} :: \vdash S \in F$ by assumption

Case: $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{S_1 \Longrightarrow S_2} \qquad \overset{\mathcal{D}_2}{S_2 \overset{*}{\Longrightarrow} S_3}}{S_1 \overset{*}{\Longrightarrow} S_3}\,\mathsf{trstep}$

$\mathcal{E}_1 ::\vdash S_1 \in F$	by assumption
$\mathcal{E}_2 ::\vdash S_2 \in F$	by Lemma 7.3 on $\mathcal{E}_1, \mathcal{D}_1$
$\mathcal{E}_3 ::\vdash S_3 \in F$	by i.h. on $\mathcal{D}_2, \mathcal{E}_2$

□

Theorem 7.9 (Termination) *We consider the evaluation of a function of type* $\forall x_1 : A_1. \ldots \forall x_n : A_n. \exists y_1 : A'_1. \ldots \exists y_m : A'_m. \top$ *applied to arguments* $M_1, \ldots, M_n$ *in a parameter context* Φ*. The termination order is* O *and all procedures (used as lemmas) terminate.*

1. *If* $S = \Phi; C \triangleright P$ *and* P *is not a value*
   then $S \overset{*}{\Longrightarrow} \Phi; C \triangleright V$
   or the computation terminates prematurely.

2. *If* $S = \Phi; C \triangleright (\psi; \delta) \sim \Omega$
   then $S \overset{*}{\Longrightarrow} \Phi; C \triangleright V$
   or the computation terminates prematurely.

Proof: by induction lexicographically on 'order $(O, M_1 \ldots M_n)$' and ($P(2)$ and $\Omega(3)$).

1. **Case:** $P = \textbf{let } D \textbf{ in } P'$

$$\begin{aligned} D = \nu\, \rho_1^{L_1}. \ldots \nu\, \rho_q^{L_q}.\ & \\ & \mathbf{y}_1 = \mathbf{x}[P'/\mathbf{x}]\ M'_1, \\ & \mathbf{y}_2 = \mathbf{y}_0\ M'_2 \\ & \vdots \\ & \mathbf{y}_m = \mathbf{y}_n\ M'_m, \\ & \langle x_1, \mathbf{y}_{m+1} \rangle = \mathbf{y}_m, \\ & \vdots \\ & \langle x_p, \mathbf{y}_{m+p} \rangle = \mathbf{y}_{m+p-1} \end{aligned}$$ by definition

Case: $\mathbf{x}$ recursion variable

$m = n$	by inversion
order $(O, M'_1 \ldots M'_n) <_O$ order $(O, M_1 \ldots M_n)$	by Condition 7.8
$P' = \Lambda x_1 : A_1. \ldots \Lambda x_n : A_n. P''$	by n inversion steps
$\Phi; C \triangleright \textbf{let } D \textbf{ in } P'$	
$\overset{*}{\Longrightarrow} \Phi; C \triangleright P''[M'_1/x_1, \ldots, M'_n/x_n]$	by n applications of trApp
$\overset{*}{\Longrightarrow} \Phi; C \triangleright V$	
or the computation terminates prematurely	by i.h. (2)

Case: $\mathbf{x}$ lemma variable

$\Phi; C \triangleright \textbf{let } D \textbf{ in } P'$
$\overset{*}{\Longrightarrow} \Phi; C \triangleright V$
or the computation terminates prematurely — by assumption

Case: $P = \langle P_1, P_2 \rangle$

$\Phi; C \triangleright \langle P_1, P_2 \rangle$
$\Longrightarrow \Phi; C, \langle \bullet, P_2 \rangle \triangleright P_1$ — by trpair
$\overset{*}{\Longrightarrow} \Phi; C, \langle \bullet, P_2 \rangle \triangleright V_1$ — by i.h.(1)
$\Longrightarrow \Phi; C \triangleright \langle V_1, P_2 \rangle$ — by $\mathsf{trpairC}$
$\Longrightarrow \Phi; C, \langle V_1, \bullet \rangle \triangleright P_2$ — by trmix
$\overset{*}{\Longrightarrow} \Phi; C, \langle V_1, \bullet \rangle \triangleright V_2$ — by i.h.(1)
$\Longrightarrow \Phi; C \triangleright \langle V_1, V_2 \rangle$ — by trmixC
or the computation terminates prematurely

Case: $P = \langle V_1, P_2 \rangle$

$\Phi; C \triangleright \langle V_1, P_2 \rangle$
$\Longrightarrow \Phi; C, \langle V_1, \bullet \rangle \triangleright P_2$ — by trmix
$\overset{*}{\Longrightarrow} \Phi; C, \langle V_1, \bullet \rangle \triangleright V_2$ — by i.h.(1)
$\Longrightarrow \Phi; C \triangleright \langle V_1, V_2 \rangle$ — by trmixC
or the computation terminates prematurely

Case: $P = \langle M, P \rangle$

$\Phi; C \triangleright \langle M, P \rangle$
$\Longrightarrow \Phi; C, \langle M, \bullet \rangle \triangleright P$ — by trinx
$\overset{*}{\Longrightarrow} \Phi; C, \langle M, \bullet \rangle \triangleright V$ — by i.h.(1)
$\Longrightarrow \Phi; C \triangleright \langle M, V \rangle$ — by trinxC
or the computation terminates prematurely

Case: $P = \mu \mathbf{x} \in F.\, P'$

$\Phi; C \triangleright \mu \mathbf{x} \in F.\, P'$
$\Longrightarrow \Phi; C \triangleright P[\mu \mathbf{x} \in F.\, P' / \mathbf{x}]$ — by trrec
$\overset{*}{\Longrightarrow} \Phi; C \triangleright V$ — by i.h.(1)
or the computation terminates prematurely

Case: $P = \text{case } (\psi; \delta) \text{ of } \Omega$

$\Phi; C \triangleright \text{case } (\psi; \delta) \text{ of } \Omega$
$\Longrightarrow \Phi; C \triangleright (\psi; \delta) \sim \Omega$ — by trcase
$\overset{*}{\Longrightarrow} \Phi; C \triangleright V$ — by i.h.(2)
or the computation terminates prematurely

2. **Case:** $\Omega, (\Psi' \triangleright \psi' \mapsto P))$ and there exists a ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$

$\Phi; C \triangleright (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P))$
$\Longrightarrow \Phi; C \triangleright P[\psi''; \delta]$ — by tryes
$\overset{*}{\Longrightarrow} \Phi; C \triangleright V$ — by i.h.(1)
or the computation terminates prematurely

Case: $\Omega, (\Psi' \triangleright \psi' \mapsto P))$ and there is no ψ'' s.t. $(\psi'; \mathrm{id}_\Delta) \circ (\psi''; \delta) = (\psi; \delta)$

$\Phi; C \triangleright (\psi; \delta) \sim (\Omega, (\Psi' \triangleright \psi' \mapsto P))$
$\Longrightarrow \Phi; C \triangleright (\psi; \delta) \sim \Omega$ by trno
$\stackrel{*}{\Longrightarrow} \Phi; C \triangleright V$ by i.h.(2)
or the computation terminates prematurely

□

Lemma 7.18 (Liveness of constant covers)
If $\mathcal{D} :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma \gg \omega$ *cover*
and $\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2$
and $\Psi \vdash$ *raise* $(\Gamma_x, \Pi\Gamma_c.\, B_c) = (\Psi_c \triangleright \Pi\Gamma_x.\, B'_c)$
and $\mathcal{F} :: \Phi; \cdot \vdash \psi; \delta \in \Psi, \Psi_c; \Delta$
and $\Sigma(c) = \Pi\Gamma_c.\, B_c$
and $\psi \in$ *unify* $(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_c, x \approx \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x))$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$

Proof: by induction on $\mathcal{D}$:

Case: $\mathcal{D} = \dfrac{}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \cdot \gg \cdot \text{ cover}}\ \textsf{ccempty}$

Impossible case $\cdot(c)$ is undefined by $\mathcal{G}$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma \gg \omega \text{ cover}\end{array}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma, d : \Pi\Gamma_d.\, B_d \gg \omega, (\Psi' \triangleright \psi_m|_\Psi) \text{ cover}}\ \textsf{ccunify}$

Case: $c \neq d$

there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$ by i.h. on $\mathcal{D}_1$
$(\Psi_0 \triangleright \psi_0) \in \omega, (\Psi' \triangleright \psi_m|_\Psi)$ trivial

Case: $c = d$

$\Psi' \vdash \psi_m = \mathrm{mgu}\ (\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_c, x \approx \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x)) \in \Psi, \Psi_c$ by side condition
there exists ψ_1, st. $\Phi \vdash \psi_m \circ \psi_1 = \psi \in \Psi, \Psi_c$ ψ_m is mgu
$\mathcal{P}_1 :: \Phi \vdash \psi_1 \in \Psi'$ by well-typedness of $\psi_m \circ \psi_1$
$\mathcal{P}_2 :: \Psi' \vdash \psi_m \in \Psi, \Psi_c$ by well-typedness of $\psi_m \circ \psi_1$
$\mathcal{P}_3 :: \Psi' \vdash \psi_m|_\Psi \in \Psi$ by Lemma 7.14
$(\Psi' \triangleright \psi_m|_\Psi) \in \omega, (\Psi' \triangleright \psi_m|_\Psi)$ trivial
$\Phi \vdash \psi_m|_\Psi \circ \psi_1 = \psi|_\Psi \in \Psi$ by Lemma 7.15

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma \gg \omega \text{ cover}\end{array}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma, c : \Pi\Gamma_c.\, B_c \gg \omega \text{ cover}}\ \textsf{ccskip}$

Case: $c \neq d$

there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$ — by i.h. on $\mathcal{D}_1$

Case: $c = d$

$\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_c, x \approx \lambda\Gamma_x.\, (c\ (\Psi_c\ \Gamma_x))$ do not unify — by assumption
$\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_c, x \approx \lambda\Gamma_x.\, (c\ (\Psi_c\ \Gamma_x))$ unifies — by assumption
Impossible case

□

Lemma 7.19 (Liveness of local parameter covers)
If $\mathcal{D} :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma \gg \omega$ *cover*
and $\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2$
and $\Psi \vdash$ *raise* $(\Gamma_x, \Pi\Gamma_p.\, B_p) = (\Psi_p \triangleright \Pi\Gamma_x.\, B'_p)$
and $\mathcal{F} :: \Phi; \cdot \vdash \psi; \delta \in \Psi, \Psi_p; \Delta$
and $\Gamma(p) = \Pi\Gamma_p.\, B_p$
and $\psi \in$ *unify* $(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p, x \approx \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x))$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$

Proof: by induction on $\mathcal{D}$:

Case: $\mathcal{D} = \dfrac{}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \cdot \gg \cdot \text{ cover}}\ \textsf{ccempty}$

Impossible case — $\cdot(p)$ is undefined by $\mathcal{G}$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma \gg \omega \text{ cover}\end{array}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma, d : \Pi\Gamma_d.\, B_d \gg \omega, (\Psi' \triangleright \psi_m|_\Psi) \text{ cover}}\ \textsf{ccunify}$

Case: $c \neq d$

there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$ — by i.h. on $\mathcal{D}_1$
$(\Psi_0 \triangleright \psi_0) \in \omega, (\Psi' \triangleright \psi_m|_\Psi)$ — trivial

Case: $p = d$

$\Psi' \vdash \psi_m = \text{mgu}\,(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p, x \approx \lambda\Gamma_x.\, c\ (\Psi_p\ \Gamma_x)) \in \Psi, \Psi_p$ — by side condition
there exists ψ_1, st. $\Phi \vdash \psi_m \circ \psi_1 \in \Psi, \Psi_p$ — ψ_m is mgu
$\mathcal{P}_1 :: \Phi \vdash \psi_1 \in \Psi'$ — by well-typedness of $\psi_m \circ \psi_1$
$\mathcal{P}_2 :: \Psi' \vdash \psi_m \in \Psi, \Psi_p$ — by well-typedness of $\psi_m \circ \psi_1$
$\mathcal{P}_3 :: \Psi' \vdash \psi_m|_\Psi \in \Psi$ — by Lemma 7.14
$(\Psi' \triangleright \psi_m|_\Psi) \in \omega, (\Psi' \triangleright \psi_m|_\Psi)$ — trivial
$\Phi \vdash \psi_m|_\Psi \circ \psi_1 = \psi|_\Psi \in \Psi$ — by Lemma 7.15

Case: $$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma \gg \omega \text{ cover}\end{array}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma, p : \Pi\Gamma_p.\, B_p \gg \omega \text{ cover}}\ \mathsf{ccskip}$$

Case: $p \neq d$

there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$ — by i.h. on $\mathcal{D}_1$

Case: $p = d$

$\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p, x \approx \lambda\Gamma_x.\, (p\ (\Psi_p\ \Gamma_x))$ do not unify — by assumption
$\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p, x \approx \lambda\Gamma_x.\, (p\ (\Psi_p\ \Gamma_x))$ unifies — by assumption
Impossible case

□

Lemma 7.20 (Liveness of global parameter covers)
If $\mathcal{D} :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \rho \gg \omega$ *cover*
and $\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2$
and $\Psi \vdash$ *raise* $(\Gamma_x, \Pi\Gamma_y.\, B_y) = (\Psi_y \triangleright \Pi\Gamma_x.\, B'_y)$
and $\mathcal{F} :: \Phi \vdash \psi \in \Psi, \Psi_3, \Psi_y$
and $\rho(\underline{y}) = \Pi\Gamma_y.\, B_y$
and $\psi \in$ *unify* $(\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_y, x \approx \lambda\Gamma_x.\, p\ (\Psi_y\ \Gamma_x))$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$

Proof: by induction on $\mathcal{D}$:

Case: $\mathcal{D} = \dfrac{}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \cdot \gg \cdot \text{ cover}}\ \textsf{ccempty}$

Impossible case $\qquad \cdot(\underline{y})$ is undefined by $\mathcal{G}$

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \rho \gg \omega \text{ cover}\end{array}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3 \vdash \rho, \underline{d} : \Pi\Gamma_d.\, B_d \gg \omega, (\Psi' \triangleright \psi_m|_\Psi) \text{ cover}}\ \textsf{ccunify}$

Case: $\underline{y} \neq \underline{d}$

there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$ $\qquad$ by i.h. on $\mathcal{D}_1$
$(\Psi_0 \triangleright \psi_0) \in \omega, (\Psi' \triangleright \psi_m|_\Psi)$ $\qquad$ trivial

Case: $\underline{y} = \underline{d}$

$\Psi' \vdash \psi_m = \text{mgu}\ (\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_y, x \approx \lambda\Gamma_x.\, c\ (\Psi_y\ \Gamma_x)) \in \Psi, \Psi_3, \Psi_y$ $\qquad$ by side cond.
there exists ψ_1, st. $\Phi \vdash \psi_m \circ \psi_1 = \psi \in \Psi, \Psi_3, \Psi_y$ $\qquad$ ψ_m is mgu
$\mathcal{P}_1 :: \Phi \vdash \psi_1 \in \Psi'$ $\qquad$ by well-typedness of $\psi_m \circ \psi_1$
$\mathcal{P}_2 :: \Psi' \vdash \psi_m \in \Psi, \Psi_3, \Psi_y$ $\qquad$ by well-typedness of $\psi_m \circ \psi_1$
$\mathcal{P}_3 :: \Psi' \vdash \psi_m|_\Psi \in \Psi$ $\qquad$ by Lemma 7.14
$(\Psi' \triangleright \psi_m|_\Psi) \in \omega, (\Psi' \triangleright \psi_m|_\Psi)$ $\qquad$ trivial
$\Phi \vdash \psi_m|_\Psi \circ \psi_1 = \psi|_\Psi \in \Psi$ $\qquad$ by Lemma 7.15

Case: $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \rho \gg \omega \text{ cover}\end{array}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \rho, \underline{d} : \Pi\Gamma_d.\, B_d \gg \omega \text{ cover}}\ \textsf{ccskip}$

Case: $\underline{y} \neq \underline{d}$

there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi|_\Psi \in \Psi$ $\qquad$ by i.h. on $\mathcal{D}_1$

Case: $\underline{y} = \underline{d}$

$\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_y, x \approx \lambda\Gamma_x.\, (\underline{y}\ (\Psi_y\ \Gamma_x))$ do not unify $\qquad$ by assumption
$\Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_y, x \approx \lambda\Gamma_x.\, (\underline{y}\ (\Psi_y\ \Gamma_x))$ unifies $\qquad$ by assumption
Impossible case

□

Lemma 7.20 (Liveness of schematic coverage)
If $\mathcal{D} :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash S \gg \omega$
and $\Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2$
and $\mathcal{F} :: \Phi \vdash \psi \in \Psi$
and $\psi(x) = \lambda\Gamma_x.\, g\ M_1..M_n$
and $\rho^L \in \Phi$
and $\rho(g) = \Pi\Gamma_g.\, B_g$
and $\mathcal{G} :: S(L) = \text{SOME } C_1.\ \text{BLOCK } C_2$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0; \circ\psi_1 = \psi \in \Psi$

Proof: by induction on $\mathcal{D}$

Case: $\mathcal{D} = \dfrac{}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \cdot \gg \cdot \text{ cover}}\ \textsf{scempty}$

Impossible case $\qquad\qquad$ $\cdot(L)$ is undefined by $\mathcal{G}$

Case: $\mathcal{D} = \dfrac{\begin{array}{l}\mathcal{D}_1 :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash S \gg \omega_1 \text{ cover} \\ \mathcal{D}_2 :: \Psi_3 \vdash \sigma \in C_1 \\ \mathcal{D}_3 :: \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2, \Psi_3 \vdash \rho \equiv [\sigma]C_2 \\ \mathcal{D}_4 :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2; \Psi_3, \rho^{L'} \vdash \rho \gg \omega_2 \text{ cover}\end{array}}{\Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash S, (\text{SOME } C_1.\ \text{BLOCK } C_2)^{L'} \gg \omega_1, \omega_2 \text{ cover}}\ \textsf{scnext}$

Case: $L = L'$

$\Psi \vdash \text{raise } (\Gamma_x, \Pi\Gamma_g.\, B_g) = (\Psi_g \triangleright \Pi\Gamma_x.\, B'_g)$	by definition
$\Psi_g = z_1 : A_1, ..., z_n : A_n$	by definition
$\mathcal{G}_1 :: \Psi, \Psi_g \vdash \lambda\Gamma_x.\, g\ (\Psi_g\ \Gamma_x) \in \Pi\Gamma_x.\, B'_g$	by Lemma 7.11
$\mathcal{E}_1 :: \Phi \vdash \sigma = \psi, \lambda\Gamma_x.\, M_1/z_1, ...\lambda\Gamma_x.\, M_n/z_n \in \Psi, \Psi_g$	by Def. Substitution
$\mathcal{P}_1 :: \Phi \vdash \lambda\Gamma_x.\, g\ (\Psi_g\ \Gamma_x)[\sigma] = \lambda[\sigma]\Gamma_x.\, g\ M_1...M_n$	by Lemma 6.7(1)
$\Phi \vdash \lambda\Gamma_x.\, g\ (\Psi_g\ \Gamma_x)[\sigma] = \lambda[\psi]\Gamma_x.\, g\ M_1...M_n$	Γ_x does not depend on Ψ_g
$\Phi \vdash \lambda\Gamma_x.\, g\ (\Psi_g\ \Gamma_x)[\sigma] \in \Pi\Gamma_x.\, B'_g[\sigma]$	by LF substitution lemma on $\mathcal{G}_1$
$\Phi \vdash \lambda\Gamma_x.\, g\ (\Psi_g\ \Gamma_x)[\sigma] \in \Pi\Gamma_x.\, B_x[\sigma]$	by Definition of substitution
$\mathcal{P}_2 :: \Phi \vdash \Pi\Gamma_x.\, B'_g[\sigma] \equiv \Pi\Gamma_x.\, B_x[\sigma]$	by Lemma 2.7
$\sigma \in \text{unify } (x \approx \lambda\Gamma_x.\, g\ (\Psi_g\ \Gamma_x), \Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_g)$	by Definition 7.10 on $\mathcal{P}_1, \mathcal{P}_2$
there exists a $(\Psi_0 \triangleright \psi_0) \in \omega_2$ and a ψ_1, s.t.	
$\Phi \vdash \psi_1 \in \Psi_0$	
$\Psi_0 \vdash \psi_0 \in \Psi$	
$\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$	by Lemma 7.20
$(\Psi_0 \triangleright \psi_0) \in \omega_1, \omega_2$	trivial

Case: $L \neq L'$

there exists a $(\Psi_0 \rhd \psi_0) \in \omega_1$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 ; \circ \psi_1 = \psi \in \Psi$ by i.h. on $\mathcal{D}_1$
$(\Psi_0 \rhd \psi_0) \in \omega_1, \omega_2$ trivial

□

Lemma 7.22 (Liveness of single coverage)
If $\mathcal{D} :: \Psi \vdash \omega$ *cover*
and $\mathcal{E} :: \Phi \vdash \psi \in \Psi$
then there exists a $(\Psi_0 \rhd \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$

Proof: by case analysis of $\mathcal{D}$:

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1 \\ \Psi = \Psi_1, x : \Pi\Gamma_x.\, B_x, \Psi_2\end{array} \qquad \begin{array}{l}\mathcal{D}_2 :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Gamma_x \gg \omega_1 \text{ cover} \\ \mathcal{D}_3 :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \Sigma \gg \omega_2 \text{ cover} \\ \mathcal{D}_4 :: \Psi_1; x : \Pi\Gamma_x.\, B_x; \Psi_2 \vdash \mathcal{S} \gg \omega_3 \text{ cover}\end{array}}{\Psi \vdash \omega_1, \omega_2, \omega_3 \;\; \text{cover}} \,\mathsf{single}$$

Let $\psi(x) = \lambda[\psi]\Gamma_x.\, h\ M_1 ... M_n$ by Theorem 2.6 and $\mathcal{D}_1$

Case: $h = c$

$\Sigma(c) = \Pi\Gamma_c.\, B_c$ well-typedness of ψ
$\Psi \vdash \text{raise}\ (\Gamma_x, \Pi\Gamma_c.\, B_c) = (\Psi_c \rhd \Pi\Gamma_x.\, B_c')$ by definition
$\Psi_c = z_1 : A_1, ..., z_n : A_n$ by definition
$\mathcal{G}_1 :: \Psi, \Psi_c \vdash \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x) \in \Pi\Gamma_x.\, B_c'$ by Lemma 7.11
$\mathcal{E}_1 :: \Phi \vdash \sigma = \psi, \lambda\Gamma_x.\, M_1/z_1, ...\lambda\Gamma_x.\, M_n/z_n \in \Psi, \Psi_c$ by Def. substitution
$\mathcal{P}_1 :: \Phi \vdash \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x)[\sigma] = \lambda[\sigma]\Gamma_x.\, c\ M_1 ... M_n$ by Lemma 6.7(1)
$\Phi \vdash \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x)[\sigma] = \lambda[\psi]\Gamma_x.\, c\ M_1 ... M_n$ Γ_x does not depend on Ψ_c
$\Phi \vdash \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x)[\sigma] \in \Pi\Gamma_x.\, B_c'[\sigma]$ by LF substitution lemma on $\mathcal{G}_1$
$\Phi \vdash \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x)[\sigma] \in \Pi\Gamma_x.\, B_x[\sigma]$ by Definition of substitution
$\mathcal{P}_2 :: \Phi \vdash \Pi\Gamma_x.\, B_c'[\sigma] \equiv \Pi\Gamma_x.\, B_x[\sigma]$ by Lemma 2.7
$\sigma \in \text{unify}\ (x \approx \lambda\Gamma_x.\, c\ (\Psi_c\ \Gamma_x), \Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B_c')$ by Definition 7.10 on $\mathcal{P}_1, \mathcal{P}_2$
there exists a $(\Psi_0 \rhd \psi_0) \in \omega_2$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$ by Lemma 7.18
$(\Psi_0 \rhd \psi_0) \in \omega_1, \omega_2, \omega_3$ trivial

Case: $h = p$

$\Gamma(p) = \Pi\Gamma_p.\, B_p$	well-typedness of ψ
$\Psi \vdash \text{raise}\ (\Gamma_x, \Pi\Gamma_p.\, B_p) = (\Psi_p \triangleright \Pi\Gamma_x.\, B'_p)$	by definition
$\Psi_p = z_1 : A_1, ..., z_n : A_n$	by definition
$\mathcal{G}_1 :: \Psi, \Psi_p \vdash \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x) \in \Pi\Gamma_x.\, B'_p$	by Lemma 7.11
$\mathcal{E}_1 :: \Phi \vdash \sigma = \psi, \lambda\Gamma_x.\, M_1/z_1, ...\lambda\Gamma_x.\, M_n/z_n \in \Psi, \Psi_p$	by Def. substitution
$\mathcal{P}_1 :: \Phi \vdash \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x)[\sigma] = \lambda[\sigma]\Gamma_x.\, p\ M_1...M_n$	by Lemma 6.7(1)
$\Phi \vdash \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x)[\sigma] = \lambda[\psi]\Gamma_x.\, p\ M_1...M_n$	Γ_x does not depend on Ψ_p
$\Phi \vdash \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x)[\sigma] \in \Pi\Gamma_x.\, B'_p[\sigma]$	by LF substitution lemma on $\mathcal{G}_1$
$\Phi \vdash \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x)[\sigma] \in \Pi\Gamma_x.\, B_x[\sigma]$	by Definition of substitution
$\mathcal{P}_2 :: \Phi \vdash \Pi\Gamma_x.\, B'_p[\sigma] \equiv \Pi\Gamma_x.\, B_x[\sigma]$	by Lemma 2.7
$\sigma \in \text{unify}\ (x \approx \lambda\Gamma_x.\, p\ (\Psi_p\ \Gamma_x), \Pi\Gamma_x.\, B_x \approx \Pi\Gamma_x.\, B'_p)$	by Definition 7.10 on $\mathcal{P}_1, \mathcal{P}_2$
there exists a $(\Psi_0 \triangleright \psi_0) \in \omega_1$ and a ψ_1, s.t.	
$\Phi \vdash \psi_1 \in \Psi_0$	
$\Psi_0 \vdash \psi_0 \in \Psi$	
$\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$	by Lemma 7.19
$(\Psi_0 \triangleright \psi_0) \in \omega_1, \omega_2, \omega_3$	trivial

Case: $h = g$

$\rho^L \in \Phi$	
$\rho(g) = \Pi\Gamma_g.\, B_g$	g is a global parameter
there exists a $(\Psi_0 \triangleright \psi_0) \in \omega_2$ and a ψ_1, s.t.	
$\Phi \vdash \psi_1 \in \Psi_0$	
$\Psi_0 \vdash \psi_0 \in \Psi$	
$\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$	by Lemma 7.21
$(\Psi_0 \triangleright \psi_0) \in \omega_1, \omega_2, \omega_3$	trivial

□

Lemma 7.23 (Liveness of multi coverage)
If $\mathcal{D} :: \Psi \vdash \omega$ *cover**
and $\Phi \vdash \psi \in \Psi$
then there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ *and a* ψ_1
s.t. $\Phi \vdash \psi_1 \in \Psi_0$
and $\Psi_0 \vdash \psi_0 \in \Psi$
and $\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$

Proof: by induction on $\mathcal{D}$

Case: $\mathcal{D} = \dfrac{\Psi \vdash \omega \text{ cover}}{\Psi \vdash \omega \text{ cover}^*}\ \mathsf{multiempty}$

there exists a $(\Psi_0 \triangleright \psi_0) \in \omega$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$

$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$ by Lemma 7.22

$$\textbf{Case: } \mathcal{D} = \dfrac{\begin{matrix}\mathcal{D}_1 \\ \Psi \vdash \omega_1, (\Psi' \rhd \psi'), \omega_2 \ \text{ cover}^*\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ \Psi' \vdash \omega' \ \text{ cover}\end{matrix}}{\Psi \vdash \omega_1, \psi' \circ \omega', \omega_2 \ \text{ cover}^*} \ \mathsf{multicons}$$

there exists a $(\Psi_0 \rhd \psi_0) \in \omega_1, (\Psi' \rhd \psi'), \omega_2$ and a ψ_1, s.t.
$\mathcal{P}_1 :: \Phi \vdash \psi_1 \in \Psi_0$
$\mathcal{P}_2 :: \Psi_0 \vdash \psi_0 \in \Psi$
$\mathcal{P}_3 :: \Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$ by i.h. on $\mathcal{D}_1$

Case: $(\Psi_0 \rhd \psi_0) = (\Psi' \rhd \psi')$

then there exists a $(\Psi_2 \rhd \psi_2) \in \omega'$ and a ψ_3 s.t.
$\mathcal{Q}_1 :: \Phi \vdash \psi_3 \in \Psi_2$
$\mathcal{Q}_2 :: \Psi_2 \vdash \psi_2 \in \Psi_0$
$\mathcal{Q}_3 :: \Phi \vdash \psi_2 \circ \psi_3 = \psi_1 \in \Psi_0$ by Lemma 7.22 on $\mathcal{P}_1, \mathcal{D}_2$
$\mathcal{R}_1 :: \Psi_2 \vdash \psi_0 \circ \psi_2 \in \Psi$ by Lemma 5.2 on $\mathcal{Q}_2, \mathcal{P}_2$
$(\Psi_2 \rhd \psi_0 \circ \psi_2) \in \psi_0 \circ \omega'$ by Definition 7.3.2
$(\Psi_2 \rhd \psi_0 \circ \psi_2) \in \omega_1, \psi' \circ \omega', \omega_2$ trivial
$\mathcal{R}_2 :: \Phi \vdash (\psi_0 \circ \psi_2) \circ \psi_3 = \psi_0 \circ \psi_1 = \psi \in \Psi$ by Def. substitution on $\mathcal{Q}_1, \mathcal{R}_1, \mathcal{Q}_3, \mathcal{P}_3$

Case: $(\Psi_0 \rhd \psi_0) \neq (\Psi' \rhd \psi')$

$(\Psi_0 \rhd \psi_0) \in \omega_1, \omega_2$ trivial
$(\Psi_0 \rhd \psi_0) \in \omega_1, \psi' \circ \omega', \omega_2$ trivial

□

Lemma 7.24 (Liveness)
If $\mathcal{D} :: \Psi \vdash \omega$ *cover**
and $\mathcal{E} :: \Phi; \cdot \vdash \psi; \delta \in \Psi; \Delta$
then there exists a $(\Psi_0 \rhd \psi_0) \in \omega$*, and a* ψ_1*, s.t.* $\Phi; \cdot \vdash \psi_1; \delta \in \Psi_0; [\psi_0]\Delta$
and $\Psi_0; [\psi_0]\Delta \vdash \psi_0; id_\Delta \in \Psi; \Delta$
and $\Phi; \cdot \vdash (\psi_0; id_\Delta) \circ (\psi_1; \delta) = (\psi; \delta) \in \Psi; \Delta$

Proof:

$\mathcal{E}_1 :: \Phi \vdash \psi \in \Psi$ by Lemma 7.17 on $\mathcal{E}$
there exists a $(\Psi_0 \rhd \psi_0) \in \omega$ and a ψ_1, s.t.
$\Phi \vdash \psi_1 \in \Psi_0$
$\Psi_0 \vdash \psi_0 \in \Psi$
$\Phi \vdash \psi_0 \circ \psi_1 = \psi \in \Psi$ by Lemma 7.23 on $\mathcal{D}, \mathcal{E}_1$

$\Phi; \cdot \vdash \psi_1; \delta \in \Psi_0; [\psi_0]\Delta$
$\Psi_0; [\psi_0]\Delta \vdash \psi_0; \mathrm{id}_\Delta \in \Psi; \Delta$
$\Phi; \cdot \vdash (\psi_0; \mathrm{id}_\Delta) \circ (\psi_1; \delta) = (\psi; \delta) \in \Psi; \Delta$ by Lemma 7.16

□

Lemma 7.25 (Progress for case)
If $S = \Phi; C \triangleright (\psi; \delta) \sim \Omega$
and there exists a $((\Psi_0 \triangleright \psi_0) \mapsto P) \in \Omega$, *and a* ψ_1
s.t. $\Phi; \cdot \vdash \psi_1; \delta \in \Psi_0; [\psi_0]\Delta$
and $\Psi_0; [\psi_0]\Delta \vdash \psi_0; id_\Delta \in \Psi; \Delta$
and $\Phi; \cdot \vdash (\psi_0; id_\Delta) \circ (\psi_1; \delta) = (\psi; \delta) \in \Psi; \Delta$
then there exists an S'
and $S \overset{*}{\Longrightarrow} S'$
and S' *is not a match state*

Proof: by induction over Ω

Case: $\Omega = \cdot$

Impossible case $((\Psi_0 \triangleright \psi_0) \mapsto P) \in \cdot$ undefined

Case: $\Omega = \Omega', ((\Psi'_0 \triangleright \psi'_0) \mapsto P')$

Case: $((\Psi_0 \triangleright \psi_0) \mapsto P) = ((\Psi'_0 \triangleright \psi'_0) \mapsto P')$

$S \Longrightarrow \Phi; C \triangleright P[\psi_1; \delta]$ by tryes
$S \overset{*}{\Longrightarrow} \Phi; C \triangleright P[\psi_1; \delta]$ by trid

Case: $((\Psi_0 \triangleright \psi_0) \mapsto P) \neq ((\Psi'_0 \triangleright \psi'_0) \mapsto P')$

$\mathcal{D}_1 :: S \Longrightarrow \Phi; C \triangleright (\psi; \delta) \sim \Omega'$ by trno
$\mathcal{D}_2 :: \Phi; C \triangleright (\psi; \delta) \sim \Omega' \overset{*}{\Longrightarrow} S'$ by i.h. on Ω'
S' is not a match state by i.h. on Ω'
$S \overset{*}{\Longrightarrow} S'$ by trstep on $\mathcal{D}_1, \mathcal{D}_2$

□

Theorem 7.26 (Progress)
If S *is a state, but not a match state*
and $S \neq \Phi; \cdot \triangleright V$
and $\mathcal{D} :: \vdash S \in F$
then there exists an S' *and an* S'' *which is not a match state*
and $S \Longrightarrow S' \overset{*}{\Longrightarrow} S''$

Proof: by case analysis of S

Case: $S = \Phi; C \triangleright P$

Case: P is not a value: $P \neq V$

$P = \langle M, P \rangle$: trinx is applicable

$P = \mathbf{let}\ D\ \mathbf{in}\ P$: trlet is applicable

$P = \mu\mathbf{x} \in F.\, P$: trrec is applicable

$P = \langle P_1, P_2 \rangle$: trpair is applicable

$P = \langle V_1, P_2 \rangle$: trmix is applicable

$P = \text{case}\ (\psi; \delta)\ \text{of}\ \Omega$:

$\mathcal{D}_1 :: \Phi; \cdot \vdash \text{case}\ (\psi; \delta) \in F_1$ — by inversion on $\mathcal{D}$

$\mathcal{E}_1 :: \Phi; \cdot \vdash \psi; \delta \in \Psi; \Delta$ — by inversion on $\mathcal{D}_1$

$\mathcal{E}_2 :: \Psi; \Delta \vdash \Omega \in F_1$ — by inversion on $\mathcal{D}_1$

$\mathcal{E}_3 :: \Psi \vdash \text{strip}\ (\Omega)\ \text{cover}^*$ — by formal side condition of $\mathcal{D}_1$

there exists a $(\Psi_0 \triangleright \psi_0) \in \text{strip}\ (\Omega)$, and a ψ_1, s.t.

$\Phi; \cdot \vdash \psi_1; \delta \in \Psi_0; [\psi_0]\Delta$

$\Psi_0; [\psi_0]\Delta \vdash \psi_0; \text{id}_\Delta \in \Psi; \Delta$

$\Phi; \cdot \vdash (\psi_0; \text{id}_\Delta) \circ (\psi_1; \delta) = (\psi; \delta) \in \Psi; \Delta$ — by Lemma 7.24

$((\Psi_0 \triangleright \psi_0) \mapsto P) \in \Omega$ — by Definition 7.13

$\mathcal{D}_1 :: S \Longrightarrow S'$ — by trcase

$\mathcal{D}_2 :: S' \overset{*}{\Longrightarrow} S''$ — by Lemma 7.25

S'' is not a match state — by Lemma 7.25

$S \overset{*}{\Longrightarrow} S''$ — by trstep on $\mathcal{D}_1, \mathcal{D}_2$

Case: P is a value: $P = V$; case analysis of C

$C = C', \langle \bullet, P \rangle$: $\mathsf{trpairC}$ is applicable

$C = C', \langle V, \bullet \rangle$: trmixC is applicable

$C = C', \langle M, \bullet \rangle$: trinxC is applicable

$C = C', (\mathbf{x} \in F = \pi_1\ \bullet, D)$: trfstC is applicable

$C = C', (\mathbf{x} \in F = \pi_2\ \bullet, D)$: trsndC is applicable

$C = C', (\langle x : A, \mathbf{y} \in F \rangle = \bullet, D)$: $\mathsf{trsplitC}$ is applicable

$C = C', (\mathbf{x} \in F = \bullet\ M, D)$: trAppC is applicable

$C = C', (\mathbf{x} \in F = \bullet\ \rho, D)$: trappC is applicable

$C = C', (\mathbf{x} \in F = \bullet, D)$: $\mathsf{trassign}$ is applicable

all other continuations impossible due to typing

Case: $S = \Phi; C \triangleright D$

$D = \cdot$: $\mathsf{trempty}$ is applicable

$D = \langle x : A, \mathbf{y} \in F \rangle = P, D$: $\mathsf{trsplit}$ is applicable

$D = \mathbf{x} \in F = P\ M, D$: trApp is applicable

$D = \mathbf{x} \in F = P\ \rho, D$: trapp is applicable

$D = \nu\ \rho^L.\ D$: trnew is applicable
$D = \mathbf{x} \in F = \pi_1\ P, D$: trfst is applicable
$D = \mathbf{x} \in F = \pi_2\ P, D$: trsnd is applicable

Case: $S = \Phi; C \triangleright \psi; \delta$: Case analysis over C

$C = C', \mathbf{let} \bullet \mathbf{in}\ P$: trletC is applicable
$C = C', (\bullet; V/\mathbf{x}, \bullet)$: trmeta is applicable
$C = C', (\lambda\rho^L.\ (\bullet; \bullet))$: trnewC is applicable
$C = C', (M/x, \bullet; V/\mathbf{y}, \bullet)$: $\mathsf{trsubst}$ is applicable
all other continuations impossible due to typing

$\square$

Theorem 7.27 (Realizability)
If $\Phi; \cdot \vdash P \in F$
then there exists a V
s.t. $\Phi; \cdot \vdash V \in F$
and $\Phi; \star \triangleright P \stackrel{*}{\Longrightarrow} \Phi; \star \triangleright V$

Proof: direct.

$\mathcal{D} :: \Phi \star \triangleright P \stackrel{*}{\Longrightarrow} S'$ by Theorem 7.9

Case: $S' = \Phi \star \triangleright V$

$\mathcal{E}_1 :: \Phi \vdash \star \in F \Rightarrow F$	by tcdone
$\mathcal{E}_2 :: \vdash (\Phi \star \triangleright P) \in F$	by tsprg on $\mathcal{E}_1$
$\mathcal{E}_3 :: \vdash (\Phi \star \triangleright V) \in F$	by Lemma 7.4 on $\mathcal{E}_2$
$\mathcal{E} :: \Phi; \cdot \vdash V \in F$	by inversion on $\mathcal{E}_3$

Case: $S' \neq \Phi \star \triangleright V$ and computation ends in S'. Case is impossible because: $S' \stackrel{1}{\Longrightarrow} S''$ by Theorem 7.26 and therefore S' canot be the state the computation ended in.

$\square$

Theorem 7.28 (Soundness of $\mathcal{M}_2^+$)

1. *If* $\mathcal{D} :: \vdash Q \in G$
then $\models G$.

2. *If* $\mathcal{D} :: \Phi; \cdot \vdash V \in F$
then $\Phi \models F$.

Proof: (1) direct, (2) by induction on the size of formulas F.

1. **Case:** $G = \Box S.\, F$:

$\mathcal{D} :: \vdash \mathbf{box}\ S.\, P \in \Box S.\, F$ by assumption
$\mathcal{D}_1 :: \cdot; \cdot \vdash P \in F$ by inversion
Let $\Phi \in [\![S]\!]$, arbitrary
$\mathcal{D}_2 :: \Phi; \cdot \vdash P \in F$ by Lemma 6.11(1)
$\mathcal{Q}_1 :: \Phi; \star \triangleright P \stackrel{*}{\Longrightarrow} \Phi; \star \triangleright V$
$\mathcal{Q}_2 :: \Phi; \cdot \vdash V \in F$ for a V by Theorem 7.27 on $\mathcal{D}_2$
$\Phi \models F$ by i.h.(2) on $\mathcal{Q}_2$
$\models \Box S.\, F'$ by Definition 5.7 discharging assumption that Φ arbitrary

2. **Case:** $F = \top$:

$\mathcal{D} :: \Phi; \cdot \vdash \langle\rangle \in \top$ by assumption
$\Phi \models \top$ by Definition 5.7

Case: $F = \exists x : A.\, F'$

$\mathcal{D} :: \Phi; \cdot \vdash \langle M, V' \rangle \in \exists x : A.\, F'$ by assumption
$\mathcal{D}_1 :: \Phi \vdash M : A$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash V' \in F'[M/x]$ by inversion on $\mathcal{D}$
$\Phi \models F'[M/x]$ by i.h.(2) on $\mathcal{E}_2, \mathcal{D}_2$
$\Phi \models \exists x : A.\, F'$ by Definition 5.7 on $\mathcal{D}_1$

Case: $F = \forall x : A.\, F'$:

$\mathcal{D} :: \Phi; \cdot \vdash \Lambda x : A.\, P \in \forall x : A.\, F'$ by assumption
$\mathcal{D}_1 :: \Phi, x : A; \cdot \vdash P \in F'$ by inversion on $\mathcal{D}$
Let M be arbitray, s.t. $\Phi \vdash M : A$
$\mathcal{P}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot \in \Phi; \cdot$ by Lemma 6.22
$\mathcal{P}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, M/x; \cdot \in \Phi, x : A; \cdot$ trivial
$\mathcal{Q}_1 :: \Phi; \cdot \vdash P[M/x] \in F'[M/x]$ by Lemma 6.20 (1) on $\mathcal{D}_1, \mathcal{P}_2$
$\mathcal{Q}_2 :: \Phi \triangleright P[M/x] \stackrel{*}{\Longrightarrow} \Phi \triangleright V$
$\mathcal{Q}_3 :: \Phi; \cdot \vdash V \in F'[M/x]$ for a V by Theorem 7.27 on $\mathcal{Q}_1$
$\Phi \models F'[M/x]$ by i.h.(2) on $\mathcal{Q}_3$
$\Phi \models \forall x : A.\, F'$ by Definition 5.7 discharging assumption that M arbitrary

Case: $F = \mathbf{\Pi} \rho^L.\, F'$:

$\mathcal{D} :: \Phi; \cdot \vdash \lambda \rho^L.\, P \in \mathbf{\Pi} \rho^L.\, F'$ by assumption
$\mathcal{D}_1 :: \Phi, \rho^L; \cdot \vdash P \in F'$ by inversion on $\mathcal{D}$
Let $\rho'^L \in \Phi$ be arbitray, s.t. $\Phi \vdash \rho \approx \rho'$
$\mathcal{P}_1 :: \Phi; \cdot \vdash \mathrm{id}_\Phi; \cdot \in \Phi; \cdot$ by Lemma 6.22
$\mathcal{P}_2 :: \Phi; \cdot \vdash \mathrm{id}_\Phi, \rho'/\rho; \cdot \in \Phi, \rho^L; \cdot$ trivial
$\mathcal{Q}_1 :: \Phi; \cdot \vdash P[\rho'/\rho] \in F'[\rho'/\rho]$ by Lemma 6.20 (1) on $\mathcal{D}_1, \mathcal{P}_2$

$\mathcal{Q}_2 :: \Phi \triangleright P[\rho'/\rho] \overset{*}{\Longrightarrow} \Phi \triangleright V$
$\mathcal{Q}_3 :: \Phi; \cdot \vdash V \in F'[\rho'/\rho]$ for a V by Theorem 7.27 on $\mathcal{Q}_1$
$\Phi \models F'[\rho'/\rho]$ by i.h.(2) on $\mathcal{Q}_3$
$\Phi \models \mathbf{\Pi}\rho^L.\, F'$ by Definition 5.7 discharging assumption that ρ'^L arbitrary

Case: $F = F_1 \wedge F_2$:

$\mathcal{D} :: \Phi; \cdot \vdash \langle V_1, V_2 \rangle \in F_1 \wedge F_2$ by assumption
$\mathcal{D}_1 :: \Phi; \cdot \vdash V_1 \in F_1$ by inversion on $\mathcal{D}$
$\mathcal{D}_2 :: \Phi; \cdot \vdash V_2 \in F_2$ by inversion on $\mathcal{D}$
$\Phi \models F_1$ by i.h.(2) on $\mathcal{D}_1, \mathcal{E}_1$
$\Phi \models F_2$ by i.h.(2) on $\mathcal{D}_2, \mathcal{E}_2$
$\Phi \models F_1 \wedge F_2$ by Definition 5.7

□

Bibliography

[AC99] Simon J. Ambler and Roy L. Crole. Mechanized operational semantics via (co)induction. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 221–238. Springer-Verlag LNCS 1690, 1999.

[AF99] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th Conference on Computer and Communications Security*, Singapore, November 1999. ACM Press. To appear.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In Thomas Reps, editor, *Conference Record of the 27th Annual Symposium on Principles of Programming Languages (POPL'00)*, pages 243–253, Boston, Massachusetts, January 2000. ACM Press.

[AH77a] K. Appel and W. Haken. Every planar map is four colorable, i. discharging. *Illinois J. Math.*, 21:429–490, 1977.

[AH77b] K. Appel and W. Haken. Every planar map is four colorable, ii. reducibility,. *Illinois J. Math.*, 21:491–567, 1977.

[AINP90] Peter B. Andrews, Sunil Issar, Dan Nesmith, and Frank Pfenning. The TPS theorem proving system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, Germany*, pages 641–642. Springer-Verlag LNCS 449, July 1990. System abstract.

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 13–28, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[Bar80] H. P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. North-Holland, 1980.

[BGLS92] Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic paramodulations and superposition. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 462–476, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607. To appear.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. ACM monograph series. Academic Press, New York, 1979.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. D. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[BvHHS91] Alan Bundy, Frank van Harmelen, Jane Hesketh, and Alan Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, Sep 1991.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CD99] Valérie Ménissier-Morain Catherine Dubois. Certification of a type inference tool for ML: Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3–4):319–346, November 1999.

[CGL94] E. Clarke, O. Grumberg, and D. Long. *A Decade of Concurrency - Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, chapter Verification tools for finite-state concurrent systems. 1994.

[CGP00] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[Coq86] Thierry Coquand. An analysis of Girard's paradox. In *Symposium on Logic Computer Science*, pages 227–236. IEEE, June 1986.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[CP97a] Iliano Cervesato and Frank Pfenning. Linear higher-order pre-unification. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Sumposium on Logic in Computer Science (LICS'97)*, pages 422–433, Warsaw, Poland, June 1997. IEEE Computer Society Press.

[CP97b] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Department of Computer Science, Carnegie Mellon University, April 1997.

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[CT95] C. Cornes and D. Terrasse. Automating inversion and inductive predicates in Coq. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 85–104, Torino, Italy, June 1995. Springer-Verlag LNCS 1158.

[dB72] N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DFH$^+$93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[DFH95] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.

[DH94] Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in Coq. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.

[DHKP96] 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.

[DL98] Joëlle Despeyroux and Pierre Leleu. A modal lambda calculus with iteration and case constructs. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 47–61, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

[DMTV99] Anatoli Degtyarev, Grigori Mints, Tanel Tammet, and Andrei Voronkov. *Handbook of Automated Reasoning*, chapter The inverse method. Elsevier Science Publishers, 1999.

[Dow92] Gilles Dowek. Third order matching is decidable. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 2–10, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[DP95] Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical Report CMU-CS-95-121, Department of Computer Science, Carnegie Mellon University, February 1995.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS 1210. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.

[FH94] Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In *Fifth International Conference on Logic Programming and Automated Reasoning.* Springer-Verlag, LNCS 822, July 1994.

[FSDF93] Cormac Flanagan, Amry Sabry, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Conference on Porgramming Language Design and Implementation. Albuquerque, New Mexico*, June 21-25 1993.

[Gal93] Jean Gallier. On the correspondence between proofs and λ-terms. Cahiers du centre de logique, Université Catholique de Louvain, January 1993.

[Gen35] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen, 1969*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1935.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der Principia Mathematica und verwandter Systeme i. *Monatshefte fr Mathematik und Physik 38*, pages 173–198, 1931.

[Göd90] Kurt Gödel. On an extension of finitary mathematics which has not yet been used. In Solomon Feferman et al., editors, *Kurt Gödel, Collected Works, Volume II*, pages 271–280. Oxford University Press, 1990.

[GP99] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.

[Häh99] Reiner Hähnle. *Handbook of Automated Reasoning*, chapter Tableaux and related methods. lsevier Science Publishers, 1999.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[HHP93] 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.

[HO80] Gérard Huet and Derek C. Oppen. Equations and rewrite rules: A survey. Technical Report STAN-CS-80-786, Stanford University, January 1980.

[Hof99] Martin Hofmann. Semantical analysis for higher-order abstract syntax. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 204–213, Trento, Italy, July 1999. IEEE Computer Society Press.

[How69] W. A. Howard. The formulae-as-types notion of construction. Unpublished manuscript, 1969. Reprinted in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1980.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1980*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard, 1979.

[How98] J. M. Howe. *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St Andrews, December 1998. Available as University of St Andrews Research Report CS/99/1.

[HP98] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.

[HP99] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. In A. Felty, editor, *Proceedings of the Workshop on Logical Frameworks and Meta-Languages (LFM'99)*, Paris, France, September 1999. Extended version available as Technical Report CMU-CS-99-159.

[HS96] D. Hutter and C. Sengler. Inka, the next generation. In *Proceedings of the 13th Confernce on Automated Deduction*, LNAI. Springer Verlag, 1996.

[HS97] Robert Harper and Christopher Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, 1997.

[Hud00] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge Univ Press, 2000.

[Hue73] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973.

[Jau99] M. Jaume. A full formalization of SLD-resolution in the Calculus of Inductive Constructions. *Journal of Automated Reasoning*, 23(3–4):347–371, November 1999.

[Kap98] Deepak Kapur. Automated geometric reasoning: Dixon resultants, grbner bases, and characteristic sets. In D. Wang, editor, *Automated Deduction in Geometry*, 1998.

[KHH98] Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, pages 317–322. Springer Verlag, LNAI 1421, 1998.

[Kre98] Christoph Kreitz. *Automated Deduction - A Basis for Applications*, chapter Program Synthesis. Kluwer, 1998.

[Kun95] Kenneth Kunen. A ramsey theorem in boyer-moore logic. *Journal of Automated Reasoning*, 15:217–235, 1995.

[Lel98] Pierre Leleu. *Induction et Syntaxe Abstraite d'Ordre Supérieur dans les Théories Typées.* PhD thesis, Ecole Nationale des Ponts et Chaussees, Marne-la-Vallee, France, December 1998.

[Low96] G. Lowe. Breaking and fixing the needham-schroeder public key protocol using fdr. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[Mag95] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[McC94] W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, USA., 1994.

[McC97] W. McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[McD97] Raymond McDowell. *Reasoning in a Logic with Definitions and Induction.* PhD thesis, University of Pennsylvania, 1997.

[MCJ97] W. Marrero, E.M. Clarke, and S. Jha. Model checking for security protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. A Preliminary version appeared as Technical Report TR-CMU-CS-97-139, Carnegie Mellon University, May 1997.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax: An extended abstract. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, pages 434–445, Warsaw, Poland, June 1997.

[MM00] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.

[MP99] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3–4):373–409, November 1999.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[Muñ97] C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en thorie de types*. PhD thesis, Paris 7, November 1997.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[Nec97] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[NM88] 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.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23(3–4):299–318, November 1999.

[NS78] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21:393–399, 1978.

[NvO98] Tobias Nipkow and David von Oheimb. Java-light is type-safe — definitely. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, January 1998. ACM Press.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Procedings of the 11th International Conference on Automated Deduction (CADE-11)*, pages 748–752, Saratoga Springs, New York, June 1992. Springer Verlag LNAI 607.

[OSRSC99] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[Pau83] Lawrence Paulson. Tactics and tacticals in cambridge LCF. Technical Report 39, University of Cambridge, Computer Laboratory, July 1983.

[Pau87] Larry Paulson. *Logic and Computation: Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[PE88] 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.

[Pfe89] 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.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe93] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 1993. To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe95] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[Pfe99] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. In preparation.

[Pfe00] Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[Pol97] Randy Pollack. *Twenty Five Years of Constructive Type Theory*, chapter How to Believe a Machine-Checked Proof. Oxford University Pres, 1997. to appear.

[PP99] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 295–309, L'Aquila, Italy, April 1999. Springer-Verlag LNCS 1581.

[PP00] Brigitte Pientka and Frank Pfenning. Termination and reduction checking in the logical framework. 2000. submitted.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

[PS98] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.

[PS99a] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, volume LNCS 1657, pages 179–193, Kloster Irsee, Germany, March 1999. Springer-Verlag.

[PS99b] 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 LNAI 1632.

[Reu99] Bernhard Reus. Formalizing synthetic domain theory. *Journal of Automated Reasoning*, 23(3–4):411–444, November 1999.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rog92] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1992.

[Roh96] Ekkehard Rohwedder. *Verifying the Meta-Theory of Deductive Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996. Forthcoming.

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[RSV99] I.V. Ramakrishnan, R. Sekar, and Andrei Voronkov. *Handbook of Automated Reasoning*, chapter Term indexing. Elsevier Science Publishers, 1999.

[SA98] R. Stata and M. Abadi. A type system for java bytecode subroutines. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 149–160, San Diego, California, jan 1998. ACM Press.

[SB98] Wilfried Sieg and John Byrnes. Normal natural deduction proofs (in classical logic). *Studia Logica*, 60(1):67–106, January 1998.

[Sch95] Carsten Schürmann. A computational meta logic for the Horn fragment of LF. Master's thesis, Carnegie Mellon University, December 1995. Available as Technical Report CMU-CS-95-218.

[SD99] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[SH93a] Peter Schroeder-Heister. Definitional reflection and the completion. In R. Dyckhoff, editor, *Proceedings of the 4th International Workshop on Extensions of Logic Programming*, pages 333–347. Springer-Verlag LNAI 798, 1993.

[SH93b] Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 222–232, Montreal, Canada, June 1993.

[Sha88] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery*, 35(3):475–522, July 1988.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*, volume 38 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1994.

[Sny91] Wayne Snyder. *A Proof Theory for General Unification*. Birkhäuser, 1991.

[SP98] 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 LNCS 1421.

[Sta99] Mark Staples. Representing wp semantics in isabelle/zf. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*. Springer-Verlag LNCS 1690, 1999.

[Tam97] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming, Second Edition*. Addison-Wesley, 1999.

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, September 1999. Available as Technical Report CMU-CS-99-167.

[Wei97] Christoph Weidenbach. Spass: Version 0.49. *Journal of Automated Reasoning*, 2(18):247–252, 1997.