A Dependently Typed Programming Language, with applications to Foundational Certified Code Systems

Susmit Sarkar

CMU-CS-09-128
May 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Karl Crary, chair
Robert Harper
Peter Lee
Andrew Appel, Princeton University

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

Copyright © 2009 Susmit Sarkar

This research was sponsored by the National Science Foundation under grant numbers CCR-0121633 and CCR-9984812. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Type Theory, Dependent Types, Functional Programming, Proof Carrying Code, Foundational Certified Code

Abstract

Certified code systems enable trust to be generated in untrusted pieces of code. This is done by requiring that a machine–verifiable certificate be packaged with code, which can then be proved safe independently. Safety is defined with respect to a defined safety policy. Recent work has focused on "foundational certified code systems", which define the safety policy as execution on a concrete machine architecture. This makes the safety guarantees of the system more concrete relative to previous systems. There are two advantages. One, we gain in flexibility since the proof producers can use different assumptions and techniques. Two, the parts of the system that must be trusted become substantially simpler.

This work describes our design of a practical foundational certified code system. Foundational systems have new proof obligations, for which we need different proof techniques and verification environments. In common with other such systems, we use an intermediate formal system such as a type system to isolate a group of programs. There are then two proof obligations. A program–specific proof verifies that the program belongs to the group so defined. This is the *type checking* problem. A generic safety proof says that all programs belonging to the group is safe to execute on the concrete machine. For a type system this is the *type safety* property.

Our first main contribution is a functional language, which we call LF/ML. Programs written in this language can be statically proved to be partially correct with respect to their specification. This requires us to program with formal representations. LF/ML, is designed to program with representations in the logical framework LF. Partial correctness of programs in LF/ML can be statically verified with respect to specifications in LF.

Our second contribution is a mechanized proof of the safety of a low-level type system, TALT, with respect to the concrete IA-32 architecture. This proof leverages the Twelf metalogical reasoning framework to prove safety. This allows us to have a complete foundational certified code system.

Contents

List of Figures

Acknowledgments

This thesis has been a long journey, and I would never have reached here if it had not been for a lot of people. I am extremely grateful to all of them, including those I have forgotten to thank below.

I will begin with my thesis committee. Karl Crary found interesting problems for me to work on, taught me a huge amount of things, both technically and about doing research, and was always available to work through a problem and think through alternatives. He also taught me to strive for clarity in thought, speech and writing.

Bob Harper and Peter Lee had many incisive comments on my work and our discussions made the thesis immeasurably better, but I have much more to thank them for. I would like to thank them here for introducing me, respectively, to the Curry-Howard isomorphism and continuation-passing style. Those two separate meetings, within the space of a week in early 2002, impressed me with the elegance and mathematical beauty of research in programming languages, and determined all that followed. I would also like to mention their obvious, and very inspiring, passion for the field.

Rounding out the committee, Andrew Appel, through the lossy channel of phone and email conversations, always had probing questions and technical insights to share. His relentless questions and deep experience helped improve the exposition remarkably.

Next, I would like to mention the incredibly supportive environment within SCS in general and the POP group in particular. Among the faculty, Frank Pfenning was always ready to advise, explain and help. Bruce Maggs deserves special mention for being my first advisor and supporting me when my research interests diverged from his. I learned a lot from the many discussions with the various members of the POP group, and in particular Kevin Watkins, Derek Dreyer, Brigitte Pientka, Tom Murphy VII, and Dan Licata. Meanwhile, Sharon Burks and then Deborah Cavlovich performed many complicated spells of magic through the years to save my skin more times than I can remember.

Pittsburgh was a very pleasant place to spend well over five years, principally because of the many good friends I made there. Thanks to all of them. Special mention is due to my housemates Arjunan Rajeswaran and Satashu Goel. In addition to their general forbearance and support, Arjunan gave much sensible advice and helped me maintain an optimum level of activity, and Satashu kept me well-supplied with tea and movies.

I finished working on this dissertation in Cambridge, where my new supervisor Peter Sewell has my gratitude for employing with good cheer a post-doc without a doctorate for years on end, and allowing and even encouraging me to finish up when there were other interesting research areas to explore. Thanks also to the convivial and supportive PLS group, Scott Owens and Tom Ridge in particular.

Finally, I end by thanking my family. My parents brought me up to question, probe and learn, and my debt to them is immeasurable. They waited with much love and affection and more than a little impatience through the much prolonged thesis process, and helped and supported me from afar as much as they had previously done from close by. I am also grateful for the tolerance, companionship and love shown by Veena Muthuraman, who during these long years became in turns confidante, fiancée and spouse, and throughout remained a friend.

Chapter 1

Introduction

Computers are now being used in ever-increasing numbers to perform a variety of tasks. Every person today interacts with multitudes of programs running on various machines every day. Unfortunately, whether by malicious intent or negligence, programs often go wrong, sometimes subtly, and sometimes catastrophically. As we come to be increasingly dependent on them, the consequences become ever graver.

1.1 Creating Trust in Trustless Environments

Let us consider the issue from the point of view of the principals. As users of programs, who we will call *code consumers*, we want to be able to trust that the programs we run have certain desirable properties. We will leave the kinds of properties we deal with unspecified until the next chapter. To motivate the reader, one particular property we will be interested in is that of memory safety. This says that the program will not access parts of the memory not explicitly allocated to it by the runtime system.

In general though, given an arbitrary program, and assuming no further information, it is intractable to deduce interesting properties of it. Code consumers should not be expected to spend unbounded resources on verifying all the programs they run. Of course, in practically all of current practice, almost no checks are attempted.

This situation is already not ideal in the circumstance of ordinary users running programs written by large corporations. It becomes untenable when the developer of the code is unknown or untrusted. Consider the widespread use of applets on the web, written by developers whose competence and motives we know nothing about. Users are understandably wary of running these applets, and they are usually given very restricted environments to execute, if at all.

To take another example, and one which motivated the current work, there has been much current interest in grid computing, where a lot of idle computing power in generic consumer hardware is potentially available connected to the Internet. Projects such as SETI@Home [SET00] or Distributed.net [Dis04] have tried taking advantage of this by utilizing these spare cycles. On the other hand, computer owners, or code consumers, are understandingly skeptical of downloading and running arbitrary pieces of code. Thus, the usage of the grid has been restricted to such developers that can obtain the trust of a large number of code consumers. Absent such trust, developers cannot utilize the idle computing resources. Also, grid applications have tended to be ones which are massively parallel, requiring no interaction except with a central trusted server. Applications which are less parallel, and require interaction with other machines on the grid have not taken off. At least part of the reason is to do with code consumers not trusting any other participants of the grid.

This brings us to another principal, the developer of the code, who we will call the *code producer*. As code producers, we want to get code consumers to trust in our programs. In many situations, the code consumer may not know or trust us. We want a mechanism whereby we can convince the consumer that our programs satisfies the properties they expect. This will help us stand out from arbitrary producers.

1.1.1 Earlier Approach: Runtime Checks

The problem of generating trust in arbitrary pieces of code is well-studied, and various solutions have been proposed in the literature. Before outlining our proposed solution, we will now review a few of these solutions.

One possible solution is to impose checks and constraints at runtime so that the code cannot perform unsafe actions. The conditions are enforced by some entity which is present at the execution site, that is, on the same machine as the code consumer. This enforcement can be done through either hardware or software means. This approach can be characterized as the use of dynamic checks.

The hardware-based approach is the primary mechanism in most modern operating systems. Here, the code to be executed is not allowed direct access to the system. Instead, it is isolated in its own address space. All accesses to global system resources are mediated by the runtime system, which can enforce policies of access. There is simply no way that the program can access prohibited resources directly.

This method is relatively simple to trust and implement, since all accesses are mediated. There is no need to trust arbitrary code. The user must trust merely the runtime system, a centralized and often used part of the system.

There are some disadvantages to this approach. First, this requires specialized hardware and operating system support. We would like not to impose major entry barriers to donating spare cycles to the grid. Second, extending the set of properties to be checked is extremely difficult. Imposing a new policy on a resource likewise requires careful planning and study of the interaction with the system.

A different approach can be taken by constraining the code by software means. In particular, a variety of systems accept code only in the form of bytecode, a species of code that is easy to analyze. This code is then executed by a trusted interpreter. Again, the code consumer need not trust arbitrary pieces of code. The centralized part that must be trusted is the interpreter, which only performs safe actions.

This approach benefits by not requiring as much special support as with hardware. Further, the design of the bytecode is by the writer of the interpreter. This means that information to guide the interpreter can be embedded in the bytecode.

Another possible software mechanism is by instrumenting the code to insert run-time checks. Any possibly unsafe code is wrapped by wrapper code that ensures the operations are aborted if they are indeed unsafe. This technique has been dubbed Software Fault Isolation [WLAG93].

There is one large problem with such dynamic checks. Errors, if any, are encountered at the time of program execution. Most commonly, an error will require aborting the program execution and performing associated cleanup. This involves complex procedures to ensure all resources are properly reclaimed. Additionally, a lot of processing work already performed has to be thrown away.

A further disadvantage in these approaches is that they impose a runtime cost. Computational resources on the grid, while plentiful, are still not absolutely free. This is particularly unfortunate for scientific computing, a big part of potential grid applications, since the whole point is executing code which is hard or impossible to run using the resources of a single machine. Certainly, approaches like dynamic compilation, the so-called just-in-time compilation strategy, or compilers inserting run-time checking code, can significantly reduce or eliminate this burden. Unfortunately, these compilers are rather complex beasts. A rather large piece of software must be trusted. The situation will only worsen with compilers performing clever optimizations to improve performance, and with expectations of different properties to be checked.

1.1.2 Our Approach: Static Checks

We would like a solution where we do not have to constrain programs at run time. Rather, we would like to be able to check programs before execution. This is technically referred to as static checking. We have noted already that it is intractable to prove properties of arbitrary programs. On the other hand, given certain additional information, such properties can be verified easily.

A proof in Martin-Löf's view [ML96] is defined as the evidence for concluding the fact or judgment it proves. The relevant issue for our purposes is that a proof is easy to verify, hard though it may be to come up with a proof. Concretely, we know how to build simple and efficient verifiers for proofs. This is the case even if we do not have enough information to produce the proofs.

The fundamental insight of Necula and Lee [NL96] is that this property of proofs can be used to create trust in trustless settings. Consider for example, a proof that the program satisfies the property of interest. Suppose that the consumer insists on getting the proof in addition to the code. Then she can easily verify the proof and convince herself that the code is in fact safe to execute.

The point for the consumer is that the proof is something that can be easily verified. In principle, any property of interest can be encoded as a proof. More generally, we can encode information with the code that can be used to prove properties of the code. The consumer need not spend a lot of resources in verifying that the code satisfies the properties she expects.

It might be asked how this proof arises in the first place. The code producer is in a position to generate the proof required. If the code is in actuality safe, the producer knows why it is safe, and it remains to transmit this knowledge to the consumer. Thus such systems shift the burden of proof from the code consumer to the code producer. It can create trust where no trust existed before, without the use of any central authority.

In general then, the idea is to insist that producers package information to help prove properties of their programs together with the program itself. Such information could be in the form of a proof of safety, as in the original Proof Carrying Code system [Nec97]. We shall call the general form of such information a *certificate*, since as we shall see, the additional information need not, and in practice will not, be a literal proof.

1.2 Minimizing Trusted Components

In any trust system, we must carefully study what components of the system must be trusted. All these components must work correctly for the system to do what it is meant to do. Conversely, any mistake in one, or in the interaction of different parts of the system, creates an opening that can be exploited by malicious code. All these components are grouped together in what is called the *trusted computing base* (TCB) of the system.

To build confidence in the system, the trusted computing base should be as small as possible. There should be as few components as possible that we must trust. This is important because the components may themselves be faulty. Particularly for software components, getting them right in isolation is difficult enough. There is also the further issue of getting it to work right with the other components, so the interface has to be absolutely correct. Further, the components should be as small as possible. Smaller components are easier both to get right as well as check independently to be correct. What is crucial here is not size in terms of lines of code or similar crude measures, but the conceptual complexity of the system.

A second, and maybe more important reason, to remove or simplify trusted components is that of flexibility. Any trusted component is one that all code producers are stuck with. For a grid application, we want to attract a wide variety of producers, with a wide variety of problems. Restrictions imposed by a particular component which may be perfectly all right for one producer may often be unduly onerous for another.

In the setting of a certified code system, the trusted computing base includes the definition of what properties the code consumer expects (technically called the *safety policy*), the mechanism to express the proofs, the process of verifying the proofs, and the execution environment. The safety and integrity of the system depends crucially on these components.

1.2.1 Previous approach to Certified Code

Early certified code systems used a type system as a key component of their trusted computing base. Their definition of what it means for a program to be safe was well-typedness in a low-level type system, whether implicitly in the PCC system [Nec97], or explicitly in the Typed Assembly Language (TAL) system [MWCG99]. This is also the idea behind the Java Virtual Machine [LY96], though in that case the language of bytecodes was further removed from the actual code being executed.

This is an entirely reasonable way of proceeding. Researchers have used type systems to serve as a method of proving safety of code using Milner's famous slogan: "Well-typed programs do not go wrong". Various

type systems have been devised to help in proving safety properties, from memory safety to guaranteeing abstraction boundaries at interfaces to resource usage limits. It is a well-studied method of ensuring such properties.

In the setting of a certified code system though, there are a few problems:

1. Code consumers do not really care about the type system in use. The type system is good only in so far as well-typed programs really do not go wrong. That this necessary property is enjoyed by the type system in use is a theorem with an often delicate proof. Some of the systems have such a published proof, and some do not. In any case, the scrutiny applied to such proofs can vary widely.

2. It is not even clear that the proofs are of interest to the end user. To pick on one example, the TALx86 [MCG+99] system was the realization of the idealized TAL system that would be used by the end user. TAL did enjoy a proof of type safety on paper that appeared in conference proceedings and journals. But it was for an idealized machine. The concrete system in use, TALx86, was never proved safe. The safety was argued by analogy with the admittedly similar TAL. The authors understandably did not want to redo the work of proving safety for a closely related language.

3. A type system is a formal abstraction. Concretely, to check programs, we need a way of verifying that the program indeed is well-typed in the type system. This is done by the type checker, a program that must also be trusted. It is perfectly plausible that the type system is safe and proved to be so, but the type checker perhaps does not correctly implement the type system and accepts unsafe programs.

1.2.2 Foundational Systems

There have been efforts to remove or minimize parts of the trusted base of the system. As already discussed, these improve the flexibility of the system, as well as make it easier to trust. Efforts in this direction have ranged from simplifying the verification mechanism [BL02] to removing the type system from the trusted base [App01]. We will be concerned with the second idea.

Recent projects, beginning with the Foundational Proof Carrying Code system [App01], do not consider the type system as part of the trusted components of the system. Rather, they go to the heart of the matter by proving code safe relative to the concrete machine the code is executed on. Of course, a type system is an useful abstraction to help in the proof. The key insight is that the type system is just a proof mechanism. It is not necessary to trust in the type system, provided the proofs can be verified. Such systems have been called *foundational*.

Foundational systems are conceptually much simpler to trust, since the type system, a large and complex component, is not trusted. There is a price to this simplicity. To generate the required level of trust, we now have a lot of proof obligations. In particular, the proofs that were implicitly assumed by the system earlier have to be now transmitted in a way that can be verified by machine.

There are concerns with practicality in foundational certified code systems. The proofs we refer to are complex, and require careful engineering so that they are small enough to transmit. Recall that the proofs are sent across the network. Further, they have to be verified at the consumer side. We do not want to use huge amount of resources in time or space to verify the proofs.

Even more important, proofs have to be developed by code producers. While we are willing to put the burden on the producers, we do not want to make it extremely hard to come up with the proofs. This point gains more importance if we think of the system as flexible and supporting multiple producers with different techniques. While producers may be willing to invest a large effort in one particular proof, they may not be willing to invest the same effort for every different little program they produce. This points to the fact that the intellectual work must be modularized. The checking work should be factored so that much of the obligation is discharged once and reused multiple times.

We have been talking of proofs in our discussion. For the purposes of certified code systems, and even more so for foundational systems, what is important is that these proofs be verifiable by machine. This is of interest to code consumers, who is not expected to be an expert in proving properties of programs or proof systems. Also, even experts can get bored very quickly. We need a verifier that cannot be fooled, and one that does not get bored. This inevitably means that all proofs must be formalized so that it can be checked by machine.

Thus, an additional motivation to study foundational certified code systems is that they are an investigation of formalizing programming language metatheory. Mechanized proofs have attracted attention [ABF+05] recently as representing the highest standards of trust in proofs. Foundational certified code systems are an ideal application domain as well as test-bed for mechanized proofs, since the proofs are of central importance.

1.3 Correctness by construction: the language-centric approach

Foundational certified code systems have a sharply demarcated trusted computing base. One key component of this trusted base is the verification that a particular program has certain good properties. There is a proof engineering issue here, since some of the work, even with careful factoring, is ultimately specific to the particular program being run. One option is to perform this check by the producer, before the code even reaches the consumer. The problem is that then the evidence for the validity of the check has to be transmitted to the consumer. This evidence has to be something the consumer can trust. There is a tension with the flexibility of the system, since the evidence must also be expressed in a generic enough metalanguage that can express different proof strategies by the producer.

A better option is to perform the check on the consumer side. But since we want to maintain flexibility, the checker must be tunable, and ideally, written by, the producer. We do not want to preconstrain the checkers that can be written to trusted users or large engineering efforts, since that detracts from the goal of being generic in the proof strategies to be used. The question then becomes how we can trust the checker. Our approach is to solve this knot by adopting a language-centric view. We design a new programming language, dubbed LF/ML for reasons we explain below, which defines a way of expressing checkers that can manifestly be trusted.

The design goal for this language thus is to be able to write programs that, by construction, satisfy strong correctness properties. We bring to bear type-theoretic ideas to design such a language. Type systems have been extensively used to design languages with this flavor, but here we require a type system going well beyond the usual memory and abstraction safety properties. We now require type systems able to prove so-called partial correctness, that is, correctness upto nontermination. Proving full correctness is both technically difficult as well as unimportant for our purposes, since the checkers written in this system will run as an untrusted process, with low privileges and bounded resources.

A subsidiary design goal of our language is that the programming language be functional, allowing functional algorithms to be expressed. A similar guarantee can be provided by some logic programming languages, in particular, the Elf language within Twelf. It is the case however that realistic checkers will need a variety of algorithms, and functional programming languages can express a richer variety of programs than can be expressed in logic programming. We do not want the producer who is writing programs in LF/ML to be precluded from writing the kind of code that she wants. Further, efficiency of the programs written in the language is also a key concern, since ultimately, the checking programs will be run by the code consumer. In our examples, we have used many different patterns of control flow, and rich (polymorphic) data structures. In another direction, in this work we have not found much use for, and indeed not worked out the theory for, the use of state, which precludes imperative features.

Our programming language is based on adding a form of dependent types to the widely used core-ML language. ML is well suited for the kinds of programs we want to write. Enriching the type structure of the language with dependent types allows us to write precise invariants of the code, and have them checked statically. The invariants are correctness specifications. The predominant specification language in specifying the logical systems we are interested in is the Logical Framework (LF). Our language is thus ML with its types enriched via indexing by LF, justifying the name LF/ML.

There are some technical challenges to this approach. While we will discuss the issues in detail in the rest of this work, one high-level point needs to be made. A key role is played in the LF methodology of representations by the use of higher-order abstract syntax. The possibility of using higher-order abstract syntax depends delicately on the metatheory of the representation language, here LF. It is often noted that LF is so well-suited for higher-order abstract syntax precisely because it is logically so weak. In particular, any use of logically strong computational devices such as case analysis or recursion destroys this delicate

balance. We isolate the problem by syntactically separating the representation layer, which is closely related to LF, from the computational language, which is closely related to ML.

We demonstrate the practicality of the LF/ML language by writing a typechecker for a low-level type system in such a language. The type system is quite sophisticated, requiring a correspondingly detailed checker. By construction, the typechecker is statically guaranteed to accept only well-typed programs.

1.4 Contributions of this thesis

This thesis makes the following claim:

> *A dependently typed functional language can be used to statically prove the partial correctness of programs, in particular, type checkers within foundational certified code systems.*

This thesis is concerned with building a practical foundational certified code system. This can be used to generate trust in trustless settings. We have used this work within the ConCert framework for grid computing [Con01] to generate trust. Towards this end, we demonstrate formalization techniques, proof techniques, and tools to produce and verify the proofs. It is flexible in that it can be instantiated with different systems that can be used within the framework.

Our major technical contribution is the language LF/ML for writing statically verifiable programs. It was designed keeping in mind the practical and theoretical demands of a foundational certified code system, but has much wider applicability. It is possible to reuse specifications of logical systems within LF, a well-understood and mature technology with various systems already encoded. LF/ML then lets us specify the partial correctness of programs with respect to any such system. It is a functional programming languages, based on core ML, including ML-style polymorphic types, algebraic datatypes, and exceptions, but no ML-style reference types.

As a demonstration of the practicality of our system, we have created one particular instantiation. This system, dubbed TALT, comes with completely mechanized proofs within the framework that we have developed. The instantiation need not be trusted, since we have produced complete proofs that can be mechanically verified. Our work goes beyond comparable systems in the expressivity of the system, that is, the programs that can be verified within the system. Our techniques, informed by advances in logical frameworks, mechanical reasoning and new type systems, help us in creating the system in a much shorter time frame than comparable projects.

The other major contribution is the proof effort for the TALT system. We have developed the first mechanized proof of safety for a practical low-level type system. The type system is practical in the sense that it can be used as the target for compiling practical programs. The proof is foundational in nature because it targets a concrete machine architecture. This proof was developed using the Twelf metalogical framework that allows reasoning about logical systems in LF. It served as the first use of the Twelf metareasoning engine for a complex proof, though others and ourselves have later used the system for other large proofs.

1.5 Related Work

1.5.1 Type Theoretic Approaches to Program Verification : Reflection

The problem of verifying functional programs against their specifications has attracted much type theoretic interest. The LCF project [GMW79] observed that so-called tactics would be guaranteed to produce valid proofs by the use of a typed language (ML) as its metalanguage. The dependent type theories studied by Martin-Löf can in fact provide stronger guarantees, as described below. This was implemented among others by the NuPRL [CAB$^+$86] theorem prover.

NuPRL went further, since it is a programming language as well as a proof language, tactics to generate proofs can be programmed in it. The use of dependent types extends the notion of validity from merely being a valid proof of some theorem to the fact of being a valid proof of the particular theorem of interest.

NuPRL can also use the idea of reflection in type theory. In such a system, the provability predicate is reflected into the language, and can be used in proofs like other predicates. This allows proofs of theorems to use metatheoretic arguments. The use of reflection allows the writing of tactics that are provably correct.

Knoblock [Kno87] discusses these methods in great detail. He divides tactics into three classes: complete tactics, partial tactics and search tactics. Complete tactics will produce a proof, while partial tactics will reduce the proof obligation. They are both guaranteed statically to succeed. Thus, if they apply, the tactic is guaranteed to terminate, and produce a solution or a reduced goal. A more general class of search tactics performs search, but if it terminates in success, is guaranteed to produce a valid proof.

1.5.2 Verifying Functional Programs through Theorem Provers

A different approach based on type theory is explored in theorem provers. This relies on the observation that the proof representation in type theory can be looked on as a functional programming language. By looking at the constructive content of proofs, one can often extract functional programs. This has been the case with the Coq system, based on the Calculus of Inductive Constructions [PM93]. In Coq, we can build proofs within CIC either directly or by using a tactic language. Coq's concrete syntax for the CIC is called Gallina, which also incorporates a variety of syntactic sugar to make expressing datatypes and proofs reminiscent of functional programming in ML. Indeed, since CIC is a theory of functions, Gallina can itself be used as a programming language, and indeed, have been so used [Ler06]. Programs in an unsafe variant of CAML can be extracted from proofs in the Coq system [PM89, Let03]. This relies on erasing portions of the proof that are noncomputational.

One feature of the proof representation language is that it usually encodes only total functions, since total functions correspond via the Curry-Howard interpretation into valid proofs. This has the effect that any programs which are extracted are actually total functions. Put another way, programs to be verified have to be written in a way that is provably terminating. In some cases, notably in a checking application for certified code, the property of termination of the program is not important. Proving total correctness (program is correct and terminates) instead of partial correctness (program is correct if it does terminate) is an additional proof burden.

The main philosophical difference between Gallina and our language, LF/ML, is the fact that Gallina or CIC integrates in one language the proof representation and the computation building the proofs, while LF/ML separates these layers. This enables LF/ML to express partial functions easily, while Gallina forbids functions which are not total, indeed, provably total by a syntactic check.

Another theorem prover called Epigram has been recently devised by McBride and McKinna [MM04, AMM05]. They work in a very rich type theory [Luo94] with a hierarchy of universes, and add constructs and notation for allowing definitions of functions by pattern matching. Again, since their language allows only total functions, they can prove total correctness of programs. As an example, they prove the correctness of a type checker for the simply typed lambda calculus. Encoding reasoning about partial programs is a planned feature of Epigram, by restricting partial functions within a partiality monad.

1.5.3 Metaprogramming

Proofs are formal objects that need to be manipulated. We draw on work in metaprogramming, which is concerned with the manipulation of formal objects that are actually programs. Metaprogramming systems have been studied from a variety of motivations, including partial evaluation and run-time code generation [WLPD98]. Studies in this area include Davies and Pfenning's series of papers [Dav96, DP01]. The MetaML programming language [TS97, TBS98] defines a functional language with additional primitives for meta programming. They describe a language with facilities for building and running code fragments. A type system has been defined for this language [MTBS99], taking ideas from Davies' work, which ensures that code fragments executed do not fail by trying to execute open code. The type system is proved safe, in the absence of effects or polymorphism. The major challenge in this line of work is to deal with open and closed code (*i.e.* code with free variables present or absent respectively) properly.

Pasalic *et al.* [PTS02] build on the MetaML idea by defining a dependently typed language for staged computation. They show how to write a typechecker in their language. They use datatypes indexed by term level constructs. They reflect the term level in special constructs at the type constructor level, and use kinds to judge validity of such constructs.

A system that also aims to manipulate proofs is Schürmann *et al*'s Delphin project [Del04]. Within this project, the theory for a functional language manipulating objects from the LF language is under development [SPS04]. The operational model of the ∇-calculus is a hybrid of logic programming and functional programming. There is a nondeterministic pattern match operator, as also uses of existential (unification) variables.

1.5.4 Dependent Type Theories in Programming

Xi's DML system [XP99, Xi98] introduced dependent types in a ML-like language. This system had types dependent on terms from index domains. This work uses the index domain of natural numbers to point out the usefulness of the concept, such as in array bound check elimination [XP98].

This work was extended by Dunfield, in the series of papers [Dun02, DP03, DP04]. These systems combine index domains such as Xi's with a powerful language of refinement types [FP91]. This extends the power of using dependent types through the use of property types and intersection and union types over such properties. This work also deals with type inference issues, by treating index domain constructs as implicitly typed. In contrast, our system is explicitly typed with the indices under programmer control.

Extending DML in a different direction, Xi *et al* [XCC03] extended the notion of algebraic datatypes to inductive types at higher kinds, which they call guarded recursive datatypes. They showed the relation to DML style dependent types, and in a later paper [CX03], presented a direct translation from a dependently typed meta programming language into guarded recursive datatypes. Peyton Jones *et al* study type inference problems for similar systems [JWW04].

Recent work in this line has led to so-called Applied Type Systems [CDX05]. These systems formalize a general system of dependent types with various index domains. They use a domain of constraints over indices which is used in the definition of type equality. One particular instantiation of the system [CX05] allows means for the programmer to construct proofs, which witness the constraints used in the system. The proof domain is an intuitionistic variant of higher-order logic together with inductive types.

1.5.5 Certified Code

The use of certified code was pioneered by Proof Carrying Code (PCC), in the series of papers [NL96, Nec97, Nec98]. These describe an architecture where proofs of the safety of a code are packaged together with the code itself. The safety policy in this system is specified by the code consumer. A trusted *verification condition generator* (VCGen) looks at the code and produces a series of verification conditions, which are theorems that must be proved. The code producer has the responsibility of providing the proofs of these theorems.

In principle, PCC can encode arbitrary safety policies and provide proofs for such policies. In practice, however, the proofs have to be produced by the code producer in an automated manner. This is usually accomplished by starting with a typed high-level language, and maintaining high level invariants through the compilation process [NL98]. The compiler could thus emit the necessary proofs, for properties such as memory safety, or satisfying the bytecode verifier of the Java Virtual Machine Language. In the elaboration of this idea into the SpecialJ compiler for Java [CLN$^+$00], the system encoded into the VCGen the knowledge that the source language was Java. In effect, the safety policy was tied to the type system of Java.

The TAL project defined a low-level type system for an idealized RISC-based assembly language in a series of papers [MWCG99, MCGW02]. Type checking of the code provided assurance that the code was memory-safe, with the standard theorem of type safety. Type systems provided a formal logical specification of a class of programs guaranteed to be memory-safe. Further, a compiler could use typed intermediate languages to maintain type information throughout the compilation process. Thus, a typed high level language could be compiled to typed assembly language. These systems would fix the target type system, thus losing the

flexibility of PCC in principle. However, the use of types could lead to a formal translation, and typing annotations should in principle be small.

The concrete realization of TAL was TALx86, a version of TAL specialized to the IA-32 architecture [MCG+99]. This specialized the idealized TAL to the IA-32 architecture. The metatheoretic statements of type safety was not proved, but was loosely justified by analogy to TAL.

1.5.6 Foundational Certified Code : Checking Type Safety

Appel *et al.* [App01, AF00] advocated a more foundational approach, in which the type system is not trusted, but explicitly proved safe. This would rely only on the axioms of a trusted, well-understood logic (higher-order logic, in the FPCC approach). The safety of programs would be proved relative to a concrete architecture.

The FPCC system developed a denotational semantics of types in higher-order logic. Complex semantic arguments had to be encoded to deal with advanced type systems with references and recursive types. Next, a concrete architecture was modeled as a formal system. Safety of code was proved relative to this concrete architecture. A low-level type system [CWAF03] was defined and explicitly proved safe.

Hamid *et al.* [HST+02] developed a different proof for a foundational system. They mechanized a standard syntactic proof of type safety. This approach, based on defining an operational semantics for the type system, and proving safety by simple structural induction techniques, is considered to be simpler to generate in paper proofs [WF94]. They defined a type system called Featherweight TAL, and proved it safe relative to an actual machine architecture.

Foundational certified code systems improve the reliability of the system by removing a key trusted component, the type system. Minimizing the trusted computing base is a step towards increasing confidence in the system, since a smaller base can be more easily trusted. We may mention Bernard and Lee's *temporal logic PCC* [BL02], which removed the VCGen from a PCC system. This was done by specifying the safety policy in temporal logic, which led to both expressive safety policies as well as the elimination of a Verification Condition Generator. This simplification of the system by removing a big trusted component contributed to better confidence in the system.

1.5.7 Proof Checking in Certified Code Systems

Certified code systems, both classic certified code and foundational systems, suffer from the problem that proofs are large. The work of Necula and Rahul [NR01] pioneered the oracle concept, which made possible extremely short encodings of proofs. This method is based on a view of guiding a theorem prover by describing the choices it must make to find a valid proof. If the choices that can be made by the theorem prover are predictable (that is, we know the theorem prover's algorithm), these choices can be encoded in an extremely compact manner. The system described in the work relied on a specialized logic, or type system.

When there is no trusted logic or type system, one typically wants a program to perform type checking of the program. Wu *et al.* [WAS03, AMSV03] developed a logic program interpreter. This was a interpreter that could be written in a very small number of lines of code. The interpreter worked in a fragment of logic very close to Prolog. Programs in this language are mechanically checked with respect to a semantics in higher-order logic. This approach leads to a verified type checker, written as a logic program. Later, Appel and Felty observe [AF04] that the use of dependent types can statically certify correctness of code. This avoids runtime errors in proof construction in the context of a theorem prover.

The Open Verifier Framework [CCNS05] advocates a similar layering of proof effort by shipping new proof verifying programs rather than proofs. This minimizes the amount of proofs sent in proof carrying code approaches, and also leads to well-understood proof techniques. On the issue of having to trust the proof verifiers, they advocate having the verifier emit a proof of the soundness of each of its steps. By having a restricted proof language, that of Horn clauses, the verifier can itself be verified dynamically.

Chapter 2

Overview of our system

We will now sketch out what the pieces of our system are, and how they fit together. We discuss how the users interact with the system, and what is expected from every principal. The technical details are postponed to later chapters, and for now we give an informal survey.

Our work was done in the context of the ConCert system [Con01], a system for managing grid applications. Code consumers in this setting are owners of computers who wish to donate spare cycles on their machines. Code producers are developers who wish to utilize these resources for large programming tasks. To set the stage for our foundational certified code system, let us examine the actions each of these classes of users perform.

A user who is willing to let his machine become part of the grid's resources registers with the ConCert system. He has to download and execute a program that runs on his machine, which we will call the steward. The steward accepts programs for execution from other machines. It performs the checks described later to ensure the program is safe to execute. Once it is satisfied, it executes the program in the background. Finally, it packages the result of the computation and returns to the caller. Thus the steward is the agent for the code consumer.

The developer for the grid is responsible for satisfying the steward that his programs are indeed safe to execute. He will do this by packaging certain extra information, or certificate, together with the code. He then waits for the result, and may use the result in further computation.

Programs may spawn other processes, provided they can simultaneously provide certificates. They can communicate with other processes. There is no requirement or expectation that the computation consists of a large number of identical copies of a program. Much more complicated patterns of parallel computation can thus be exploited on this grid.

Further, there is no need to trust a central authority. Code producers can directly communicate with code consumers in providing code and certificates. Different protocols can then be run to ensure properties such as fairness and resource allocation, but that is outside the scope of this thesis.

We will now focus on what the code consumer expects, and what kind of certificate is needed from the code producer. This is the mechanism of creating trust in this decentralized trustless setting. We will create this trust by using a foundational certified code system.

2.1 System Overview

2.1.1 Policy

The policy is what the code consumer expects to be satisfied by every program that is executed. Security policies are defined in the literature [Sch00a] as an arbitrary predicate on sets of executions of the code. A program represents all possible sets of executions of the code, and satisfies the security policy exactly when all the possible executions satisfy the predicate.

A more restrictive notion is that of a security property [AS85]. This is the class of security policies that can be defined by looking only at an individual execution. This excludes interesting policies that depend on relating different executions, such as an information flow policy that prohibits correlations between some sensitive data and program behavior.

Security properties can be further divided into two classes: *safety properties* and *liveness properties*. Safety properties [Lam77] intuitively state that a defined class of bad things does not happen, and includes such things as memory safety (access to unallocated memory does not happen), partial correctness (completion of execution without satisfying post-condition does not happen), and resource usage policies (access of resource without following protocol does not happen). Liveness properties [Lam77] say that something good eventually happens, and includes such properties as availability of some resource (resource is eventually made available), and termination (program eventually terminates). It has been proved [AS85] that any security property can be expressed as an intersection of a safety property and a liveness property.

In our system, the policy is restricted to stating safety properties only. We thus call our policy the *safety policy* of the system. This may seem to restrict the kinds of things we can check. However, in practice, we can approximate liveness properties by a stronger form of liveness called a *bounded liveness property*, which specifies that the good thing happens in a given amount of time. Bounded liveness properties are safety properties, for example, bounded termination says that the program terminates within a specified amount of time.

The code consumer publishes the safety policy that it expects to be satisfied by all programs. In the current system, the safety policy is memory safety for x86 code. This ensures that unallocated memory is never accessed. The runtime system executes pure x86 programs and provides a library of support functions. The support functions provide facilities like input/output, interaction with the grid, and memory allocation primitives. Support functions specify their input and output behavior, and programs are not allowed to assume anything about the functions apart from the specifications. Memory allocated to the support functions is held abstract and not allowed to be accessed by the program. The safety policy is described in detail in chapter 3.

The safety policy in a certified code system is a formal artifact. This is the case since satisfaction of the policy has to be verified by machine. This brings up the question of how to represent formally the policy. In common with a lot of certified code projects [Nec97, App01], we have used the Logical Framework (LF) [HHP93] to formalize the policy. Some other projects [HST+02] have used the Calculus of Inductive Constructions as implemented in Coq [PM93]. We have found that LF provides elegant and short encodings of formal systems. Further, the standard methodology of representation in LF provides straightforward proofs of the "adequacy property", which states that the formal representation is isomorphic to the notion to be formalized.

2.1.2 Certificates

The code producer is supposed to satisfy the steward that a program he has written is indeed safe within the meaning of the safety policy in operation. We imagine that code producers will want to write multiple programs. The work of proving a program safe can and should be factored. This means that a major part of the proof be done at one time, leaving smaller portions of the proof to be done for every new program to be written by the producer.

The factoring of the proof is started by isolating a group of programs. The code producer declares that he will be writing programs within a group that he defines himself. It is possible, and indeed plausible, that different producers will differ on the idea of which group of programs they are interested in writing. We call such a group defined by a *safety condition*, which is producer-defined. The group may consist of just one program of interest, or a variety of programs. The producer at this stage is allowed to write a condition which accepts all possible programs too. Clearly, we should not trust whatever condition the producer comes up with. The reader may imagine that this group is defined by a type system, as it will be in practice, but this is certainly not dictated by our system.

Next, the producer provides a proof that all programs which lie within the declared group are safe within the meaning of the safety policy. This proof relates two formal systems, the *safety condition* and our *safety*

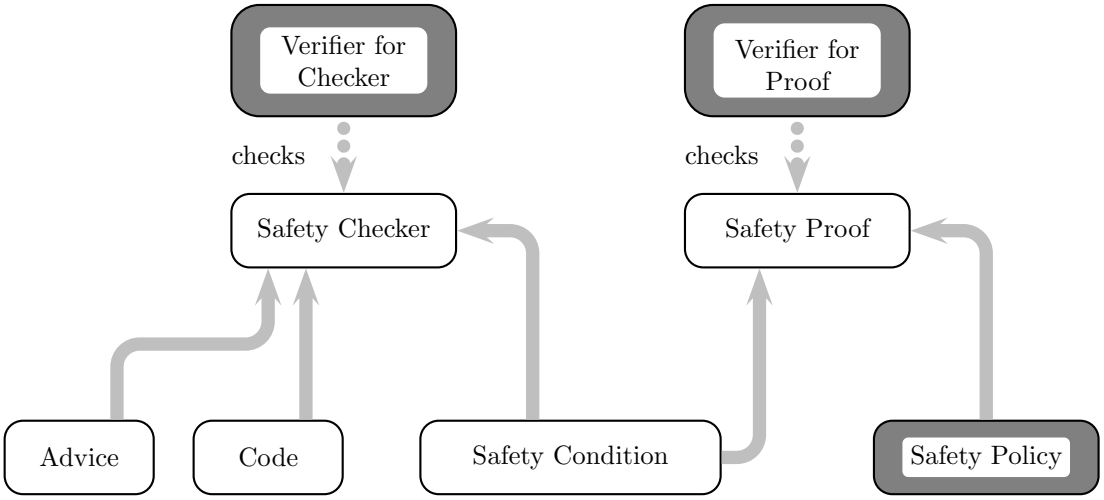

Figure 2.1: System Components

policy. It is called the *safety proof* for the condition. If the safety condition is indeed a type system, then this safety proof is the familiar proof of type safety for the system. The type safety is with respect to the semantics implied by the safety policy, in our case the operational semantics of x86 code.

The final step in the proof is to show that the particular program that the producer wants executed actually lies within the group defined by the *safety condition*. Combined with the safety proof, this implies that the program actually is safe with respect to the safety policy. Supposing again that the safety condition is defined as a type system, this step is showing that the program is well-typed within the type system.

In fact, we invite the producer to provide his own checker for his system, which we will call the safety checker. The safety checker checks whether the program belongs to the defined group. This checking program can inspect the program, and may need some advice as other data. It is possible that no advice is needed, and we leave this design decision up to the producer. For a type system based approach, the safety checker would be a type checker for the type system. The advice would be annotations or metadata included with the program to help the checker.

The components of the system are presented schematically in figure 2.1. The trusted computing base is marked in gray, and everything else is untrusted. The solid arrows represent the flow of information. Thus the safety checker takes code and some possible advice, and checks that it is indeed within the safety condition. The correctness of the checker is verified by a trusted component of the system. The safety proof witnesses the fact that all programs which pass the safety condition are safe for the safety policy. This likewise is verified by a trusted component of the system. The vertical direction in the figure denotes roughly the level of abstraction of the component. We notice that raising the abstraction level of the trusted components lets us be flexible with respect to safety conditions in use.

The required components which should accompany executable code are presented schematically in figure 2.2. This is a conceptual view of what the steward will expect of the producer to run the code. In this figure, the code (first component of the package) is raw binary code that will be run. The other components together help prove the code safe.

The view is more conceptual than practical. In particular, since we imagine the producer will want to reuse the safety condition for multiple programs, it is advantageous to cache as much of the package as possible. In particular, the lower block of the package (excluding the program and advice) is generic and

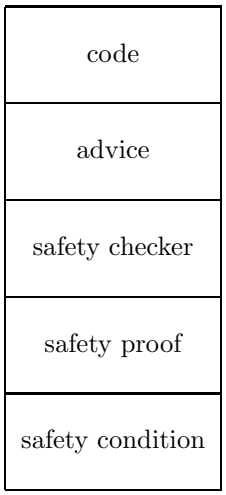

Figure 2.2: A Certified Binary : Conceptual View

program-independent. Thus it can be checked once and for all and then cached on the consumer side, if so desired. Programs using this safety condition then need only be the raw code and such annotations necessary for the particular safety checker.

We will now focus on the different components of the system. We will discuss how the safety proof and the safety checker are expressed. It is clearly critical for the integrity of the system that these are correct and trustworthy. We will talk in general terms about the technology we use. More details will be given in subsequent chapters.

2.2 Safety Proof

The safety proof is supposed to prove that any program satisfying the safety condition is safe according to the safety policy. This is thus a relation between two formal systems, the safety condition and the safety policy, also known technically as a metatheoretic property (the prefix "meta" indicating that it is reasoning about a formal system. The required property is soundness of the safety condition with respect to the safety policy. If we consider the safety policy as given by the semantics of the concrete machine and the safety condition as specified by a type system, this is the familiar type safety property.

One standard way of proving such meta theoretic properties is by rule induction. This works by induction on the structure of the derivations within the logical systems formalized. This is often termed the *syntactic method* of proving soundness and type safety. The logical framework LF uniformly represents derivations within it. We use it as our representation language for both the safety policy as well as the safety condition. Schürmann and Pfenning have created a system to formalize valid metalogical proofs, that is, proofs about systems formalized in LF [PS99]. This is implemented within the Twelf system. We use Twelf to formalize the safety proof.

In our experience, Twelf has allowed quick proof development because proofs are expressed in a natural fashion. We will discuss the safety proof in more detail in chapter 4. It is appropriate here to mention though that the current version of Twelf is restricted in the kinds of proofs that can be expressed and checked. Only theorems lying within the Π_2 fragment of logic, that is, statements of the form $\forall\exists P$, where P is quantifier-free, can be expressed. This forbids some proof-theoretically strong methods of proof. Fortunately, the syntactic method of proving type safety lies within this fragment of logic.

Different logical principles have been used by other projects building foundational certified code systems. The Princeton group [App01, CWAF03] has chosen to follow a different way of proving type safety. They build a denotational model for types and prove the rules of typing sound with respect to this model [AF00]. The

model is novel in that it is based on concrete machine architectures instead of mathematical objects such as domains. Their representation language like ours is the Logical Framework. For performing metareasoning, or proofs about represented logics, they use a variant of higher-order logic.

A separate project at Yale has followed, like us, the syntactic method for proving safety [HST+02]. Their formalization is done within the Coq system. The Coq system uses the Calculus of Inductive Constructions as its logic representation language as well as its metalogic. This leads to a certain uniformity of representation techniques. On the other hand, it conflates representations and proofs about representations, which preclude the use of convenient representation techniques.

Our trusted base thus includes the LF representation of the safety policy and the meta logic checker as implemented in Twelf. The safety condition, in the form of a formalized LF system, is explicitly not trusted. Trust is built by verifying the proof that it is safe, which is given as a formalized proof that can be checked by Twelf.

A cost of the approach is that the metalogic is fairly complicated. The proof checker for this metalogical framework is thus inherently more complex than that for a simpler logic such as LF. We believe that this is justified in terms of the substantial simplification of the process of generating proofs. The system is still perfectly flexible in terms of different safety conditions that can be used, and hence satisfies our goals of a flexible proof which is also specified with respect to concrete machine architectures.

2.3 Safety Checker

We now turn to the safety checker. We note that the producer provides the checker for his safety condition, and thus the checker itself is untrusted. We must verify that the checker is correct. However, it is not essential that the checker be complete, that is, accepts all programs satisfying the safety condition. It is in the producer's interest for the checker to accept as much as possible, but from the consumer's point of view, a checker need only accept some of the programs satisfying the safety condition. In other words, only one direction of correctness is critical for safety. Actually, nontermination is also perfectly all right, since programs that do not type check in some amount of time can be rejected.

This leads us to look for a method to verify partial correctness of programs. Since the safety condition is specified in the Logical Framework, we want to check the correctness of a program with respect to a specification in LF. We have designed a functional language we call LF/ML, which can verify partial correctness. This is a dependently-typed language which is based on and refines the core ML language. Types in LF/ML are indexed over objects from the LF type theory. This allows partial correctness to be stated with respect to a specification in LF. The statement can be verified by type-checking in this theory.

The technology of LF/ML is of independent interest, since it provides a way of proving functional programs partially correct. The specification technology is the logical framework LF, which has been used to encode various logical systems. It will be described in more detail in chapters 5 to 8.

The design of LF/ML is inspired by previous work by Xi [Xi98]. Similar to DML, our language is based on the core ML language, a functional language with polymorphic types. The type structure includes a set of datatypes with their respective constructors. Datatypes are refined to allow indexing. We go further than DML in allowing the index domain to range over objects from the LF theory. This provides information to the checker about the specification of the program. Further, we allow the set of constants of LF available (referred to as the signature) to be specified by the programmer. This means that the programmer can be flexible about the logical system he is going to use. Type checking in LF/ML then verifies partial correctness of the program with respect to the LF signature.

2.4 Alternative Approaches to Safety Checker

LF/ML is a key technical contribution of this thesis. However, it may be asked why we need to build such a general system. Here we will discuss some simpler approaches that could be tried, and why they did not work in our system. We emphasize that these approaches have to be flexible with regards to the safety condition. We cannot simply trust a checker particular to one safety condition, since we do not trust a safety condition

and may want to use different safety conditions and checkers. This means that approaches such as in Proof Carrying Code of having a trusted verifier is not sufficient.

2.4.1 Typing Derivations

One simple idea is to send derivations within the safety condition as proof that the particular program adheres to the condition. Since safety conditions in practice are type systems, this would then be a typing derivation. This approach is quite general, since checking the derivation is not tied to checking any particular formal system. The representation in a framework such as LF is uniform as to all safety conditions, and derivations are also represented uniformly.

In this method, checking that the program belongs to the safety condition is done by checking that the derivation is valid in the given type system. This problem is simply checking representations within the framework (LF), which we have to do anyway to check the safety condition. This can be checked by a small verifier, which is easy to trust [AMSV03].

Implemented naively, the derivations are typically very large in size. They are many orders of magnitude bigger than the piece of code they are proving properties of. The blowup has been noted to be of a factor of thousand or more. This is bad since this is supposed to be the certificate for code coming across the network. Further, the consumer has to spend resources checking these large formal objects.

2.4.2 Oracle Based Checking

One possible refinement is to compress the derivations from the previous section using the so-called "oracle" technique, due to Necula and Rahul [NR01]. Conceptually, checking that a proof is valid can be done as follows. The proof checker tries to prove the theorem. At any point where the proof tree branches, that is, a nondeterministic choice has to be made, the proof checker looks at the proof to see which branch to take. If the proof is valid, this procedure terminates in success without any backtracking. The information within the proof term can be represented by the sequence of choices made, and this sequence can be compactly encoded as a bit-string. Such bit-strings are called oracles in the literature [NR01].

In our system, since we do not trust the logic of the safety condition, we need to have a generic oracle-based proof system. On the code consumer side, this involves taking a proposition in a particular logic, and an oracle, and running the prover to verify that the oracle can produce a valid proof of the proposition. The prover consults the oracle whenever it faces a choice to determine which one to take. This procedure can verify theorems without any backtracking.

Dually, on the producer side, we have to produce this oracle string. This has to be done with knowledge of the verifying algorithm. The producer needs to know when a choice is necessary on the verifier side, and output the corresponding sequence of bits. Essentially, the same algorithm can be run on the producer, with the proof search taking as input a proof term instead of an oracle. When a choice is necessary, this procedure can consult the proof term to determine which branch to take, and output the necessary bits.

We implemented such an approach within the Twelf system [SPC05]. The problem with this approach is a subtle one. In the system sketched out above, notice that the oracle producer takes in a proposition and a proof for the proposition, and converts that into an oracle. We have already discussed that the size of the derivation terms are very large. It turns out that they are unbearably large for even relatively modest sized programs. The terms do not fit in memory even for the initial typechecking and conversion at the producer side.

A possible solution would be to not have a generic oracle-producer. Instead, the assembler of the code generates the required oracles. This seems infeasible to engineer. The assembler has to have detailed knowledge of the verifying proof search algorithm to know when it will face a choice. This is an added level of complexity over the already complex task of assembling low-level code and verifying it is well-typed.

2.4.3 Typed Logic Programs

We abandon the effort to have derivation based approaches, and consider checking the code at the consumer side. If the safety condition was a trusted component, the checker could also be a trusted component of the system. Unfortunately, in our system, the safety condition or type system is untrusted and provided by the code producer. Then the checker also has to be provided by the producer. We need a guarantee that the checker correctly implements the purported safety condition. It should only accept programs belonging to the safety condition. Thus, we are looking at a way to provide a provably correct implementation.

One way of doing this is to use dependently typed logic programming, as illustrated by Appel and Felty [AF04]. We will illustrate this in the Twelf system. Twelf provides an operational interpretation of LF [Pfe91a]. Proof search can be done by systematically searching for derivations within LF. Then, any LF signature can be looked on as a logic program.

If proof search terminates in success, a proof is produced. Assuming the proof search procedure is correct, this proof is a valid proof of the query proposition. Note that we do not have to run the program to know this fact. The type of the program is sufficient to ensure that if it terminates, the proposition belongs to the logic. This observation depends crucially on the fact that LF is a strongly typed, and in fact, dependently typed framework.

The logical specification of the safety condition can thus itself be looked on as a certifying checker. We know that if it terminates in success, the program lies within the represented safety condition. In other words, partial correctness of the checker can be checked statically. This is particularly nice since the specification of the safety condition within LF has already been done for the generic part of the proof.

The problem with using logic programming is that it uses a very restricted control strategy, that of depth first search. This imposes costs since some algorithms cannot be naturally expressed. Programs have to be written to work with the proof search operational model. Further, the backtracking based approach imposes costs on programs. It is difficult to take advantage of negative information, that is, that some fact does not hold.

These conditions point us to use functional programming to express our safety checkers. LF/ML is a functional programming language that is informed by the lessons from the logic programming language ELF, since both use LF as the means of specifying formal systems.

Chapter 3

The Safety Policy

Arguably the most crucial component of the trusted computing base is the safety policy. This is the definition of what the code consumer is willing to accept as safe. Our system allows us to state safety properties, that is, roughly, that a certain class of bad things will never occur. Our current safety policy focuses on memory safety. In more detail, programs are allocated memory by the runtime system on request. The program should never access any part of the memory not allocated to it. Further, it should not write into parts of memory allocated to hold its code section. This policy is stated in terms of executable code on a concrete architecture. The target of our system is the x86 architecture, the most commonly used architecture of potential grid machines. Our safety policy is memory safety.

The policy is stated as follows. We give a transition semantics for the code. This formalizes the operational semantics of a subset of the x86 architecture. This subset includes commonly used instructions used by programs. The formalization starts with defining the state of the machine, and then formalizes a transition relation between states. The transition relation specifies the behavior of the machine in a step-by-step manner. This style of semantics is known in the literature as small-step semantics.

The facilities provided by the runtime can be accounted for easily in this setting. This might be thought of as a notion of "system calls" for our system. We specify what inputs are expected, and provide direct transitions in the formal system to a state representing the completion of the function, or function return. The inner workings of the facilities provided need not be formalized and are held abstract. This means that the details are not known by target programs and cannot be used.

We now notice that not all the states of the machine are allowed or safe. For example, we want to consider a machine state where the code jumps to forbidden locations in memory as unsafe. We call such a state a "wrong" state. In our formalization, we remove any transition which ends up in a wrong state. Now a program that is about to perform such a forbidden transition finds itself with no transition in the formalized system. In the standard terms, the state is *stuck*. We must ensure that no state can have both a safe and an unsafe transition. We in fact maintain (and prove) a stronger property, that the transition system is deterministic. We discuss the issue further in Section 3.2.2.

The safety policy can then be stated simply as the property that after loading a program and executing it for any number of steps, the formalized machine is never stuck. This has the consequence that the formalized machine always has a transition it can make. Since only safe or allowable transitions are present in the formalized machine, the formalized machine is always making safe transitions. Also, since the formalized machine is in correspondence with the real machine, the real machine is also making safe transitions only. Stated in the contra-positive, if the real machine were ever to perform an unsafe transition, the corresponding formalized machine would be making one. Since the formalized machine does not have the unsafe transition, it would be stuck in that instance. The safety policy asks for a proof that this cannot happen. This method hinges crucially on the fact that a bad thing (a safety violation) can be detected by looking at a run of a program, and the exact step the violation occurs can be pinpointed. Thus our method is restricted to safety properties.

One issue is that safe programs do stop executing at some time. As so far stated, the policy seems to

Kinds	K ::=	$\mathsf{type} \mid \Pi x{:}A.K$
Families	A ::=	$a \mid \Pi x{:}A_1.A_2 \mid A\,M$
Objects	M ::=	$c \mid x \mid \lambda x{:}A.M \mid M_1\,M_2$
Signatures	Σ ::=	$\cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A$
Contexts	Γ ::=	$\cdot \mid \Gamma, x{:}A$

Figure 3.1: Abstract Syntax of LF

exclude terminating programs. We can special case this by saying that safe programs are not stuck unless they have reached a safe halt state. We follow an equivalent strategy by designating a safe halt state, and specifying an outgoing transition from the halt state to itself. This is the only outgoing transition from the halt state. This is merely a convenience in stating the safety policy as never stuck, with this being interpreted as always performing safe transitions or making some number of safe transitions and then entering the safe halt state.

We will now discuss the details of the encoding of the policy. Our policy is encoded within the Logical Framework. To make the presentation self-contained, we will start by a brief reminder to the reader of how formal systems are encoded within LF. We will then discuss the encoding of the above policy, and show how states and transitions of the concrete architecture are encoded.

3.1 Background: Representation in LF

The logical framework LF has been widely used to represent formal systems. Languages, type systems, operational semantics and logics can all be uniformly represented. To help the reader, we will briefly summarize the methodology of representing formal systems in LF [HHP93]. A more careful and modern treatment is given in Harper and Licata [HL06]. We will also present the concrete notation for Twelf [PS99], the implementation of LF that we will use. For a more leisurely introduction, the reader should consult examples in the Twelf distribution [PS02], and the handbook article by Pfenning [Pfe01].

3.1.1 The language LF

LF is at its core a minimal dependent type theory. There are three levels to the language. The first level is that of objects, which are classified by types. The level of types is generalized to type families, which in turn are classified by kinds. A type is a member of the class of type families which lives in the special kind type. The only type constructor is the dependent function type or Π types. LF differentiates between variables bound at introduction and constants, which are described by a signature. A formal system is specified within LF by postulating a signature containing new constants, both at the type and term levels. The syntax of LF is given in Figure 3.1.

The well-formed terms of LF are defined by a set of judgments. The judgments (except the signature formation judgment itself) assume that the constants are given types and terms by a well-formed signature. The rules, given in figure 3.2 are standard for a dependent type theories, with the rule for application having to substitute the applicand into the body of the dependent function type. Variables and constants are looked up in the context and the signature respectively. Also, when typing an abstraction, the domain of the variable is checked to be well-formed before adding the assumption to the context. This ensures that the well-formedness of contexts are preserved, and thus the leaves of the derivation tree do not need to check for well-formedness of the context. Also notice that we allow type conversion (and kind conversion) while checking well-formedness.

In the LF system, terms which are $\beta\eta$-equivalent to each other are considered equal. A judgment form of definitional equality axiomatizes $\beta\eta$ equality at each level of the language. These judgment forms and

$\boxed{\vdash \Sigma : \mathsf{sig}}$ (Σ is a well-formed signature)

$$\frac{}{\vdash \cdot : \mathsf{sig}}\ \text{LFSig-Nil} \qquad \frac{\vdash \Sigma : \mathsf{sig} \qquad \cdot \vdash_\Sigma \mathsf{K} : \mathsf{kind}}{\vdash \Sigma, \mathsf{a}{:}\mathsf{K} : \mathsf{sig}}\ \text{LFSig-Fam} \qquad \frac{\vdash \Sigma : \mathsf{sig} \qquad \cdot \vdash_\Sigma \mathsf{A} : \mathsf{type}}{\vdash \Sigma, \mathsf{c}{:}\mathsf{A} : \mathsf{sig}}\ \text{LFSig-Obj}$$

$\boxed{\vdash \Gamma : \mathsf{ctx}}$ (Γ is a well-formed LF context)

$$\frac{}{\vdash \cdot : \mathsf{ctx}}\ \text{LFCtx-Nil} \qquad \frac{\vdash \Gamma : \mathsf{ctx} \qquad \Gamma \vdash \mathsf{A} : \mathsf{type}}{\vdash \Gamma, \mathsf{x}{:}\mathsf{A} : \mathsf{ctx}}\ \text{LFCtx-Cons}$$

$\boxed{\Gamma \vdash \mathsf{M} : \mathsf{A}}$ (M_1 is a well-formed LF object at type A)

$$\frac{\Gamma(\mathsf{x}) = \mathsf{A}}{\Gamma \vdash \mathsf{x} : \mathsf{A}}\ \text{O-Var} \qquad \frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Gamma \vdash \mathsf{c} : \mathsf{A}}\ \text{O-Const} \qquad \frac{\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \quad \Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \text{O-Abs}$$

$$\frac{\Gamma \vdash \mathsf{M}_1 : \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 \quad \Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Gamma \vdash \mathsf{M}_1\ \mathsf{M}_2 : [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_1}\ \text{O-App} \qquad \frac{\Gamma \vdash \mathsf{M} : \mathsf{A}_1 \quad \Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}}{\Gamma \vdash \mathsf{M} : \mathsf{A}_2}\ \text{O-Fameq}$$

$\boxed{\Gamma \vdash \mathsf{A} : \mathsf{K}}$ (A is a well-formed LF type family at kind K)

$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Gamma \vdash \mathsf{a} : \mathsf{K}}\ \text{F-Const} \qquad \frac{\Gamma \vdash \mathsf{A}_1 : \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{K} \quad \Gamma \vdash \mathsf{M} : \mathsf{A}_2}{\Gamma \vdash \mathsf{A}_1\ \mathsf{M} : [\mathsf{M}/\mathsf{x}]\,\mathsf{K}}\ \text{F-App}$$

$$\frac{\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \quad \Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}}{\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}}\ \text{F-Pi} \qquad \frac{\Gamma \vdash \mathsf{A} : \mathsf{K}_1 \quad \Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}}{\Gamma \vdash \mathsf{A} : \mathsf{K}_2}\ \text{F-Keq}$$

$\boxed{\Gamma \vdash \mathsf{K} : \mathsf{kind}}$ (K is a well-formed LF kind)

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

Figure 3.2: Well-formedness judgments of LF

defining rules are summarized in figure 3.3. Notice that these judgments are defined mutual recursively with the well-formedness judgment, since LF is a dependent type theory, and well-formedness uses definitional equality.

Formalizing a logical system inevitably raises questions about the correctness of the formalization. To be correct, reasoning about the formalization must be "just as good" as reasoning in ordinary pen-and-paper style. The notion of *adequate* representations [HHP93] provides the necessary assurance of being just as good for reasoning purposes. This notion insists that there be an isomorphism between the on-paper (or informal) logical system and its formalization. This ensures that no information is lost by the formalization, so that every distinct informal notion has a distinct encoding. The other direction of the isomorphism says that every formal notion is an encoding of some informal notion, and thus we are only reasoning about things we meant to formalize.

The LF method of formalizing systems declares new types to formalize the judgments, and the derivations of the judgment are formalized by terms inhabiting the associated type. As mentioned before, terms which are $\beta\eta$-equivalent to each other are considered equal.We pick so-called long $\beta\eta$-normal forms as canonical representatives of the equivalence classes under $\beta\eta$ conversion, and call these the canonical forms of the theory. Adequacy can then be stated as a bijection between the informal presentation and canonical forms of the associated type. It has been shown [HP00] that all well-formed terms of LF are definitionally equivalent to a canonical form at the same type, and further, a simple algorithm can produce this canonical form.

3.1.2 Formalizing within LF

We will now use the framework LF to formalize some formal systems. As a small example, consider the system of natural numbers, denoted in standard notation as follows:

$$\text{Natural Numbers} \quad \mathsf{N} \quad ::= \quad \mathsf{0} \mid \mathsf{succ}\ \mathsf{N}$$

To formalize this, we start with defining a type of natural numbers which we can call `nat`. Terms belonging to this term are constructed by using new object constants, which we name `z` and `s`. Thus, the corresponding LF signature is given as:

```
nat : type.

z : nat.
s : nat -> nat.
```

To ensure that we have formalized correctly, we need to state the adequacy theorem, which in this case states that there is an isomorphism between natural numbers N and canonical forms of the type `nat`. Indeed, this is easy, as the only canonical inhabitants of the type `nat` are `z` and `s n`, where `n` is a canonical inhabitant of type `nat`. Then the following representation function can easily be shown to be a bijection:

$$\ulcorner \mathsf{0} \urcorner = \mathtt{z} \qquad\qquad \ulcorner \mathsf{succ}\ \mathsf{N} \urcorner = \mathtt{s}\ \ulcorner \mathsf{N} \urcorner$$

More interesting logical systems and programming languages have a notion of binding. This leads to defining notions of α-conversion, the α-equivalence classes induced by this conversion, and capture-avoiding substitution. Reasoning with these systems are often really reasoning about the α-equivalence classes. The method of formalization in LF smoothly represents such equivalence classes using the technique of *higher-order abstract syntax* [PE88]. The basic idea is to represent object language variables by LF variables, and use the framework's notion of α-equivalence and capture-avoiding substitution for the corresponding notions at the object language level. The tedious reasoning about bound variables for object languages is done once and for all as part of the framework's metatheory, and does not have to be redone for every new object language.

As an illustration, consider the simply typed lambda calculus, or STLC, whose syntax is given below:

$$\begin{array}{lllll} \text{Types} & \tau & ::= & \mathsf{b} \mid \tau_1 \rightarrow \tau_2 \\ \text{Terms} & \mathsf{M} & ::= & \mathsf{u} \mid \mathsf{x} \mid \lambda \mathsf{x}{:}\tau.\mathsf{M} \mid \mathsf{M}_1\ \mathsf{M}_2 \end{array}$$

$\boxed{\Gamma \vdash M_1 \equiv M_2 : A}$ (M_1 and M_2 are definitionally equal at type A)

$$\frac{\Gamma(x) = A}{\Gamma \vdash x \equiv x : A}\ \textsc{Objeq-Var} \qquad \frac{\Sigma(c) = A}{\Gamma \vdash c \equiv c : A}\ \textsc{Objeq-Const}$$

$$\frac{\begin{array}{l}\Gamma \vdash M_{11} \equiv M_{21} : \Pi x{:}A_2.A_1 \\ \Gamma \vdash M_{12} \equiv M_{22} : A_2\end{array}}{\Gamma \vdash M_{11}\, M_{12} \equiv M_{21}\, M_{22} : [M_{12}/x]\, A_1}\ \textsc{Objeq-App} \qquad \frac{\begin{array}{l}\Gamma \vdash A_{11} \equiv A_1 : \mathsf{type} \\ \Gamma \vdash A_{12} \equiv A_1 : \mathsf{type} \\ \Gamma, x{:}A_1 \vdash M_1 \equiv M_2 : A_2\end{array}}{\Gamma \vdash \lambda x{:}A_{11}.M_1 \equiv \lambda x{:}A_{12}.M_2 : \Pi x{:}A_1.A_2}\ \textsc{Objeq-Abs}$$

$$\frac{\begin{array}{l}\Gamma \vdash A_1 : \mathsf{type} \\ \Gamma \vdash M_1 : \Pi x{:}A_1.A_2 \\ \Gamma \vdash M_2 : \Pi x{:}A_1.A_2 \\ \Gamma, x{:}A_1 \vdash M_1\, x \equiv M_2\, x : A_2\end{array}}{\Gamma \vdash M_1 \equiv M_2 : \Pi x{:}A_1.A_2}\ \textsc{Objeq-ExtPi} \qquad \frac{\begin{array}{l}\Gamma \vdash A_1 : \mathsf{type} \\ \Gamma, x{:}A_1 \vdash M_{12} \equiv M_{22} : A_2 \\ \Gamma \vdash M_{11} \equiv M_{21} : A_1\end{array}}{\Gamma \vdash (\lambda x{:}A_1.M_{12})\, M_{11} \equiv [M_{21}/x]\, M_{22} : [M_{11}/x]\, A_2}\ \textsc{Objeq-BetaPi}$$

$$\frac{\Gamma \vdash M_2 \equiv M_1 : A}{\Gamma \vdash M_1 \equiv M_2 : A}\ \textsc{Objeq-Symm} \qquad \frac{\begin{array}{l}\Gamma \vdash M_1 \equiv M_2 : A \\ \Gamma \vdash M_2 \equiv M_3 : A\end{array}}{\Gamma \vdash M_1 \equiv M_3 : A}\ \textsc{Objeq-Trans} \qquad \frac{\begin{array}{l}\Gamma \vdash A_1 \equiv A_2 : \mathsf{type} \\ \Gamma \vdash M_1 \equiv M_2 : A_1\end{array}}{\Gamma \vdash M_1 \equiv M_2 : A_2}\ \textsc{Objeq-Fameq}$$

$\boxed{\Gamma \vdash A_1 \equiv A_2 : K}$ (A_1 and A_2 are definitionally equal at kind K)

$$\frac{\Sigma(a) = K}{\Gamma \vdash a \equiv a : K}\ \textsc{Fameq-Const} \qquad \frac{\begin{array}{l}\Gamma \vdash A_{11} \equiv A_{21} : \mathsf{type} \\ \Gamma \vdash A_{11} : \mathsf{type} \\ \Gamma, x{:}A_{11} \vdash A_{12} \equiv A_{22} : \mathsf{type}\end{array}}{\Gamma \vdash \Pi x{:}A_{11}.A_{12} \equiv \Pi x{:}A_{21}.A_{22} : \mathsf{type}}\ \textsc{Fameq-Pi}$$

$$\frac{\Gamma \vdash A_2 \equiv A_1 : K}{\Gamma \vdash A_1 \equiv A_2 : K}\ \textsc{Fameq-Symm} \qquad \frac{\begin{array}{l}\Gamma \vdash A_1 \equiv A_2 : K \\ \Gamma \vdash A_2 \equiv A_3 : K\end{array}}{\Gamma \vdash A_1 \equiv A_3 : K}\ \textsc{Fameq-Trans} \qquad \frac{\begin{array}{l}\Gamma \vdash K_1 \equiv K_2 : \mathsf{kind} \\ \Gamma \vdash A_1 \equiv A_2 : K_1\end{array}}{\Gamma \vdash A_1 \equiv A_2 : K_2}\ \textsc{Fameq-Keq}$$

$\boxed{\Gamma \vdash K_1 \equiv K_2 : \mathsf{kind}}$ (K_1 and K_2 are definitionally equal)

$$\frac{}{\Gamma \vdash \mathsf{type} \equiv \mathsf{type} : \mathsf{kind}}\ \textsc{Keq-Type} \qquad \frac{\begin{array}{l}\Gamma \vdash A_1 \equiv A_2 : \mathsf{type} \\ \Gamma \vdash A_1 : \mathsf{type} \\ \Gamma, x{:}A_1 \vdash K_1 \equiv K_2 : \mathsf{kind}\end{array}}{\Gamma \vdash \Pi x{:}A_1.K_1 \equiv \Pi x{:}A_2.K_2 : \mathsf{kind}}\ \textsc{Keq-Pi}$$

$$\frac{\Gamma \vdash K_2 \equiv K_1 : \mathsf{kind}}{\Gamma \vdash K_1 \equiv K_2 : \mathsf{kind}}\ \textsc{Keq-Symm} \qquad \frac{\begin{array}{l}\Gamma \vdash K_1 \equiv K_2 : \mathsf{kind} \\ \Gamma \vdash K_2 \equiv K_3 : \mathsf{kind}\end{array}}{\Gamma \vdash K_1 \equiv K_3 : \mathsf{kind}}\ \textsc{Keq-Trans}$$

Figure 3.3: Definitional Equality Judgments of LF

The case for types is simple. We define a new LF type `tp`, which has the appropriate constructors.

```
tp    : type.

b     : tp.
arrow : tp -> tp -> tp.
```

For terms, we define a new LF type `tm`. We will have no constant for variables, since these are represented by LF variables. Since variables are introduced by abstractions, the constant for abstraction must take in the body of the function using the abstracted variable. The body is thus of a higher LF type.

```
tm    : type.

u     : tm.
lam   : tp -> (tm -> tm) -> tm.
app   : tm -> tm -> tm.
```

For adequacy, we must have a bijection between types of the simply-typed calculus and canonical LF terms of the LF type `tp`. This is as simple as the natural numbers case, since types in STLC do not have binding structure. For the case of terms of the calculus, we require canonical LF terms of the LF type `tm`. We notice that we are required to account for open LF terms in this bijection, that is, terms which possibly depend on bound variables. This is stated as follows:

Proposition 3.1.1 (Adequacy for Syntax of STLC) *Suppose X is a set of terms of the simply typed calculus. Define Γ_X to be a LF context of the form $\mathtt{x}_1{:}\mathtt{tm}, \ldots, \mathtt{x}_n{:}\mathtt{tm}$. Then there is a bijection between simply typed terms with free variables in X and canonical forms of LF of type `tm` in context Γ_X (assuming the signature containing the above constants).*

Proof

The encoding function (assuming the obvious encoding function for types $\epsilon(\tau)$) is given below:

$$\begin{array}{lcl} \epsilon_X(\mathtt{x}) & = & \mathtt{x} \\ \epsilon_X(\mathtt{u}) & = & \mathtt{u} \\ \epsilon_X(\lambda \mathtt{x}{:}\tau.\mathsf{M}) & = & \mathtt{lam}(\lambda \mathtt{x}{:}\epsilon(\tau).\epsilon_{X,x}(\mathsf{M})) \\ \epsilon_X(\mathsf{M}_1\ \mathsf{M}_2) & = & \mathtt{app}\ \epsilon_X(\mathsf{M}_1)\ \epsilon_X(\mathsf{M}_2) \end{array}$$

Notice that variables are encoded by LF variables, and for abstraction, the encoding is done in an extended context for variables.

It can now be shown by induction on the structure of a simply-typed lambda term that its encoding is a canonical inhabitant of the LF type `tm` in the context Γ_X.

To show that this encoding is a bijection, we now exhibit the decoding function $\delta(\mathtt{M})$ (again assuming a decoding function $\delta(\tau)$ for types):

$$\begin{array}{lcl} \delta_{\Gamma_X}(\mathtt{x}) & = & \mathtt{x} \\ \delta_{\Gamma_X}(\mathtt{u}) & = & \mathtt{u} \\ \delta_{\Gamma_X}(\mathtt{lam}(\lambda \mathtt{x}{:}\tau.\mathtt{M})) & = & \lambda \mathtt{x}{:}\delta(\tau).\delta_{\Gamma_X,\mathtt{x}:\mathtt{tm}}(\mathtt{M}) \\ \delta_X(\mathtt{app}\ \mathtt{M}_1\ \mathtt{M}_2) & = & \delta_X(\mathtt{M}_1)\ \delta_X(\mathtt{M}_2) \end{array}$$

Again by induction on the structure of a canonical inhabitant `M` of `tm` in a context Γ_X, we see that $\delta(\mathtt{M})$ is a simply typed lambda term.

Further, by induction on the structure of simply typed lambda terms, we can prove that for any M, $\delta_{\Gamma_X}(\epsilon_X(\mathsf{M})) = \mathsf{M}$. □

We have seen that the formalization enjoys the crucial property of adequacy. A further important feature of the encoding is that it is compositional with respect to capture avoiding substitution on the term structure. This justifies the use of substitution in the framework to model substitution in the object language (here STLC). Specifically, the property we are interested in is the following:

Judgment form $\Gamma \vdash \mathsf{M} : \tau$	M value	$M_1 \Longrightarrow M_2$

$$\frac{\Gamma(\mathsf{x}) = \tau}{\Gamma \vdash \mathsf{x} : \tau} \qquad \lambda\mathsf{x}{:}\tau.M \text{ value} \qquad \frac{}{(\lambda\mathsf{x}{:}\tau.M_{12})\, M \Longrightarrow M[M_{12}/\mathsf{x}]}$$

$$\frac{}{\Gamma \vdash \mathsf{u} : \mathsf{b}} \qquad \frac{M_1 \Longrightarrow M_2}{M_1\, M \Longrightarrow M_2\, M}$$

$$\frac{\Gamma \vdash \mathsf{M}_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash \mathsf{M}_2 : \tau_2}{\Gamma \vdash \mathsf{M}_1\ \mathsf{M}_2 : \tau_2} \qquad \frac{M \text{ value} \quad M_1 \Longrightarrow M_2}{M\, M_1 \Longrightarrow M\, M_2}$$

$$\frac{\Gamma, \mathsf{x}{:}\tau_1 \vdash \mathsf{M} : \tau_2}{\Gamma \vdash \lambda\mathsf{x}{:}\tau_1.\mathsf{M} : \tau_1 \to \tau_2}$$

Figure 3.4: STLC: Semantics in ordinary notation

Proposition 3.1.2 (Compositionality of Encoding) *There is a compositional bijection ϵ_X between the terms of STLC with free variables in X and canonical LF terms in the context Γ_X, assuming the definitions of X and Γ_X as in proposition 3.1.1 in the following sense:*

If M_1 is a STLC term with free variables in X_1, x, X_2 and M_2 is an STLC term with free variables in X_1,

$$\epsilon_{X_1,X_2}(\mathsf{M}_1[\mathsf{M}_2/x]) = (\epsilon_{X_1,x,X_2}(\mathsf{M}_1))[\epsilon_{X_1}(\mathsf{M}_2)/x]$$

Proof

In fact, the encoding function in the proof for proposition 3.1.1 is compositional. This can be proved by induction on the structure of the term M_1, with necessary weakening for abstractions. □

We now turn to the semantics of the simply typed lambda calculus. This is in two parts, a static semantics which assigns types to terms in a context, and a dynamic semantics which states how terms compute. The static semantics is defined by a typing relation given inductively. Since this is an explicitly typed variant, there is no nondeterminism in the rules, and every term has at most one type. The dynamic semantics is given in the form of a small step operational semantics. Notice that the semantics is a call-by-value semantics, and requires judging terms to be values. These semantics are given in figure 3.4.

We now show representations in LF. The method is to postulate new LF types to correspond with judgments, and inhabiting terms to correspond with valid derivations of the judgment. Turning to the static semantics, this is a relation between terms and types in a context. The technique of higher-order abstract syntax obviates the need to represent the context explicitly. The encoding of the judgment is a type family `of` indexed by terms and types, but not the context. Its kind is thus `tm -> tp -> type`, and $\Gamma \vdash \mathsf{M} : \tau$ is encoded by `of` $\epsilon_\Gamma(\mathsf{M})\ \epsilon(\tau)$. The context $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$ is encoded by the LF context of the form $x_1{:}$`tm`$, d_1{:}$`of` $x_1\ \tau_1, \ldots, x_n{:}$`tm`$, d_n{:}$`of` $x_n\ \tau_n$.

```
of : tm -> tp -> type.

of_u   : of u b.
of_app : of (app M1 M2) T2
           <- of M1 (arrow T1 T2)
           <- of M2 T1.
```

```
of_lam  : of (lam T1 M) (arrow T1 T2)
           <- ({x:tm} of x T1
                    -> of (M x) T2).
```

The dynamic semantics requires the definition of value forms for the language. This is encoded by a judgment `value`, indexed by the term to be judged a value. The evaluation relation itself is encoded by a relation `evalsto` between the target and result of evaluation. The `evalsto` relation makes use of the `value` judgment form. Notice in the encoding of the `beta` rule the use of application in the framework to encode substitution.

```
value       : tm -> type.

value_lam   : value (lam _ _).



evalsto       : tm -> tm -> type.

evalsto_beta  : evalsto (app (lam T M1) M2) (M1 M2)
                  <- value M2.
evalsto_appL  : evalsto (app M1 M) (app M2 M)
                  <- evalsto M1 M2.
evalsto_appR  : evalsto (app M M1) (app M M2)
                  <- value M
                  <- evalsto M1 M2.
```

Having shown the encodings, we now have to check that they indeed encode the same thing as the informal definitions. In other words, adequacy has to be proved for the typing, value and evaluation judgments.

First, however, we should ensure that our encodings of syntax are adequate. There is a subtlety in that we proved adequacy with respect to a signature containing just the definitions for syntax. To be useful definitions, we need them to be adequate with respect to the bigger signature containing the new constants defined. We can check that adequacy is not falsified by the addition of these constants, and the previous proofs still go through. In general though, adding new constants might falsify adequacy, since new inhabitants of types may be created. A more modular approach is worked out in Harper and Licata [HL06].

Assuming we have proved adequacy then for terms and types, we can now state and prove adequacy of the encoding of the judgment forms. We have a different type family corresponding to each of the three judgment forms. The statement of adequacy then is captured by the following proposition.

Proposition 3.1.3 (Adequacy for Semantics of STLC) *Suppose* Γ *is a context of the simply typed lambda calculus of the form* $x_1{:}t_1, \ldots, x_n{:}t_n$. *Define the encoding of the context* $\ulcorner\Gamma\urcorner$ *to be* $x_1{:}\mathtt{tm}, d_1{:}\mathtt{of}\ x_1\ t_1, \ldots, x_n{:}\mathtt{tm}, d_n{:}\mathtt{of}\ x_n\ t_n$. *Then there is*

1. *a bijection between derivations of* $\Gamma \vdash \mathsf{M} : \tau$ *and canonical LF-terms of type* $\mathtt{of}\ \ulcorner M\urcorner\ \ulcorner\tau\urcorner$ *in the LF-context* $\ulcorner\Gamma\urcorner$,
2. *a bijection between derivations of* M *value and canonical LF-terms of type* $\mathtt{value}\ \ulcorner M\urcorner$ *in the LF-context* $\ulcorner\Gamma\urcorner$, *and*
3. *a bijection between derivations of* $\mathsf{M}_1 \Longrightarrow \mathsf{M}_2$ *and canonical LF-terms of type* $\mathtt{evalsto}\ \ulcorner M_1\urcorner\ \ulcorner M_2\urcorner$ *in the LF-context* $\ulcorner\Gamma\urcorner$.

Proof

We prove for the typing judgment, the argument for the other judgments are similar.

In the forward direction, we proceed by induction on the structure of the typing judgment. There are four cases, for variables, for unit, for application, and for abstraction. Assume then that the typing judgment is for a variable x. By the encoding within LF, this is represented by the LF variable `x`. Now, since the variable lies within the typing context, the corresponding typing assumption lies within the encoding of the context. This is just a variable typing rule, now at the level of LF. For unit, by encoding of syntax we

know $\ulcorner u \urcorner = \texttt{u}$, and $\ulcorner b \urcorner = \texttt{b}$. Using the rule for typing the LF constant `of_u`, we get the required result. For application, we inductively get LF terms for the subderivations. Using the rule for the LF constant `of_app`, we get the required result. The case for abstraction is interesting because a variable is added to the context. By induction, we get a LF term in the context with two extra assumptions, a LF variable and a typing assumption (in that order). We then use the LF abstraction rule to abstract over these two assumptions, creating a term of higher type (Π type). Using the rule for the LF constant `of_lam`, we get the required result.

In the reverse direction, we proceed by induction on the structure of the canonical term. By canonical forms within LF, the term can only be a variable, or have one of the constants `of_u`, `of_app`, or `of_lam` at its head. Assume then that it is a variable. Since we have restricted the form of the LF context, it can only come from the encoding of a typing assumption for a variable. We then use the STLC variable typing rule to get the required derivation. If the LF term is the constant `of_u`, then the STLC unit typing rule gives us the required derivation. If the LF term has the constant `of_app` at its head, it must have two canonical terms as immediate subterms. Applying the inductive hypothesis and the STLC application typing rule gives us the required result. Finally, assume that the LF term has the constant `of_lam` at its head. The argument must be a canonical form of the function type. Applying inversion on this canonical form twice, we get a term where we can apply induction, but in a larger context. The context can be checked to be of the right form (one variable and its typing assumption added), and thus induction together with the abstraction typing rule of STLC gives us the required result.

The encodings given can be checked to be inverses of each other by induction on the derivation and the canonical LF term respectively, case by case being exact inverses of each other. □

3.2 Formalized IA-32 Semantics: States and Transitions

As discussed before, we formalize the operational semantics of our runtime system. This is designed to be a subset of the IA-32 architecture, augmented with a few convenient runtime facilities. The formalization is intended be in close correspondence with the actual operations of the machine. We have used the semantics as specified in the IA-32 architecture manual as our guide. This is not precise in various places. We will discuss some of these issues. Also recall that all unsafe transitions (memory-unsafe) are omitted from the semantics. We will discuss the formalization in this section.

3.2.1 Data Representation

Data in the system is built up of bits, which (naturally) can take on either a one or a zero. These are built up to form binary numbers, which are merely strings of bytes. It is convenient for many purposes to specify the length of the binary numbers, so we index the datatype of binary numbers by their length. The length is given as a natural number in unary notation, which we call `nat`. Here then are the basic constituents of data.

```
bit : type.

zero            : bit.
one             : bit.


binary : nat -> type.

nil$    : binary z.
$       : bit -> binary N -> binary (s N).
%infix right 4 $.
```

The x86 architecture is a byte-oriented architecture. In some cases, particularly when addresses are required, four bytes are treated as a unit, and this unit is called a word.

```
bw  : nat = 8.   %% byte width in bits
ww  : nat = 4.   %% word width in bytes
wwb : nat = 32.  %% word width in bits (should be bw * ww)
dw  : nat = 8.   %% double word width

%abbrev abyte = binary bw.   %%  "actual byte"
%abbrev aword = binary wwb.  %%  "actual word"
%abbrev address = aword.
```

Finally, we also have streams of bytes, which we call string. This again is a type that is indexed by its length. This is a list with its nil element represented as `#` and its cons represented as `/` in infix notation.

```
string : nat -> type.

#       : string z.
/       : abyte -> string N -> string (s N).
%infix right 4 /.

%abbrev word = string ww.
%abbrev byte = string 1.
```

3.2.2 Indeterminism

There are various situations when the behavior of the system is indeterminate. One example is that of input operations, when the input can be conceivably any of a set of values. Further, under some conditions, part of the system state of the IA-32 architecture is defined to be *undefined*, that is, it could conceivably be any value. To be safe the program should be able to continue no matter what values they encounter in such situations. We should stress that such situations are not necessarily error states, and programs can plausibly reach states with partially undefined outcomes.

To take one particular example, IA-32 has a set of condition codes which can be used to guide jumps. The state of some of these condition codes after a shift or rotate instruction is undefined. It is certainly not an error to perform shifts or rotates. The safety of the program cannot, however, depend on any particular value of the flags after the instruction (say, even if we notice that the flag is always reset, this is not behavior that the program's safety should depend upon).

This situation also arises when the outcome of the behavior is too difficult to formalize. As we shall discuss, we have memory management primitives in our runtime system. The particular result of garbage collection is hard to formalize, and should not be something that programs should rely upon to prove safety. The situation is analogous, and we insist that programs should be able to deal with any particular result.

We must insist that all possible actions that the machine takes nevertheless result in safety. The question is to formalize this notion of all possible actions. One obvious solution is to make the transition relation of the system nondeterministic, that is, allow multiple target states from one state. The transition relation then is no longer a function.

The obvious solution, however, creates a few problems. Suppose we work with a system that allows multiple transitions out of a single state. Recall that our safety policy is formulated by discarding unsafe transitions and checking that the formalized machine not be stuck. In this case though, it is no longer sufficient to remove all the unsafe transitions. If there is both a safe transition and an unsafe transition that can be made out of a state, the state itself is unsafe, and should be stuck. That is, whenever there is one unsafe transition out of a state, all the outgoing transitions should be removed.

To illustrate, suppose that the program wants to transfer control based on an indeterminate value. This is only safe if every possible transfer point is safe. However, it is quite subtle to design the possible relations to exclude all transitions if any of them are bad. It would be difficult to design a policy which makes it abundantly clear that the above property was satisfied. This is particularly so since the transition relations are defined by making use of interacting auxiliary relations as premises. For example, in the transfer control example, a natural encoding of the transition is to make use of auxiliary premises which choose among the

target addresses, put the target address in the instruction pointer and then perform a fetch. The fetch may filter out a bad address, but that will require that all of the other (superficially valid) transitions are filtered out too.

There is a further reason to not have a nondeterministic transition relation. A transition relation that is deterministic is easy to reason about. We will look at one particular safety proof in the next chapter, where we will see that a deterministic transition relation simplifies the proof considerably.

Our solution is to force the transition relation to be deterministic by encoding the nondeterministic choices in an oracle. This oracle is a fictional part of the state. When there is a position where the semantics must make a nondeterministic choice, the semantics notionally reads the oracle and follows the advice. The safety policy is stated in terms of quantifying over all possible finite oracles. In other words, the quantification is over all finite sequences of choices. This is sufficient since a key property of safety properties is that unsafe behavior occurs (if at all) in a finite time.

Oracles are just sequences of choices. We encode choices as a stream of bits.

```
oracle : type.

oracle_nil      : oracle.
oracle_cons     : bit -> oracle -> oracle.
```

Operations that are nondeterministic are now rewritten to consult the oracle. Consulting the oracle in this setting is merely looking at the head of the stream, pulling off as many bits as are required to make the choice. For example, an operation such as allocation that needs to fill up n bytes of memory with unknown contents consumes $8n$ bits from the oracle to determine the new contents of memory. Note that this indeterminism is in addition to the indeterminate nature of the allocated address, which is also handled by the oracle.

3.2.3 States

Machine states for execution are formalized as a tuple of their components. The components are the memory, a register file, the condition flags visible to the programs, the instruction pointer, and also the oracle referred to in the previous section.

```
state : type.

state_  : memory          %% main memory
          -> regs numregs %% register file
          -> flags        %% condition flags
          -> address      %% instruction pointer
          -> oracle
          -> state.
```

Further, we allow a safe halt state to represent normal completion of a program. This is a special sort of state. If the machine manages to reach this state, it is known to be safe. The halt state can possibly transition to itself (this makes stating some theorems easier), but for safety, it is essential that this state not transition to any other state.

```
stopped : state.          %% Final state. Safe by definition.
```

We will now look at each of the components of a normal state of a machine in turn.

Memory Memory is formalized as a list of sections of memory accessible to the program. Sections are contiguous blocks of memory of some size, and distinct sections are nonoverlapping. Each section is arranged in the space of addresses that can possibly be addressed by programs. This means that memory addresses lie between 0 and $2^{32} - 1$. It is convenient to have a representation of the size of each section.

```
memory : type.
```

```
mnil    : memory.
mcons   : address -> {n:nat} section n -> memory -> memory.
%%        start,     size,   contents,    rest of memory
```

The sections themselves are of two different sorts. One sort of section is one which the program is allowed to hold pointers to, and these hold data that can possibly be read or written to. The data itself contained within such sections is in the form of a string. The second kind of section is a reserved part of memory. The program knows of its location and extent, but nothing is known of the data contained within this section. Further, we may not dereference pointers to the middle of these reserved sections. They may allow access to the very beginning of the section, which is how calls to runtime utilities is made. Reserved sections contain parts of the runtime system, various inaccessible parts of memory, and any other sections blocked off for any reason.

```
section : nat -> type.

section_valid           : segment -> string N -> section (s N).
section_reserved        : rsection N -> section N.
```

Valid sections are classified by the segment they belong to, into either heap, code, or stack segment. Heap and stack segments can be read from or written into, while code segments can only be read from. Further, the specification of the garbage collector states that stack segments are not garbage collected.

```
segment : type.

ss  : segment.   %%% Stack segment
cs  : segment.   %%% Code segment
hs  : segment.   %%% Heap segment
```

Registers and Flags The IA-32 architecture has a bank of eight general purpose registers, EAX, EBX, ECX, EDX, EBP, ESI, EDI and ESP. Each of these can hold a word. The lower 8 and 16 bits can also be individually addressed. We collect all of them into a register file which we call `regs`. We do not treat the segment registers available to the programmer to perform address calculations. The EFLAGS register is treated separately.

```
numregs : nat = 8.

regs : nat -> type.

regs_nil        : regs z.
regs_cons       : word -> regs N -> regs (s N).
```

The IA-32 has a variety of flags available to the programmer in the EFLAGS register. For our system, the only flags visible to the programmer are the CARRY, ZERO, SIGN, and OVERFLOW flags. Each of these holds one bit, that is, it can be checked whether they are set or reset.

```
flags : type.

flags_  : bit       % cf
          -> bit    % zf
          -> bit    % sf
          -> bit    % of
          -> flags.
```

3.2.4 Transitions

Having encoded states of the machine, we now encode transitions as a relation between the from-state and the to-state. Recall that only safe transitions are encoded, and any transition that is unsafe is omitted from the definition of the system.

```
transition        : state -> state -> type.


transition*       : iinst -> state -> state -> type.
transition_runfac : runfac _ -> state -> state -> type.
```

Transitions are broken into three kinds. The most common kind of transition performs an instruction fetch, and case analyzes on the instruction. We encode the case analysis by a helper relation `transition*`. Another kind of transition occurs when a call is made to a runtime facility. These calls are treated as any other calls, and the target address is put in the instruction pointer. Recall that runtime facilities live in reserved parts of memory. An ordinary instruction fetch would fail in such a case. Instead, we look up the address in our list of addresses of runtime facilities, and perform the action of the facility (atomically). Finally, to easily state safety as the property that there is always a transition, we have a notional transition from the stopped state to itself. These three notions are detailed below.

```
transition_           : transition ST ST'
                          <- fetch ST IN
                          <- transition* IN ST ST'.

transition_runtime    : transition ST ST'
                          <- at_runtime_address ST IRF
                          <- transition_runfac IRF ST ST'.

trans_stopped         : transition stopped stopped.
```

The work of performing the case analysis of an instruction is handled by the helper relation `transition*`. We exhibit a few of the cases to give the general idea of the encoding of instructions.

For a `mov` instruction, we have to find the contents of the source operand (encoded by the relation `oload`), and store these contents at the destination operand, also called an effective address in Intel parlance (encoded by the relation `store`). For this simple instruction, this is all there is to the main job of the instruction. We now have to get the machine to update its instruction pointer. Because instruction encoding is irregular on the IA-32 architecture, we encapsulate the calculation of the next instruction address into a helper relation `next`. Thus the case for `mov` looks as follows.

```
trans_mov             : transition* (ii_mov Nsz E O) ST ST'
                          <- oload ST O W
                          <- store ST E W ST1
                          <- next ST A
                          <- puteip ST1 A ST'.
```

For a `jmp` instruction, the normal way of calculating the next address is not carried out. Instead, we load the next address from the operand argument. Recall that loads give us words, a sequence of four bytes. We convert this to a 32-bit address representation by the helper relation `implode_word`. Thus the case for `jmp` looks as follows.

```
trans_jmp             : transition* (ii_jmp O) ST ST'
                          <- oload ST O W
                          <- implode_word W A
                          <- puteip ST A ST'.
```

Similar encodings are performed for each of the different instructions supported. They follow the same pattern, though some are understandably more complex. By factoring the state transitions into common patterns of behavior, and encapsulating these as helper relations, encodings of state transitions is performed in about ten lines of LF each (the most complicated is the `call` instruction, at 17 lines, the least is `nop`, at 2 lines).

Delving a little deeper into the relations, we can exhibit the flavor by looking at the definition of the `next` relation, which finds the address of the next instruction in the stream. This finds the current instruction

pointer and state of memory, and finds the string at that position. We then make use of a relation `idecode`, which decodes the string, and returns the length of the string decoded and an abstract syntax description of the instruction. In this case, we do not actually care about the instruction. We merely take the length of the encoding, convert it to binary, and add the resulting number to the current value of the instruction pointer.

```
next : state -> address -> type.

next_           : next ST A'
                  <- geteip ST A
                  <- getmem ST H
                  <- mload* H A S
                  <- idecode S I N _
                  <- represents N B
                  <- add A B A' _.
```

3.2.5 Initial states and reachability

We now have to encode the set of states reachable if we start with a given piece of code. This is done as follows.

First, our pieces of code are encoded in binary as strings of bytes. We could reuse the `string N` type from before, but it is clearer to not mention the length explicitly. We thus define a very similar type of "actual strings", `astring` for short, as follows:

```
astring : type.

##      : astring.
!       : abyte -> astring -> astring.
%infix right 4 !.
```

Next, we define a small helper relation that relates a piece of code (encoded as a `astring`), and the set of states that are legal initial states for that piece of code.

```
initial_state : astring -> state -> type.
```

This relation nondeterministically picks an address for the piece of code, assumes a memory with the code at that address, and unspecified chunks of memory which are reserved by the system. The instruction pointer is assumed to point to the start of this address. The stack pointer and the global table registers are initialized, and the rest of the registers and flags are nondeterministically picked. Here also, a stream of bits is picked nondeterministically as the oracle.

Finally, we define the set of states reachable when a particular piece of code (again, encoded as a string) has executed for some finite number of steps. This is either a state that is a valid initial state, or a state that is related by the transition relation to a state already known to be reachable.

```
reaches : astring -> state -> type.

reaches_z       : reaches AS ST
                  <- initial_state AS ST.
reaches_s       : reaches AS ST2
                  <- reaches AS ST1
                  <- transition ST1 ST2.
```

3.2.6 Safety

We want to state that for any reachable state from a safe code, there always exists at least one state that can be transitioned to. The notion of safety is left unspecified, up to the definition of a relation `check` between

an unspecified type of certificates and pieces of code. Code producers are expected to provide their own notions of `certificate` and `check`.

```
certificate : type.

check : certificate -> astring -> type.
```

Safety is then the metatheorem that for all pieces of code `AS`, if there exists a certificate `CERT` such that `check CERT AS` holds, and for all states `ST` such that the relation `reaches AS ST` holds, there exists a state `ST'` such that the relation `transition ST ST'` holds.

How to state and prove such metatheorems in the formal system of Twelf is the subject of the next chapter. Here we will only mention that this theorem and its proof can be encoded and checked effectively. Together with the guarantees of adequacy sketched out, this then implies that all programs which satisfy the producer's certifying system are safe to execute on the IA-32. Thus the IA-32 semantics sketched out in this chapter form a part of the trusted computing base, but the proof of safety itself is not.

Chapter 4

The Safety Proof

Our method of building a foundational certified code system uses a safety condition as an intermediary in the proof. This is a means of isolating a set of programs. Then we have the proof obligation of showing that any program lying within the safety condition is safe with respect to the safety policy. We note that the safety condition is not trusted. The proof, which we call the safety proof, is the means of generating trust that the safety condition in use is any good.

The safety condition itself is a formal system. In fact, in practical usage, the safety condition is going to be a type system. In terms of a type based approach, the safety proof is a proof of the safety of the type system with respect to the semantics given by the safety policy. This proof is a property about the two formal systems, the safety condition and the safety policy, together. We have already described how such formal systems can be encoded within the Logical Framework. What we now need is to encode reasoning about these formal systems. In other words, we want to encode metalogical reasoning.

Various reasoning frameworks have been devised to encode metalogical reasoning. The one we use is the metalogical framework in Twelf [PS99]. This was designed [SP98] to easily encode inductive proofs over encodings within LF, while allowing the use of natural LF encodings by higher-order abstract syntax. It has been particularly designed for structural inductive proofs for programming language and logical systems. We were the first to create a large-scale proof within the Twelf metalogic. Other, larger proofs have since been written. In all these projects, we have found the tool rather pleasant, and helpful to create large systems rapidly.

Other researchers have used other systems for encoding metalogical reasoning. In particular, since the task is similar to encoding logical systems, they have reused their logical frameworks to encode metalogical reasoning. The FTAL project at Yale has used the Calculus of Inductive Constructions to both encode their systems as well as proofs about the system. The FPCC project at Princeton has used an encoding of higher-order logic within LF to serve as their metalogical framework. Our separation of encoding framework and metalogical proof framework has given us a pragmatic advantage in terms of proof development. This comes at the price of having to trust a larger, more complex system, the Twelf framework.

We will now describe the safety proof in our framework. Recall that this must be a machine-checkable proof that a safety condition is sound with respect to the safety policy outlined in the previous chapter. Our example safety condition is known as TALT, and is a particular type system for low-level code. Its safety proof is fully formalized within the Twelf metalogical framework.

4.1 Background: Metatheory in Twelf

The Twelf system [PS99] contains an implementation of metareasoning for languages built on top of LF. This builds on work by Pfenning and Schürmann on Twelf and its predecessor system Elf [PR92, Sch00b]. We will give a brief tutorial of the technique in this section. Harper and Licata [HL06] give a more comprehensive explanation. The Twelf manual [PS02] and the Twelf wiki [The07] are other sources of information. The

techniques behind checking such proofs have been described in technical papers on the Twelf implementation [RP96, PP00].

4.1.1 Stating totality assertions

Recall that LF has type families to express relations. These relations are not restricted to be on syntax (such as typing relating terms and types) but can be higher level, or semantic, relations (such as relations between two typing derivations). Twelf allows stating and proving assertions that relations are total, in that they are well-defined for any input. These totality assertions are stated by two directives, a *mode* declaration and a *worlds* declaration. Mode declarations mark elements of the relation as input (+ mode) or output (- mode). Worlds declarations declare the LF contexts the relation is total in. For our work, we will need only simple closed, or empty, worlds.

As an example, consider the type of natural numbers nat from section 3.1. Suppose we were to define a relation between natural numbers called sum, which relates N1, N2 and N3 if and only if $N1 + N2 = N3$, as follows:

```
sum    : nat -> nat -> nat -> type.

sum/z  : sum z N N.
sum/s  : sum (s N1) N2 (s N3)
           <- sum N1 N2 N3.
```

We notice that given two ground natural numbers, there is always a third ground natural number which is related to them by the sum relation above. Further, this holds in the empty context (where there are no natural numbers assumed in the context). In other words, the sum relation is total in its first two arguments, and in the empty context. This is stated in Twelf using the following two directives, the mode and the world declarations:

```
%mode sum +N1 +N2 -N3.

%worlds () (sum N1 N2 N3).
```

We can look on the declaration of the sum type family together with the above mode and world declarations as a *specification*, asserting totality for the sum relation. The definition of the sum/z and sum/s objects is the *justification* for the totality assertion. Twelf can be asked to verify that the justifications taken together indeed satisfy the specification, by means of a totality assertion. This viewpoint of specification and justification acquires power in the higher-order setting when we state relations about other relations, and allows a check of metatheoretical facts, as we will describe in the next section.

4.1.2 Mechanizing Metatheory via Totality

An important class of theorems in programming language metatheory is the Π_2 class, also known as the class of $\forall\exists$ statements, which represents statements of the form

For all derivations $\mathsf{D}_1, \ldots, \mathsf{D}_n$, there exist certain other derivations $\mathsf{E}_1, \ldots, \mathsf{E}_m$.

A constructive proof of a $\forall\exists$ theorem is a function from the input arguments (the $\forall$ parts) to the output (the $\exists$ parts), which is defined for all inputs. Recasting functions as relations, we see how such statements can be proved within Twelf. The theorem is then a particular relation, together with the specification of which arguments are input and which are output, and the context indexing the relation. The proof is realized as inhabitants of this relation. Checking that the proof is valid can be recast as checking that the relation is total in its arguments. Thus we can use the method of totality assertions in the previous section to state and prove meta theorems.

We exhibit an example theorem, the preservation theorem for the simply typed calculus, which says that whenever a closed term M_1 which is well-typed at type τ evaluates to a closed term M_2, M_2 also has the type τ.

Theorem 4.1.1 (Preservation, informally) For all closed terms M_1 and M_2, and type τ, if there is a derivation D_1 of $\cdot \vdash M_1 : \tau$ and a derivation D_{ev} of $M_1 \Longrightarrow M_2$, then there is a derivation D_2 of $\cdot \vdash M_2 : \tau$.

By adequacy of encodings, we can formulate this as the totality of the relation preservation as follows:

Theorem 4.1.2 (Preservation, Twelf version)

```
preservation    : {M1:tm} {M2:tm} {T:tp}
                  of M1 T
                  -> evalsto M1 M2
                  -> of M2 T -> type.
%mode preservation +M1 +M2 +T +D1 +D2 -D3.

%worlds () (preservation M1 M2 T D1 D2 D3).
```

Notice that preservation is stated (and proved) in the empty context (the worlds declaration is empty). This is typical of all the theorems we will be talking about in this dissertation, and henceforth, we will be omitting the worlds declaration. Also, we can take advantage of Twelf's reconstruction of implicit arguments to simplify the statement of preservation, with corresponding changes to mode and world declarations, as follows:

```
preservation : of M1 T -> evalsto M1 M2 -> of M2 T -> type.
```

The Twelf proof of preservation is given in figure 4.1. Notice how closely it corresponds with the usual informal proof given below.

Proof

We case analyze on the evaluation step followed.

Case 1: Evaluation on the left of an application

Since application typing is the only rule that can apply, we perform inversion on the typing judgment to get typing derivations for the function position and the argument positions. We apply inductive hypothesis on the smaller derivations in the function position to get the result is well-typed at the same type. Using the application typing rule, we can put this derivation together with the argument's typing derivation to get the required typing.

Case 2: Evaluation on the right of an application

Since application typing is the only rule that can apply, we can again perform inversion on the typing judgment to get typing derivations for the function position and the argument positions. We now apply inductive hypothesis on the smaller derivations in the argument position to get the result is well-typed at the same type. Using the application typing rule, we can put this derivation together with the function's typing derivation to get the required typing.

Case 3: Beta-reduction

Again, application typing is the only rule that can apply. We invert this to get a type for the function and the argument. The abstraction rule is the only typing rule possible for the function, so we can invert it to get a typing for the body under an extended context. We then appeal to substitution, and substitute the argument's typing to get the required typing. Notice that in Twelf, the substitution lemma is expressible in the language as a simple application.

□

4.1.3 Verification of meta proofs

We have noted that meta proofs are represented by total relations. Thus the verification of totality by Twelf is of supreme importance in establishing validity of the proofs. We will now give an informal description of the checks Twelf performs when asked to verify totality. Since we assume that the signature has passed type checking, the relation cannot fail at run time with a type error. There are now five different ways a relation can fail to be total, and hence not be a good proof. These are failure of mode checking, of termination checking, of input coverage checking, of output coverage checking, and of world checking.

```
-1 : preservation
      (of_app Darg Dfun1)
      (evalsto_appL Deval)
      (of_app Darg Dfun2)
      <- preservation Dfun1 Deval Dfun2.
-2 : preservation
      (of_app Darg1 Dfun)
      (evalsto_appR Deval Dvalue)
      (of_app Darg2 Dfun)
      <- preservation Darg1 Deval Darg2.
-3 : preservation
      (of_app Darg (of_lam Dbody))
      (evalsto_beta Dvalue)
      (Dbody _ Darg).
```

Figure 4.1: Preservation proof, Twelf

To illustrate these failures, we will give putative proofs of the false meta proposition all_b which says that any well typed closed term is well typed at the base type in the empty context:

```
all_b    : of M T -> of M b -> type.

%mode all_b +D1 -D2.

%worlds () (all_b _ _).
```

Mode checking Mode checking checks for the ways that a relation can fail to be well-moded. Given well-specified, or ground, inputs, the outputs of the relation must be well-specified (ground) themselves. Failing to check this can allow bogus proofs of the theorem as follows:

```
- : all_b (Din : of M T) (Dout : of M b).
```

The other possibility of failing mode checking is to pass ill-specified inputs to calls of metatheorem relations, either the same one inductively or a different metatheorem used as a lemma. Failing to observe this rule allows bogus proofs, such as the following, which appeals to a nonexistent evaluation step and applies the valid metatheorem preservation.

```
- : all_b (Din : of M T) (Dout : of M b)
     <- preservation (of_u : of u b)
        (Deval : evalsto u M)
        (Dout : of M b).
```

Termination checking Termination checking checks that the relation terminates when called with any input. This is important in the presence of recursive calls to the relation itself as an induction argument, or in more complex cases to chains of mutually inductive calls between relations defined together. An invalid induction can produce bogus proofs easily, such as the purported proof below that calls itself inductively on an argument that does not get smaller.

```
- : all_b (D1 : of M T) (D2 : of M b)
     <- all_b D1 D2.
```

Twelf can check termination with some help from the programmer in stating what the induction argument is. Lexicographic and simultaneous termination orders, and checking these on mutually recursive predicates, are supported.

Input coverage checking Checking input coverage means that Twelf checks that all ground inputs are covered by the relation. This is similar to checking that all cases of the theorem are covered. A failure of input coverage leads to incomplete and hence bogus proofs. Thus, in the proof below, the case that the term is the unit term (u) is correctly proved, but the other cases are ignored.

```
- : all_b (of_u : of u b) of_u.
```

Output coverage checking Checking output coverage means that the output of any relations called as lemmas must not be over-constrained. Recall that we are proving $\forall\exists$ theorems. Ignoring output coverage would indicate relying on particular instantiations of the existence predicate for a lemma. This can give rise to bogus proofs, as in the following (contrived) example. Suppose that we have proved as a helper lemma the valid, trivial lemma that given a derivation of typing, we have a derivation of typing the same term at some type (provided as output).

```
helper : {T1:tp} {T2:tp} of M T1 -> of M T2 -> type.
%mode helper +T1 -T2 +D1 -D2.
```

The purported proof below constrains the output of the helper lemma in a way that violates the checking of output coverage.

```
- : all_b (Dof1 : of M T) (Dof2 : of M b)
     <- helper T b Dof1 Dof2.
```

World checking World checking is a check that any relations used within the proof, whether other relations used as lemmas, or the same relation used inductively on a smaller argument, are called with a legal context. Recall that the specification of a theorem includes the shapes of the contexts it is valid in. Theorems are only valid in certain contexts.

A simple-minded restriction (this is not what Twelf does) would be to forbid adding anything to the context whatsoever. However, proofs of many theorems (though not actually any discussed in this thesis) require us to descend within bodies of abstractions, and thus add to the context. This is handled within Twelf by explicitly declaring the shapes of the contexts a theorem is valid in by the worlds declaration, and checking that these constraints are satisfied by any additions made to the context.

To illustrate what could go wrong if this check is violated, imagine that we create an uninhabited type, with, as the name suggests, no inhabiting closed terms. We could then easily prove a lemma that, in the empty context, assuming there is an inhabitant of uninhabited, anything else is possible, in particular, any given term is well-typed at the base type.

```
lemma: {M:tm} {u:uninhabited} of M b -> type.
%mode lemma +M +u -D.
%worlds () lemma (_ _ _).
```

Notice the importance of the empty world declaration as part of the lemma statement above. This lemma is obviously not true in any context that contains an inhabitant of uninhabited. If world checking were to be ignored, the purported proof below which adds a variable of uninhabited type and then calls on lemma would go through.

```
-: all_b (Din : of M T) Dout
   <- ({u:uninhabited}
         lemma M u Dout).
```

4.2 Safety Policy in Twelf

We can now use the metalogic capabilities of Twelf to state the safety policy. Recall from chapter 3 that the safety policy is formulated in terms of state transitions. Specifically, if we have a binary S which the code producer can check, any state ST reachable on executing S has at least one more transition.

We thus have defined a predicate relation reaches between binaries and machine states that formalizes reachability.

```
reaches  : astring -> state -> type.
```

We also have an unspecified type of certificate used for metadata used to check binaries and an unspecified predicate check that holds for a certificate and program the code producer considers safe.

```
certificate   : type.

check         : certificate -> astring -> type.
```

The names of the type certificate and the judgment form check are special to the system, but their definitions are left to be filled in by the untrusted code producer. We ask that the code producer convince us of the safety of their system by proving within Twelf the following metatheorem:

```
safety : check _ S
          -> reaches S ST
          -> transition ST ST' -> type.
%mode safety +D1 +D2 -D.
```

The statement of this theorem says that if we can check a binary S (with some certificate), and state ST is reachable in an execution of S, then there must be a further transition from ST.

The code producer is responsible for filling in the proof of this theorem. After getting the proof, we will ask Twelf to verify that this theorem is valid. This will give us the formal guarantee required to ensure trust in the producer's system, or safety condition.

4.3 Safety Proof

The example safety proof we have developed is for the TALT type system. At its heart, it is a simple syntactic proof of type safety. This method of proof defines a type system, an operational semantics, and proves safety by means of proving lemmas usually called progress and preservation. There are some issues with the approach at both ends. First, it is convenient to prove safety while holding some details of the implementation abstract. This however means that the abstract semantics has to be related to the concrete semantics of the IA-32 sketched out in the previous chapter. Second, it is convenient to prove safety for a system like TALT which is declarative, and not designed with type checking in mind. This means that TALT is not effectively checkable, with many parts of the system requiring search, and even unbounded search. In practice, we would like to add annotations so that the type checker can effectively check programs. This is then a closely related, but formally different variant of TALT known as XTALT (for eXplicit TALT). The relationship between XTALT and TALT has to be formalized and proved correct.

The proof is thus structured in three parts.

— **Static** In this part of the proof, the XTALT type system is proved sound with respect to the TALT type system. This proves that the explicit annotations added can be removed in a sound manner via a process of elaboration.

— **Abstract** In this part of the proof, the TALT type system is proved to be sound with respect to an abstract semantics, known as the TALT dynamic semantics.

— **Concrete** In this part of the proof, it is shown that the TALT dynamic semantics is realized by the IA-32 dynamic semantics. This is a simulation argument that relates the two semantics.

4.3.1 Preliminaries

We wish to reason about binary pieces of code. Recall that the safety condition is a (so-far unspecified) relation check between a (likewise unspecified) type of certificate's and binary code. In the TALT instatiation, we start by defining a type program which represents the abstract syntax of XTALT program. It is convenient to consider such a program to be the required metadata for the binary, that is, the certificate type is instantiated by an injection from the type of program.

```
certificate_   : program -> certificate.
```

The safety condition check is then instantiated by satisfying two steps. First, the binary S to be checked must be parsable as the program P claimed by the certificate. This is given by a relation between the two parse_program. Second, the program P must be well-typed in the XTALT type system, given by a predicate check_program.

```
check_          : check (certificate_ P) S
                  <- parse_program S P
                  <- check_program P.
```

4.3.2 Static Soundness of XTALT

First, we will prove that the XTALT system is sound with respect to TALT. The top level structure of the XTALT system is given by a type of program, which represents the abstract syntax of XTALT programs. Well-typedness of XTALT program's is given by a predicate check_program, holding for well-typed programs.

```
check_program  : program -> type.
```

In a foundational system, we are reasoning about pieces of low-level code. We thus will get pieces of code (in binary) representing the raw machine code of the program. Thus a second component of the XTALT system formalizes the binary parser, relating binary code to the abstract syntax of XTALT. Here the type of astring formalizes a string of bytes, representing the code.

```
parse_program  : astring -> program -> type.
```

On the TALT side, the top level concept is that of an abstract machine, called machine, representing the state of execution of a program. Well-typed abstract machines are represented by a predicate machineok.

```
machineok      : machine -> type.
```

Relating XTALT to TALT then involves computing the initial state of a TALT machine when loaded with an XTALT program. Further, XTALT has additional annotations which must be stripped away. Both of these tasks are performed by a relation of elaboration, defined by the relation elab_program.

```
elab_program : program -> machine -> type.
```

The soundness of the XTALT type checker then involves saying that for all XTALT programs P and TALT machines M such that P elaborates to M, if P is well-typed in XTALT, then the machine M is well-typed in TALT. This is the content of the metatheorem sound_program:

```
sound_program : elab_program P M
                    -> check_program P
                    -> machineok M -> type.
%mode sound_program +D1 +D2 -D3.
```

This proof has to proceed by the structure of the program, breaking it into its component instructions, operands and values. At each of these components, we have a corresponding notion of elaboration to closely related concepts in TALT.

On the concrete level, we have a type of state as discussed in the safety policy. We have a relation implements between concrete state and TALT machine that ensures the state realizes the abstract machine. We will discuss this in greater detail in the concrete part of the proof.

We then have a second theorem which ensures that the binary parser is sound, that is if a binary S is parsed as an XTALT program P, and the initial state of the concrete machine when S is loaded is ST, then there is a TALT machine M that the program P elaborates to, and further, this machine M is realized by ST. We call this theorem initial_impl.

```
initial_impl : parse_program S P
                  -> initial_state S ST
                  -> elab_program P M
                  -> implements ST M -> type.
%mode initial_impl +D1 +D2 -D3 -D4.
```

This theorem has to relate the initialization of the abstract machine to the set of possible initial states of the concrete machine. The concrete machine leaves many low-level details indeterminate, as we have discussed in chapter 3. Such details include, for example, the actual starting address of the code and the contents of the registers which do not contain known data.

4.3.3 Abstract Safety of TALT

Next, we have to prove TALT safe. This is done by Crary [Cra03], and we reuse this proof as a part of our proof. Briefly, we have as discussed, a type of abstract machines (machine). This is given a small-step operational semantics which is quite low-level and close to the machine. Some details such as the concrete identity of pointers is held abstract at this stage. The semantics is given in terms of a relation stepsto.

```
stepsto   : machine -> machine -> type.
```

Type safety is then proved in the standard syntactic style [WF94, Har94] by proving the theorems of progress and preservation. Progress says that all well-typed machines can make a step.

```
progress : machineok M1
            -> stepsto M1 M2 -> type.
%mode progress +D1 -D2.
```

Preservation says that if a machine is well-typed and makes a step, the result is itself well-typed.

```
preservation : machineok M1
                -> stepsto M1 M2
                -> machineok M2 -> type.
%mode preservation +D1 +D2 -D3.
```

The abstract semantics does not make use of the oracle described for the concrete machine, and thus possesses an indeterminate semantics in many cases. An example is the set of memory locations collected by the garbage collector. Another example is pointer arithmetic, when condition codes are determined by the actual numeric values. The TALT abstract machine holds pointers abstract, and hence has no way of knowing the resulting condition codes. Since the simulation result of the next section shows that the operational semantics of TALT is realized by the concrete semantics, this mismatch has to be handled in that part of the proof.

4.3.4 Concrete Simulation of Semantics

Finally, we have to show that the concrete semantics correctly realizes the abstract semantics. Recall that we have a concrete state which represents the state of the IA-32 machine, and abstract machine at the TALT level. The realization is mediated by a relation of implements.

```
implements   : state -> machine -> type.
```

The implementation relation matches components of the abstract machine to the corresponding components of the concrete architecture. It is at this stage that the technicalities of the concrete architecture have to be correctly handled. One crucial issue is that the relation is not a one to one mapping, but rather a many to many mapping. For example, a TALT machine holds various data abstract, such as the actual representation of pointers. At the concrete level, pointers are just particular 32 bit values. The realizing relation has to perform this mapping, while still mapping enough structure to carry out the proof of simulation.

A second important way in which the abstract machine is more abstract is that it has complex instructions that are implemented by a sequence of concrete instructions. We may mention the cmpjcc instruction from TALT [Cra03], which for precise typing reasons amalgamates a compare and a conditional jump instruction. What this means is that one step of the abstract semantics is implemented by a variable number of transitions of the concrete system. We thus define a relation transitions for multiple transitions, indexing it by the number of transitions as follows.

```
transitions   : nat -> state -> state -> type.
transitions_z : transitions z ST ST.
```

```
transitions_s : transitions (s N) ST1 ST3
                 <- transition ST1 ST2
                 <- transitions N ST2 ST3.
```

With this relation, we can now proceed to prove the two important lemmas of the concrete stage, that of simulation, and that of determinism.

Simulation

The main body of the work in the concrete stage is to prove the simulation lemma. This asks that we imagine we have a concrete state ST, which we know implements an abstract machine M. Further assume that the abstract machine steps to an abstract machine M1. Then it must be the case that there are concrete transitions (one or more) to another concrete state ST'. We allow for multiple transitions, but eventually the concrete machine must reach a state that implements a TALT abstract machine. But what can we say about this machine? We know that ST' may not implement the abstract machine M1, since the abstract machine has a set of indeterminate choices. What we can prove is that ST' implements a machine M2 which is also reachable in the abstract semantics. M2 may or may not be the same as M1. In other words, we prove that we can match some possible abstract step, as long as there is at least one possible step.

```
simulate : implements ST M
            -> stepsto M M1
            -> transitions (s N) ST ST'
            -> stepsto M M2
            -> implements ST' M2 -> type.
%mode simulate +D1 +D2 -D3 -D4 -D5.
```

Determinism

A second lemma important for the proof is that of determinism, which says that the target of a concrete transition is uniquely determined. This is expressed in Twelf by creating an equality type on states `state_eq` with a single constructor `state_eq_` for reflexivity.

```
state_eq    : state -> state -> type.
state_eq_   : state_eq ST ST.

determinism : transition ST ST1
               -> transition ST ST2
               -> state_eq ST1 ST2 -> type.
%mode determinism +D1 +D2 -D3.
```

With the definition of equality on states, determinism says that whenever a concrete state ST transitions to both ST1 and ST2, ST1 and ST2 are provably identical.

4.3.5 Concrete Progress and Preservation

We will now prove a version of progress and preservation for concrete states. We isolate a set of states to be ok. At a first cut, states which implement a well-typed TALT abstract machine are ok. However, there is a complication. Recall that the concrete system may make more than one transition to implement an abstract step. We certainly want the intermediate concrete states to be ok too. Thus our final definition says that a state is ok if it eventually (in a finite number of steps) transitions to a state which implements a well-typed TALT machine.

```
ok  : state -> type.
ok_ : ok ST
       <- transitions _ ST ST'
       <- implements ST' M
       <- machineok M.
```

Using ok as a notion of well-typed states, we can prove progress and preservation.

Lemma 4.3.1 (Concrete Progress) Concrete progress says that ok states always have a transition.

```
cprogress : ok ST -> transition ST ST' -> type.
%mode cprogress +D1 -D2.
```

Proof

Breaking out the definition of ok, we know that ST steps in zero or more steps to some ST' that implements a well-typed machine. We case analyze on the number of transitions.

Case 1: ST transitions in one or more steps.

Then we have the required transition by picking the first of the sequence.

```
- : cprogress (ok_ _ _ (transitions_s _ Dtrans)) Dtrans.
```

Case 2: ST transitions in zero steps.

Then ST implements a well-typed abstract machine M. Since M is well-typed, it has an abstract step by progress. Simulation then says that ST has a transition sequence of one or more steps. We can just take the first of this sequence.

```
- : cprogress (ok_ Dmok Dimpl transitions_z) Dtrans
     <- progress Dmok Dstep
     <- simulate Dimpl Dstep (transitions_s _ Dtrans) _ _.
```

□

Lemma 4.3.2 (Concrete Preservation) Concrete preservation says that if a state ST is ok and transitions to another state ST', then ST' is also ok.

```
cpreservation : ok ST
                -> transition ST ST'
                -> ok ST' -> type.
%mode cpreservation +D1 +D2 -D3.
```

Proof

Breaking out the definition of ok, we know that ST steps in zero or more steps to some ST'' that implements a well-typed machine. We case analyze on the number of transitions.

Case 1: ST transitions in one or more steps.

Then there is a sequence of transitions, the first of which is to some state ST1, such that ST1 transitions in zero or more steps to ST''. Since ST'' implements a well-typed machine, ST1 is ok. We use determinism to prove that ST1 and ST' must in fact be the same state. Twelf requires us to change equals to equals in the ok relation to make use of this equality. Thus we use a helper lemma which says that the ok relation respects equality of states.

```
- : cpreservation (ok_ Dmok Dimpl (transitions_s Dmtrans Dtrans'))
                  Dtrans Dok
     <- determinism Dtrans' Dtrans Deq
     <- ok_resp Deq (ok_ Dmok Dimpl Dmtrans) Dok.
```

Case 2: ST transitions in zero steps.

Now ST itself implements a well-typed abstract TALT machine M. We can apply progress for TALT to know that M steps to some abstract machine M1. We can apply simulate to get a transition in one or more steps to a concrete state ST''. Further, ST'' implements a machine M2 which M can also transition to. Applying preservation for TALT, we get that M2 is also well-typed. Thus ST'' implements a well-typed machine. But the sequence of concrete transitions from ST to ST'' is nonempty, so there must be a state ST1 that ST transitions to, and ST1 transitions to ST'' in zero or more steps. By the definition of ok, ST1 is ok. Now applying determinism, we get that ST1 must be in fact the same state as ST'.

```
- : cpreservation (ok_ Dmok Dimpl transitions_z) Dtrans Dok
     <- progress Dmok Dstep
```

```
<- simulate Dimpl Dstep (transitions_s Dmtrans Dtrans')
   Dstep' Dimpl'
<- preservation Dmok Dstep' Dmok'
<- determinism Dtrans' Dtrans Deq
<- ok_resp Deq (ok_ Dmok' Dimpl' Dmtrans) Dok.
```

□

4.3.6 Top level safety of the system

We will now put together the static, abstract and concrete parts of the proof to prove safety for the system.

The proof of safety works inductively on the states transitioned to by the program starting from an initial state. Using the ok predicate as our notion of well-typed concrete machines, we prove that when we first load the program, it is ok. Next we prove that any reachable states are also ok. By concrete progress, all ok states have at least one transition (in fact, by determinism, an unique one).

Lemma 4.3.3 (Initial states ok) If we have a binary S which can be checked to be a program checked in XTALT, and if the initial state of execution with the program S is ST, then ST is ok.

```
initial_ok : check _ S
               -> initial_state S ST
               -> ok ST -> type.
%mode initial_ok +D1 +D2 -D3.
```

Proof

The premise that S can be checked to be a program of XTALT can be inverted to get that there must be an XTALT program in abstract syntax P which S is parseable as (the relation parse_program S P holds). Further, P can be checked within the XTALT type system (the predicate check_program P holds).

We apply initial_impl to know that there is an elaboration of P into an abstract TALT machine M which is implemented by ST. The soundness of XTALT type checking then gives us that the machine M is well-typed in TALT (the predicate machineok M holds). Thus, by the definition of ok, ST is ok since it transitions (in zero steps) to a state which implements a well-typed abstract machine.

```
- : initial_ok (check_ Dcheck Dparse) Dinitial
     (ok_ Dmok Dimpl transitions_z)
     <- initial_impl Dparse Dinitial Delab Dimpl
     <- sound_program Delab Dcheck Dmok.
```

□

Lemma 4.3.4 (Reachable states are ok) If we have a binary S which can be checked to be a program checked in XTALT, and if a state ST is reachable when S is executing, then ST is ok.

```
safety' : check _ S
           -> reaches S ST
           -> ok ST -> type.
%mode safety' +D1 +D2 -D3.
```

Proof

We proceed by induction on the reachability judgment.

Case 1: ST is an initial state for the binary S

In this case, we apply lemma 4.3.3 to know that S is ok.

```
- : safety' Dcheck (reaches_z Dinitial) Dok
     <- initial_ok Dcheck Dinitial Dok.
```

Case 2: ST is reachable by a transition from a reachable state ST'

In this case, we apply induction to get that ST' is ok. Then concrete preservation gives us that ST is also ok.

```
- : safety' Dcheck (reaches_s Dtrans Dreach) Dok'
     <- safety' Dcheck Dreach Dok
     <- cpreservation Dok Dtrans Dok'.
```

□

Finally, we can provide a top-level proof of safety, the theorem required by the system. This is an immediate consequence of the above lemma.

Theorem 4.3.5 (Safety of system) If we check the binary S using the XTALT system, and the concrete state ST is reachable when S is executing, then ST always has a transition to some state ST'.

```
safety : check _ S
           -> reaches S ST
           -> transition ST ST' -> type.
%mode safety +D1 +D2 -D.
```

Proof

By the previous lemma safety', the state ST is ok. By concrete progress, it always has a transition.

```
- : safety Dcheck Dreaches Dtrans
     <- safety' Dcheck Dreaches Dok
     <- cprogress Dok Dtrans.
```

□

Chapter 5

Checking Safety of a Program

Our method of proving the safety of programs is factored into first defining and proving safe a whole class of programs, and then showing that the program we are interested in belongs to the class so defined. We are now interested in the second problem, that of showing that a particular program belongs to the defined group of programs.

We have defined a safety condition (for example, TALT) to isolate the group of programs. This safety condition is not trusted, and in fact can be different depending on the wish of the code producer. It is convenient to state this condition in a framework, such as LF. Our checking procedure thus must be provably correct with respect to an untrusted specification within LF.

Our method of solving this problem is to let the code producer provide her own checking procedure for her own safety condition. We will verify the procedure with respect to the safety condition. That programs compute the right result if they terminate is a partial correctness result. We will use dependent type theory to express and prove partial correctness. The programming language will be defined in such a way that the correctness of programs is expressible within the type structure of the language.

The language we develop for this task is a functional language based on the core ML language. We call it LF/ML, for ML with LF dependencies. The notion of safety for LF/ML is much more extended from the usual memory safety, to partial correctness with respect to specifications expressed in LF.

The technical definition of LF/ML will occupy the next three chapters. In this chapter, we motivate the design of this language, and provide an overview of the language and programming within it. Particular design choices made will be justified by means of examples.

The running example for this chapter is a type checker for the simply typed lambda calculus. We imagine practical safety conditions will be designed as a type system, such as XTALT itself is. Checking programs in this case is then the type checker for the system. The simply typed lambda calculus is a particularly simple type system. However, it already possesses variable binding, a key sticking point for reasoning systems. Studying this example will let us isolate the issues.

We will begin by recalling how the type system looks like in an implementation in a functional language, and how it looks like within an LF encoding, in section 5.1. We will then begin extending ML by a notion of dependent types over LF representations in section 5.2. We will then depict this first system in use to check a few cases in section 5.3. Dealing with variables and binding will oblige us to extend our representation language. We depict our solution in section 5.4. We then look back at the language LF/ML and consider the proof obligations in section 5.5.

5.1 Representations in ML and LF

We will start with a core functional language similar to ML. This is a simple language with recursive functions and pairs. The syntax of this language is shown in figure 5.1. We equip this language with a conventional call-by-value semantics, which we elide in this presentation.

Types	$\mathsf{c}, \tau ::=$	Unit	Unit Type
		$\tau_1 \rightarrow \tau_2$	Arrow Type
		$\tau_1 \times \tau_2$	Product Type
		D	Datatype
Terms	$e ::=$	unit	Unit
		x	Variables
		c	Constants
		$\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e$	Functions
		$e_1\, e_2$	Applications
		$\langle e_1, e_2 \rangle$	Pairs
		$\pi_i\, e$	Projection from Pair
		error	Error
		$\mathsf{case}\, e\, \mathsf{of}\, ms\, \mathsf{end}$	Case
		$\mathsf{let\ val}\, x = e_1\, \mathsf{in}\, e_2\, \mathsf{end}$	Let form
		$\mathsf{C}\, e$	Datatype Constructors
Matches	$ms ::=$	$\cdot$	Empty
		$c\, x \Rightarrow e \vert ms$	Cons

Figure 5.1: A Core Functional Language

5.1.1 A functional type checker

Given such a language, we might imagine writing a typechecker for the simply typed lambda calculus in deBruijn form. We will use the language above, with a few additions for clarity. In particular, we will allow ourselves to write multiple declarations in a let form for clarity, and tuple arguments to a function for clarity. Then the code will look something like the following:

```
datatype Type = Base | Arrow of Type * Type
datatype Context = Nil | Cons of Type * Context
datatype Exp = Var of int | Lam of Type * Exp | App of Exp * Exp | Unit

fun typecheck (c:Context,e:Exp) : Type.
  case e of
    Var i => lookup c i
  | Unit _ => Base ()
  | Lam (t,e) =>
    let
       val cnew = Cons (t,c)
       val tbody = typecheck (cnew,e)
    in
       Arrow (t,tbody)
    end
  | App (e1, e2) =>
    let
       val t1 = typecheck (c,e1)
       val t2 = typecheck (c,e2)
    in
       case t1 of
          Arrow (t11,t12) =>
              if sametype (t11, t2) then t12 else error
```

```
        | Base _ => error
    end
```

Looking over this program, we notice that we start with defining a bunch of datatypes for expressions and types. We then write a recursive function, which case analyzes its input. Some cases are simple enough to return an immediate answer, while others require a recursive call. Also, we may need to call other functions, or abort on error.

Turning now to look closer at the variables, we notice that the program assumes its input is in deBruijn notation. Thus variables are annotated with their deBruijn index, and contexts are merely lists of types abstracted over. For our purposes, we have chosen this notation instead of a named form notation which will require checking name equality. This notation is more suited to reasoning and relating to LF, but does require preprocessing if a concrete named representation is available. Also, for ease of explanation, we assume all lambda forms are explicitly typed to ignore issues of type inference, even though this is not essential to our method.

5.1.2 A proof representation in LF

Recall also the LF notation for simply typed lambda calculus, which we reproduce for convenience.

```
tp    : type.

b     : tp.
arrow : tp -> tp -> tp.

tm    : type.

u     : tm.
lam   : tp -> (tm -> tm) -> tm.
app   : tm -> tm -> tm.


of : tm -> tp -> type.

of_u    : of u b.
of_app  : of (app M1 M2) T2
           <- of M1 (arrow T11 T2)
           <- of M2 T1
           <- eqtp T1 T11.
of_lam  : of (lam T1 M) (arrow T1 T2)
           <- ({x:tm} of x T1
                    -> of (M x) T2).
```

This representation defines a set of (LF) types corresponding to the language, and declares constants at those types. The typing judgment has different cases for each sort of term, for unit, applications, and abstractions. These cases might have subderivations of typing, or may call upon other relations. Thus far the representation is similar in spirit to the functional one.

The LF representation differs from the functional one in its treatment of binding. By the principle of higher-order abstract syntax, there is no case for variables in expressions, and correspondingly for typing. All possible uses of variable are put in the LF context by the rule for typing abstractions. The LF context is implicit, and does not even appear in the signature of the `of` type family, unlike the explicit context argument in the functional case.

5.2 Indexed types

We now begin to relate the ML datatypes with the LF types. This leads us to create a restricted form of dependent types (both Π and Σ types) as in Dependent ML [XP99]. The types depend on objects from a static index domain, which in our case is our representation language, LF.

We thus add a new syntactic class of indices, which are synonyms for closed LF objects (or type families). These will be classified by closed LF families (or kinds), which we call sorts. Indices can be used in the types of the language by means of universal dependent types (Π types) and existential dependent types (Σ types). The index domain will be explicitly typed. Inhabitants of Π types are functions over index arguments, and they are eliminated by application to indices. Σ types are introduced by a `pack` construct, and eliminated by a `let pack` construct. We will also need, for purposes explained later while typechecking case analyses, a set of constraints, which are conjunctions of (well-sorted) equations on indices. We show these additions in figure 5.2

Indices	$\mathcal{P}$ ::=	M	LF objects (extended, as in Figure 5.3)
	|	A	LF families (extended, as in Figure 5.3)
Sorts	$\mathcal{Q}$ ::=	A	LF families (extended, as in Figure 5.3)
	|	K	LF kinds
Types	c, τ ::=	...	
	|	$\Pi w{:}\mathcal{Q}.\mathsf{c}$	Universal Dependent Types
	|	$\Sigma w{:}\mathcal{Q}.\mathsf{c}$	Existential Dependent Types
	|	$\mathsf{D}(\mathcal{P})$	Datatypes
Matches	ms ::=	$\cdot$	
	|	$\mathsf{C}\,[w, x] \Rightarrow e \mid ms$	
Terms	e ::=	...	
	|	$\mathsf{Fun}\, x(w{:}\mathcal{Q}){:}\mathsf{c}.e$	
	|	$e\ [\mathcal{P}]$	
	|	$\mathsf{pack}\,\langle \mathcal{P}, e\rangle$	
	|	$\mathsf{let\ pack}\,\langle w, x\rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$	
	|	$\mathsf{C}\,[\mathcal{P}, e]$	Datatype Constructors
	|	$\mathsf{case}^{\mathsf{c}}\ e_1\ \mathsf{of}\ ms\ \mathsf{end}$	Case Expression
Signatures	S ::=	$\cdot$	
	|	$S, \mathsf{D}{:}\mathcal{Q} \to \mathsf{TYPE}$	
	|	$S, \mathsf{C}{:}(\Pi w{:}\mathcal{Q}.\mathsf{c}_1 \to \mathsf{c}_2)$	
Contexts	Υ ::=	...	
	|	$\Upsilon, w{:}\mathcal{Q}$	
Constraints	$\mathcal{C}$ ::=	true	True Constraint
	|	$\mathcal{P}_1 =^{?} \mathcal{P}_2 : \mathcal{Q} \wedge \mathcal{C}$	Conjunction

Figure 5.2: Additions for Indices and Sorts

In typing these new constructs, we will need to carry around assumptions on the new index variables, and the term variables. The role of the third set of assumptions $\mathcal{C}$ will be explained during the match rule, and the reader should ignore it for now. We start with the Π types. The introduction form for these types is a function taking index arguments. The domain must be valid in LF, and the return type must be a good type with assumption on the index variable.

$$\frac{\cdot \vdash \mathsf{A} : \mathsf{type} \quad \Psi, \mathsf{X_M}{:}\mathsf{A} \vdash \tau : \mathbb{T} \quad \Psi, \mathsf{X_M}{:}\mathsf{A}; \Upsilon, f{:}(\Pi \mathsf{X_M}{:}\mathsf{A}.\tau); \mathcal{C} \vdash e : \mathsf{c}}{\Psi; \Upsilon; \mathcal{C} \vdash \mathsf{Fun}\, f(\mathsf{X_M}{:}\mathsf{A}){:}\tau.e : \Pi \mathsf{X_M}{:}\mathsf{A}.\mathsf{c}}$$

$$\frac{\cdot \vdash \mathsf{K} : \mathsf{kind} \quad \Psi, \mathsf{X_A}{:}\mathsf{K} \vdash \tau : \mathbb{T} \quad \Psi, \mathsf{X_A}{:}\mathsf{K}; \Upsilon, f{:}(\Pi \mathsf{X_A}{:}\mathsf{K}.\tau); \mathcal{C} \vdash e : \mathsf{c}}{\Psi; \Upsilon; \mathcal{C} \vdash \mathsf{Fun}\, f(\mathsf{X_A}{:}\mathsf{K}){:}\tau.e : \Pi \mathsf{X_A}{:}\mathsf{K}.\mathsf{c}}$$

Correspondingly, the elimination form is the application to a LF term, where we merely substitute in the body of the Π.

$$\frac{\Psi; \Upsilon; \mathcal{C} \vdash e : \Pi w{:}\mathcal{Q}.\tau \quad \cdot \vdash \mathcal{P} : \mathcal{Q}}{\Psi; \Upsilon; \mathcal{C} \vdash e\ [\mathcal{P}] : [\mathcal{P}/w]\,\tau}$$

The Σ types are introduced by a package form, and eliminated by a unpackaging construct, as in the following.

$$\frac{\cdot \vdash \mathcal{P} : \mathcal{Q} \quad \Psi, w{:}\mathcal{Q} \vdash \tau : \mathbb{T} \quad \Psi; \Upsilon; \mathcal{C} \vdash e : [\mathcal{P}/w]\,\tau}{\Psi; \Upsilon; \mathcal{C} \vdash \mathsf{pack}\, \langle \mathcal{P}, e \rangle : \Sigma w{:}\mathcal{Q}.\tau}$$

$$\frac{\begin{array}{c}\Psi \vdash \tau : \mathbb{T} \\ \Psi; \Upsilon; \mathcal{C} \vdash e_1 : \Sigma w{:}\mathcal{Q}.\tau_1 \quad \Psi, w{:}\mathcal{Q}; \Upsilon, x{:}\tau_1; \mathcal{C} \vdash e_2 : \tau\end{array}}{\Psi; \Upsilon; \mathcal{C} \vdash \mathsf{let\ pack}\, \langle w, x \rangle = e_1\, \mathsf{in}\, e_2\, \mathsf{end} : \tau}$$

We now turn to the datatypes. The introduction form for these are the datatype constructors, which take as we noted indices as well as expression arguments. Notice in particular that the constructors are nonuniform in that they target particular instances of datatypes at indices.

$$\frac{\begin{array}{c}\mathcal{S}(\mathsf{C}) = \Pi w{:}\mathcal{Q}.\tau_1 \to \mathsf{D}(\mathcal{P}_2) \\ \cdot \vdash \mathcal{P} : \mathcal{Q} \\ \Psi; \Upsilon; \mathcal{C} \vdash e : [\mathcal{P}/w]\,\tau_1\end{array}}{\Psi; \Upsilon; \mathcal{C} \vdash \mathsf{C}\,[\mathcal{P}, e] : \mathsf{D}([\mathcal{P}/w]\,\mathcal{P}_2)}$$

We now have to eliminate the datatypes. The elimination form is a case statement with matches. In checking matches, we defer to a judgment that knows both the target of the match and the expected result.

$$\frac{\begin{array}{c}\Psi; \Delta; \Upsilon; \mathcal{C} \vdash e_1 : \mathsf{D}(\mathcal{P}) \quad \Psi \vdash \tau : \mathbb{T} \\ \Psi; \Upsilon; \mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}) \to \tau\end{array}}{\Psi; \Upsilon; \mathcal{C} \vdash \mathsf{case}^{\tau}\ e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau}$$

Checking matches is where the system has to do some work. All arguments to the judgment should be thought of as input arguments, since we know while calling these rules all the types involved. We syntactically restrict case analyses to only analyze datatypes. Recall that constructors target different instances of the indexed family of the head datatype. Thus, in checking a branch, we should only check cases which are reachable at runtime. In other words, we have to perform unification of the target type of the datatype constructor and the type being analyzed.

If unification fails, therefore, we need not check the case at all. If unification succeeds, we may discover some new facts that will help us check this branch. Unfortunately, unification of LF involves higher-order unification, which is undecidable in general. Thus, we may be simply unable to solve some equations. The question then becomes what to do in such cases. The safe approach is to conservatively reject all such

unsolvable equations, and insist on fully solvable equations. This is bothersome in practice. We adopt a more liberal approach of postponing such equations, which we record in the constraint list $\mathcal{C}$. Such equations may become solvable when we further clarify the equations. If we reach the end of type checking with postponed equations, we then issue a type checking failure. This approach is used in the Twelf system for logic programming, and also works well for our example system.

To reiterate, unification is attempted at every case branch. If this fails, the branch need not be checked, since it is unreachable. If it succeeds, we continue. If it is unsolvable, it is postponed, until it is reexamined at another case branch. The unification algorithm (not described until chapter 7) is axiomatized by the judgement forms $\Psi;\Gamma \vdash \mathcal{C} \Longrightarrow \mathsf{false}$ for failure and $\Psi;\Gamma \vdash \mathcal{C} \Longrightarrow (\rho, \Psi, \mathcal{C})$ for (possibly partial) success.

The rule for typechecking matches is then as follows:

$$\frac{}{\Psi;\Upsilon;\mathcal{C} \vdash \cdot : \mathsf{D}(\mathcal{P}) \to \tau}$$

$$\frac{\begin{array}{l}\mathcal{S}(\mathsf{C}) = \Pi w{:}\mathcal{Q}.\tau_1 \to \mathsf{D}(\mathcal{P}_2) \\ \mathcal{S}(\mathsf{D}) = \Pi w'{:}\mathcal{Q}'.\mathbb{T} \\ \Psi, w{:}\mathcal{Q};\cdot \vdash \mathcal{P}_2 \doteq \mathcal{P}_1 : \mathcal{Q}' \wedge \mathcal{C} \Longrightarrow (\rho, \Psi_1, \mathcal{C}_1) \\ \Psi_1;\Upsilon[\rho], x{:}\tau_1[\rho];\mathcal{C}_1 \vdash e[\rho] : \tau[\rho] \\ \Psi;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1) \to \tau\end{array}}{\Psi;\Upsilon;\mathcal{C} \vdash \mathsf{C}\,[w, x] \Rightarrow e | ms : \mathsf{D}(\mathcal{P}_1) \to \tau}$$

$$\frac{\begin{array}{l}\mathcal{S}(\mathsf{C}) = \Pi w{:}\mathcal{Q}.\tau_1 \to \mathsf{D}(\mathcal{P}_2) \\ \mathcal{S}(\mathsf{D}) = \Pi w'{:}\mathcal{Q}'.\mathbb{T} \\ \Psi, w{:}\mathcal{Q};\cdot \vdash \mathcal{P}_2 \doteq \mathcal{P}_1 : \mathcal{Q}' \wedge \mathcal{C} \Longrightarrow \mathsf{false} \\ \Psi;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1) \to \tau\end{array}}{\Psi;\Upsilon;\mathcal{C} \vdash \mathsf{C}\,[w, x] \Rightarrow e | ms : \mathsf{D}(\mathcal{P}_1) \to \tau}$$

5.3 A first checker

Using the system above, we can now start writing the checking program. We extend the datatypes of types and terms to be indexed by the corresponding LF types. Similarly, datatype constructors are indexed to produce particular elements of the LF type family. Definitions in this section are temporary, and will have to be changed in the next section. We use `{a:S}b` as concrete syntax for $\Pi a{:}S.b$ forms, and `<a:S>b` as concrete syntax for $\Sigma a{:}S.b$ forms. We indicate LF representations in gray.

```
datatype Type :: tp -> TYPE

const Base    :: Type [b]
const Arrow   :: {t1:tp}{t2:tp} (Type [t1] * Type [t2] -> Type [arrow t1 t2]


datatype Context  :: TYPE
(* Constants later *)

datatype Exp      :: tm -> TYPE
(* Constants later *)
```

Given these declarations of datatypes, we want the type checker to take a `Context` and a `Exp` and return a `Tp`. We can go further and also express the correctness of the type checker by insisting it also produce a type derivation in the LF system. In other words, let us try writing a type checking function of the following type:

```
typecheck : {e:tm} (Context * Exp [e]) ->
              <t:tp}> <d:of e t> Tp [t]
```

Notice that the output of the function should be a package of the LF representation of the output type and a typing derivation together with the term-level representation of the type.

5.3.1 The case for unit

The simplest case for the checker is for the unit, where we can give a type and a typing derivation outright. Assume then that the constant for unit is declared to produce a term-level expression indexed by u as follows:

```
const U           :: Exp [u]
```

The typechecking function for this case is easy then to write.

```
Fun typecheck (e:exp). fn (ctx, e) =>
  case e of
      U => pack <b, of_u, Base>
```

In this function we have taken the liberty of conflating two successive packages into a single statement. We can check that this satisfies the specification, since we return a LF representative of the base type b as well as a derivation of_u that the input expression u has type b. Notice that for this to be well-typed, we have to unify the target type of U with the input expression type to get more precise information for this case branch.

5.3.2 The case for application

Moving on to the application case, here there are two components of the expression, which have to be recursively checked. The constant declaration looks like:

```
const App :: {e1:tm} {e2:tm} (Exp [e1] * Exp [e2]) ->
               Exp [app e1 e2]
```

Here we have to perform recursive call to the type checking function for the two parts. Finally, we have to check that the function has an arrow type, and in fact the domain type matches that of the argument.

```
...
  case e of
     App (<e1,e2,(e1,e2)>) =>
         let
            pack <t1,dfn,t1>   = typecheck [e1] (c,e1)
            pack <t2,darg,t2>  = typecheck [e2] (c,e2)
         in
            case t1 of
                Arrow (<t11,t12,(t11,t12)>) =>
                     let
                         val result = sametype [t11][t2] (t11,t2)
                     in
                         case result of
                             SOME (<deqtp,()>) =>
                                 <t12,of_app deqtp darg dfn,t12>
                           | NONE => error
                     end
              | Base => error
         end
```

We first perform the two recursive calls, passing the LF representatives found from unpacking the constant App. Then we analyze the function type. If it is a base type we issue an error, while if it is an arrow type, we call a helper function to check the domain type against the argument's type. For ease of explanation,

LF Families	$A ::=$	$\ldots$	
		$\mid 1$	Unit Family
		$\mid \Sigma x{:}A_1.A_2$	Dependent Sum Type
LF Objects	$M ::=$	$\ldots$	
		$\mid \langle\rangle$	Unit Object
		$\mid \langle M_1, M_2\rangle^A$	Pairs
		$\mid \pi_i M$	Projection

Figure 5.3: Σ and Unit Types in LF

we have written the LF signature to take an explicit equality argument, so we can package the type and derivation together, as required.

5.4 Binding and variables: extending the language

We now have some practice in writing closed first order terms in LF/ML. However, when we turn to open terms, we have a problem. The LF representation represents binding by putting assumptions into the LF context. The functional program on the other hand manipulates explicit contexts, in a deBruijn form or otherwise.

Our solution is to reify the LF contexts and manipulate representatives of these contexts. This approach is closely related to the closure conversion done in a compiler to abstract the environment of a function abstraction. In this case we are creating a closure containing both the free computational variables as well as the corresponding representatives in LF. The LF contexts will be represented as products of the abstracted types. Since LF is a dependent language and terms in the context can depend on the earlier part of the context, this will require dependent product, or Σ types. The unit for these products will be a new unit type, which will represent the null context. Open LF terms, or in other words LF terms living in a context, will be represented as functions from contexts to terms. This notion of abstracting over contexts is also done within Twelf to represent implicit bindings.

We thus extend LF to a bigger language that contains Σ and unit types. The additions are detailed in figure 5.3.

These have the usual typing rules for dependent Σ types and an extensionally strong unit type (that is, there is just one canonical inhabitant of unit). We excerpt the typing rules below.

$$\frac{\Gamma \vdash A_1 : \mathsf{type} \quad \Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}}{\Gamma \vdash \Sigma x{:}A_1.A_2 : \mathsf{type}}\ \text{F-SIGMA} \qquad \frac{}{\Gamma \vdash 1 : \mathsf{type}}\ \text{F-UNIT}$$

Notice in particular the typing rule for the dependent pair, where the second component must have a type depending on the first component. For ease of checking, we annotate pairs with the type they are supposed to have.

$$\frac{\begin{array}{l}\Gamma \vdash \Sigma x{:}A_1.A_2 : \mathsf{type} \\ \Gamma \vdash M_1 : A_1 \\ \Gamma \vdash M_2 : [M_1/x]\,A_2\end{array}}{\Gamma \vdash \langle M_1, M_2\rangle^{\Sigma x:A_1.A_2} : \Sigma x{:}A_1.A_2}\ \text{O-PAIR}$$

$$\frac{\Gamma \vdash M : \Sigma x{:}A_1.A_2}{\Gamma \vdash \pi_1 M : A_1}\ \text{O-PROJ1} \qquad \frac{\Gamma \vdash M : \Sigma x{:}A_1.A_2}{\Gamma \vdash \pi_2 M : [\pi_1 M/x]\,A_2}\ \text{O-PROJ2} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}\ \text{O-UNIT}$$

We can now use these new types to represent the contexts of LF. To take an example, consider the context the type checking algorithm must use. These represent the term level contexts (`x1:t1,...,xn:tn`), and in LF are adequately represented by LF contexts of the form `x1:tm,d1:(of x1 t1),...,xn:tm,dn:(of xn tn)`. The context must contain blocks following this "regular worlds" assumption. We enforce this by our constructors for the datatype `Context` as follows:

```
datatype Context  :: type -> TYPE

const Nil         :: Context [1]
const Cons        :: {ctx:type}{t:tp} (Context [ctx] * Tp [tp]) ->
                       Context [<ctx:type><evar:tm>(of evar t)]
```

Notice that we need to have family level metavariables in LF to represent the contexts.

The datatype of expressions has to represent open terms, or in our methodology, functions from contexts to terms. This is similar to the way function abstractions are represented after closure conversion as pairs of environments and a function from an environment to the body. Thus the final version of the datatype is

```
datatype Exp      :: {ctx:type} {e:ctx -> tm} TYPE
```

The application constant, for example, takes as input a context and two expressions living in that context. In our representation, this is a type for the context and two functions from that context to terms. The produced term is a function from the same context to an application form in LF.

```
const App :: {ctx:type}{e1:ctx -> tm} {e2:ctx -> tm}
              (Exp [ctx][e1 ctx] * Exp [ctx][e2 ctx]) ->
                Exp [ctx][λ gamma:ctx.app (e1 gamma) (e2 gamma)]
```

With this change, the signature of the `typecheck` function changes to relate the derivation found to be in the representation of the LF context passed in. It is now:

```
typecheck : {ctx:type}{e:ctx -> tm} (Context * Exp [ctx][e]) ->
              <t:tp}> <d:{gam:ctx} of (e gam) t> Tp [t]
```

Notice that the type of the derivation is now a function type from the context to typing derivations.

5.4.1 The case for abstraction

Abstraction is interesting because it adds to the context. We can use the new types we have introduced to write this case of the typechecker.

The first order of business is to define the constant for abstraction. We must first take a context where this abstraction lives. Also, the type annotation is a parameter. Finally, we take a body of the abstraction. However, recall that this body is not in the ambient context, but is in the context with the addition of a term-level variable. Typechecking this body will involve adding two objects to the context, the term variable and its typing assumption. We can fuse the typing context and the term variable context since we have projections, in a manner that implementations of LF have one standard LF context.

Thus the body of the abstraction can be assumed to live in a bigger context, which is a product of the previous context with a `tm` variable and a typing assumption. The body of the abstraction in LF, ebody,

does not depend on the typing assumption, so we project out the parts needed. This is depicted in the following constant.

```
const Lam :: {ctx:type}{t:tp}{ebody:ctx -> tm -> tm}
             (Tp [t] * Exp[<gam:ctx><evar:tm> of evar t]
                                [λ gam.lam t (ebody (#1 gam) (#1 (#2 gam)))]) ->
             Exp[ctx][λ gam:ctx. lam t (λ x:tm. ebody gam x)]
```

In typechecking the abstraction, we have to pass the recursive call a new extended context. We will get a result derivation living in this bigger context. We will then have to package this derivation to be well-formed in the current context with the additional assumptions.

```
case e of
  | Lam (<gam,tdom,ebody,(tdom,ebody)) =>
       let
          pack <tbody,dbody,tbody> =
              typecheck [<gam:ctx><evar:tm>of evar tdom]
                        [λ gamma.ebody (#1 gamma) (#1 (#2 gamma))]
                        (Cons [gam][tdom](ctx,tdom), ebody)
       in
          pack <arrow tdom tbody,
                λ gamma:ctx.
                      of_lam (λ x:tm.λ d:of x tbody. dbody (gamma,x,d)),
                Arrow [tdom][tbody] (tdom,tbody)>
      end
```

5.4.2 The case for variables

We also have to type check variables. In the LF case, this case is automatic, since the appropriate typing assumption is already in the context, while here we have to project out the necessary assumptions from the context. A variable can only be a well-formed term if it is actually present in the context. We encode this by encapsulating the variable as a projection function projecting out the relevant variable. We ensure this by an auxiliary datatype `Index` for deBruijn index, counting from zero down within the context.

```
datatype Index   :: {ctx:type}{e:ctx -> tm} TYPE

const Z          :: {ctx:type}{t:tp} unit ->
                     Index [<gam:ctx><e:tm> of e t]
                           [λ gam. #1 (#2 g)]
const S          :: {ctx:type}{t:tp}{ein:(ctx -> tm)}
                     Index [ctx][e] ->
                     Index [<gam:ctx><e:tm> of e t]
                           [λ gam. ein (#1 gam)]
```

In the `Z` case, the term is merely the top variable in the context. In the `S` case, we pass in the inner function on the smaller context, and package it to live in the bigger context.

Given this auxiliary datatype, the variable constant for `Exp` is simply a package for an `Index`.

```
const Var        :: {ctx:type}{e:ctx -> tm}
                    Index [ctx][e] -> Exp [ctx][e]
```

Correspondingly, the type checking function merely defers to a lookup function for variables.

```
case e of
 | Var (<gam,v,v) =>
      getTypeFromCtx [gam][e] (ctx,v)
```

The lookup function must search through the context for the variable in question. Here note that well-formed variable indices must always be valid for the context, so we do not have to go off the end of the context. Here is a case where type checking ensures that code is unreachable.

```
Fun getTypeFromCtx (ctx:type, e:c -> exp). fn (ctx, e) =>
    case (ctx,e) of
         (Nil, Z _) => ()   (* Unreachable case !! *)
       | (Nil, S _) => ()   (* Unreachable case !! *)
```

Now consider where we have reached the index `Z`. The required type is embedded in the datatype, and the typing derivation for the expression is the last assumption on the context. Thus we have:

```
    case (ctx, e) of
       | (Cons (<gam, tp1, (ctx1, tp1)>), Z _) =>
            pack <tp1,
                  λ gam. (#2 (#2 g)),
                 tp1>
```

In the case where we have to look within the inner context, we perform a recursive call. The output derivation from the recursive call lives in the smaller context, so it has to be packaged appropriately.

```
    case (ctx, e) of
       | (Cons (<gam, tp1, (ctx1, tp1)>),
          S (<gam', tp1', ein, ein>)) =>
            let
                pack <tout, dout, tout> =
                      getTypeFromCtx [gam] [ein] (ctx1, ein)
            in
                pack <tout,
                      λ gam. dout (#1 gam),
                       tout>
            end
    end
```

This completes the presentation of the simple type checker. Looking back, we see how little the structure of the program has to change from what would have been written in core ML. There are additional annotations and packages of LF representations, but these are all purely static objects. In effect, they can be erased to produce a program well-typed in ML. This creates a simple implementation strategy for our language, in that after type checking we perform a source to source translation by erasure from LF/ML to ML.

5.5 The Language LF/ML

We have seen in this chapter a user's view of the language LF/ML. We recapitulate the key features now, and then will talk about the proof obligations.

LF/ML is a two-level dependently-typed calculus, with a functional programming language at its core. The type structure includes dependent function and pair types, with dependencies over a representation language.

The representation language is an extension of LF with Σ types and a unit type with one canonical inhabitant. Further, we must add metavariables to LF at both object and family levels to stand for our index variables. These metavariables from the external language always stand for closed terms. We call this extended language $\mathrm{LF}^{\Sigma,1+}$, and it is studied in chapter 6. To be a good representation language for our methodology, it should have canonical forms which are used in our adequacy proofs. We would also like the theory to be conservative over LF, so that we can import preexisting LF signatures.

Typechecking the language LF/ML involves in the case analysis case unification between arbitrary (closed) LF objects and type families. This problem is undecidable in general. We isolate a practically useful class

of problems which is decidable, and give a unification algorithm in chapter 7. We postpone equations that do not fall in this class for further clarification.

Finally, in chapter 8, we study the type and term structure of LF/ML itself, and prove important safety properties of the system. Safety for LF/ML allows us to conclude partial correctness of programs written within the language. We use this property to write certified type checkers within our system.

Chapter 6

The representation language $\mathbf{LF}^{\Sigma,1+}$

In this chapter, we introduce a variant of the LF type theory [HHP93] to serve as our representation language. This will serve as our representation layer for the language LF/ML. The LF language is widely used as a logical framework. For our purposes, we need to introduce metavariables which can be substituted by closed terms. It turns out to be necessary for solving unification problems to generalize slightly and introduce metavariables representing terms in arbitrary contexts, not just the empty context. This is following on the description of unification sketched out in contextual modal type theory [NPP05]. The other change we introduce is to have dependent sum (so-called *sigma*-types) and an extensionally strong unit (which has just one canonical inhabitant) to represent reified contexts of pure LF.

The main technical result needed for the type theory is that every well-formed object has an equivalent canonical form. This is important since canonical forms are related to representations via the adequacy properties. As we have described earlier, adequacy is crucial in arguing about the representations used within LF/ML.

A second useful result is an algorithm to check type equality. This enables us to effectively check well-typedness. The main issue with type checking is that deciding typing inevitably requires checking type equivalence, as is typical in a dependent type theory. We have a fairly rich notion of definitional type equivalence, incorporating the usual rules of β and η conversions in a typed setting.

Fortunately, the two technical problems can be solved together. The algorithm we give depends on a type directed equivalence check that deals cleanly with both the η and β rules. The algorithm can be shown to be correct with respect to the declarative system using a standard logical relations argument. Also, the algorithm can be instrumented to produce an equivalent representative in canonical form. Thus, all well-typed terms turn out to have an equivalent canonical representation. The algorithm we describe is used as the basis for the implementation of $\mathrm{LF}^{\Sigma,1+}$ within the LF/ML system.

A further significant technical result is that this theory is a conservative extension of LF. This lets us reuse LF representations used in the metalogical proofs within Twelf as the basis for checking programs written in LF/ML.

6.1 Abstract Syntax

We start with defining the abstract syntax of $\mathrm{LF}^{\Sigma,1+}$. This is generated by the grammar in figure 6.1.

There are three levels of terms: objects, type families and kinds. Kinds classify families, and families belonging to kind type are called *types*. Types classify objects. The family level includes Π and Σ types, a unit type, as well as family-level abstractions and applications. We have object and family-level constants. A signature records the types and kinds assigned to these constants. We have object-level variables, and these are provided types by contexts. The other object level constructs are abstractions and applications, pairs and projections, and a canonical inhabitant of the unit type.

To those familiar with the LF type theory [HHP93], our calculus extends it with dependent sum and unit

Kinds	K ::=	$\mathsf{type} \mid \Pi x{:}A.K$
Families	A ::=	$a \mid \Pi x{:}A_1.A_2 \mid \Sigma x{:}A_1.A_2 \mid 1 \mid \lambda x{:}A_1.A_2 \mid A\,M \mid X_A[\sigma]$
Objects	M ::=	$c \mid x \mid \lambda x{:}A.M \mid M_1\,M_2 \mid \langle M_1, M_2\rangle^A \mid \pi_i M \mid \langle\rangle \mid X_M[\sigma]$
Substitutions	σ ::=	$\cdot \mid \sigma, M/x$
Signatures	Σ ::=	$\cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A$
Contexts	Γ ::=	$\cdot \mid \Gamma, x{:}A$
Metavariable Contexts	Ψ ::=	$\cdot \mid \Psi, X_A{::}(\Gamma \vdash K) \mid \Psi, X_M{::}(\Gamma \vdash A)$
Metavariable Substitutions	ρ ::=	$\cdot \mid \rho, M/X_M \mid \rho, A/X_A$

Figure 6.1: Abstract Syntax of LFS

$$\frac{}{\vdash \cdot : \mathsf{sig}}\ \text{LFSig-Nil} \qquad \frac{\vdash \Sigma : \mathsf{sig} \qquad \cdot;\cdot \vdash_\Sigma K : \mathsf{kind}}{\vdash \Sigma, a{:}K : \mathsf{sig}}\ \text{LFSig-Fam}$$

$$\frac{\vdash \Sigma : \mathsf{sig} \qquad \cdot;\cdot \vdash_\Sigma A : \mathsf{type}}{\vdash \Sigma, c{:}A : \mathsf{sig}}\ \text{LFSig-Obj}$$

Figure 6.2: Well-formed signatures

types. Further, there are metavariables at both object and family levels. The metavariables always appear within terms associated with an explicit substitution, which mediates between the current context and the declared context for the metavariable.

Contexts Γ, which assign types to variables, can be assumed to be ordered lists. Ordering is important because of dependencies. We assume that a variable is not bound more than once in contexts. This can always be assured in the standard way in all judgments by tacitly renaming newly bound variables if necessary. Similarly, we have metavariable contexts Ψ to record assumptions for object and family level metavariables. The assumptions note both the expected type as well as the expected context any substituent must live in. Again, we assume a metavariable is not bound more than once.

We define a partial order on contexts (ordinary and metavariable) syntactically. $\Gamma_1 \subseteq \Gamma_2$ holds if $\Gamma_1(x) = \Gamma_2(x)$ for every $x \in \mathsf{dom}(\Gamma_1)$. Thus if $\Gamma_1 \subseteq \Gamma_2$ then $\mathsf{dom}(\Gamma_1) \subseteq \mathsf{dom}(\Gamma_2)$, and Γ_1 appears as a subsequence of Γ_2.

6.2 Static Semantics

The semantics of the calculus is given by a set of judgments defining what it means for objects, type families and kinds to be well-formed, taking into account the classifier type for objects and kind for type families. As is typical for a dependent type theory, the judgments are mutually inductively defined. Further, the important rule of type conversion (and analogously kind conversion) incorporates a notion of definitional equivalence in assigning types.

We begin describing the system by defining well-formed signatures and contexts in figures 6.2,6.3, and 6.4. Broadly, these ensure that every defined constant and variable gets a type well-formed in the ambient context.

To simplify the presentation, from now on we will assume fixed a valid signature Σ and omit it from the

$$\frac{}{\vdash \cdot}\ \text{M-Nil} \qquad \frac{\begin{array}{l}\vdash \Psi \\ \Psi \vdash \Gamma : \mathsf{ctx} \\ \Psi;\Gamma \vdash \mathsf{K}_b : \mathsf{kind}\end{array}}{\vdash \Psi, \mathsf{X}_\mathsf{A} :: (\Gamma \vdash \mathsf{K})}\ \text{M-Fam} \qquad \frac{\begin{array}{l}\vdash \Psi \\ \Psi \vdash \Gamma : \mathsf{ctx} \\ \Psi;\Gamma \vdash \mathsf{A}_b : \mathsf{type}\end{array}}{\vdash \Psi, \mathsf{X}_\mathsf{M} :: (\Gamma \vdash \mathsf{A})}\ \text{M-Obj}$$

Figure 6.3: Well-formed meta variable contexts

$$\frac{}{\Psi \vdash \cdot : \mathsf{ctx}}\ \text{LFCtx-Nil} \qquad \frac{\Psi \vdash \Gamma : \mathsf{ctx} \qquad \Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}}{\Psi \vdash \Gamma, \mathsf{x}{:}\mathsf{A} : \mathsf{ctx}}\ \text{LFCtx-Cons}$$

Figure 6.4: Well-formed contexts

judgments. All further judgments of the calculus assume a fixed well-formed signature.

Substitutions are well-formed just when each substituent is well-formed in the ambient context, as is described in figure 6.5.

With these preliminaries out of the way, we can now describe well-typed objects in figure 6.6, well-kinded type families in figure 6.7, and well-formed kinds in figure 6.8.

Notice in particular the rules O-Feq and F-Keq, which allow the conversion of types and kinds. These rely on a notion of definitional equality for the system.

We now turn to the presentation of definitional equality which is also defined mutually recursively with the typing judgments. This is intuitively a congruence closure of the β and η conversion rules on well-typed terms. Substitutions are equal precisely when each component of the two substitutions are equal in the ambient context, as shown in figure 6.9.

Turning now to objects, we define equality to be a congruence relation on the structure of the object (figure 6.10). We have explicit symmetry and transitivity rules for equality. Reflexivity is not an explicit rule, but will turn out to be admissible within this system.

The interesting rules for object equality are the conversion rules, given in figure 6.11. We have extensionality rules for Π and Σ types, as well as for the unit type. The unit in particular is extensionally strong, which means that it has only one canonical inhabitant. This has the consequence that any two well-typed objects of this type must themselves be equal. There are also parallel β conversion rules for both function applications and pair projections.

Type families have a corresponding notion of definitional equality, given in figure 6.12. This is also a congruence, and further, since we have type level abstractions, there is both beta and extensionality rules at the higher kind. Notice also that component objects of type families are judged equal by appeal to the definitional equality judgment for objects.

$$\frac{}{\Psi;\Gamma \vdash \cdot : \cdot}\ \text{Sub-Nil} \qquad \frac{\Psi;\Gamma \vdash \sigma : \Gamma_1 \quad \Psi;\Gamma \vdash \mathsf{A} : \mathsf{type} \quad \Psi;\Gamma \vdash \mathsf{M} : [\sigma]\mathsf{A}}{\Psi;\Gamma \vdash (\sigma, \mathsf{M}/\mathsf{x}) : (\Gamma_1, \mathsf{x}{:}\mathsf{A})}\ \text{Sub-Cons}$$

Figure 6.5: Well-formed substitutions

$$\frac{\Gamma(\mathsf{x}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{x} : \mathsf{A}}\ \text{O-VAR} \qquad \frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{c} : \mathsf{A}}\ \text{O-CONST} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \text{O-ABS}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 : \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 \quad \Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{M}_1\ \mathsf{M}_2 : [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_1}\ \text{O-APP} \qquad \frac{\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type} \quad \Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1 \quad \Psi;\Gamma \vdash \mathsf{M}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2}{\Psi;\Gamma \vdash \langle \mathsf{M}_1, \mathsf{M}_2 \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \text{O-PAIR}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_1 \mathsf{M} : \mathsf{A}_1}\ \text{O-PROJ1} \qquad \frac{\Psi;\Gamma \vdash \mathsf{M} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_2 \mathsf{M} : [\pi_1 \mathsf{M}/\mathsf{x}]\,\mathsf{A}_2}\ \text{O-PROJ2} \qquad \frac{}{\Psi;\Gamma \vdash \langle\rangle : 1}\ \text{O-UNIT}$$

$$\frac{\Psi(\mathsf{X}_\mathsf{M}) = \Gamma_1 \vdash \mathsf{A} \quad \Psi;\Gamma_2 \vdash \sigma : \Gamma_1}{\Psi;\Gamma_2 \vdash \mathsf{X}_\mathsf{M}[\sigma] : [\sigma]\mathsf{A}}\ \text{O-META} \qquad \frac{\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}_1 \quad \Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}}{\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}_2}\ \text{O-FAMEQ}$$

Figure 6.6: Well-formed objects

$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{a} : \mathsf{K}}\ \text{F-CONST} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{K}}{\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{K}}\ \text{F-LAM} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{K} \quad \Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{A}_1\ \mathsf{M} : [\mathsf{M}/\mathsf{x}]\,\mathsf{K}}\ \text{F-APP}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}}{\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}}\ \text{F-PI} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}}{\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}}\ \text{F-SIGMA} \qquad \frac{}{\Psi;\Gamma \vdash 1 : \mathsf{type}}\ \text{F-UNIT}$$

$$\frac{\Psi(\mathsf{X}_\mathsf{A}) = \Gamma_1 \vdash \mathsf{K} \quad \Psi;\Gamma_2 \vdash \sigma : \Gamma_1}{\Psi;\Gamma_2 \vdash \mathsf{X}_\mathsf{A}[\sigma] : [\sigma]\mathsf{K}}\ \text{F-META} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}_1 \quad \Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}}{\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}_2}\ \text{F-KEQ}$$

Figure 6.7: Well-formed type families

$$\frac{}{\Psi;\Gamma \vdash \mathsf{type} : \mathsf{kind}}\ \text{K-Type} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type} \qquad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{K} : \mathsf{kind}}{\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K} : \mathsf{kind}}\ \text{K-Pi}$$

Figure 6.8: Well-formed kinds

$$\frac{}{\Psi;\Gamma \vdash \cdot \equiv \cdot : \cdot}\ \text{Subeq-Nil} \qquad \frac{\begin{array}{c}\Psi;\Gamma \vdash \sigma_1 \equiv \sigma_2 : \Gamma_1 \\ \Psi;\Gamma \vdash \mathsf{A} \equiv \mathsf{A} : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : [\sigma_1]\mathsf{A}\end{array}}{\Psi;\Gamma \vdash (\sigma_1, \mathsf{M}_1/\mathsf{x}) \equiv (\sigma_2, \mathsf{M}_2/\mathsf{x}) : (\Gamma_1, \mathsf{x}{:}\mathsf{A})}\ \text{Subeq-Cons}$$

Figure 6.9: Equal substitutions

Finally, kinds are judged equal by a simple structural congruence, given in figure 6.13. There are no nontrivial conversion rules at the level of kinds.

6.3 Overview of Proof

The main complication in typing is the presence of the conversion rules, which allows us to convert types in the middle of the derivation. The usual strategy is to rely on normalization and canonical forms. This means that the typing algorithm either produces a canonical form type, or while checking equivalence of types, both types are fully normalized and compared syntactically. Unfortunately, in the presence of the extensionality rules, this strategy is rather difficult. The canonical form for a term now depends on both the context and the type at which comparison is done. For example, the Obj-ExtUnit rule says that any two well-typed terms at unit type are equal, since there is only one canonical inhabitant of the extensionally strong unit type.

The idea then is to perform the equivalence check in a type directed way. If terms are being compared at a higher type (or kind), the algorithm drives down to base type by η-expansion. At base types, the two terms are weak-head normalized, and their root structure is compared structurally. Checking subterms (such as in application) may require checking in a type directed way again. The full type does not matter in the type directed phase, but only its structure. Thus the algorithm relies on the simple types and kinds got by erasing dependency information. This erasure simplifies the proof effort considerably by eliminating false dependencies, such as in the Σ type case.

The correctness of the algorithm with respect to the deductive system involves checking that it is complete (that is, all equivalent terms are declared equivalent) as well as sound (that is, all terms declared equivalent are indeed definitionally equal).

Completeness is an interesting problem since a judgment of definitional equality might have arbitrary interleavings of β and η conversions. As we have noted, the extensionality rules are context dependent. The proof proceeds by strengthening the induction hypothesis in higher type cases. This is well-known as the method of Kripke logical relations [Cra05].

In checking soundness of the algorithm, the main preliminary is to prove subject reduction. the algorithm performs weak head reduction, which has to be shown to preserve types. Soundness of the algorithm is then proved by considering each step of the algorithm and proving that they produce a valid typing derivation or preserve typing. We also notice that the algorithm actually canonicalizes all well-typed terms, while failing early when checking ill-typed terms. This allows us to extend the proof in a minor fashion to also produce

$$\frac{\Gamma(\mathsf{x}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{x} \equiv \mathsf{x} : \mathsf{A}}\ \textsc{Objeq-Var} \qquad \frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{c} \equiv \mathsf{c} : \mathsf{A}}\ \textsc{Objeq-Const}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 \\ \Psi;\Gamma \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : \mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} \equiv \mathsf{M}_{21}\,\mathsf{M}_{22} : [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_1}\ \textsc{Objeq-App}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_1 : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{A}_{12} \equiv \mathsf{A}_1 : \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{M}_1 \equiv \lambda \mathsf{x}{:}\mathsf{A}_{12}.\mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{Objeq-Abs}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \mathsf{A}_1 \\ \Psi;\Gamma \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \langle \mathsf{M}_{11}, \mathsf{M}_{12} \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} \equiv \langle \mathsf{M}_{21}, \mathsf{M}_{22} \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{Objeq-Pair}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_1 \mathsf{M}_1 \equiv \pi_1 \mathsf{M}_2 : \mathsf{A}_1}\ \textsc{Objeq-Proj1} \qquad \frac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_2 \mathsf{M}_1 \equiv \pi_2 \mathsf{M}_2 : [\pi_1 \mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2}\ \textsc{Objeq-Proj2}$$

$$\frac{\begin{array}{l}\Psi(\mathsf{X}_\mathsf{M}) = \Gamma_1 \vdash \mathsf{A} \\ \Psi;\Gamma_2 \vdash \sigma_1 \equiv \sigma_2 : \Gamma_1\end{array}}{\Psi;\Gamma_2 \vdash \mathsf{X}_\mathsf{M}[\sigma_1] \equiv \mathsf{X}_\mathsf{M}[\sigma_2] : [\sigma_1]\mathsf{A}}\ \textsc{Objeq-Meta} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_1\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_2}\ \textsc{Objeq-Fameq}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_1 : \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}}\ \textsc{Objeq-Symm} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A} \\ \Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_3 : \mathsf{A}\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_3 : \mathsf{A}}\ \textsc{Objeq-Trans}$$

Figure 6.10: Equal objects: Congruence and type conversion

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{M}_1 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \\ \Psi;\Gamma \vdash \mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1\,\mathsf{x} \equiv \mathsf{M}_2\,\mathsf{x} : \mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{Objeq-ExtPi}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : \mathsf{A}_2 \\ \Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \mathsf{A}_1\end{array}}{\Psi;\Gamma \vdash (\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{12})\,\mathsf{M}_{11} \equiv [\mathsf{M}_{21}/\mathsf{x}]\,\mathsf{M}_{22} : [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2}\ \textsc{Objeq-BetaPi}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_3 : \mathsf{A}_1 \\ \Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \pi_1\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} \equiv \mathsf{M}_3 : \mathsf{A}_1}\ \textsc{Objeq-BetaProj1}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1 \\ \Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_3 : \mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \pi_2\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} \equiv \mathsf{M}_3 : \mathsf{A}_2}\ \textsc{Objeq-BetaProj2}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{M}_1 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \\ \Psi;\Gamma \vdash \mathsf{M}_2 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \\ \Psi;\Gamma \vdash \pi_1 \mathsf{M}_1 \equiv \pi_1 \mathsf{M}_2 : \mathsf{A}_1 \\ \Psi;\Gamma \vdash \pi_2 \mathsf{M}_1 \equiv \pi_2 \mathsf{M}_2 : [\pi_1 \mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{Objeq-ExtSigma}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 : 1 \quad \Psi;\Gamma \vdash \mathsf{M}_2 : 1}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : 1}\ \textsc{Objeq-ExtUnit}$$

Figure 6.11: Equal objects: extensionality and beta

$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{a} \equiv \mathsf{a} : \mathsf{K}}\ \textsc{Fameq-Const} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_1 : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{A}_{21} \equiv \mathsf{A}_1 : \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{K}\end{array}}{\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \lambda \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{K}}\ \textsc{Fameq-Abs}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \Pi \mathsf{x}{:}\mathsf{A}_3.\mathsf{K} \\ \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_3\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1\,\mathsf{M}_1 \equiv \mathsf{A}_2\,\mathsf{M}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{K}}\ \textsc{Fameq-App} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{A}_1 : \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K} \\ \Psi;\Gamma \vdash \mathsf{A}_2 : \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{A}_1\,\mathsf{x} \equiv \mathsf{A}_2\,\mathsf{x} : \mathsf{K}\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K}}\ \textsc{Fameq-Ext}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K} \\ \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}\end{array}}{\Psi;\Gamma \vdash (\lambda \mathsf{x}{:}\mathsf{A}.\mathsf{A}_1)\,\mathsf{M}_1 \equiv [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{K}}\ \textsc{Fameq-Beta}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{A}_{11} : \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{type}\end{array}}{\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Pi \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}}\ \textsc{Fameq-Pi}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{A}_{11} : \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{type}\end{array}}{\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}}\ \textsc{Fameq-Sigma} \qquad \frac{}{\Psi;\Gamma \vdash 1 \equiv 1 : \mathsf{type}}\ \textsc{Fameq-Unit}$$

$$\frac{\begin{array}{l}\Psi(\mathsf{X}_\mathsf{A}) = \Gamma_1 \vdash \mathsf{K} \\ \Psi;\Gamma_2 \vdash \sigma_1 \equiv \sigma_2 : \Gamma_1\end{array}}{\Psi;\Gamma_2 \vdash \mathsf{X}_\mathsf{A}[\sigma_1] \equiv \mathsf{X}_\mathsf{A}[\sigma_2] : [\sigma_1]\mathsf{K}}\ \textsc{Fameq-Meta} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_1 : \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}}\ \textsc{Fameq-Symm}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K} \\ \Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_3 : \mathsf{K}\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_3 : \mathsf{K}}\ \textsc{Fameq-Trans} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \\ \Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}_1\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}_2}\ \textsc{Fameq-Keq}$$

Figure 6.12: Equal type families

$$\frac{}{\Psi;\Gamma \vdash \mathsf{type} \equiv \mathsf{type} : \mathsf{kind}}\ \textsc{Keq-Type} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}\end{array}}{\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{K}_1 \equiv \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{K}_2 : \mathsf{kind}}\ \textsc{Keq-Pi}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{K}_2 \equiv \mathsf{K}_1 : \mathsf{kind}}{\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}}\ \textsc{Keq-Symm} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \\ \Psi;\Gamma \vdash \mathsf{K}_2 \equiv \mathsf{K}_3 : \mathsf{kind}\end{array}}{\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_3 : \mathsf{kind}}\ \textsc{Keq-Trans}$$

Figure 6.13: Equal kinds

a canonical form (that is, a β-normal η-long form) for all well-typed terms.

Finally, using the algorithm for checking equivalence as an intermediary, we can produce an algorithm for full type checking. This gets us decidability of the system. Further, the algorithm is also practical, and serves as the basis for an implementation.

The property that every well-typed term has an equivalent canonical form gets us our requirement of conservativity over the LF type theory. The algorithm on canonical forms can be seen to be totally syntax directed. Turning to LF, briefly, LF can be considered to be the fragment of the above system where we disallow metavariables, pairs, projections, and unit from appearing in the term (and the signature). Assuming no such constants appear in the signature, canonical forms then assures us that terms within the LF fragment have typing derivations lying entirely within LF.

We start with proving a variety of basic properties of the system, in section 6.4. These include the fact that all well-typed terms are definitionally equal to themselves. We also need to prove that the substitution property holds of all judgments, which is vital in a dependent setting. We will also need to pick apart typing derivations using the type inversion lemmas. Finally, an important property required in the rest of the proof is that of regularity (terms appearing in valid derivations are themselves well-formed). This requires some technical lemmas such as functionality of substitutions, that is, equal substitutions take equal elements to equal elements.

The other useful technical lemma in picking apart derivations in the rest of the proof is a property called injectivity of product and sum types. If we have equal Π (or Σ) types, we often need that the components are equal too. This property is not obvious since we have type level abstractions, so nontrivial conversions at the type level may be used in the equality derivation. We adopt a solution of Vanderwaart and Crary [VC02] to maintain stronger induction invariants by a restricted form of logical relations in section 6.5.

With these preliminary key lemmas out of the way, we define our algorithm for checking equality in section 6.6, and proceed to prove completeness and soundness. We will need basic facts about the algorithm for proving completeness, such as the fact that it is symmetric and transitive, and supports weakening the context, for the purposes of the proof. We proceed to prove completeness using Kripke logical relations in section 6.7. Soundness is proved in section 6.8. As we mentioned before, canonical forms is proved together with soundness. We then use this algorithm to produce an algorithm for full type checking in section 6.9, and also use the algorithm to prove conservativity over LF in section 6.10.

6.4 Elementary Properties

We start with some basic properties of the system. The most important lemma in this section is the regularity lemma, which states that terms appearing in valid derivations are themselves well typed. To get there, we first prove some basic structural facts about the typing judgments can be proved by easy structural

inductions.

Lemma 6.4.1 (Weakening) *If* $\Psi;\Gamma_1 \vdash \mathcal{J}$ *and* $\Gamma_1 \subseteq \Gamma_2$*, then* $\Psi;\Gamma_2 \vdash \mathcal{J}$*.*

Proof

By structural induction on the derivation of the judgment. □

Lemma 6.4.2 (Free Variable Containment) *If* $\Psi \vdash \Gamma : \mathsf{ctx}$ *and* $\Psi;\Gamma \vdash \mathcal{J}$*, then* $FV(\mathcal{J}) \in \mathsf{dom}(\Gamma)$*.*

Proof

By structural induction on the derivation of the judgment $\mathcal{J}$. □

Next we show that reflexivity is admissible for our definition of definitional equality. This lemma shows that definitional equality is an equivalence relation (it is already symmetric and transitive by rules).

Lemma 6.4.3 (Reflexivity)

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

Proof

By structural induction on the derivation of the judgment. In each case, the inductive hypotheses provide the required facts to put together a derivation of equality. □

One of the important properties of a declarative system is the admissibility of a substitution property for hypotheses. To prove this, we need some basic facts about substitutions, such as that we can always come up with an identity substitution for a well-typed context, and that substitutions can be extended to have no effect on terms not in the domain of the substitution.

Lemma 6.4.4 (Identity Substitutions) *If* $\Psi \vdash \Gamma : \mathsf{ctx}$ *then* $\Psi;\Gamma \vdash \mathsf{id}_\Gamma : \Gamma$ *and* $\Psi;\Gamma \vdash \mathsf{id}_\Gamma = \mathsf{id}_\Gamma : \Gamma$*.*

Proof

By induction on the construction of the context. □

Lemma 6.4.5 (Extending Substitutions)

1. *If* $\Gamma_1 \vdash \sigma_1 : \Gamma$*,* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$ *and* $\mathsf{x} \notin \mathsf{dom}(\Psi)\Gamma \cup \mathsf{dom}(\Psi)\Gamma_1$ *then* $\Gamma_1, \mathsf{x}{:}\mathsf{A}[\sigma_1] \vdash \sigma_1, \mathsf{x}/\mathsf{x} : \Gamma, \mathsf{x}{:}\mathsf{A}$*.*
2. *If* $\Gamma_1 \vdash \sigma_1 = \sigma_2 : \Gamma$*,* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$ *and* $\mathsf{x} \notin \mathsf{dom}(\Psi)\Gamma \cup \mathsf{dom}(\Psi)\Gamma_1$ *then* $\Gamma_1, \mathsf{x}{:}\mathsf{A}[\sigma_1] \vdash \sigma_1, \mathsf{x}/\mathsf{x} = \sigma_2, \mathsf{x}/\mathsf{x} : \Gamma, \mathsf{x}{:}\mathsf{A}$*.*

Proof

Case 1:

Direct, by the definition of substitution typing, using Weakening.

Case 2:

Direct, by the definition of substitution equality and Weakening. □

Next, we show that the notion of substitutions can be lifted to entire judgments.

Lemma 6.4.6 (Substitution) *If* $\Gamma_1 \vdash \mathcal{J}$ *and* $\Gamma_2 \vdash \sigma : \Gamma_1$*, then* $\Gamma_2 \vdash \mathcal{J}[\sigma]$*.*

Proof

By induction over the structure of the given derivation of $\mathcal{J}$. □

With the substitution property already proved, an useful fact about contexts can now be proved. We often want to change declarations in the context to bind a definitionally equal type. This is now easy to prove.

Lemma 6.4.7 (Context Conversion) *Assume* $\Psi \vdash \Gamma, \mathsf{x}{:}\mathsf{A}_1 : \mathsf{ctx}$*,* $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{type}$*, and* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$*. If* $\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathcal{J}$*, then* $\Gamma, \mathsf{x}{:}\mathsf{A}_2 \vdash \mathcal{J}$*.*

Proof

Direct, by weakening and substitution.

$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_2 \vdash \mathsf{x} : \mathsf{A}_2$	By rule (variable)
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$	By assumption
$\Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_1 : \mathsf{type}$	By rule (symmetry)

$\Psi;\Gamma, x{:}A_2 \vdash x : A_1$ By rule (type conversion)
$\Gamma, x{:}A_1 \vdash \mathcal{J}$ By assumption
$\Gamma, x_1{:}A_1 \vdash [x_1/x]\mathcal{J}$ By substitution
$\Gamma, x{:}A_2, x_1{:}A_1 \vdash [x_1/x]\mathcal{J}$ By weakening
$\Gamma, x{:}A_2 \vdash [x/x_1]([x_1/x]\mathcal{J})$ By substitution
$\Gamma, x{:}A_2 \vdash \mathcal{J}$ By properties of substitution

□

A critical lemma for our purposes is the lemma that is usually called functionality. This says that equal substitutions are "functional", *i.e.* given equal elements, they produce equal elements. Such a property is needed, for example, in our proofs of completeness of algorithmic equality with respect to the definitional equality given above. Our proof of this property needs regularity. However, our proof of regularity itself seems to need this property. Fortunately, the form of functionality needed in the regularity proof is less general. We need the fact that applying equal substitutions to *the same* element produces equal elements. This less general lemma can be proved directly, without an appeal to regularity. After we prove regularity, we can come back and prove the more general version we will need later.

Lemma 6.4.8 (Functionality for Typing) *Assume* $\Gamma_1 \vdash \sigma_1 : \Gamma$, $\Gamma_1 \vdash \sigma_2 : \Gamma$, *and* $\Gamma_1 \vdash \sigma_1 = \sigma_2 : \Gamma$.

1. *If* $\Psi;\Gamma \vdash M : A$ *then* $\Psi;\Gamma_1 \vdash M[\sigma_1] \equiv M[\sigma_2] : A[\sigma_1]$.
2. *If* $\Psi;\Gamma \vdash A : K$ *then* $\Psi;\Gamma_1 \vdash A[\sigma_1] \equiv A[\sigma_2] : K[\sigma_1]$.
3. *If* $\Psi;\Gamma \vdash K : \mathsf{kind}$ *then* $\Psi;\Gamma_1 \vdash K[\sigma_1] \equiv K[\sigma_2] : \mathsf{kind}$.

Proof

By induction on the structure of the derivation of the typing judgment. □

One last lemma is needed before we give our proof of regularity. We need for sum and product types and product kinds the fact that the components of these types are well-formed if the types themselves are well-formed. This is applied in the sum and product type cases in the proof of regularity.

Lemma 6.4.9 (Inversion on Products and Sums)

1. *If* $\Psi;\Gamma \vdash \Pi x{:}A_1.A_2 : K$ *then* $\Psi;\Gamma \vdash A_1 : \mathsf{type}$ *and* $\Psi;\Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$.
2. *If* $\Psi;\Gamma \vdash \Sigma x{:}A_1.A_2 : K$ *then* $\Psi;\Gamma \vdash A_1 : \mathsf{type}$ *and* $\Psi;\Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$.
3. *If* $\Psi;\Gamma \vdash \Pi x{:}A.K : \mathsf{kind}$ *then* $\Psi;\Gamma \vdash A : \mathsf{type}$ *and* $\Psi;\Gamma, x{:}A \vdash K : \mathsf{kind}$.

Proof

By induction on the structure of the given derivation. □

We are now in a position to prove our required regularity property.

Lemma 6.4.10 (Regularity) *Assume* $\Psi \vdash \Gamma : \mathsf{ctx}$.

1. *If* $\Psi;\Gamma \vdash M : A$ *then* $\Psi;\Gamma \vdash A : \mathsf{type}$.
2. *If* $\Psi;\Gamma \vdash M_1 \equiv M_2 : A$ *then* $\Psi;\Gamma \vdash M_1 : A$, $\Psi;\Gamma \vdash M_2 : A$, *and* $\Psi;\Gamma \vdash A : \mathsf{type}$.
3. *If* $\Psi;\Gamma \vdash A : K$ *then* $\Psi;\Gamma \vdash K : \mathsf{kind}$.
4. *If* $\Psi;\Gamma \vdash A_1 \equiv A_2 : K$ *then* $\Psi;\Gamma \vdash A_1 : K$, $\Psi;\Gamma \vdash A_2 : K$, *and* $\Psi;\Gamma \vdash K : \mathsf{kind}$.
5. *If* $\Psi;\Gamma \vdash K_1 \equiv K_2 : \mathsf{kind}$ *then* $\Psi;\Gamma \vdash K_1 : \mathsf{kind}$, *and* $\Psi;\Gamma \vdash K_2 : \mathsf{kind}$.

Proof

By structural induction on the derivation of the judgment. We will show a few representative cases.

Case 1:
$$\frac{\Psi;\Gamma \vdash A_1 : \mathsf{type} \qquad \Psi;\Gamma, x{:}A_1 \vdash M_2 : A_2}{\Psi;\Gamma \vdash \lambda x{:}A_1.M_2 : \Pi x{:}A_1.A_2}\ \text{O-Abs}$$

$\Psi;\Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash \Pi x{:}A_1.A_2 : \mathsf{type}$ By rule

Case 2:
$$\frac{\Psi;\Gamma \vdash M : \Sigma x{:}A_1.A_2}{\Psi;\Gamma \vdash \pi_2 M : [\pi_1 M/x]\,A_2}\ \text{O-Proj2}$$

$\Psi;\Gamma \vdash \Sigma x{:}A_1.A_2 : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash A_1 : \mathsf{type}$,
$\Psi;\Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$ By Inversion on Sums
$\Psi;\Gamma \vdash M : \Sigma x{:}A_1.A_2$ By premise
$\Psi;\Gamma \vdash \pi_1 M : A_1$ By rule (projection)
$\Psi;\Gamma_2 \vdash [\pi_1 M/x]\, A_2 : \mathsf{type}$ By Substitution

Case 3:
$$\frac{\Psi;\Gamma \vdash A_{11} \equiv A_1 : \mathsf{type} \qquad \Psi;\Gamma \vdash A_{12} \equiv A_1 : \mathsf{type} \qquad \Psi;\Gamma, x{:}A_1 \vdash M_1 \equiv M_2 : A_2}{\Psi;\Gamma \vdash \lambda x{:}A_{11}.M_1 \equiv \lambda x{:}A_{12}.M_2 : \Pi x{:}A_1.A_2}\ \textsc{Objeq-Abs}$$

$\Psi;\Gamma \vdash A_{11} : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash A_{21} : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash A_1 : \mathsf{type}$ By induction
$\Psi;\Gamma, x{:}A_1 \vdash M_1 : A_2$,
$\Psi;\Gamma, x{:}A_1 \vdash A_2 : A_2$,
$\Psi;\Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$ By induction
$\Psi;\Gamma, x{:}A_1 \vdash A_2 \equiv A_2 : \mathsf{type}$ By Reflexivity
$\Psi;\Gamma \vdash \Pi x{:}A_1.A_2 : \mathsf{type}$ By rule
$\Psi;\Gamma \vdash \Pi x{:}A_1.A_2 \equiv \Pi x{:}A_{11}.A_2 : \mathsf{type}$ By rule
$\Psi;\Gamma \vdash \Pi x{:}A_1.A_2 \equiv \Pi x{:}A_{21}.A_2 : \mathsf{type}$ By rule
$\Psi;\Gamma, x{:}A_{11} \vdash M_1 : A_2$ By Context Conversion
$\Psi;\Gamma \vdash \lambda x{:}A_{11}.M_1 : \Pi x{:}A_{11}.A_2$ By rule (abstraction)
$\Psi;\Gamma_1 \vdash \lambda x{:}A_{11}.M_1 : \Pi x{:}A_1.A_2$ By rule (type conversion)
$\Psi;\Gamma, x{:}A_{21} \vdash M_2 : A_2$ By Context Conversion
$\Psi;\Gamma_1 \vdash \lambda x{:}A_{21}.M_2 : \Pi x{:}A_{21}.A_2$ By rule (abstraction)
$\Psi;\Gamma_1 \vdash \lambda x{:}A_{21}.M_2 : \Pi x{:}A_1.A_2$ By rule (type conversion)

Case 4:
$$\frac{\Psi;\Gamma \vdash A_1 : \mathsf{type} \qquad \Psi;\Gamma, x{:}A_1 \vdash M_{12} \equiv M_{22} : A_2 \qquad \Psi;\Gamma \vdash M_{11} \equiv M_{21} : A_1}{\Psi;\Gamma \vdash (\lambda x{:}A_1.M_{12})\, M_{11} \equiv [M_{21}/x]\, M_{22} : [M_{11}/x]\, A_2}\ \textsc{Objeq-BetaPi}$$

$\Psi;\Gamma, x{:}A_1 \vdash M_{12} : A_2$,
$\Psi;\Gamma, x{:}A_1 \vdash M_{22} : A_2$,
$\Psi;\Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash M_{11} : A_1$,
$\Psi;\Gamma \vdash M_{21} : A_1$,
$\Psi;\Gamma \vdash A_1 : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash [M_{11}/x]\, A_2 : \mathsf{type}$ By Substitution
$\Psi;\Gamma \vdash [M_{11}/x]\, A_2 \equiv [M_{21}/x]\, A_2 : \mathsf{type}$ By Functionality
$\Psi;\Gamma \vdash [M_{21}/x]\, M_{22} : [M_{21}/x]\, A_2$ By Substitution
$\Psi;\Gamma \vdash [M_{21}/x]\, M_{22} : [M_{11}/x]\, A_2$ By rule (type conversion)
$\Psi;\Gamma \vdash \lambda x{:}A_1.M_{12} : \Pi x{:}A_1.A_2$ By rule
$\Psi;\Gamma \vdash (\lambda x{:}A_1.M_{12})\, M_{11} : [M_{11}/x]\, A_2$ By rule

Case 5:
$$\frac{\begin{array}{c}\Psi;\Gamma \vdash \Sigma x{:}A_1.A_2 : \mathsf{type} \\ \Psi;\Gamma \vdash M_{11} \equiv M_{21} : A_1 \\ \Psi;\Gamma \vdash M_{12} \equiv M_{22} : [M_{11}/x]\, A_2\end{array}}{\Psi;\Gamma \vdash \langle M_{11}, M_{12}\rangle^{\Sigma x:A_1.A_2} \equiv \langle M_{21}, M_{22}\rangle^{\Sigma x:A_1.A_2} : \Sigma x{:}A_1.A_2}\ \textsc{Objeq-Pair}$$

$\Psi;\Gamma \vdash A_1 : \mathsf{type}$,
$\Psi;\Gamma, x{:}A_1 \vdash A_2 : \mathsf{type}$ By Inversion on Sums

$\Psi;\Gamma \vdash \mathsf{M}_{11} : \mathsf{A}_1$,
$\Psi;\Gamma \vdash \mathsf{M}_{21} : \mathsf{A}_1$ — By induction
$\Psi;\Gamma \vdash [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2 \equiv [\mathsf{M}_{21}/\mathsf{x}]\,\mathsf{A}_2 : \mathsf{type}$ — By Functionality
$\Psi;\Gamma \vdash \mathsf{M}_{12} : [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2$,
$\Psi;\Gamma \vdash \mathsf{M}_{22} : [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2$ — By induction
$\Psi;\Gamma \vdash \mathsf{M}_{22} : [\mathsf{M}_{21}/\mathsf{x}]\,\mathsf{A}_2$ — By rule (type conversion)
$\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$ — By assumption
$\Psi;\Gamma \vdash \langle \mathsf{M}_{11}, \mathsf{M}_{12}\rangle^{\Sigma\mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$ — By rule
$\Psi;\Gamma \vdash \langle \mathsf{M}_{21}, \mathsf{M}_{22}\rangle^{\Sigma\mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$ — By rule

Case 6:
$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_3 : \mathsf{A}_2}{\Psi;\Gamma \vdash \pi_2\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} \equiv \mathsf{M}_3 : \mathsf{A}_2}\ \textsc{Objeq-BetaProj2}$$

$\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2$,
$\Psi;\Gamma \vdash \mathsf{M}_3 : \mathsf{A}_2$,
$\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{type}$ — By induction
$\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type}$ — By induction
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}$ — By Weakening
$\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$ — By rule
$\Psi;\Gamma \vdash \mathsf{M}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2$ — Since $\mathsf{x} \notin FV(\mathsf{A}_2)$
$\Psi;\Gamma \vdash \langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$ — By rule (pair)
$\Psi;\Gamma \vdash \pi_2\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} : [\pi_1\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2$ — By rule (projection)
$\Psi;\Gamma \vdash \pi_2\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} : \mathsf{A}_2$ — Since $\mathsf{x} \notin FV(\mathsf{A}_2)$

□

With regularity now proved, we can prove two important lemmas used many times in the later sections. One is a generalization of the functionality property to the equality judgments. The other is an inversion on typing judgments.

Lemma 6.4.11 (Functionality for Equality) *Assume* $\Gamma_1 \vdash \sigma_1 = \sigma_2 : \Gamma$.

1. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$ *then* $\Psi;\Gamma_1 \vdash \mathsf{M}_1[\sigma_1] \equiv \mathsf{M}_2[\sigma_2] : \mathsf{A}[\sigma_1]$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$ *then* $\Psi;\Gamma_1 \vdash \mathsf{A}_1[\sigma_1] \equiv \mathsf{A}_2[\sigma_2] : \mathsf{K}[\sigma_1]$.
3. *If* $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$ *then* $\Psi;\Gamma_1 \vdash \mathsf{K}_1[\sigma_1] \equiv \mathsf{K}_2[\sigma_2] : \mathsf{kind}$.

Proof

We show one case.

$\Psi;\Gamma_1 \vdash \sigma_1 : \Gamma$,
$\Psi;\Gamma_1 \vdash \sigma_2 : \Gamma$ — By definition of substitution equality

$\Psi;\Gamma_1 \vdash \mathsf{M}_1[\sigma_1] \equiv \mathsf{M}_2[\sigma_1] : \mathsf{A}[\sigma_1]$ — By Substitution
$\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$ — By Regularity
$\Psi;\Gamma_1 \vdash \mathsf{M}_2[\sigma_1] \equiv \mathsf{M}_2[\sigma_2] : \mathsf{A}[\sigma_1]$ — By Functionality
$\Psi;\Gamma_1 \vdash \mathsf{M}_1[\sigma_1] \equiv \mathsf{M}_2[\sigma_2] : \mathsf{A}[\sigma_1]$ — By rule (transitivity)

□

Lemma 6.4.12 (Typing Inversion) *Assume* $\Psi \vdash \Gamma : \mathsf{ctx}$.

1. *If* $\Psi;\Gamma \vdash \mathsf{x} : \mathsf{A}_1$ *then* $\Gamma(\mathsf{x}) = \mathsf{A}_2$ *and* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{c} : \mathsf{A}_1$ *then* $\Sigma(\mathsf{c}) = \mathsf{A}_2$ *and* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$.
3. *If* $\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_1 : \mathsf{A}$ *then* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type}$ *and* $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1 : \mathsf{A}_2$ *and* $\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \equiv \mathsf{A} : \mathsf{type}$.

4. *If* $\Psi;\Gamma \vdash \mathsf{M}_1\,\mathsf{M}_2 : \mathsf{A}$ *then* $\Psi;\Gamma \vdash \mathsf{M}_1 : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$, $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_1$ *and* $\Psi;\Gamma \vdash [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_2 \equiv \mathsf{A} : \mathsf{type}$.
5. *If* $\Psi;\Gamma \vdash \langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\Sigma\mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \mathsf{A}$ *then* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1$, $\Psi;\Gamma \vdash \mathsf{M}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2$ *and* $\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \equiv \mathsf{A} : \mathsf{type}$.
6. *If* $\Psi;\Gamma \vdash \pi_1\mathsf{M}_1 : \mathsf{A}$ *then* $\Psi;\Gamma \vdash \mathsf{M}_1 : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$ *and* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A} : \mathsf{type}$.
7. *If* $\Psi;\Gamma \vdash \pi_2\mathsf{M}_1 : \mathsf{A}$ *then* $\Psi;\Gamma \vdash \mathsf{M}_1 : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$ *and* $\Psi;\Gamma \vdash [\pi_1\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2 \equiv \mathsf{A} : \mathsf{type}$.
8. *If* $\Psi;\Gamma \vdash \langle\rangle : \mathsf{A}$ *then* $\Psi;\Gamma \vdash 1 \equiv \mathsf{A} : \mathsf{type}$.
9. *If* $\Psi;\Gamma \vdash \mathsf{a} : \mathsf{K}_1$ *then* $\Sigma(\mathsf{a}) = \mathsf{K}_2$ *and* $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$.
10. *If* $\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{K}$ *then* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type}$ *and* $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{K}_1$ *and* $\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{K}_1 \equiv \mathsf{K} : \mathsf{kind}$.
11. *If* $\Psi;\Gamma \vdash \mathsf{A}\,\mathsf{M} : \mathsf{K}$ *then* $\Psi;\Gamma \vdash \mathsf{A} : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{K}_1$, $\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}_1$ *and* $\Psi;\Gamma \vdash [\mathsf{M}/\mathsf{x}]\,\mathsf{K}_1 \equiv \mathsf{K} : \mathsf{kind}$.
12. *If* $\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{K}$ *then* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type}$, $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}$ *and* $\Psi;\Gamma \vdash \mathsf{type} \equiv \mathsf{K} : \mathsf{kind}$.
13. *If* $\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{K}$ *then* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type}$, $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}$ *and* $\Psi;\Gamma \vdash \mathsf{type} \equiv \mathsf{K} : \mathsf{kind}$.
14. *If* $\Psi;\Gamma \vdash 1 : \mathsf{K}$ *then* $\Psi;\Gamma \vdash \mathsf{type} \equiv \mathsf{K} : \mathsf{kind}$.

Proof

By structural induction on the derivation of the judgment. We show a few representative cases.

Case 1:
$$\frac{\Psi;\Gamma \vdash \mathsf{M} : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_2\mathsf{M} : [\pi_1\mathsf{M}/\mathsf{x}]\,\mathsf{A}_2}\ \text{O-Proj2}$$

$\Psi;\Gamma \vdash [\pi_1\mathsf{M}/\mathsf{x}]\,\mathsf{A}_2 : \mathsf{type}$ By Regularity
$\Psi;\Gamma \vdash [\pi_1\mathsf{M}/\mathsf{x}]\,\mathsf{A}_2 \equiv [\pi_1\mathsf{M}/\mathsf{x}]\,\mathsf{A}_2 : \mathsf{type}$ By Reflexivity

Case 2:
$$\frac{\Psi;\Gamma \vdash \langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}}{\Psi;\Gamma \vdash \langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} : \mathsf{A}_2}\ \text{O-FamEq}$$

$\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_3$,
$\Psi;\Gamma \vdash \mathsf{M}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_4$,
$\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_3.\mathsf{A}_4 \equiv \mathsf{A}_1 : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_3.\mathsf{A}_4 \equiv \mathsf{A}_2 : \mathsf{type}$ By rule (symmetry)

Case 3:
$$\frac{\Psi;\Gamma \vdash \langle\rangle : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}}{\Psi;\Gamma \vdash \langle\rangle : \mathsf{A}_2}\ \text{O-FamEq}$$

$\Psi;\Gamma \vdash \langle\rangle : 1$,
$\Psi;\Gamma \vdash 1 \equiv \mathsf{A}_1 : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash 1 \equiv \mathsf{A}_2 : \mathsf{type}$ By rule (transitivity)

□

Lemma 6.4.13 (Extending Equal Substitutions) *If* $\Psi \vdash \Gamma : \mathsf{ctx}$, $\Gamma_1 \vdash \sigma_1 = \sigma_2 : \Gamma$, $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$ *and* $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$ *and* $\mathsf{x} \notin \mathsf{dom}(\Psi)\Gamma$ *then* $\Gamma_1 \vdash \sigma_1, \mathsf{M}_1/\mathsf{x} = \sigma_2, \mathsf{M}_2/\mathsf{x} : \Gamma, \mathsf{x}{:}\mathsf{A}$.

Proof

Direct, by Regularity and Functionality for Equality.

□

Lemma 6.4.14 (Equality Inversion on Kinds)

1. *If* $\Psi;\Gamma \vdash \mathsf{K} \equiv \mathsf{type} : \mathsf{kind}$ *or* $\Psi;\Gamma \vdash \mathsf{type} \equiv \mathsf{K} : \mathsf{kind}$ *then* $\mathsf{K} = \mathsf{type}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{K}' \equiv \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{K} : \mathsf{kind}$ *or* $\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{K} \equiv \mathsf{K}' : \mathsf{kind}$ *then* $\mathsf{K}' = \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{K}_1$ *with* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A} : \mathsf{type}$ *and* $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K} : \mathsf{kind}$.

Proof

By a straightforward induction on the structure of the derivations. □

$$\frac{}{(\lambda x{:}A_1.M_2)\,M_1 \xrightarrow{\text{whr}} [M_1/x]\,M_2}\ \textsc{Wobj-Beta} \qquad \frac{M_1 \xrightarrow{\text{whr}} M_2}{M_1\,M \xrightarrow{\text{whr}} M_2\,M}\ \textsc{Wobj-Congleft}$$

$$\frac{}{\pi_i\langle M_1, M_2\rangle^{A} \xrightarrow{\text{whr}} M_i}\ \textsc{Wobj-Rbeta} \qquad \frac{M_1 \xrightarrow{\text{whr}} M_2}{\pi_i M_1 \xrightarrow{\text{whr}} \pi_i M_2}\ \textsc{Wobj-CongPair}$$

$$\frac{}{(\lambda x{:}A_1.A_2)\,M \xrightarrow{\text{whr}} [M/x]\,A_2}\ \textsc{Wfam-Beta} \qquad \frac{A_1 \xrightarrow{\text{whr}} A_2}{A_1\,M \xrightarrow{\text{whr}} A_2\,M}\ \textsc{Wfam-Congleft}$$

Figure 6.14: Weak Head Reduction

6.5 Injectivity

We need a property usually called injectivity. Briefly, this says that given equal dependent function types, the components of the product types are themselves equal (at the kind type). A similar property should hold for dependent pair types. In Harper and Pfenning [HP00], this follows by an easy induction analogous to the case for dependent function kinds (Lemma 6.4.14). However, we now have a nontrivial equality relation at the level of types due to the presence of family-level abstractions and applications. The problem in an inductive proof comes with the use of the transitivity rule. While judging equality between two products (say), the mediating family need not be of the same form. To get around this, we follow Vanderwaart and Crary [VC02] and prove by the method of logical relations. In defining the logical relation we will need, we need the auxiliary notion of weak-head reduction. This notion is important in its own right, and will be reused in the definition of the equality algorithm.

6.5.1 Weak Head Reduction

Weak head reduction for objects and type families is defined in figure 6.14. This applies the beta rules for lambda abstractions and applications, as well as the one for projections, but only if the beta-redex is at the root of the term.

The reduction is obviously determinate, in that a reducible term can have just one reduct.

Lemma 6.5.1 (Determinacy of Weak Head Reduction)

1. *If* $M_1 \xrightarrow{\text{whr}} M_2$ *and* $M_1 \xrightarrow{\text{whr}} M_3$ *then* $M_2 = M_3$.
2. *If* $A_1 \xrightarrow{\text{whr}} A_2$ *and* $A_1 \xrightarrow{\text{whr}} A_3$ *then* $A_2 = A_3$.

Proof

By case analysis on the derivation of the first judgment.

□

We write $\xrightarrow{\text{whr}}{}^{*}$ for the reflexive transitive closure of the weak-head reduction relation $\xrightarrow{\text{whr}}$. On occasion, we write $M \not\xrightarrow{\text{whr}}$ to indicate that there exists no M_1 such that $M \xrightarrow{\text{whr}} M_1$ according to the rules set out here. An analogous notation $A \not\xrightarrow{\text{whr}}$ is defined for families.

6.5.2 A Logical Relation

The logical relation we use is defined by induction on kinds. The notable feature is that at higher kinds, we require definitional equality instead of logical relatedness for the arguments. Since the structure of kinds (*i.e.* whether they are a dependent function kind or type) is not affected by substitution of object variables, the definition of the relation is well-founded.

- $\Psi;\Gamma \vdash A_1 \approx A_2 : [\![\mathsf{type}]\!]$ iff all these hold:
  - $A_1 \xrightarrow{\text{whr}}{}^* \Pi x{:}A_{11}.A_{12}$ iff $A_2 \xrightarrow{\text{whr}}{}^* \Pi x{:}A_{21}.A_{22}$.
  - $A_1 \xrightarrow{\text{whr}}{}^* \Sigma x{:}A_{11}.A_{12}$ iff $A_2 \xrightarrow{\text{whr}}{}^* \Sigma x{:}A_{21}.A_{22}$.
  - If $A_1 \xrightarrow{\text{whr}}{}^* \Pi x{:}A_{11}.A_{12}$ and $A_2 \xrightarrow{\text{whr}}{}^* \Pi x{:}A_{21}.A_{22}$ then $\Psi;\Gamma \vdash A_{11} \equiv A_{21} : \mathsf{type}$ and $\Psi;\Gamma, x{:}A_{11} \vdash A_{12} \equiv A_{22} : \mathsf{type}$.
  - If $A_1 \xrightarrow{\text{whr}}{}^* \Sigma x{:}A_{11}.A_{12}$ and $A_2 \xrightarrow{\text{whr}}{}^* \Sigma x{:}A_{21}.A_{22}$ then $\Psi;\Gamma \vdash A_{11} \equiv A_{21} : \mathsf{type}$ and $\Psi;\Gamma, x{:}A_{11} \vdash A_{12} \equiv A_{22} : \mathsf{type}$.
- $\Psi;\Gamma \vdash A_1 \approx A_2 : [\![\Pi x{:}A.K]\!]$ iff for every pair of objects M_1 and M_2 such that $\Psi;\Gamma \vdash M_1 \equiv M_2 : A$ we have $\Psi;\Gamma \vdash A_1\, M_1 \approx A_2\, M_2 : [\![[M_1/x]\, K]\!]$.
- $\Psi;\Gamma \vdash K_1 \approx K_2 : [\![\mathsf{kind}]\!]$ iff for every pair of families A_1 and A_2, $\Psi;\Gamma \vdash A_1 \approx A_2 : [\![K_1]\!]$ iff $\Psi;\Gamma \vdash A_1 \approx A_2 : [\![K_2]\!]$.

The fundamental theorem of Logical Relations is that definitionally equal type families and kinds are logically related under all substitutions. To handle the symmetry and transitivity cases we will need that the logical relation itself is symmetric and transitive. To handle the abstraction case, we will need that the logical relation is closed under weak head expansion. To handle the family-level constant case, we will need that paths are logically related to each other. We now prove these standard lemmas before setting out to prove the fundamental lemma.

Lemma 6.5.2 (Paths are Logically Related)
If $\Sigma(a) = \Pi x_1{:}A_1 \ldots \Pi x_k{:}A_k.K$ *($k \geq 0$) and* $\Psi;\Gamma \vdash M_i \equiv M_i' : [M_1 \ldots M_{i-1}/x_1 \ldots x_{i-1}]\, A_i$ *for all* i*,*
then $\Psi;\Gamma \vdash a\, M_1 \ldots M_i \approx a\, M_1' \ldots M_i' : [\![[M_1 \ldots M_i/x_1 \ldots x_i]\, K]\!]$.

Proof

By induction on the size of the kind K.

Case 1: $K = \mathsf{type}$

$a\, M_1 \ldots M_i \not\xrightarrow{\text{whr}}$	By inspection of rules
$\Psi;\Gamma \vdash a\, M_1 \ldots M_i \approx a\, M_1' \ldots M_i' : [\![\mathsf{type}]\!]$	By definition of relation

Case 2: $K = \Pi x{:}A.K_1$

$\Psi;\Gamma \vdash M \equiv M' : A$	New assumption
$\Psi;\Gamma \vdash a\, M_1 \ldots M_i M \approx a\, M_1' \ldots M_i' M' : [\![[M_1 \ldots M_i M/x_1 \ldots x_i x]\, K_1]\!]$	By induction
$\Psi;\Gamma \vdash a\, M_1 \ldots M_i \approx a\, M_1' \ldots M_i' : [\![\Pi x{:}A.([M_1 \ldots M_i/x_1 \ldots x_i]\, K_1)]\!]$	By definition of relation
$\Psi;\Gamma \vdash a\, M_1 \ldots M_i \approx a\, M_1' \ldots M_i' : [\![[M_1 \ldots M_i/x_1 \ldots x_i]\, (\Pi x{:}A.K_1)]\!]$	By properties of substitution

□

Lemma 6.5.3 (Closure Under Weak Head Expansion) *If we have* $\Psi;\Gamma \vdash A_{11} \approx A_{21} : [\![K]\!]$, $\Psi;\Gamma \vdash A_{12} : K$, $\Psi;\Gamma \vdash A_{22} : K$, $A_{12} \xrightarrow{\text{whr}}{}^* A_{11}$ *and* $A_{22} \xrightarrow{\text{whr}}{}^* A_{21}$ *then* $\Psi;\Gamma \vdash A_{12} \approx A_{22} : [\![K]\!]$.

Proof

By induction on the size of K. □

We shall need a small fact about substituting logically related families into kinds for proving symmetry of the logical relation.

Lemma 6.5.4 (Equivalent Substitutions of a Valid Kind Related) *If* $\Psi \vdash \Gamma : \mathsf{ctx}$, $\Psi;\Gamma \vdash K : \mathsf{kind}$, $\Gamma_1 \vdash \sigma_1 = \sigma_2 : \Gamma$ *and* $\Psi;\Gamma_1 \vdash A_1 \approx A_2 : [\![K[\sigma_1]]\!]$ *then* $\Psi;\Gamma_1 \vdash A_1 \approx A_2 : [\![K[\sigma_2]]\!]$.

Proof

By induction on the size of K. □

We need symmetry and transitivity of the relation, for the main theorem.

Lemma 6.5.5 (Symmetry of Logical Relation) *If* $\Psi \vdash \Gamma : \mathsf{ctx}$, $\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}$, *and* $\Psi;\Gamma \vdash \mathsf{A}_1 \approx \mathsf{A}_2 : [\![\mathsf{K}]\!]$ *then*
$\Psi;\Gamma \vdash \mathsf{A}_2 \approx \mathsf{A}_1 : [\![\mathsf{K}]\!]$.

Proof

By induction on the size of K. We show only the product kind case.

Case $\mathsf{K} = \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K}_1$

$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$	New assumption
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_1 : \mathsf{A}$	By rule (symmetry)
$\Psi;\Gamma \vdash \mathsf{A}_1\,\mathsf{M}_2 \approx \mathsf{A}_2\,\mathsf{M}_1 : [\![[\mathsf{M}_2/\mathsf{x}]\,\mathsf{K}_1]\!]$	By definition of relation
$\Psi;\Gamma \vdash \mathsf{A}_2\,\mathsf{M}_1 \approx \mathsf{A}_1\,\mathsf{M}_2 : [\![[\mathsf{M}_2/\mathsf{x}]\,\mathsf{K}_1]\!]$	By induction
$\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$	By Inversion
$\Psi \vdash \Gamma, \mathsf{x}{:}\mathsf{A} : \mathsf{ctx}$	By rule
$\Gamma \vdash \mathsf{id}_\Gamma = \mathsf{id}_\Gamma : \Gamma$	By reflexivity of substitution equality
$\Gamma \vdash \mathsf{id}_\Gamma, \mathsf{M}_2/\mathsf{x} = \mathsf{id}_\Gamma, \mathsf{M}_1/\mathsf{x} : \Gamma, \mathsf{x}{:}\mathsf{A}$	By Extending Substitutions by Equal Elements
$\Psi;\Gamma \vdash \mathsf{A}_1\,\mathsf{M}_2 \approx \mathsf{A}_2\,\mathsf{M}_1 : [\![[\mathsf{M}_1/\mathsf{x}]\,\mathsf{K}_1]\!]$	By Equivalent Substitutions are Related
$\Psi;\Gamma \vdash \mathsf{A}_2 \approx \mathsf{A}_1 : [\![\Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K}_1]\!]$	By definition of relation

□

Lemma 6.5.6 (Transitivity of Logical Relation) *If* $\Psi \vdash \Gamma : \mathsf{ctx}$, $\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}$, $\Psi;\Gamma \vdash \mathsf{A}_1 \approx \mathsf{A}_2 : [\![\mathsf{K}]\!]$, *and* $\Psi;\Gamma \vdash \mathsf{A}_2 \approx \mathsf{A}_3 : [\![\mathsf{K}]\!]$ *then* $\Psi;\Gamma \vdash \mathsf{A}_1 \approx \mathsf{A}_3 : [\![\mathsf{K}]\!]$.

Proof

By induction on the size of K.

□

We are now in a position to prove the fundamental lemma of logical relations. This says that type families (and kinds) that are judged to be equal are related to each other by the logical relation. The significance of this relation is that we know by definition of the relation that we have injectivity at dependent product and sum types. The reason we have to go through the logical relation is that in the case of the beta-conversion rule, we need a stronger induction hypothesis. The logical relation packages up the hypothesis we need.

Lemma 6.5.7 (Definitionally Equal Terms are Logically Related Under Subsitutions)

1. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$ *and* $\Gamma_1 \vdash \sigma_1 = \sigma_2 : \Gamma$ *and* $\Psi \vdash \Gamma : \mathsf{ctx}$ *then* $\Psi;\Gamma_1 \vdash \mathsf{A}_1[\sigma_1] \approx \mathsf{A}_2[\sigma_2] : [\![\mathsf{K}[\sigma_1]]\!]$.
2. *If* $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$ *and* $\Gamma_1 \vdash \sigma_1 = \sigma_2 : \Gamma$ *and* $\Psi \vdash \Gamma : \mathsf{ctx}$ *then* $\Psi;\Gamma_1 \vdash \mathsf{K}_1[\sigma_1] \approx \mathsf{K}_2[\sigma_2] : [\![\mathsf{kind}]\!]$.

Proof

By induction on derivations. We show some of the more interesting cases.

Case 1:
$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{a} \equiv \mathsf{a} : \mathsf{K}}\ \textsc{Fameq-Const}$$

$\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind}$	By definition of valid signature
$FV(\mathsf{K}) = \cdot$	By Free Variable Containment
$\Psi;\Gamma_1 \vdash \mathsf{a} \approx \mathsf{a} : [\![\mathsf{K}]\!]$	By Logically Related Paths
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{a} \approx [\sigma_2]\mathsf{a} : [\![[\sigma_1]\mathsf{K}]\!]$	By definition of substitution

Case 2:
$$\frac{\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_1 : \mathsf{type} \quad \Psi;\Gamma \vdash \mathsf{A}_{21} \equiv \mathsf{A}_1 : \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{K}}{\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \lambda \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{K}}\ \textsc{Fameq-Abs}$$

$\Psi;\Gamma_1 \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : [\sigma_1]\mathsf{A}_1$	New hypothesis
$\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type}$	By Regularity
$\Psi \vdash \Gamma, \mathsf{x}{:}\mathsf{A}_1 : \mathsf{ctx}$	By rule
$\Psi;\Gamma_1 \vdash \sigma_1, \mathsf{M}_1/\mathsf{x} \equiv \sigma_2, \mathsf{M}_2/\mathsf{x} : \Gamma, \mathsf{x}{:}\mathsf{A}$	By Extending Substitutions by Equal Elements

$\Psi;\Gamma_1 \vdash [\sigma_1, \mathsf{M}_1/\mathsf{x}]\mathsf{A}_{12} \approx [\sigma_2, \mathsf{M}_2/\mathsf{x}]\mathsf{A}_{22} : [\![[\sigma_1, \mathsf{M}_1/\mathsf{x}]\mathsf{K}]\!]$ By induction
$\Psi;\Gamma_1 \vdash [\sigma_1](\mathsf{A}_{12}\,\mathsf{M}_1) \approx [\sigma_2](\mathsf{A}_{22}\,\mathsf{M}_2) : [\![[\sigma_1]([\mathsf{M}_1/\mathsf{x}]\mathsf{K})]\!]$ From previous
$\Psi;\Gamma_1 \vdash (\lambda \mathsf{x}{:}[\sigma_1]\mathsf{A}_{11}.[\sigma_1]\mathsf{A}_{12})\,\mathsf{M}_1 \approx (\lambda \mathsf{x}{:}[\sigma_2]\mathsf{A}_{21}.[\sigma_2]\mathsf{A}_{22})\,\mathsf{M}_2 : [\![[\sigma_1]([\mathsf{M}_1/\mathsf{x}]\mathsf{K})]\!]$ By Closure Under Expansion
$\Psi;\Gamma_1 \vdash \lambda \mathsf{x}{:}[\sigma_1]\mathsf{A}_{11}.[\sigma_1]\mathsf{A}_{12} \approx \lambda \mathsf{x}{:}[\sigma_2]\mathsf{A}_{21}.[\sigma_2]\mathsf{A}_{22} : [\![\Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K}]\!]$ By definition of relation

Case 3:
$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \Pi \mathsf{x}{:}\mathsf{A}_3.\mathsf{K} \qquad \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_3}{\Psi;\Gamma \vdash \mathsf{A}_1\,\mathsf{M}_1 \equiv \mathsf{A}_2\,\mathsf{M}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{K}}\ \textsc{Fameq-App}$$

$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_1 \approx [\sigma_2]\mathsf{A}_2 : [\![\Pi \mathsf{x}{:}[\sigma_1]\mathsf{A}_3.[\sigma_1]\mathsf{K}]\!]$ By induction
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{M}_1 \equiv [\sigma_2]\mathsf{M}_2 : [\sigma_1]\mathsf{A}_3$ By Functionality of Equality
$\Psi;\Gamma_1 \vdash ([\sigma_1]\mathsf{A}_1)\,([\sigma_1]\mathsf{M}_1) \approx ([\sigma_2]\mathsf{A}_2)\,([\sigma_2]\mathsf{M}_2) : [\![[\sigma_1]([\mathsf{M}_1[\sigma_1]/\mathsf{x}]\,\mathsf{K})]\!]$ By definition of relation
$\Psi;\Gamma_1 \vdash [\sigma_1](\mathsf{A}_1\,\mathsf{M}_1) \approx [\sigma_2](\mathsf{A}_2\,\mathsf{M}_2) : [\![[\sigma_1]([\mathsf{M}_1[\sigma_1]/\mathsf{x}]\,\mathsf{K})]\!]$ By definition of substitution

Case 4:
$$\frac{\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type} \qquad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K} \qquad \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}}{\Psi;\Gamma \vdash (\lambda \mathsf{x}{:}\mathsf{A}.\mathsf{A}_1)\,\mathsf{M}_1 \equiv [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{K}}\ \textsc{Fameq-Beta}$$

$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{M}_1 \equiv [\sigma_2]\mathsf{M}_2 : [\sigma_1]\mathsf{A}$ By Functionality of Equality
$\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$ By assumption
$\Psi \vdash \Gamma, \mathsf{x}{:}\mathsf{A} : \mathsf{ctx}$ By rule
$\Psi;\Gamma_1 \vdash \sigma_1, [\sigma_1]\mathsf{M}_1/\mathsf{x} \equiv \sigma_2, [\sigma_2]\mathsf{M}_2/\mathsf{x} : \Gamma, \mathsf{x}{:}\mathsf{A}$ By Extending Substitutions by Equal Elements
$\Psi;\Gamma_1 \vdash [\sigma_1]\,[[\sigma_1]\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_1 \approx [\sigma_2]\,[[\sigma_2]\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_2 : [\![[\sigma_1]\,[[\sigma_1]\mathsf{M}_1/\mathsf{x}]\,\mathsf{K}]\!]$ By induction
$\Psi;\Gamma_1 \vdash [[\sigma_1]\mathsf{M}_1/\mathsf{x}]\,([\sigma_1]\mathsf{A}_1) \approx [[\sigma_2]\mathsf{M}_2/\mathsf{x}]\,([\sigma_2]\mathsf{A}_2) : [\![[[\sigma_1]\mathsf{M}_1/\mathsf{x}]\,([\sigma_1]\mathsf{K})]\!]$ From previous
$\Psi;\Gamma_1 \vdash (\lambda \mathsf{x}{:}[\sigma_1]\mathsf{A}.[\sigma_1]\mathsf{A}_1)\,[\sigma_1]\mathsf{M}_1 \approx [[\sigma_2]\mathsf{M}_2/\mathsf{x}]\,([\sigma_2]\mathsf{A}_2) : [\![[[\sigma_1]\mathsf{M}_1/\mathsf{x}]\,([\sigma_1]\mathsf{K})]\!]$
By Closure Under Expansion

Case 5:
$$\frac{\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type} \qquad \Psi;\Gamma \vdash \mathsf{A}_{11} : \mathsf{type} \qquad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{type}}{\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}}\ \textsc{Fameq-Sigma}$$

$\Sigma \mathsf{x}{:}[\sigma_1]\mathsf{A}_{11}.[\sigma_1]\mathsf{A}_{12} \xrightarrow{\text{whr}} \mathsf{A}_1$ implies $\mathsf{A}_1 = \Sigma \mathsf{x}{:}[\sigma_1]\mathsf{A}_{11}.[\sigma_1]\mathsf{A}_{12}$ By definition of reduction
$\Sigma \mathsf{x}{:}[\sigma_2]\mathsf{A}_{21}.[\sigma_2]\mathsf{A}_{22} \xrightarrow{\text{whr}} \mathsf{A}_2$ implies $\mathsf{A}_2 = \Sigma \mathsf{x}{:}[\sigma_2]\mathsf{A}_{21}.[\sigma_2]\mathsf{A}_{22}$ By definition of reduction
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_{11} \equiv [\sigma_2]\mathsf{A}_{21} : \mathsf{type}$ By Functionality of Equality
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_{11} : \mathsf{type}$ By Regularity
$\Psi \vdash \Gamma_1, \mathsf{x}{:}[\sigma_1]\mathsf{A}_{11} : \mathsf{ctx}$ By rule
$\Psi;\Gamma \vdash \mathsf{A}_{11} : \mathsf{type}$ By Regularity
$\Psi \vdash \Gamma, \mathsf{x}{:}\mathsf{A}_{11} : \mathsf{ctx}$ By rule
$\Psi;\Gamma_1, \mathsf{x}{:}[\sigma_1]\mathsf{A}_{11} \vdash \sigma_1, \mathsf{x}/\mathsf{x} \equiv \sigma_2, \mathsf{x}/\mathsf{x} : \Gamma, \mathsf{x}{:}\mathsf{A}_{11}$ By Extending Substitutions
$\Psi;\Gamma_1, \mathsf{x}{:}\mathsf{A}_{11} \vdash [\sigma_1, \mathsf{x}/\mathsf{x}]\mathsf{A}_{12} \equiv [\sigma_2, \mathsf{x}/\mathsf{x}]\mathsf{A}_{22} : \mathsf{type}$ By Functionality
$\Psi;\Gamma_1, \mathsf{x}{:}\mathsf{A}_{11} \vdash [\sigma_1]\mathsf{A}_{12} \equiv [\sigma_2]\mathsf{A}_{22} : \mathsf{type}$ From previous
$\Psi;\Gamma_1 \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \approx \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : [\![\mathsf{type}]\!]$ By definition of relation

Case 6:
$$\frac{\Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_1 : \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}}\ \textsc{Fameq-Symm}$$

$\Gamma_1 \vdash \sigma_2 = \sigma_1 : \Gamma$ By symmetry of substitution equality
$\Psi;\Gamma_1 \vdash [\sigma_2]\mathsf{A}_2 \approx [\sigma_1]\mathsf{A}_1 : [\![[\sigma_2]\mathsf{K}]\!]$ By induction
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_1 \approx [\sigma_2]\mathsf{A}_2 : [\![[\sigma_2]\mathsf{K}]\!]$ By Symmetry of Relation
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_1 \approx [\sigma_2]\mathsf{A}_2 : [\![[\sigma_1]\mathsf{K}]\!]$ By Relation of Equal Substitutions

Case 7:
$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K} \qquad \Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_3 : \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_3 : \mathsf{K}}\ \textsc{Fameq-Trans}$$

$\Psi;\Gamma_1 \vdash \sigma_2 \equiv \sigma_1 : \Gamma$	By symmetry of substitution equality
$\Psi;\Gamma_1 \vdash \sigma_1 \equiv \sigma_1 : \Gamma$	By transitivity of substitution equality
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_1 \approx [\sigma_1]\mathsf{A}_2 : [\![[\sigma_1]\mathsf{K}]\!]$	By induction
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_2 \approx [\sigma_2]\mathsf{A}_3 : [\![[\sigma_1]\mathsf{K}]\!]$	By induction
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_1 \approx [\sigma_2]\mathsf{A}_3 : [\![[\sigma_1]\mathsf{K}]\!]$	By Transitivity of Relation

Case 8:
$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \qquad \Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}}{\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{K}_1 \equiv \Pi\mathsf{x}{:}\mathsf{A}_2.\mathsf{K}_2 : \mathsf{kind}}\ \textsc{Keq-Pi}$$

The definition of relation at product kinds involves checking a bi-implication. We show one direction, the other is similar.

$\Psi;\Gamma_1 \vdash \mathsf{A}_3 \approx \mathsf{A}_4 : [\![\Pi\mathsf{x}{:}[\sigma_2]\mathsf{A}_2.[\sigma_2]\mathsf{K}_2]\!]$	New hypothesis
$\Psi;\Gamma_1 \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : [\sigma_1]\mathsf{A}_1$	New hypothesis
$\Psi;\Gamma_1 \vdash [\sigma_1]\mathsf{A}_1 \equiv [\sigma_2]\mathsf{A}_2 : \mathsf{type}$	By Functionality of Equality
$\Psi;\Gamma_1 \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : [\sigma_2]\mathsf{A}_2$	By rule (type conversion)
$\Psi;\Gamma_1 \vdash \mathsf{A}_3\,\mathsf{M}_1 \approx \mathsf{A}_4\,\mathsf{M}_2 : [\![[\mathsf{M}_1/\mathsf{x}]\,[\sigma_2]\mathsf{K}_2]\!]$	By definition of relation
$\Psi;\Gamma_1 \vdash \sigma_1, \mathsf{M}_1/\mathsf{x} \equiv \sigma_2, \mathsf{M}_1/\mathsf{x} : \Gamma, \mathsf{x}{:}\mathsf{A}$	By Extending Substitution
$\Psi;\Gamma_1 \vdash [\sigma_1, \mathsf{M}_1/\mathsf{x}]\mathsf{K}_1 \approx [\sigma_2, \mathsf{M}_1/\mathsf{x}]\mathsf{K}_2 : [\![\mathsf{kind}]\!]$	By induction
$\Psi;\Gamma_1 \vdash \mathsf{A}_3\,\mathsf{M}_1 \approx \mathsf{A}_4\,\mathsf{M}_2 : [\![[\mathsf{M}_1/\mathsf{x}]\,[\sigma_1]\mathsf{K}_1]\!]$	By definition of relation
$\Psi;\Gamma_1 \vdash \mathsf{A}_3 \approx \mathsf{A}_4 : [\![\Pi\mathsf{x}{:}[\sigma_1]\mathsf{A}_1.[\sigma_1]\mathsf{K}_1]\!]$	By definition of relation

□

We can now prove the injectivity property we are interested in.

Theorem 6.5.8 (Injectivity of Products and Sums)

1. *If* $\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}$ *then* $\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type}$ *and* $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{type}$.
2. *If* $\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Sigma\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}$ *then* $\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type}$ *and* $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{type}$.

Proof

Direct, from the previous lemma and definition of the relation. We show one case, the other is analogous

Case $\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}$

$\Psi;\Gamma \vdash \mathsf{id}_\Gamma \equiv \mathsf{id}_\Gamma : \Gamma$	By reflexivity of substitutions
$\Psi;\Gamma \vdash [\mathsf{id}_\Gamma](\Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12}) \approx [\mathsf{id}_\Gamma](\Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22}) : [\![\mathsf{type}]\!]$	By Definitionally Equal Terms Related
$\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \approx \Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : [\![\mathsf{type}]\!]$	From previous
$\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type}$, $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{type}$	By definition of relation

□

6.6 Equality Algorithm

An algorithm for deciding equality is given following Harper and Pfenning [HP00]. This uses "erased" types and kinds to direct the comparison. The problem with having unerased types and kinds is that since we have dependent types, we have to pick some object to substitute in for the application or projection cases. Then it is not obvious that the algorithm is symmetric, or transitive. Fortunately, the comparison depends only on the shape of the type or kind, which is what the erasure function gets at.

6.6.1 Erasure

We erase all dependencies in families and kinds for the algorithm. This process identifies types that differ only in the terms appearing in them. We first define the simple kinds and types, which are essentially all the original kinds and types with dependencies erased. Thus dependent Π types (and kinds) erase to nondependent arrow types (and kinds), and dependent Σ types erase to nondependent product types.

Simple Kinds	$\kappa ::=$	type^-	simple kind of types
	$\mid$	$\tau \to \kappa$	simple arrow kind
Simple Types	$\tau ::=$	a^-	simple type constants
	$\mid$	$\mathsf{X_A}^-$	simple metavariable type
	$\mid$	1^-	unit
	$\mid$	$\tau_1 \to \tau_2$	simple arrow type
	$\mid$	$\tau_1 \times \tau_2$	simple product type
Simple Contexts	$\Gamma^- ::=$	$\cdot$	empty
	$\mid$	$\Gamma^-, \mathsf{x}{:}\tau$	context extension
Simple Metavariable Contexts	$\Psi^- ::=$	$\cdot$	empty
	$\mid$	$\Psi^-, \mathsf{X_M}{:}\tau$	object metavariable
	$\mid$	$\Psi^-, \mathsf{X_A}{:}\kappa$	family metavariable

In the following, we will isolate a notion of atomic types. The simple atomic types α consist of simple type constants a^- and simple metavariable types $\mathsf{X_A}^-$. The other simple types (arrow and product) are called higher types. The significance of this distinction is that comparison of objects at higher types is by the use of extensionality. Notice that the unit type is defined as a higher type from this point of view.

The erasure function is defined as follows:

$$
\begin{aligned}
(\mathsf{a})^- &= \mathsf{a}^- \\
(\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2)^- &= \mathsf{A}_2{}^- \\
(\mathsf{A}\,\mathsf{M})^- &= \mathsf{A}^- \\
(\Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2)^- &= \mathsf{A}_1{}^- \to \mathsf{A}_2{}^- \\
(\Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2)^- &= \mathsf{A}_1{}^- \times \mathsf{A}_2{}^- \\
(1)^- &= 1^- \\
((\mathsf{X_A}[\sigma]))^- &= \mathsf{X_A}{}^-
\end{aligned}
\qquad
\begin{aligned}
(\mathsf{type})^- &= \mathsf{type}^- \\
(\Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K})^- &= \mathsf{A}^- \to \mathsf{K}^- \\
(\cdot)^- &= \cdot \\
(\Gamma, \mathsf{x}{:}\mathsf{A})^- &= \Gamma^-, \mathsf{x}{:}\mathsf{A}^- \\
(\cdot)^- &= \cdot \\
(\Psi, \mathsf{X_M}{:}(\Gamma \vdash \mathsf{A}))^- &= \Psi^-, \mathsf{X_M}{:}\mathsf{A}^- \\
(\Psi, \mathsf{X_A}{:}(\Gamma \vdash \mathsf{K}))^- &= \Psi^-, \mathsf{X_A}{:}\mathsf{K}^-
\end{aligned}
$$

We will need some basic facts about erasure. We notice that erasure eliminates dependencies on terms, and further, for type-level abstractions and eliminations, erasure returns just the head family. Given this informal description, it is not surprising that erasure is invariant on substitution, and further, definitionally equal types and kinds erase to the same simple type and kind.

Lemma 6.6.1 (Erasure Preservation)

1. *For any* A, x *and* M, $([\mathsf{M}/\mathsf{x}]\,\mathsf{A})^- = \mathsf{A}^-$.
2. *For any* K, x *and* M, $([\mathsf{M}/\mathsf{x}]\,\mathsf{K})^- = \mathsf{K}^-$.

Proof

By induction on the structure of the type family or kind in the premise.

□

Lemma 6.6.2 (Erasure Preservation of Equality)

1. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$ *then* $\mathsf{A}_1{}^- = \mathsf{A}_2{}^-$
2. *If* $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$ *then* $\mathsf{K}_1{}^- = \mathsf{K}_2{}^-$

Proof

By induction on the derivation of equality.

□

Since we will case analyze on erased types, we need to say something about what type families erase to a particular erased type. We notice that a variety of equal types can erase to the same erased type, but fortunately, all of these have the same weak head normal form.

Lemma 6.6.3

1. *If* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$, $\mathsf{A}^- = 1^-$ *and* $\mathsf{A} \not\xrightarrow{\text{whr}}$, *then* $\mathsf{A} = 1$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$, $\mathsf{A}^- = \tau_1 \to \tau_2$ *and* $\mathsf{A} \not\xrightarrow{\text{whr}}$, *then* $\mathsf{A} = \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$.
3. *If* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$, $\mathsf{A}^- = \tau_1 \times \tau_2$ *and* $\mathsf{A} \not\xrightarrow{\text{whr}}$, *then* $\mathsf{A} = \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$.

Proof

By inspection of the construction of A.

□

6.6.2 Algorithmic Equality

We are now in a position to give the algorithm. This is given by five mutually recursive judgments. The idea of the algorithm is that extensionality is used whenever terms are compared at nonatomic types, to drive the type at which comparison is done to an atomic type. Once at atomic type, the comparison is done by weak-head normalizing both sides and comparing structurally. The structural phase compares the primitive head of the term, and uses the type-directed phase for terms down the spine. An analogous process is carried out at the level of type families.

Judgments

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau$	Type Directed Object Equality
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$	Structural Object Equality
$\Psi^-;\Gamma^- \vdash \tau_1 \Longleftrightarrow \tau_2 : \kappa$	Kind Directed Type Equality
$\Psi^-;\Gamma^- \vdash \tau_1 \longleftrightarrow \tau_2 : \kappa$	Structural Type Equality
$\Psi^-;\Gamma^- \vdash \kappa_1 \longleftrightarrow \kappa_2 : \mathsf{kind}^-$	Structural Kind Equality

6.6.3 Inference Rules

$\boxed{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau}$

For the type directed phase, we drive the comparison down to a base type by using the η expansion. Thus for comparison at arrow types, we add to the context a new variable, and for comparison at product types, we check each projection. If we are comparing at unit types, since terms are already well-typed at the type unit, they are equal. Once we get to a base type, we weak head normalize both sides, and then switch to the structural object equality part of the algorithm.

$$\frac{\mathsf{M}_1 \xrightarrow{\text{whr}} \mathsf{M}_2 \quad \Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M} : \alpha}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M} : \alpha}\ \textsc{Tobj-Whleft} \qquad \frac{\mathsf{M}_1 \xrightarrow{\text{whr}} \mathsf{M}_2 \quad \Psi^-;\Gamma^- \vdash \mathsf{M} \Longleftrightarrow \mathsf{M}_2 : \alpha}{\Psi^-;\Gamma^- \vdash \mathsf{M} \Longleftrightarrow \mathsf{M}_1 : \alpha}\ \textsc{Tobj-Whright}$$

$$\frac{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \alpha}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \alpha}\ \textsc{Tobj-Struct} \qquad \frac{\Psi^-;\Gamma^-, \mathsf{x}{:}\tau_1 \vdash \mathsf{M}_1\,\mathsf{x} \Longleftrightarrow \mathsf{M}_2\,\mathsf{x} : \tau_2}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau_1 \to \tau_2}\ \textsc{Tobj-Arrow}$$

$$\frac{\Psi^-;\Gamma^- \vdash \pi_1\mathsf{M}_1 \Longleftrightarrow \pi_1\mathsf{M}_2 : \tau_1 \quad \Psi^-;\Gamma^- \vdash \pi_2\mathsf{M}_1 \Longleftrightarrow \pi_2\mathsf{M}_2 : \tau_2}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau_1 \times \tau_2}\ \textsc{Tobj-Prod} \qquad \frac{}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : 1}\ \textsc{Tobj-Unit}$$

$\boxed{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau}$

In this phase of the algorithm, we check that the root structure of the terms is the same. We might need to go back to the type directed phase to check the arguments of application, so we also produce a type. The simple type here may be thought of as an output argument of the judgment.

$$\frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Psi^-;\Gamma^- \vdash \mathsf{c} \longleftrightarrow \mathsf{c} : \mathsf{A}^-}\ \textsc{Sobj-Const} \qquad \frac{\Gamma^-(\mathsf{x}) = \tau}{\Psi^-;\Gamma^- \vdash \mathsf{x} \longleftrightarrow \mathsf{x} : \tau}\ \textsc{Sobj-Var}$$

$$\frac{\Psi^-(\mathsf{X}_\mathsf{M}) = \tau}{\Psi^-;\Gamma^- \vdash \mathsf{X}_\mathsf{M}[\sigma_1] \longleftrightarrow \mathsf{X}_\mathsf{M}[\sigma_2] : \tau}\ \textsc{Sobj-MVar}$$

$$\frac{\Psi^-;\Gamma^- \vdash \mathsf{M}_{11} \longleftrightarrow \mathsf{M}_{21} : \tau_2 \to \tau_1 \quad \Psi^-;\Gamma^- \vdash \mathsf{M}_{12} \Longleftrightarrow \mathsf{M}_{22} : \tau_2}{\Psi^-;\Gamma^- \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} \longleftrightarrow \mathsf{M}_{21}\,\mathsf{M}_{22} : \tau_1}\ \textsc{Sobj-App} \qquad \frac{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau_1 \times \tau_2}{\Psi^-;\Gamma^- \vdash \pi_i \mathsf{M}_1 \longleftrightarrow \pi_i \mathsf{M}_2 : \tau_i}\ \textsc{Sobj-Proj}$$

The type family and kind levels of the algorithm is similar to the object level.

$\boxed{\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \kappa}$

Similarly to objects, for the type directed phase, we drive the comparison down to a base kind by using η expansion. Once we get to a base kind, we weak head normalize both sides, and then switch to the structural family equality part of the algorithm.

$$\frac{\mathsf{A}_1 \xrightarrow{\text{whr}} \mathsf{A}_2 \quad \Psi^-;\Gamma^- \vdash \mathsf{A}_2 \Longleftrightarrow \mathsf{A} : \mathsf{type}^-}{\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A} : \mathsf{type}^-}\ \textsc{Kfam-Whleft} \qquad \frac{\mathsf{A}_1 \xrightarrow{\text{whr}} \mathsf{A}_2 \quad \Psi^-;\Gamma^- \vdash \mathsf{A} \Longleftrightarrow \mathsf{A}_2 : \mathsf{type}^-}{\Psi^-;\Gamma^- \vdash \mathsf{A} \Longleftrightarrow \mathsf{A}_1 : \mathsf{type}^-}\ \textsc{Kfam-Whright}$$

$$\frac{\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \mathsf{type}^-}{\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \mathsf{type}^-}\ \textsc{Kfam-Struct} \qquad \frac{\Psi^-;\Gamma^-, \mathsf{x}{:}\tau \vdash \mathsf{A}_1\,\mathsf{x} \Longleftrightarrow \mathsf{A}_2\,\mathsf{x} : \kappa}{\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \tau \to \kappa}\ \textsc{Kfam-Arrow}$$

$\boxed{\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa}$

Similarly to objects, in the structural phase, we check the root structure of the type family. The simple kind here may be thought of as an output argument of the judgment.

$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Psi^-;\Gamma^- \vdash \mathsf{a} \longleftrightarrow \mathsf{a} : \mathsf{K}^-}\ \textsc{Sfam-Const} \qquad \frac{}{\Psi^-;\Gamma^- \vdash 1 \longleftrightarrow 1 : \mathsf{type}^-}\ \textsc{Sfam-Unit}$$

$$\frac{\Psi^-(\mathsf{X}_\mathsf{A}) = \kappa}{\Psi^-;\Gamma^- \vdash \mathsf{X}_\mathsf{A}[\sigma_1] \longleftrightarrow \mathsf{X}_\mathsf{A}[\sigma_2] : \kappa}\ \textsc{Sfam-MVar}$$

$$\frac{\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \tau \to \kappa \quad \Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau}{\Psi^-;\Gamma^- \vdash \mathsf{A}_1\,\mathsf{M}_1 \longleftrightarrow \mathsf{A}_2\,\mathsf{M}_2 : \kappa}\ \textsc{Sfam-App} \qquad \frac{\Psi^-;\Gamma^- \vdash \mathsf{A}_{11} \Longleftrightarrow \mathsf{A}_{21} : \mathsf{type}^- \quad \Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \vdash \mathsf{A}_{12} \Longleftrightarrow \mathsf{A}_{22} : \mathsf{type}^-}{\Psi^-;\Gamma^- \vdash \Pi \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \longleftrightarrow \Pi \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}^-}\ \textsc{Sfam-Pi}$$

$$\frac{\Psi^-;\Gamma^- \vdash \mathsf{A}_{11} \Longleftrightarrow \mathsf{A}_{21} : \mathsf{type}^- \quad \Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \vdash \mathsf{A}_{12} \Longleftrightarrow \mathsf{A}_{22} : \mathsf{type}^-}{\Psi^-;\Gamma^- \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \longleftrightarrow \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}^-}\ \textsc{Sfam-Sigma}$$

$\boxed{\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-}$

Kind equality is purely structural.

$$\frac{}{\Psi^-;\Gamma^- \vdash \mathsf{type} \longleftrightarrow \mathsf{type} : \mathsf{kind}^-}\ \textsc{Skind-Type} \qquad \frac{\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \mathsf{type}^- \quad \Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_1{}^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-}{\Psi^-;\Gamma^- \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{K}_1 \longleftrightarrow \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{K}_2 : \mathsf{kind}^-}\ \textsc{Skind-Pi}$$

We will show this algorithm to be sound and complete for the equality judgments. We will need some basic facts about the algorithm, such as that weakening of the context is allowed, and the fact that the algorithm is deterministic.

Lemma 6.6.4 (Weakening of Algorithmic Equality) *For all algorithmic judgments $\mathcal{J}$, if $\Psi^-;\Gamma^- \vdash \mathcal{J}$ and $\Gamma^- \subseteq \Gamma^{-+}$, then $\Psi^-;\Gamma^{-+} \vdash \mathcal{J}$.*

Proof

By induction on the first judgment.

□

Lemma 6.6.5 (Determinacy of Algorithmic Equality)

1. *If $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$ then $\mathsf{M}_1 \overset{\text{whr}}{\not\longrightarrow}$.*
2. *If $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$ then $\mathsf{M}_2 \overset{\text{whr}}{\not\longrightarrow}$.*
3. *If $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau_1$ and $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_3 : \tau_2$ then $\tau_1 = \tau_2$.*
4. *If $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$ then $\mathsf{A}_1 \overset{\text{whr}}{\not\longrightarrow}$.*
5. *If $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$ $\mathsf{A}_2 \overset{\text{whr}}{\not\longrightarrow}$.*
6. *If $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa_1$ and $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_3 : \kappa_2$ then $\kappa_1 = \kappa_2$.*

Proof

By induction on the derivation of the premise.

□

Notice also that families judged to be algorithmically equal have the same erasure.

Lemma 6.6.6 (Erasure Preservation of Algorithmic Equality)

1. *If $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \kappa$ then $\mathsf{A}_1{}^- = \mathsf{A}_2{}^-$.*
2. *If $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$ then $\mathsf{A}_1{}^- = \mathsf{A}_2{}^-$.*
3. *If $\mathsf{A}_1 \overset{\text{whr}}{\longrightarrow} \mathsf{A}_2$ then $\mathsf{A}_1{}^- = \mathsf{A}_2{}^-$.*

Proof

By induction on the derivation.

□

To prove that the algorithm is complete, in the cases for the use of symmetry and transitivity rules, we will need that the algorithm itself is symmetric and transitive.

Lemma 6.6.7 (Symmetry of Algorithmic Equality)

1. *If $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau$ then $\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M}_1 : \tau$.*
2. *If $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$ then $\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M}_1 : \tau$.*
3. *If $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \kappa$ then $\Psi^-;\Gamma^- \vdash \mathsf{A}_2 \Longleftrightarrow \mathsf{A}_1 : \kappa$.*
4. *If $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$ then $\Psi^-;\Gamma^- \vdash \mathsf{A}_2 \longleftrightarrow \mathsf{A}_1 : \kappa$.*
5. *If $\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-$ then $\Psi^-;\Gamma^- \vdash \mathsf{K}_2 \longleftrightarrow \mathsf{K}_1 : \mathsf{kind}^-$.*

Proof

By induction on the derivation.

□

Lemma 6.6.8 (Transitivity of Algorithmic Equality)

1. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M}_3 : \tau$, *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_3 : \tau$.
2. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \longleftrightarrow \mathsf{M}_3 : \tau$, *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_3 : \tau$.
3. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \kappa$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_2 \Longleftrightarrow \mathsf{A}_3 : \kappa$, *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_3 : \kappa$.
4. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_2 \longleftrightarrow \mathsf{A}_3 : \kappa$, *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_3 : \kappa$.
5. *If* $\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{K}_2 \longleftrightarrow \mathsf{K}_3 : \mathsf{kind}^-$, *then* $\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_3 : \mathsf{kind}^-$.

Proof

By induction on the two judgments in the premises.

□

6.7 Completeness of the Algorithm

We now turn to proving the algorithm complete with respect to the definitional equality judgment. The problem is that the definitional equality judgment might use the rules of beta and eta equality at arbitrary points of the derivation. At these points, the obvious induction hypothesis is too weak. We strengthen the hypothesis by making use of a Kripke logical relation. The point is that at higher types, this relation strengthens the hypothesis by talking of all extended contexts and all redices. This naturally requires us to prove the strengthened hypothesis in cases where the context is extended, such as in the abstraction case.

6.7.1 A Kripke Logical Relation

The logical relation we use is defined by induction on the simple types and kinds.

- $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\mathsf{a}^-]\!]$ iff $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{a}^-$.
- $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \to \tau_2]\!]$ iff for every context $\Gamma^-{}_1$ with $\Psi^- \subseteq \Gamma^-\Psi^-\Gamma^-{}_1$ and every pair of objects M_1' and M_2' such that $\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_1'$ **is** M_2' **in** $[\![\tau_1]\!]$ we have $\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_1\,\mathsf{M}_1'$ **is** $\mathsf{M}_2\,\mathsf{M}_2'$ **in** $[\![\tau_2]\!]$.
- $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \times \tau_2]\!]$ iff $\Psi^-;\Gamma^- \vdash \pi_1\mathsf{M}_1$ **is** $\pi_1\mathsf{M}_2$ **in** $[\![\tau_1]\!]$ and $\Psi^-;\Gamma^- \vdash \pi_1\mathsf{M}_2$ **is** $\pi_2\mathsf{M}_2$ **in** $[\![\tau_2]\!]$.
- $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![1^-]\!]$ always.
- $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$ iff $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \mathsf{type}^-$.
- $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$ iff for every context $\Gamma^-{}_1$ with $\Gamma^- \subseteq \Gamma^-{}_1$ and every pair of objects M_1 and M_2 such that $\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau]\!]$ we have $\Psi^-;\Gamma^-{}_1 \vdash \mathsf{A}_1\,\mathsf{M}_1$ **is** $\mathsf{A}_2\,\mathsf{M}_2$ **in** $[\![\mathsf{A}_2\,\mathsf{M}_2]\!]$.
- $\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![\cdot]\!]$ iff $\sigma_1 = \sigma_2 = \cdot$.
- $\Psi^-;\Gamma^-{}_1 \vdash [\mathsf{M}_1/\mathsf{x}]\,\sigma_1$ **is** $[\mathsf{M}_2/\mathsf{x}]\,\sigma_2$ **in** $[\![\Gamma^-, \mathsf{x}{:}\tau]\!]$ iff $\Psi^-;\Gamma^-{}_1 \vdash \sigma_1$ **is** σ_2 **in** $[\![\Gamma^-]\!]$ and $\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau]\!]$.

The logical relation is defined at base types in terms of algorithmic equality. It is not surprising that this can be extended to all types, to get the following basic lemma.

Lemma 6.7.1 (Logically Related Terms are Algorithmically Equal [Fundamental Lemma])

1. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau$.
2. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \kappa$.
3. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau]\!]$.
4. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$.

Proof

By simultaneous structural induction on the simple types, kinds and contexts in the premises. We will show parts 1 and 4, to do with terms. The argument for type families and substitutions is similar.

Case 1: Part (1), $\tau = \mathsf{a}^-$.

By definition of relation, $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{a}^-$.

Case 2: Part (1), $\tau = \mathsf{X}_\mathsf{A}{}^-$.

By definition of relation, $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{X}_\mathsf{A}{}^-$.

Case 3: Part (1), $\tau = \tau_1 \to \tau_2$.

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \to \tau_2]\!]$	By assumption
$\Psi^-;\Gamma^-, \mathsf{x}{:}\tau_1 \vdash \mathsf{x} \longleftrightarrow \mathsf{x} : \tau_1$	By rule (variable)
$\Psi^-;\Gamma^-, \mathsf{x}{:}\tau_1 \vdash \mathsf{x}$ **is** x **in** $[\![\tau_1]\!]$	By induction (part 3)
$\Psi^-;\Gamma^-, \mathsf{x}{:}\tau_1 \vdash \mathsf{M}_1\,\mathsf{x}$ **is** $\mathsf{M}_2\,\mathsf{x}$ **in** $[\![\tau_2]\!]$	By definition of relation
$\Psi^-;\Gamma^-, \mathsf{x}{:}\tau_1 \vdash \mathsf{M}_1\,\mathsf{x} \Longleftrightarrow \mathsf{M}_2\,\mathsf{x} : \tau_2$	By induction (part 1) on smaller type τ_2
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau_1 \to \tau_2$	By rule (functions)

Case 4: Part (1), $\tau = \tau_1 \times \tau_2$.

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \times \tau_2]\!]$	By assumption
$\Psi^-;\Gamma^- \vdash \pi_1\mathsf{M}_1$ **is** $\pi_1\mathsf{M}_2$ **in** $[\![\tau_1]\!]$,	
$\Psi^-;\Gamma^- \vdash \pi_2\mathsf{M}_1$ **is** $\pi_2\mathsf{M}_2$ **in** $[\![\tau_2]\!]$,	By definition of relation
$\Psi^-;\Gamma^- \vdash \pi_1\mathsf{M}_1$ **is** $\pi_1\mathsf{M}_2$ **in** $[\![\tau_1]\!]$,	
$\Psi^-;\Gamma^- \vdash \pi_2\mathsf{M}_1$ **is** $\pi_2\mathsf{M}_2$ **in** $[\![\tau_2]\!]$,	By induction (part 1)
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau_1 \times \tau_2$	By rule (products)

Case 5: Part (1), $\tau = 1^-$.

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : 1^-$	By rule (unit)

Case 6: Part (3), $\tau = \mathsf{a}^-$.

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \mathsf{a}^-$	By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{a}^-$	By rule (constants)
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\mathsf{a}^-]\!]$	By definition of relation

Case 7: Part (3), $\tau = \mathsf{X_A}^-$.

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \mathsf{X_A}^-$	By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{X_A}^-$	By rule
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\mathsf{X_A}^-]\!]$	By definition of relation

Case 8: Part (3), $\tau = \tau_1 \to \tau_2$.

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau_1 \to \tau_2$	By assumption
$\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_{11}$ **is** M_{21} **in** $[\![\tau_1]\!]$ for arbitrary $\Gamma^-{}_1$	New hypothesis
$\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_{11} \Longleftrightarrow \mathsf{M}_{21} : \tau_1$	By induction (part 1)
$\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau_1 \to \tau_2$	By Weakening
$\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_1\,\mathsf{M}_{11} \longleftrightarrow \mathsf{M}_2\,\mathsf{M}_{21} : \tau_2$	By rule (application)
$\Psi^-;\Gamma^-{}_1 \vdash \mathsf{M}_1\,\mathsf{M}_{11}$ **is** $\mathsf{M}_2\,\mathsf{M}_{21}$ **in** $[\![\tau_2]\!]$	By induction (part 3) on smaller type τ_2
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \to \tau_2]\!]$	By definition of rule

Case 9: Part (3), $\tau = \tau_1 \times \tau_2$.

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau_1 \times \tau_2$	By assumption
$\Psi^-;\Gamma^- \vdash \pi_i\mathsf{M}_1 \longleftrightarrow \pi_i\mathsf{M}_2 : \tau_i$	By rule (projection)
$\Psi^-;\Gamma^- \vdash \pi_i\mathsf{M}_1$ **is** $\pi_i\mathsf{M}_2$ **in** $[\![\tau_i]\!]$	By induction (part 3)
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \times \tau_2]\!]$	By definition of relation

Case 10: Part (3), $\tau = 1^-$.

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![1^-]\!]$ By definition of relation

□

Similar to the simpler logical relations argument for proving injectivity, we will need closure under head expansion for proving completeness for the abstraction case. We will also need the symmetry and transitivity of the logical relations, which will use the corresponding properties at base types and lift to function and product types.

Lemma 6.7.2 (Closure Under Head Expansion)

1. *If* $\mathsf{M}_{11} \xrightarrow{\text{whr}} \mathsf{M}_{12}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_{12}$ **is** M_2 **in** $[\![\tau]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_{11}$ **is** M_2 **in** $[\![\tau]\!]$.
2. *If* $\mathsf{A}_{11} \xrightarrow{\text{whr}} \mathsf{A}_{12}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_{12}$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_{11}$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$.
3. *If* $\mathsf{M}_{21} \xrightarrow{\text{whr}} \mathsf{M}_{22}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_{22} **in** $[\![\tau]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_{21} **in** $[\![\tau]\!]$.
4. *If* $\mathsf{A}_{21} \xrightarrow{\text{whr}} \mathsf{A}_{22}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_{22} **in** $[\![\mathsf{A}_{22}]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_{21} **in** $[\![\mathsf{A}_{21}]\!]$.

Proof

By induction on the simple type or kind in the premises.

□

Lemma 6.7.3 (Weakening of Logical Relations)

1. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau]\!]$ *and* $\Gamma^- \subseteq \Gamma^{-^+}$, *then* $\Psi^-;\Gamma^{-^+} \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau]\!]$.
2. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$ *and* $\Gamma^- \subseteq \Gamma^{-^+}$, *then* $\Psi^-;\Gamma^{-^+} \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$.
3. *If* $\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![{\Gamma^-}_1]\!]$ *and* $\Gamma^- \subseteq \Gamma^{-^+}$, *then* $\Psi^-;\Gamma^{-^+} \vdash \sigma_1$ **is** σ_2 **in** $[\![\sigma_2]\!]$.

Proof

By induction on the simple type, kind or context in the premise.

□

Lemma 6.7.4 (Symmetry of Logical Relations)

1. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_1 **in** $[\![\tau]\!]$.
2. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_2$ **is** A_1 **in** $[\![\mathsf{A}_1]\!]$.
3. *If* $\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![{\Gamma^-}_1]\!]$ *then* $\Psi^-;\Gamma^- \vdash \sigma_2$ **is** σ_1 **in** $[\![{\Gamma^-}_1]\!]$.

Proof

By induction on the simple type, kind or context in the premise. We will show the cases for objects. The other cases are similar.

Case 1: $\tau = \mathsf{a}^-$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\mathsf{a}^-]\!]$	By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{a}^-$	By definition of relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M}_1 : \mathsf{a}^-$	By Symmetry of Algorithm
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_1 **in** $[\![\mathsf{a}^-]\!]$	By definition of relation

Case 2: $\tau = {\mathsf{X}_\mathsf{A}}^-$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![{\mathsf{X}_\mathsf{A}}^-]\!]$	By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : {\mathsf{X}_\mathsf{A}}^-$	By definition of relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M}_1 : {\mathsf{X}_\mathsf{A}}^-$	By Symmetry of Algorithm
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_1 **in** $[\![{\mathsf{X}_\mathsf{A}}^-]\!]$	By definition of relation

Case 3: $\tau = \tau_1 \to \tau_2$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \to \tau_2]\!]$	By assumption
$\Psi^-;\Gamma^{-^+} \vdash \mathsf{M}_{21}$ **is** M_{11} **in** $[\![\tau_1]\!]$ for $\Gamma^{-^+} \supseteq \Gamma^-$	New hypothesis

$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_{11}$ **is** M_{21} **in** $[\![\tau_1]\!]$ — By induction
$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_1\,\mathsf{M}_{11}$ **is** $\mathsf{M}_2\,\mathsf{M}_{21}$ **in** $[\![\tau_2]\!]$ — By definition of relation
$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_2\,\mathsf{M}_{21}$ **is** $\mathsf{M}_1\,\mathsf{M}_{11}$ **in** $[\![\tau_2]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_1 **in** $[\![\tau_1 \to \tau_2]\!]$ — By definition of relation

Case 4: $\tau = \tau_1 \times \tau_2$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \times \tau_2]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \pi_i\mathsf{M}_1$ **is** $\pi_i\mathsf{M}_2$ **in** $[\![\tau_i]\!]$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \pi_i\mathsf{M}_2$ **is** $\pi_i\mathsf{M}_1$ **in** $[\![\tau_i]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_1 **in** $[\![\tau_1 \to \tau_2]\!]$ — By definition of relation

Case 5: $\tau = 1^-$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_1 **in** $[\![1^-]\!]$ — By definition of relation

□

Lemma 6.7.5 (Transitivity of Logical Relations)

1. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau]\!]$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_3 **in** $[\![\tau]\!]$, *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_3 **in** $[\![\tau]\!]$.
2. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_2$ **is** A_3 **in** $[\![\mathsf{A}_3]\!]$, *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_3 **in** $[\![\mathsf{A}_3]\!]$.
3. *If* $\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![{\Gamma^-}_1]\!]$ *and* $\Psi^-;\Gamma^- \vdash \sigma_2$ **is** σ_3 **in** $[\![{\Gamma^-}_1]\!]$, *then* $\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_3 **in** $[\![{\Gamma^-}_1]\!]$.

Proof

By induction on the simple type, kind or context in the premise. We will show the cases for objects. The other cases are similar.

Case 1: $\tau = \mathsf{a}^-$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\mathsf{a}^-]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_3 **in** $[\![\mathsf{a}^-]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{a}^-$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M}_3 : \mathsf{a}^-$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_3 : \mathsf{a}^-$ — By Transitivity of Algorithm
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_3 **in** $[\![\mathsf{a}^-]\!]$ — By definition of relation

Case 2: $\tau = {\mathsf{X}_\mathsf{A}}^-$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![{\mathsf{X}_\mathsf{A}}^-]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_3 **in** $[\![{\mathsf{X}_\mathsf{A}}^-]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : {\mathsf{X}_\mathsf{A}}^-$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M}_3 : {\mathsf{X}_\mathsf{A}}^-$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_3 : {\mathsf{X}_\mathsf{A}}^-$ — By Transitivity of Algorithm
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_3 **in** $[\![{\mathsf{X}_\mathsf{A}}^-]\!]$ — By definition of relation

Case 3: $\tau = \tau_1 \to \tau_2$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \to \tau_2]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_3 **in** $[\![\tau_1 \to \tau_2]\!]$ — By assumption
$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_{11}$ **is** M_{21} **in** $[\![\tau_1]\!]$ for $\Gamma^{-+} \supseteq \Gamma^-$ — New hypothesis
$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_{21}$ **is** M_{11} **in** $[\![\tau_1]\!]$ — By Symmetry of Relation
$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_{21}$ **is** M_{21} **in** $[\![\tau_1]\!]$ — By induction
$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_1\,\mathsf{M}_{11}$ **is** $\mathsf{M}_2\,\mathsf{M}_{21}$ **in** $[\![\tau_2]\!]$ — By definition of relation

$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_2\,\mathsf{M}_{21}$ **is** $\mathsf{M}_3\,\mathsf{M}_{21}$ **in** $[\![\tau_2]\!]$ — By definition of relation
$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_1\,\mathsf{M}_{11}$ **is** $\mathsf{M}_3\,\mathsf{M}_{21}$ **in** $[\![\tau_2]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_3 **in** $[\![\tau_1 \to \tau_2]\!]$ — By definition of relation

Case 4: $\tau = \tau_1 \times \tau_2$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\tau_1 \times \tau_2]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2$ **is** M_3 **in** $[\![\tau_1 \times \tau_2]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \pi_i\mathsf{M}_1$ **is** $\pi_i\mathsf{M}_2$ **in** $[\![\tau_i]\!]$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \pi_i\mathsf{M}_2$ **is** $\pi_i\mathsf{M}_3$ **in** $[\![\tau_i]\!]$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \pi_i\mathsf{M}_1$ **is** $\pi_i\mathsf{M}_3$ **in** $[\![\tau_i]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_3 **in** $[\![\tau_1 \to \tau_2]\!]$ — By definition of relation

Case 5: $\tau = 1^-$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_3 **in** $[\![1^-]\!]$ — By definition of relation

□

We now come to the main theorem about logical relations. This says that all terms judged equal are logically related to each other by related substitutions. The logical relation at higher types provides just the right inductive hypothesis to make this lemma go through. This will enable us to directly get completeness of algorithmic equality, since we have already seen by the fundamental lemma that logically related terms are also equal by the algorithm.

Lemma 6.7.6 (Definitionally Equal Terms are Logically Related Under Substitutions)

1. *If* $\Psi;\Gamma_1 \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$ *and* $\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![{\Gamma_1}^-]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}^-]\!]$.
2. *If* $\Psi;\Gamma_1 \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$ *and* $\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![{\Gamma_1}^-]\!]$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1[\sigma_1]$ **is** $\mathsf{A}_2[\sigma_2]$ **in** $[\![\mathsf{A}_2[\sigma_2]]\!]$.

Proof

By induction on the judgment of definitional equality. We will show some representative cases.

Case 1: $\dfrac{\Gamma(\mathsf{x}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{x} \equiv \mathsf{x} : \mathsf{A}}$ Objeq-Var

$\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![{\Gamma_1}^-]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\mathsf{A}^-]\!]$ for $[\mathsf{M}_1/\mathsf{x}] \in \sigma_1$, $[\mathsf{M}_2/\mathsf{x}] \in \sigma_2$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \mathsf{x}[\sigma_1]$ **is** $\mathsf{x}[\sigma_2]$ **in** $[\![\mathsf{A}^-]\!]$ — By definition of substitution

Case 2: $\dfrac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{c} \equiv \mathsf{c} : \mathsf{A}}$ Objeq-Const

$\Psi^-;\Gamma^- \vdash \mathsf{c} \longleftrightarrow \mathsf{c} : \mathsf{A}^-$ — By rule (constant)
$\Psi^-;\Gamma^- \vdash \mathsf{c}$ **is** c **in** $[\![\mathsf{A}^-]\!]$ — By Fundamental Theorem
$\Psi^-;\Gamma^- \vdash \mathsf{c}[\sigma_1]$ **is** $\mathsf{c}[\sigma_2]$ **in** $[\![\mathsf{A}^-]\!]$ — By definition of substitution

Case 3: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \Pi\mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} \equiv \mathsf{M}_{21}\,\mathsf{M}_{22} : [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_1}$ Objeq-App

$\Psi^-;\Gamma^- \vdash \mathsf{M}_{11}[\sigma_1]$ **is** $\mathsf{M}_{21}[\sigma_2]$ **in** $[\![{\mathsf{A}_2}^- \to {\mathsf{A}_1}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_{12}[\sigma_1]$ **is** $\mathsf{M}_{22}[\sigma_2]$ **in** $[\![{\mathsf{A}_2}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash (\mathsf{M}_{11}[\sigma_1])\,(\mathsf{M}_{12}[\sigma_2])$ **is** $(\mathsf{M}_{21}[\sigma_2])\,(\mathsf{M}_{22}[\sigma_2])$ **in** $[\![{\mathsf{A}_1}^-]\!]$ — By definition of relation
$\Psi^-;\Gamma^- \vdash (\mathsf{M}_{11}\,\mathsf{M}_{12}[\sigma_1])$ **is** $(\mathsf{M}_{21}\,\mathsf{M}_{22}[\sigma_2])$ **in** $[\![{\mathsf{A}_1}^-]\!]$ — By definition of substitution

Case 4:

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma \vdash \mathsf{A}_{12} \equiv \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{M}_1 \equiv \lambda \mathsf{x}{:}\mathsf{A}_{12}.\mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{Objeq-Abs}$$

$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_{11}$ **is** M_{21} **in** $[\![\mathsf{A}_1{}^-]\!]$ for $\Gamma^{-+} \supseteq \Gamma^-$ — New hypothesis
$\Psi^-;\Gamma^{-+} \vdash \sigma_1$ **is** σ_2 **in** $[\![\Gamma_1{}^-]\!]$ — By weakening of relation
$\Psi^-;\Gamma^{-+} \vdash \sigma_1, \mathsf{M}_{11}/\mathsf{x}$ **is** $\sigma_2, \mathsf{M}_{21}/\mathsf{x}$ **in** $[\![\Gamma_1{}^-, \mathsf{x}{:}\mathsf{A}_1{}^-]\!]$ — By definition of relation
$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_1[\sigma_1, \mathsf{M}_{11}/\mathsf{x}]$ **is** $\mathsf{M}_2[\sigma_2, \mathsf{M}_{21}/\mathsf{x}]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By induction
$\Psi^-;\Gamma^{-+} \vdash (\lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{M}_1[\sigma_1, \mathsf{x}/\mathsf{x}])\,\mathsf{M}_{11}$ **is** $\mathsf{M}_2[\sigma_2, \mathsf{M}_{21}/\mathsf{x}]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By Closure Under Head Expansion
$\Psi^-;\Gamma^{-+} \vdash (\lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{M}_1[\sigma_1, \mathsf{x}/\mathsf{x}])\,\mathsf{M}_{11}$ **is** $(\lambda \mathsf{x}{:}\mathsf{A}_{12}.\mathsf{M}_2[\sigma_2, \mathsf{x}/\mathsf{x}])\,\mathsf{M}_{21}$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By Closure Under Head Expansion
$\Psi^-;\Gamma^{-+} \vdash ((\lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{M}_1)[\sigma_1])\,\mathsf{M}_{11}$ **is** $((\lambda \mathsf{x}{:}\mathsf{A}_{12}.\mathsf{M}_2)[\sigma_2])\,\mathsf{M}_{21}$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By definition of substitution
$\Psi^-;\Gamma^- \vdash (\lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{M}_1)[\sigma_1]$ **is** $(\lambda \mathsf{x}{:}\mathsf{A}_{12}.\mathsf{M}_2)[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^- \to \mathsf{A}_2{}^-]\!]$ — By definition of relation

Case 5:

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{M}_1 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \\ \Psi;\Gamma \vdash \mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1\,\mathsf{x} \equiv \mathsf{M}_2\,\mathsf{x} : \mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{Objeq-ExtPi}$$

$\Psi^-;\Gamma^{-+} \vdash \mathsf{M}_{11}$ **is** M_{21} **in** $[\![\mathsf{A}_1{}^-]\!]$ for $\Gamma^{-+} \supseteq \Gamma^-$ — New hypothesis
$\Psi^-;\Gamma^{-+} \vdash \sigma_1$ **is** σ_2 **in** $[\![\Gamma_1{}^-]\!]$ — By weakening of relation
$\Psi^-;\Gamma^{-+} \vdash \sigma_1, \mathsf{M}_{11}/\mathsf{x}$ **is** $\sigma_2, \mathsf{M}_{21}/\mathsf{x}$ **in** $[\![\Gamma_1{}^-, \mathsf{x}{:}\mathsf{A}_1{}^-]\!]$ — By definition of relation
$\Psi^-;\Gamma^{-+} \vdash (\mathsf{M}_1\,\mathsf{x})[\sigma_1, \mathsf{M}_{11}/\mathsf{x}]$ **is** $(\mathsf{M}_2\,\mathsf{x})[\sigma_2, \mathsf{M}_{21}/\mathsf{x}]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By induction
$\Psi^-;\Gamma^{-+} \vdash (\mathsf{M}_1[\sigma_1])\,\mathsf{M}_{11}$ **is** $(\mathsf{M}_2[\sigma_2])\,\mathsf{M}_{21}$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By definition of substitution
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^- \to \mathsf{A}_2{}^-]\!]$ — By definition of relation

Case 6:

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : \mathsf{A}_2 \qquad \Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \mathsf{A}_1}{\Psi;\Gamma \vdash (\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{12})\,\mathsf{M}_{11} \equiv [\mathsf{M}_{21}/\mathsf{x}]\,\mathsf{M}_{22} : [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2}\ \textsc{Objeq-BetaPi}$$

$\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![\Gamma_1{}^-]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \mathsf{M}_{11}$ **is** M_{21} **in** $[\![\mathsf{A}_1{}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \sigma_1, (\mathsf{M}_{11}[\sigma_1])/\mathsf{x}$ **is** $\sigma_2, (\mathsf{M}_{21}[\sigma_2])/\mathsf{x}$ **in** $[\![\Gamma_1{}^-, \mathsf{x}{:}\mathsf{A}_1{}^-]\!]$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_{12}[\sigma_1, (\mathsf{M}_{11}[\sigma_1])/\mathsf{x}]$ **is** $\mathsf{M}_{22}[\sigma_2, (\mathsf{M}_{21}[\sigma_2])/\mathsf{x}]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash (\mathsf{M}_{12}[\sigma_1, \mathsf{x}/\mathsf{x}])[(\mathsf{M}_{11}[\sigma_1])/\mathsf{x}]$ **is** $\mathsf{M}_{22}[\sigma_2, (\mathsf{M}_{21}[\sigma_2])/\mathsf{x}]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By definition of substitution
$\Psi^-;\Gamma^- \vdash (\mathsf{M}_{12}[\sigma_1, \mathsf{x}/\mathsf{x}])[(\mathsf{M}_{11}[\sigma_1])/\mathsf{x}]$ **is** $(\mathsf{M}_{22}[\mathsf{M}_{21}/\mathsf{x}])[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By definition of substitution
$\Psi^-;\Gamma^- \vdash (\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{12}[\sigma_1, \mathsf{x}/\mathsf{x}])\,\mathsf{M}_{11}[\sigma_1]$ **is** $(\mathsf{M}_{22}[\mathsf{M}_{21}/\mathsf{x}])[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By Closure Under Head Expansion
$\Psi^-;\Gamma^- \vdash (\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{12}\,\mathsf{M}_{11})[\sigma_1]$ **is** $(\mathsf{M}_{22}[\mathsf{M}_{21}/\mathsf{x}])[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By definition of substitution
$\Psi^-;\Gamma^- \vdash (\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{12}\,\mathsf{M}_{11})[\sigma_1]$ **is** $(\mathsf{M}_{22}[\mathsf{M}_{21}/\mathsf{x}])[\sigma_2]$ **in** $[\![[\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2{}^-]\!]$ — By Erasure Preservation

Case 7:

$$\frac{\begin{array}{c}\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \mathsf{A}_1 \\ \Psi;\Gamma \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \langle \mathsf{M}_{11}, \mathsf{M}_{12} \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} \equiv \langle \mathsf{M}_{21}, \mathsf{M}_{22} \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{Objeq-Pair}$$

$\Psi^-;\Gamma^- \vdash \mathsf{M}_{11}[\sigma_1]$ **is** $\mathsf{M}_{21}[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \pi_1\langle(\mathsf{M}_{11}[\sigma_1]), (\mathsf{M}_{12}[\sigma_1])\rangle^{(\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2[\sigma_1])}$ **is** $\mathsf{M}_{21}[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^-]\!]$ — By Closure Under Head Expansion
$\Psi^-;\Gamma^- \vdash \pi_1\langle(\mathsf{M}_{11}[\sigma_1]), (\mathsf{M}_{12}[\sigma_1])\rangle^{(\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2[\sigma_1])}$ **is** $\pi_1\langle(\mathsf{M}_{21}[\sigma_2]), (\mathsf{M}_{22}[\sigma_2])\rangle^{(\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2[\sigma_2])}$ **in** $[\![\mathsf{A}_1{}^-]\!]$ — By Closure Under Head Expansion

$\Psi^-;\Gamma^- \vdash \pi_1(\langle \mathsf{M}_{11}, \mathsf{M}_{12}\rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2}[\sigma_1])$ **is** $\pi_1(\langle \mathsf{M}_{21}, \mathsf{M}_{22}\rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2}[\sigma_2])$ **in** $[\![\mathsf{A}_1{}^-]\!]$ By definition of substitution
$\Psi^-;\Gamma^- \vdash \mathsf{M}_{12}[\sigma_1]$ **is** $\mathsf{M}_{22}[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By induction
$\Psi^-;\Gamma^- \vdash \pi_2\langle(\mathsf{M}_{11}[\sigma_1]), (\mathsf{M}_{12}[\sigma_1])\rangle^{(\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2[\sigma_1])}$ **is** $\mathsf{M}_{22}[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By Closure Under Head Expansion
$\Psi^-;\Gamma^- \vdash \pi_2\langle(\mathsf{M}_{11}[\sigma_1]), (\mathsf{M}_{12}[\sigma_1])\rangle^{(\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2[\sigma_1])}$ **is** $\pi_2\langle(\mathsf{M}_{21}[\sigma_2]), (\mathsf{M}_{22}[\sigma_2])\rangle^{(\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2[\sigma_2])}$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By Closure Under Head Expansion
$\Psi^-;\Gamma^- \vdash \pi_2(\langle \mathsf{M}_{11}, \mathsf{M}_{12}\rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2}[\sigma_1])$ **is** $\pi_2(\langle \mathsf{M}_{21}, \mathsf{M}_{22}\rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2}[\sigma_2])$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By definition of substitution
$\Psi^-;\Gamma^- \vdash (\langle \mathsf{M}_{11}, \mathsf{M}_{12}\rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2})[\sigma_1]$ **is** $(\langle \mathsf{M}_{21}, \mathsf{M}_{22}\rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2})[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^- \times \mathsf{A}_2{}^-]\!]$ By definition of relation

Case 8: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_1\mathsf{M}_1 \equiv \pi_1\mathsf{M}_2 : \mathsf{A}_1}$ Objeq-Proj1

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^- \times \mathsf{A}_2{}^-]\!]$ By induction
$\Psi^-;\Psi^- \vdash \Gamma^-$ **is** $\pi_1(\mathsf{M}_1[\sigma_1])$ **in** $[\![\pi_1(\mathsf{M}_2[\sigma_2])]\!]\mathsf{A}_1{}^-$ By definition of relation
$\Psi^-;\Gamma^- \vdash (\pi_1\mathsf{M}_1)[\sigma_1]$ **is** $(\pi_1\mathsf{M}_2)[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^-]\!]$ By definition of substitution

Case 9: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_2\mathsf{M}_1 \equiv \pi_2\mathsf{M}_2 : [\pi_1\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2}$ Objeq-Proj2

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^- \times \mathsf{A}_2{}^-]\!]$ By induction
$\Psi^-;\Gamma^- \vdash \pi_2(\mathsf{M}_1[\sigma_1])$ **is** $\pi_2(\mathsf{M}_2[\sigma_2])$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By definition of relation
$\Psi^-;\Gamma^- \vdash (\pi_2\mathsf{M}_1)[\sigma_1]$ **is** $(\pi_2\mathsf{M}_2)[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By definition of substitution

Case 10: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_1 : 1 \quad \Psi;\Gamma \vdash \mathsf{M}_2 : 1}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : 1}$ Objeq-ExtUnit

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![1^-]\!]$ By definition of relation

Case 11: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_3 : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \pi_1\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} \equiv \mathsf{M}_3 : \mathsf{A}_1}$ Objeq-BetaProj1

$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_3[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^-]\!]$ By induction
$\Psi^-;\Gamma^- \vdash \pi_1\langle(\mathsf{M}_1[\sigma_1]), (\mathsf{M}_2[\sigma_1])\rangle^{(\mathsf{A}[\sigma_1])}$ **is** $\mathsf{M}_3[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^-]\!]$ By Closure Under Head Expansion
$\Psi^-;\Gamma^- \vdash (\pi_1\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}})[\sigma_1]$ **is** $\mathsf{M}_3[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^-]\!]$ By definition of substitution

Case 12: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_3 : \mathsf{A}_2}{\Psi;\Gamma \vdash \pi_2\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}} \equiv \mathsf{M}_3 : \mathsf{A}_2}$ Objeq-BetaProj2

$\Psi^-;\Gamma^- \vdash \mathsf{M}_2[\sigma_1]$ **is** $\mathsf{M}_3[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By induction
$\Psi^-;\Gamma^- \vdash \pi_2\langle(\mathsf{M}_1[\sigma_1]), (\mathsf{M}_2[\sigma_1])\rangle^{(\mathsf{A}[\sigma_1])}$ **is** $\mathsf{M}_3[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By Closure Under Head Expansion
$\Psi^-;\Gamma^- \vdash (\pi_2\langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}})[\sigma_1]$ **is** $\mathsf{M}_3[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By definition of substitution

Case 13: $\dfrac{\begin{array}{c}\Psi;\Gamma \vdash \mathsf{M}_1 : \Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2 \quad \Psi;\Gamma \vdash \mathsf{M}_2 : \Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2 \\ \Psi;\Gamma \vdash \pi_1\mathsf{M}_1 \equiv \pi_1\mathsf{M}_2 : \mathsf{A}_1 \quad \Psi;\Gamma \vdash \pi_2\mathsf{M}_1 \equiv \pi_2\mathsf{M}_2 : [\pi_1\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2}$ Objeq-ExtSigma

$\Psi^-;\Gamma^- \vdash (\pi_1\mathsf{M}_1)[\sigma_1]$ **is** $(\pi_1\mathsf{M}_2)[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^-]\!]$ By induction
$\Psi^-;\Gamma^- \vdash (\pi_2\mathsf{M}_1)[\sigma_1]$ **is** $(\pi_2\mathsf{M}_2)[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ By induction

$\Psi^-;\Gamma^- \vdash \pi_1(\mathsf{M}_1[\sigma_1])$ **is** $\pi_1(\mathsf{M}_2[\sigma_2])$ **in** $[\![\mathsf{A}_1{}^-]\!]$ — By definition of substitution
$\Psi^-;\Gamma^- \vdash \pi_2(\mathsf{M}_1[\sigma_1])$ **is** $\pi_2(\mathsf{M}_2[\sigma_2])$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By definition of substitution
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^- \times \mathsf{A}_2{}^-]\!]$ — By definition of relation

Case 14: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_1 : \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}}$ Objeq-Symm

$\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![\Gamma_1{}^-]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \sigma_2$ **is** σ_1 **in** $[\![\Gamma_1{}^-]\!]$ — By Symmetry of Relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2[\sigma_2]$ **is** $\mathsf{M}_1[\sigma_1]$ **in** $[\![\mathsf{A}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}^-]\!]$ — By Symmetry of Relation

Case 15: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A} \qquad \Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_3 : \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_3 : \mathsf{A}}$ Objeq-Trans

$\Psi^-;\Gamma^- \vdash \sigma_1$ **is** σ_2 **in** $[\![\Gamma_1{}^-]\!]$ — By assumption
$\Psi^-;\Gamma^- \vdash \sigma_2$ **is** σ_1 **in** $[\![\Gamma_1{}^-]\!]$ — By Symmetry of Relation
$\Psi^-;\Gamma^- \vdash \sigma_2$ **is** σ_2 **in** $[\![\Gamma_1{}^-]\!]$ — By Transitivity of Relation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2[\sigma_2]$ **is** $\mathsf{M}_3[\sigma_2]$ **in** $[\![\mathsf{A}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_3[\sigma_2]$ **in** $[\![\mathsf{A}^-]\!]$ — By Transitivity of Relation

Case 16: $\dfrac{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \qquad \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_1}$ Objeq-Fameq

$\mathsf{A}_1{}^- = \mathsf{A}_2{}^-$ — By Erasure Preservation
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}_2{}^-]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1[\sigma_1]$ **is** $\mathsf{M}_2[\sigma_2]$ **in** $[\![\mathsf{A}_1{}^-]\!]$ — From previous

Case 17: $\dfrac{\begin{array}{c}\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{A}_{11} : \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{type}\end{array}}{\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}}$ Fameq-Sigma

$\Psi^-;\Gamma^- \vdash \mathsf{A}_{11}[\sigma_1]$ **is** $\mathsf{A}_{21}[\sigma_2]$ **in** $[\![\mathsf{A}_{21}[\sigma_2]]\!]$ — By induction
$\Psi^-;\Gamma^- \vdash \mathsf{A}_{11}[\sigma_1] \Longleftrightarrow \mathsf{A}_{21}[\sigma_2] : \mathsf{type}^-$ — By definition of relation
$\Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \vdash \mathsf{x} \longleftrightarrow \mathsf{x} : \mathsf{A}_{11}{}^-$ — By rule (variable)
$\Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \vdash \mathsf{x}$ **is** x **in** $[\![\mathsf{A}_{11}{}^-]\!]$ — By Fundamental Theorem
$\Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \vdash [\mathsf{x}/\mathsf{x}]\,\sigma_1$ **is** $[\mathsf{x}/\mathsf{x}]\,\sigma_2$ **in** $[\![\mathsf{A}_{11}{}^-]\!]$ — By definition of relation
$\Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \vdash \mathsf{A}_{12}[\sigma_1, \mathsf{x}/\mathsf{x}]$ **is** $\mathsf{A}_{22}[\sigma_2, \mathsf{x}/\mathsf{x}]$ **in** $[\![\mathsf{A}_{22}[\sigma_2, \mathsf{x}/\mathsf{x}]]\!]$ — By induction
$\Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \vdash \mathsf{A}_{12}[\sigma_1, \mathsf{x}/\mathsf{x}] \Longleftrightarrow \mathsf{A}_{22}[\sigma_2, \mathsf{x}/\mathsf{x}] : \mathsf{type}^-$ — By definition of relation
$\Psi^-;\Gamma^- \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}[\sigma_1].\mathsf{A}_{12}[\sigma_1, \mathsf{x}/\mathsf{x}] \longleftrightarrow \Sigma \mathsf{x}{:}\mathsf{A}_{21}[\sigma_2].\mathsf{A}_{22}[\sigma_2, \mathsf{x}/\mathsf{x}] : \mathsf{type}^-$ — By rule (sums)
$\Psi^-;\Gamma^- \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}[\sigma_1].\mathsf{A}_{12}[\sigma_1, \mathsf{x}/\mathsf{x}]$ **is** $\Sigma \mathsf{x}{:}\mathsf{A}_{21}[\sigma_2].\mathsf{A}_{22}[\sigma_2, \mathsf{x}/\mathsf{x}]$ **in** $[\![\Sigma \mathsf{x}{:}\mathsf{A}_{21}[\sigma_2].\mathsf{A}_{22}[\sigma_2, \mathsf{x}/\mathsf{x}]]\!]$ — By definition of relation
$\Psi^-;\Gamma^- \vdash (\Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12})[\sigma_1]$ **is** $(\Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22})[\sigma_2]$ **in** $[\![(\Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22})[\sigma_2]]\!]$ — By definition of substitution

Case 18: $\dfrac{}{\Psi;\Gamma \vdash 1 \equiv 1 : \mathsf{type}}$ Fameq-Unit

$\Psi^-;\Gamma^- \vdash 1 \longleftrightarrow 1 : \mathsf{type}^-$ — By rule (unit)

$\Psi^-;\Gamma^- \vdash 1$ **is** 1 **in** $[\![1]\!]$ By definition of relation
$\Psi^-;\Gamma^- \vdash 1[\sigma_1]$ **is** $1[\sigma_2]$ **in** $[\![1[\sigma_2]]\!]$ By definition of substitution

□

We are almost done with proving completeness. We first need an easy lemma to show that identity substitutions are logically related to themselves.

Lemma 6.7.7 (Identity Substitutions are Logically Related) $\Psi^-;\Gamma^- \vdash \mathsf{id}_\Gamma$ **is** id_Γ **in** $[\![\Gamma^-]\!]$.

Proof

By induction on the structure of Γ.

□

Now we can eliminate the need for related substitutions in the main lemma, since we can always create an identity substitution, related to itself.

Theorem 6.7.8 (Definitionally Equal Terms are Logically Related)

1. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1$ **is** M_2 **in** $[\![\mathsf{A}^-]\!]$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1$ **is** A_2 **in** $[\![\mathsf{A}_2]\!]$.

Proof

Direct, by Lemma 6.7.6 and Lemma 6.7.7. □

Putting together the fact above that terms judged definitionally equal are related by the logical relation, and the fact that logically related terms are indeed judged equal by the algorithm, we get the required completeness property.

Theorem 6.7.9 (Completeness of Equality Algorithm)

1. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{A}^-$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$ *then* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \mathsf{K}^-$.

Proof

Direct, by Lemma 6.7.8 and Lemma 6.7.1. □

6.8 Soundness of the Algorithm and Canonical Forms

In this section we will try to prove our algorithm to be sound. We will need to first prove a few preliminary lemmas. The first one says that the weak-head reduction relation is sound. This is important because at base types the algorithm performs weak-head reductions.

Lemma 6.8.1 (Subject Reduction)

1. *If* $\mathsf{M}_1 \xrightarrow{\text{whr}} \mathsf{M}_2$ *and* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$ *then* $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$ *and* $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$.
2. *If* $\mathsf{A}_1 \xrightarrow{\text{whr}} \mathsf{A}_2$ *and* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$ *then* $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$ *and* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$.

Proof

By induction on the definition of weak head reduction.

□

The second lemma we require is a fact about what types erase to the simple types. Notice that the algorithm depends on the simple types we compare at. To relate this to the definitional equality judgment, we require the corresponding type to be of the same shape. However, due to the presence of redices at the type family level, the shape may not be exactly the same. Fortunately, we can say that the weak-head normal forms must at least be of the required shape.

Lemma 6.8.2 (Inversion on Erasure)

1. *If* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$ *and* $\mathsf{A}^- = 1^-$ *then* $\mathsf{A} \xrightarrow{\text{whr}}{}^* 1$ *and* $\Psi;\Gamma \vdash \mathsf{A} \equiv 1 : \mathsf{type}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$ *and* $\mathsf{A}^- = \tau_1 \to \tau_2$ *then* $\mathsf{A} \xrightarrow{\text{whr}}{}^* \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$ *and* $\Psi;\Gamma \vdash \mathsf{A} \equiv \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$.

3. If $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$ *and* $\mathsf{A}^- = \tau_1 \times \tau_2$ *then* $\mathsf{A} \xrightarrow{\text{whr}}{}^* \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$ *and* $\Psi;\Gamma \vdash \mathsf{A} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$.

Proof

By induction on the length of a weak head normal reduction starting from A.

Case 1: $\mathsf{A} \not\xrightarrow{\text{whr}}$

By Erasure Corresponds to Weak Head Normal Terms and Subject Reduction

Case 2: $\mathsf{A} \xrightarrow{\text{whr}} \mathsf{A}'$

By induction and transitivity

□

6.8.1 Canonical Forms

We will actually prove soundness and canonical forms properties together for our language. We now define canonical and atomic objects, families and kinds, which are a syntactic subset of the corresponding terms, defined by the following grammar.

Canonical Kinds	K^{N}	::=	type	kind of types
		|	$\Pi \mathsf{x}{:}\mathsf{A}^{\mathsf{N}}.\mathsf{K}^{\mathsf{N}}$	dependent product kind
Atomic Families	A^{A}	::=	a	family constants
		|	$\mathsf{A}^{\mathsf{A}}\,\mathsf{M}^{\mathsf{N}}$	family application
		|	$\Pi \mathsf{x}{:}\mathsf{A}^{\mathsf{N}}{}_1.\mathsf{A}^{\mathsf{N}}{}_2$	family of functions
		|	$\Sigma \mathsf{x}{:}\mathsf{A}^{\mathsf{N}}{}_1.\mathsf{A}^{\mathsf{N}}{}_2$	family of products
		|	1	unit type
Canonical Families	A^{N}	::=	A^{A}	atomic families
		|	$\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}^{\mathsf{N}}{}_2$	family level abstraction
Atomic Objects	M^{A}	::=	c	object constants
		|	x	object variables
		|	$\mathsf{M}^{\mathsf{A}}{}_1\,\mathsf{M}^{\mathsf{N}}{}_2$	object level application
		|	$\pi_i \mathsf{M}^{\mathsf{A}} \quad (i = 1, 2)$	projections from pairs
Canonical Objects	M^{N}	::=	M^{A}	atomic objects
		|	$\langle \mathsf{M}^{\mathsf{N}}{}_1, \mathsf{M}^{\mathsf{N}}{}_2 \rangle^{\mathsf{A}}$	pairs of objects
		|	$\langle\rangle$	unit object
		|	$\lambda \mathsf{x}{:}\mathsf{A}.\mathsf{M}^{\mathsf{N}}$	object functions

Notice that these are different from the original notion of canonical forms. The difference from the original canonical forms is that type annotations of abstractions at both term and family levels need not be in canonical form. A better term might be quasi-canonical forms, or almost canonical forms, but that term is already used in Harper and Pfenning for something else. Our definition is the same in spirit to the quasi-canonical form of Harper and Pfenning [HP00], but those forms elide type annotations on abstractions, so that the canonical forms do not belong syntactically to the language of LF.

We can now prove our main lemma, which implies both soundness of the algorithm for equality as well as existence of canonical forms.

Lemma 6.8.3 (Algorithm produces Canonical Mediating Terms)

1. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$, $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{A}^-$, *then there is a canonical object* M^{N} *such that* $\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{N}} : \mathsf{A}$, $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$ *and* $\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1$, $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$, *then there is an atomic object* M^{A} *such that* $\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{A}} : \mathsf{A}_1$, $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}^{\mathsf{A}} : \mathsf{A}_1$ *and* $\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}^{\mathsf{A}} : \mathsf{A}_1$ *and* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$ *and* $\mathsf{A}_1{}^- = \mathsf{A}_2{}^- = \tau$.
3. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$, $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \mathsf{K}^-$, *then there exists a canonical family* A^{N} *such that* $\Psi;\Gamma \vdash \mathsf{A}^{\mathsf{N}} : \mathsf{K}$, $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}^{\mathsf{N}} : \mathsf{K}$ *and* $\Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}^{\mathsf{N}} : \mathsf{K}$.

4. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}_1$, $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}_2$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$, *then there exists an atomic family* A^{A} *such that* $\Psi;\Gamma \vdash \mathsf{A}^{\mathsf{A}} : \mathsf{K}_1$, $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}^{\mathsf{A}} : \mathsf{K}_1$ *and* $\Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}^{\mathsf{A}} : \mathsf{K}_1$ *and* $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$ *and* ${\mathsf{K}_1}^- = {\mathsf{K}_2}^- = \kappa$.
5. *If* $\Psi;\Gamma \vdash \mathsf{K}_1 : \mathsf{kind}$, $\Psi;\Gamma \vdash \mathsf{K}_2 : \mathsf{kind}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-$, *then there exists a canonical kind* K^{N} *such that* $\Psi;\Gamma \vdash \mathsf{K}^{\mathsf{N}} : \mathsf{kind}$, $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}^{\mathsf{N}} : \mathsf{kind}$ *and* $\Psi;\Gamma \vdash \mathsf{K}_2 \equiv \mathsf{K}^{\mathsf{N}} : \mathsf{kind}$.

Proof

By induction on the algorithmic judgment.

We will show some representative cases.

Case 1: $$\frac{\mathsf{M}_1 \xrightarrow{\text{whr}} \mathsf{M}_2 \quad \Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M} : \mathsf{a}^-}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M} : \mathsf{a}^-}\ \textsc{Tobj-Whleft}$$

$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$	By Subject Reduction
$\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$	By Regularity
There exists a canonical object M^{N} such that	
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{N}} : \mathsf{A}$,	
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$,	
$\Psi;\Gamma \vdash \mathsf{M} \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$	By induction
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$	By rule (transitivity)

Case 2: $$\frac{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \mathsf{a}^-}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{a}^-}\ \textsc{Tobj-Struct}$$

There exists an atomic term M^{A} such that	
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{A}} : \mathsf{A}$,	
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}^{\mathsf{A}} : \mathsf{A}$,	
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}^{\mathsf{A}} : \mathsf{A}$	By induction

Case 3: $$\frac{\Psi^-;\Gamma^-, \mathsf{x}{:}\tau_1 \vdash \mathsf{M}_1\,\mathsf{x} \Longleftrightarrow \mathsf{M}_2\,\mathsf{x} : \tau_2}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau_1 \to \tau_2}\ \textsc{Tobj-Arrow}$$

$\Psi;\Gamma \vdash \mathsf{A} \equiv \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$,	
${\mathsf{A}_1}^- = \tau_1$ and ${\mathsf{A}_2}^- = \tau_2$	By Inversion on Erasure
$\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$	By Regularity
$\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type}$	By Inversion on Typing
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}$	By Inversion on Typing
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1\,\mathsf{x} : \mathsf{A}_2$	By rule (application typing)
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_2\,\mathsf{x} : \mathsf{A}_2$	By rule (application typing)
There exists a canonical object M^{N} such that	
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}^{\mathsf{N}} : \mathsf{A}_2$,	
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1\,\mathsf{x} \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}_2$	By induction
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_2\,\mathsf{x} \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}_2$	By induction
$\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}^{\mathsf{N}} : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	By rule
$\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}^{\mathsf{N}} : \mathsf{A}$	By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}^{\mathsf{N}} : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	By rule (extensionality)
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}^{\mathsf{N}} : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	By rule (extensionality)
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}^{\mathsf{N}} : \mathsf{A}$	By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}^{\mathsf{N}} : \mathsf{A}$	By rule (type conversion)

Case 4: $\dfrac{\Psi^-;\Gamma^- \vdash \pi_1 \mathsf{M}_1 \Longleftrightarrow \pi_1 \mathsf{M}_2 : \tau_1 \quad \Psi^-;\Gamma^- \vdash \pi_2 \mathsf{M}_1 \Longleftrightarrow \pi_2 \mathsf{M}_2 : \tau_2}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau_1 \times \tau_2}$ Tobj-Prod

$\Psi;\Gamma \vdash \mathsf{A} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$,	
${\mathsf{A}_1}^- = \tau_1$ and ${\mathsf{A}_2}^- = \tau_2$	By Inversion on Erasure
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$	By Regularity
$\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type}$	By Inversion on Typing
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}$	By Inversion on Typing
$\Psi;\Gamma \vdash \mathsf{M}_1 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	By assumption
$\Psi;\Gamma \vdash \mathsf{M}_2 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	By assumption
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_1 : \mathsf{A}_1$	By rule (projection)
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_2 : \mathsf{A}_1$	By rule (projection)
$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_1 : [\pi_1 \mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2$	By rule (projection)
$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_2 : [\pi_1 \mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_2$	By rule (projection)
There exists a canonical object ${\mathsf{M}^{\mathsf{N}}}_1$ such that	
$\Psi;\Gamma \vdash {\mathsf{M}^{\mathsf{N}}}_1 : \mathsf{A}_1$,	
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_1 \equiv {\mathsf{M}^{\mathsf{N}}}_1 : \mathsf{A}_1$,	
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_2 \equiv {\mathsf{M}^{\mathsf{N}}}_1 : \mathsf{A}_1$	By induction
$\Psi;\Gamma \vdash [\pi_1 \mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_2 \equiv [\pi_1 \mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2 : \mathsf{type}$	By Functionality
$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_2 : [\pi_1 \mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2$	By rule (type conversion)
There exists a canonical object ${\mathsf{M}^{\mathsf{N}}}_2$ such that	
$\Psi;\Gamma \vdash {\mathsf{M}^{\mathsf{N}}}_2 : [\pi_1 \mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2$,	
$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_1 \equiv {\mathsf{M}^{\mathsf{N}}}_2 : [\pi_1 \mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2$,	
$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_2 \equiv {\mathsf{M}^{\mathsf{N}}}_2 : [\pi_1 \mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2$	By induction
$\Psi;\Gamma \vdash \langle {\mathsf{M}^{\mathsf{N}}}_1, {\mathsf{M}^{\mathsf{N}}}_2 \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	By rule
$\Psi;\Gamma \vdash \langle {\mathsf{M}^{\mathsf{N}}}_1, {\mathsf{M}^{\mathsf{N}}}_2 \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \mathsf{A}$	By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \langle {\mathsf{M}^{\mathsf{N}}}_1, {\mathsf{M}^{\mathsf{N}}}_2 \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	By rule (extensionality)
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \langle {\mathsf{M}^{\mathsf{N}}}_1, {\mathsf{M}^{\mathsf{N}}}_2 \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	By rule (extensionality)
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \langle {\mathsf{M}^{\mathsf{N}}}_1, {\mathsf{M}^{\mathsf{N}}}_2 \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \mathsf{A}$	By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \langle {\mathsf{M}^{\mathsf{N}}}_1, {\mathsf{M}^{\mathsf{N}}}_2 \rangle^{\Sigma \mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \mathsf{A}$	By rule (type conversion)

Case 5: $\dfrac{}{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : 1}$ Tobj-Unit

$\Psi;\Gamma \vdash \mathsf{A} \equiv 1 : \mathsf{type}$	By Inversion on Erasure
$\Psi;\Gamma \vdash \mathsf{M}_1 : 1$	By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_2 : 1$	By rule (type conversion)
$\Psi;\Gamma \vdash \langle\rangle : 1$	By rule
$\Psi;\Gamma \vdash \langle\rangle : \mathsf{A}$	By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \langle\rangle : 1$	By rule
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \langle\rangle : 1$	By rule
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \langle\rangle : \mathsf{A}$	By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \langle\rangle : \mathsf{A}$	By rule (type conversion)

Case 6: $\dfrac{\Gamma^-(\mathsf{x}) = \tau}{\Psi^-;\Gamma^- \vdash \mathsf{x} \longleftrightarrow \mathsf{x} : \tau}$ Sobj-Var

$\Psi;\Gamma \vdash \mathsf{x} : \mathsf{A}_1$	By assumption
$\Psi;\Gamma \vdash \mathsf{x} : \mathsf{A}_2$	By assumption
$\Gamma(\mathsf{x}) = \mathsf{A}$, $\Psi;\Gamma \vdash \mathsf{A} \equiv \mathsf{A}_1 : \mathsf{type}$, $\Psi;\Gamma \vdash \mathsf{A} \equiv \mathsf{A}_2 : \mathsf{type}$	By Inversion
$\Psi;\Gamma \vdash \mathsf{x} \equiv \mathsf{x} : \mathsf{A}$	By rule (variable equality)

$\Psi;\Gamma \vdash \mathsf{x} \equiv \mathsf{x} : \mathsf{A}_1$ By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$ By rules (symmetry, transitivity)
$\mathsf{A}_1{}^- = \mathsf{A}_2{}^- = \mathsf{A}^- = \tau$ By Erasure Preservation

Case 7: $\dfrac{\Psi^-;\Gamma^- \vdash \mathsf{M}_{11} \longleftrightarrow \mathsf{M}_{21} : \tau_2 \to \tau_1 \quad \Psi^-;\Gamma^- \vdash \mathsf{M}_{12} \longleftrightarrow \mathsf{M}_{22} : \tau_2}{\Psi^-;\Gamma^- \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} \longleftrightarrow \mathsf{M}_{21}\,\mathsf{M}_{22} : \tau_1}$ Sobj-App

$\Psi;\Gamma \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} : \mathsf{A}_1$ By assumption
$\Psi;\Gamma \vdash \mathsf{M}_{21}\,\mathsf{M}_{22} : \mathsf{A}_2$ By assumption
$\Psi;\Gamma \vdash \mathsf{M}_{11} : \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12}$,
$\Psi;\Gamma \vdash \mathsf{M}_{12} : \mathsf{A}_{11}$,
$\Psi;\Gamma \vdash [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_{12} \equiv \mathsf{A}_1 : \mathsf{type}$ By typing inversion
$\Psi;\Gamma \vdash \mathsf{M}_{21} : \Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22}$,
$\Psi;\Gamma \vdash \mathsf{M}_{22} : \mathsf{A}_{21}$,
$\Psi;\Gamma \vdash [\mathsf{M}_{22}/\mathsf{x}]\,\mathsf{A}_{22} \equiv \mathsf{A}_2 : \mathsf{type}$ By typing inversion
There is an atomic object M^{A} such that
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{A}} : \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12}$,
$\Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}^{\mathsf{A}} : \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12}$,
$\Psi;\Gamma \vdash \mathsf{M}_{21} \equiv \mathsf{M}^{\mathsf{A}} : \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12}$,
$\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}$
$(\Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12})^- = (\Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22})^- = \tau_2 \to \tau_1$ By induction
$\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type}$,
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{22} : \mathsf{type}$ By Injectivity of Products
$\mathsf{A}_{11}{}^- = \mathsf{A}_{21}{}^- = \tau_2$ By definition of $()^-$
$\mathsf{A}_{12}{}^- = \mathsf{A}_{22}{}^- = \tau_1$ By definition of $()^-$
$\Psi;\Gamma \vdash \mathsf{M}_{22} : \mathsf{A}_{11}$ By rule (type conversion)
There exists a canonical object M^{N} such that
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{N}} : \mathsf{A}_{11}$,
$\Psi;\Gamma \vdash \mathsf{M}_{12} \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}_{11}$,
$\Psi;\Gamma \vdash \mathsf{M}_{22} \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}_{11}$ By induction
$\Psi;\Gamma \vdash \left[\mathsf{M}^{\mathsf{N}}/\mathsf{x}\right]\,\mathsf{A}_{12} \equiv [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_{12} : \mathsf{type}$ By Functionality
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{A}}\,\mathsf{M}^{\mathsf{N}} : \left[\mathsf{M}^{\mathsf{N}}/\mathsf{x}\right]\,\mathsf{A}_{12}$ By rule (application)
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{A}}\,\mathsf{M}^{\mathsf{N}} : [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_{12}$ By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} \equiv \mathsf{M}^{\mathsf{A}}\,\mathsf{M}^{\mathsf{N}} : [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_{12}$ By rule (application)
$\Psi;\Gamma \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} \equiv \mathsf{M}^{\mathsf{A}}\,\mathsf{M}^{\mathsf{N}} : \mathsf{A}_1$ By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{M}_{21}\,\mathsf{M}_{22} \equiv \mathsf{M}^{\mathsf{A}}\,\mathsf{M}^{\mathsf{N}} : [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_{12}$ By rule (application)
$\Psi;\Gamma \vdash \mathsf{M}_{21}\,\mathsf{M}_{22} \equiv \mathsf{M}^{\mathsf{A}}\,\mathsf{M}^{\mathsf{N}} : \mathsf{A}_1$ By rule (type conversion)
$\Psi;\Gamma \vdash [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_{12} \equiv [\mathsf{M}_{22}/\mathsf{x}]\,\mathsf{A}_{22} : \mathsf{type}$ By Functionality
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$ By rules (symmetry, transitivity)
$\mathsf{A}_1{}^- = \mathsf{A}_{12}{}^- = \mathsf{A}_{22}{}^- = \mathsf{A}_2{}^- = \tau_1$ By Erasure Preservation

Case 8: $\dfrac{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau_1 \times \tau_2}{\Psi^-;\Gamma^- \vdash \pi_1\mathsf{M}_1 \longleftrightarrow \pi_1\mathsf{M}_2 : \tau_1}$ Sobj-Proj

$\Psi;\Gamma \vdash \pi_1\mathsf{M}_1 : \mathsf{A}_1$ By assumption
$\Psi;\Gamma \vdash \pi_1\mathsf{M}_2 : \mathsf{A}_2$ By assumption
$\Psi;\Gamma \vdash \mathsf{M}_1 : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_{11}$ By Inversion
$\Psi;\Gamma \vdash \mathsf{M}_2 : \Sigma\mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_{21}$ By Inversion
There exists an atomic object M^{A} such that
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{A}} : \Sigma\mathsf{x}{:}\mathsf{A}_3.\mathsf{A}_{31}$,

$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}^{\mathsf{A}} : \Sigma \mathsf{x}{:}\mathsf{A}_3.\mathsf{A}_{31}$,
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}^{\mathsf{A}} : \Sigma \mathsf{x}{:}\mathsf{A}_3.\mathsf{A}_{31}$,
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_{11} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_3.\mathsf{A}_{31} : \mathsf{type}$,
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_{21} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_3.\mathsf{A}_{31} : \mathsf{type}$,
$\Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_{11}{}^- = \Sigma \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_{21}{}^- = \Sigma \mathsf{x}{:}\mathsf{A}_3.\mathsf{A}_{31}{}^- = \tau_1 \times \tau_2$ By induction
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}^{\mathsf{A}} : \mathsf{A}_3$ By rule
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_1 \equiv \pi_1 \mathsf{M}^{\mathsf{A}} : \mathsf{A}_3$ By rule (projection)
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_2 \equiv \pi_1 \mathsf{M}^{\mathsf{A}} : \mathsf{A}_3$ By rule (projection)
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_3 : \mathsf{type}$ By Injectivity
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_1 \equiv \pi_1 \mathsf{M}^{\mathsf{A}} : \mathsf{A}_1$ By rule (type conversion)
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_2 \equiv \pi_1 \mathsf{M}^{\mathsf{A}} : \mathsf{A}_1$ By rule (type conversion)
$\Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_3 : \mathsf{type}$ By Injectivity
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$ By rules (symmetry, transitivity)
$\mathsf{A}_1{}^- = \mathsf{A}_2{}^- = \mathsf{A}_3{}^- = \tau_1$ By Erasure Preservation

Case 9: $\dfrac{\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau_1 \times \tau_2}{\Psi^-;\Gamma^- \vdash \pi_2 \mathsf{M}_1 \longleftrightarrow \pi_2 \mathsf{M}_2 : \tau_2}$ Sobj-Proj

$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_1 : \mathsf{A}_1$ By assumption
$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_2 : \mathsf{A}_2$ By assumption
$\Psi;\Gamma \vdash \mathsf{M}_1 : \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12}$,
$\Psi;\Gamma \vdash [\pi_1 \mathsf{M}_1/\mathsf{x}]\, \mathsf{A}_{12} \equiv \mathsf{A}_1 : \mathsf{type}$ By Inversion
$\Psi;\Gamma \vdash \mathsf{M}_2 : \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22}$,
$\Psi;\Gamma \vdash [\pi_1 \mathsf{M}_2/\mathsf{x}]\, \mathsf{A}_{22} \equiv \mathsf{A}_2 : \mathsf{type}$ By Inversion
There exists an atomic object M^{A} such that
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{A}} : \Sigma \mathsf{x}{:}\mathsf{A}_{31}.\mathsf{A}_{32}$,
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}^{\mathsf{A}} : \Sigma \mathsf{x}{:}\mathsf{A}_{31}.\mathsf{A}_{32}$,
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}^{\mathsf{A}} : \Sigma \mathsf{x}{:}\mathsf{A}_{31}.\mathsf{A}_{32}$,
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_{31}.\mathsf{A}_{32} : \mathsf{type}$,
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_{31}.\mathsf{A}_{32} : \mathsf{type}$,
$(\Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12})^- = (\Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22})^- = (\Sigma \mathsf{x}{:}\mathsf{A}_{31}.\mathsf{A}_{32})^- = \tau_1 \times \tau_2$ By induction
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_1 \equiv \pi_1 \mathsf{M}^{\mathsf{A}} : \mathsf{A}_{31}$ By rule (projection)
$\Psi;\Gamma \vdash \pi_1 \mathsf{M}_2 \equiv \pi_1 \mathsf{M}^{\mathsf{A}} : \mathsf{A}_{31}$ By rule (projection)
$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_1 \equiv \pi_2 \mathsf{M}^{\mathsf{A}} : [\pi_1 \mathsf{M}_1/\mathsf{x}]\, \mathsf{A}_{32}$ By rule (projection)
$\Psi;\Gamma \vdash \pi_2 \mathsf{M}_2 \equiv \pi_2 \mathsf{M}^{\mathsf{A}} : [\pi_1 \mathsf{M}_2/\mathsf{x}]\, \mathsf{A}_{32}$ By rule (projection)
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{31} \vdash \mathsf{A}_{12} \equiv \mathsf{A}_{32} : \mathsf{type}$,
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{31} \vdash \mathsf{A}_{22} \equiv \mathsf{A}_{32} : \mathsf{type}$, By Injectivity
$\Psi;\Gamma \vdash [\pi_1 \mathsf{M}_1/\mathsf{x}]\, \mathsf{A}_{12} \equiv [\pi_1 \mathsf{M}_2/\mathsf{x}]\, \mathsf{A}_{22} : \mathsf{type}$ By Functionality
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$ By rules (symmetry, transitivity)
$\mathsf{A}_1{}^- = \mathsf{A}_2{}^- = \mathsf{A}_{32}{}^- = \tau_2$ By Erasure Preservation

Case 10: $\dfrac{\Psi^-;\Gamma^- \vdash \mathsf{A}_{11} \Longleftrightarrow \mathsf{A}_{21} : \mathsf{type}^- \quad \Psi^-;\Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \vdash \mathsf{A}_{12} \Longleftrightarrow \mathsf{A}_{22} : \mathsf{type}^-}{\Psi^-;\Gamma^- \vdash \Pi \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \longleftrightarrow \Pi \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}^-}$ Sfam-Pi

$\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} : \mathsf{K}_1$ By assumption
$\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{K}_2$ By assumption
$\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{type} : \mathsf{kind}$,
$\Psi;\Gamma \vdash \mathsf{K}_2 \equiv \mathsf{type} : \mathsf{kind}$ By Inversion
$\Psi;\Gamma \vdash \mathsf{A}_{11} : \mathsf{type}$,
$\Psi;\Gamma \vdash \mathsf{A}_{21} : \mathsf{type}$ By Inversion

There is a canonical family $\mathsf{A^N}_1$ such that
$\Psi;\Gamma \vdash \mathsf{A^N}_1 : \mathsf{type}$,
$\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A^N} : \mathsf{type}$,
$\Psi;\Gamma \vdash \mathsf{A}_{21} \equiv \mathsf{A^N} : \mathsf{type}$ By induction
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} : \mathsf{type}$,
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{21} \vdash \mathsf{A}_{22} : \mathsf{type}$ By inversion
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A}_{22} : \mathsf{type}$ By Context Conversion
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A}_{22} : \mathsf{type}$ By Context Conversion
There is a canonical family $\mathsf{A^N}_2$ such that
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A^N}_2 : \mathsf{type}$,
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A}_{12} \equiv \mathsf{A^N}_2 : \mathsf{type}$ By induction
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A}_{22} \equiv \mathsf{A^N}_2 : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A^N}_1.\mathsf{A^N}_2 : \mathsf{type}$ By rule
$\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Pi \mathsf{x}{:}\mathsf{A^N}_1.\mathsf{A^N}_2 : \mathsf{type}$ By rule (Product Equality)
$\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} \equiv \Pi \mathsf{x}{:}\mathsf{A^N}_1.\mathsf{A^N}_2 : \mathsf{type}$ By rule (Product Equality)
$\mathsf{K}_1{}^- = \mathsf{K}_2{}^- = \mathsf{type}^-$ By Erasure Preservation

Case 11:
$$\frac{\Psi^-;\Gamma^- \vdash \mathsf{A}_{11} \Longleftrightarrow \mathsf{A}_{21} : \mathsf{type}^- \quad \Psi;\Psi^- \vdash \Gamma^-, \mathsf{x}{:}\mathsf{A}_{11}{}^- \Longleftrightarrow \mathsf{A}_{12} : \mathsf{A}_{22}\mathsf{type}^-}{\Psi^-;\Gamma^- \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \longleftrightarrow \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}^-}\ \textsc{Sfam-Sigma}$$

$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} : \mathsf{K}_1$ By assumption
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{K}_2$ By assumption
$\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{type} : \mathsf{kind}$,
$\Psi;\Gamma \vdash \mathsf{K}_2 \equiv \mathsf{type} : \mathsf{kind}$ By Inversion
$\Psi;\Gamma \vdash \mathsf{A}_{11} : \mathsf{type}$,
$\Psi;\Gamma \vdash \mathsf{A}_{21} : \mathsf{type}$ By Inversion
There is a canonical family $\mathsf{A^N}_1$ such that
$\Psi;\Gamma \vdash \mathsf{A^N}_1 : \mathsf{type}$,
$\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A^N} : \mathsf{type}$,
$\Psi;\Gamma \vdash \mathsf{A}_{21} \equiv \mathsf{A^N} : \mathsf{type}$ By induction
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} : \mathsf{type}$,
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{21} \vdash \mathsf{A}_{22} : \mathsf{type}$ By inversion
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A}_{22} : \mathsf{type}$ By Context Conversion
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A}_{22} : \mathsf{type}$ By Context Conversion
There is a canonical family $\mathsf{A^N}_2$ such that
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A^N}_2 : \mathsf{type}$,
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A}_{12} \equiv \mathsf{A^N}_2 : \mathsf{type}$ By induction
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A^N}_1 \vdash \mathsf{A}_{22} \equiv \mathsf{A^N}_2 : \mathsf{type}$ By induction
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A^N}_1.\mathsf{A^N}_2 : \mathsf{type}$ By rule
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} \equiv \Sigma \mathsf{x}{:}\mathsf{A^N}_1.\mathsf{A^N}_2 : \mathsf{type}$ By rule (Product Equality)
$\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} \equiv \Sigma \mathsf{x}{:}\mathsf{A^N}_1.\mathsf{A^N}_2 : \mathsf{type}$ By rule (Product Equality)
$\mathsf{K}_1{}^- = \mathsf{K}_2{}^- = \mathsf{type}^-$ By Erasure Preservation

□

Theorem 6.8.4 (Soundness)

1. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$, $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{A}^-$, *then* $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1$, $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$, *then* $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_1$, $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$ *and* $\mathsf{A}_1{}^- = \mathsf{A}_2{}^- = \tau$.
3. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$, $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \mathsf{K}^-$, *then* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$.
4. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}_1$, $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}_2$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$, *then* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}_1$, $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$ *and* $\mathsf{K}_1{}^- = \mathsf{K}_2{}^- = \kappa$.
5. *If* $\Psi;\Gamma \vdash \mathsf{K}_1 : \mathsf{kind}$, $\Psi;\Gamma \vdash \mathsf{K}_2 : \mathsf{kind}$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-$, *then* $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$.

Proof

Directly, from previous lemma. We will show the cases for objects, the cases for families and kinds are similar.

Case 1:

$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$,
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$ By Canonical Mediating Terms for Algorithm
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{N}} \equiv \mathsf{M}_2 : \mathsf{A}$ By rule (symmetry)
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$ By rule (transitivity)

Case 2:

$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}^{\mathsf{A}} : \mathsf{A}_1$,
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}^{\mathsf{A}} : \mathsf{A}_1$,
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$,
${\mathsf{A}_1}^- = {\mathsf{A}_2}^- = \tau$ By Canonical Mediating Terms for Algorithm
$\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{A}} \equiv \mathsf{M}_2 : \mathsf{A}_1$ By rule (symmetry)
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_1$ By rule (transitivity)

□

We can now also prove canonical forms by appeal to lemma 6.8.3.

Theorem 6.8.5 (Canonical Forms)

1. *If* $\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}$ *then there exists a canonical object* M^{N} *such that* $\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{N}} : \mathsf{A}$ *and* $\Psi;\Gamma \vdash \mathsf{M} \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}$ *then there exists a canonical family* A^{N} *such that* $\Psi;\Gamma \vdash \mathsf{A}^{\mathsf{N}} : \mathsf{K}$ *and* $\Psi;\Gamma \vdash \mathsf{A} \equiv \mathsf{A}^{\mathsf{N}} : \mathsf{K}$.
3. *If* $\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}$ *then there exists a canonical kind* K^{N} *such that* $\Psi;\Gamma \vdash \mathsf{K}^{\mathsf{N}} : \mathsf{kind}$ *and* $\Psi;\Gamma \vdash \mathsf{K} \equiv \mathsf{K}^{\mathsf{N}} : \mathsf{kind}$.

Proof

Direct, from Canonical Mediating Terms for Equality Algorithm. We will show the case for objects.

$\Psi;\Gamma \vdash \mathsf{M} \equiv \mathsf{M} : \mathsf{A}$ By Reflexivity
$\Psi^-;\Gamma^- \vdash \mathsf{M} \Longleftrightarrow \mathsf{M} : \mathsf{A}^-$ By Completeness of Algorithm
There exists a canonical object M^{N} such that $\Psi;\Gamma \vdash \mathsf{M}^{\mathsf{N}} : \mathsf{A}$,
$\Psi;\Gamma \vdash \mathsf{M} \equiv \mathsf{M}^{\mathsf{N}} : \mathsf{A}$ By Canonical Mediating Terms for Equality Algorithm

□

6.9 Decidability of Type Checking

Terms which are related by the algorithm for equality are called by Harper and Pfenning "normalizing". This terminology is justified by our result on canonical forms, since these terms are provably equal to canonical forms. We first prove that equality between normalizing terms is decidable. This will imply decidability of equality for all well-typed terms, by the use of completeness of the algorithm for deciding equality.

Lemma 6.9.1 (Decidability of Equality for Normalizing Terms)

1. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \tau$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_3 \Longleftrightarrow \mathsf{M}_4 : \tau$ *then it is decidable whether* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_3 : \tau$.
2. *If* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau_1$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{M}_3 \longleftrightarrow \mathsf{M}_4 : \tau_2$ *then it is decidable whether* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_3 : \tau_3$ *for any* τ_3.
3. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \kappa$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_3 \Longleftrightarrow \mathsf{A}_4 : \kappa$ *then it is decidable whether* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_3 : \kappa$.
4. *If* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa_1$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{A}_3 \longleftrightarrow \mathsf{A}_4 : \kappa_2$ *then it is decidable whether* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_3 : \kappa_3$ *for any* κ_3.

5. *If* $\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-$ *and* $\Psi^-;\Gamma^- \vdash \mathsf{K}_3 \longleftrightarrow \mathsf{K}_4 : \mathsf{kind}^-$ *then it is decidable whether* $\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_3 : \mathsf{kind}^-$.

Proof

By structural induction on the derivation of the judgment.

□

Theorem 6.9.2 (Decidability of Algorithmic Equality)

1. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$ *and* $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$*, then it is decidable whether* $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{A}^-$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$ *and* $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$*, then it is decidable whether* $\Psi^-;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \mathsf{K}^-$.
3. *If* $\Psi;\Gamma \vdash \mathsf{K}_1 : \mathsf{kind}$ *and* $\Psi;\Gamma \vdash \mathsf{K}_2 : \mathsf{kind}$*, then it is decidable whether* $\Psi^-;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-$.

Proof

Direct from decidability for normalizing terms, using reflexivity of equality. We show one case, others are analogous.

$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_1 : \mathsf{A}$ By Reflexivity of Equality
$\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_1 : \mathsf{A}^-$ By Completeness of Algorithmic Equality
$\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_2 : \mathsf{A}$ By Reflexivity of Equality
$\Psi^-;\Gamma^- \vdash \mathsf{M}_2 \Longleftrightarrow \mathsf{M}_2 : \mathsf{A}^-$ By Completeness of Algorithmic Equality
It is decidable whether $\Psi^-;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : \mathsf{A}^-$
By Decidability of Algorithmic Equality for Normalizing Terms

□

Corollary 6.9.3 (Decidability of Definitional Equality)

1. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$ *and* $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$*, then it is decidable whether* $\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$ *and* $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$*, then it is decidable whether* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$.
3. *If* $\Psi;\Gamma \vdash \mathsf{K}_1 : \mathsf{kind}$ *and* $\Psi;\Gamma \vdash \mathsf{K}_2 : \mathsf{kind}$*, then it is decidable whether* $\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$.

Proof

By soundness and completeness of algorithmic equality, it is enough to have decidability of algorithmic equality. □

6.9.1 Algorithm for Type Checking

We now use the algorithm for equality as a subroutine to provide an algorithm for type checking in figures 6.15, 6.16 and 6.17. These algorithms are directed by the structure of the term, and produce the classifier (types and kinds) if the term is well-typed. At various points, the algorithm must decide equality of the types, notably in the application and pair formation rules. Here we use the algorithm for equality as a subroutine, knowing the two types to be compared are both well-kinded at the same kind.

It is easy to show the soundness and completeness of the algorithm for typechecking.

Lemma 6.9.4 (Soundness of Algorithmic Typechecking)

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

Proof

By structural induction on the derivation.

□

To show completeness under the presence of a nontrivial equality at the type level (induced by the presence of type-level abstractions), we need the following technical lemma.

Lemma 6.9.5 (Inversion on Product and Sum Families)

$$\frac{\Gamma(\mathsf{x}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{x} \Rightarrow \mathsf{A}}\ \textsc{OAlg-Var} \qquad \frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{c} \Rightarrow \mathsf{A}}\ \textsc{OAlg-Const}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{M}_1 \Rightarrow \mathsf{A} \\ \mathsf{A} \xrightarrow{\text{whr}}{}^{*} \Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_1 \\ \Psi;\Gamma \vdash \mathsf{M}_2 \Rightarrow \mathsf{A}_{22} \\ \Psi^-;\Gamma^- \vdash \mathsf{A}_{21} \Longleftrightarrow \mathsf{A}_{22} : \mathsf{type}^-\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1\,\mathsf{M}_2 \Rightarrow [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_1}\ \textsc{OAlg-App} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{A}_1 \Rightarrow \mathsf{type} \\ \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_2 \Rightarrow \mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_2 \Rightarrow \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{OAlg-Fun}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \Rightarrow \mathsf{type} \qquad \Psi;\Gamma \vdash \mathsf{M}_1 \Rightarrow \mathsf{A}_{11} \\ \Psi^-;\Gamma^- \vdash \mathsf{A}_{11} \Longleftrightarrow \mathsf{A}_1 : \mathsf{type}^- \qquad \Psi;\Gamma \vdash \mathsf{M}_2 \Rightarrow \mathsf{A}_{21} \\ \Psi^-;\Gamma^- \vdash \mathsf{A} \Longleftrightarrow [\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2 : \mathsf{type}^-\end{array}}{\Psi;\Gamma \vdash \langle \mathsf{M}_1, \mathsf{M}_2 \rangle^{\Sigma\mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} \Rightarrow \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}\ \textsc{OAlg-Pair} \qquad \frac{}{\Psi;\Gamma \vdash \langle\rangle \Rightarrow 1}\ \textsc{OAlg-Unit}$$

$$\frac{\begin{array}{l}\Psi(\mathsf{X}_\mathsf{M}) = \Gamma_1 \vdash \mathsf{A} \\ \Psi;\Gamma_2 \vdash \sigma : \Gamma_1\end{array}}{\Psi;\Gamma_2 \vdash \mathsf{X}_\mathsf{M}[\sigma] \Rightarrow [\sigma]\mathsf{A}}\ \textsc{OAlg-Meta}$$

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{M} \Rightarrow \mathsf{A} \\ \mathsf{A} \xrightarrow{\text{whr}}{}^{*} \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \pi_1\mathsf{M} \Rightarrow \mathsf{A}_1}\ \textsc{OAlg-Proj1} \qquad \frac{\begin{array}{l}\Psi;\Gamma \vdash \mathsf{M} \Rightarrow \mathsf{A} \\ \mathsf{A} \xrightarrow{\text{whr}}{}^{*} \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \pi_2\mathsf{M} \Rightarrow [\pi_1\mathsf{M}/\mathsf{x}]\,\mathsf{A}_2}\ \textsc{OAlg-Proj2}$$

Figure 6.15: Algorithmic type checking:Objects

$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{a} \Rightarrow \mathsf{K}}\ \textsc{FAlg-Const} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A}_1 \Rightarrow \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 \Rightarrow \mathsf{K}}{\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \Rightarrow \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{K}}\ \textsc{FAlg-Abs}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 \Rightarrow \Pi \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{K}_1 \quad \Psi;\Gamma \vdash \mathsf{M} \Rightarrow \mathsf{A}_{22} \quad \Psi^-;\Gamma^- \vdash \mathsf{A}_{21} \Longleftrightarrow \mathsf{A}_{22} : {\mathsf{K}_1}^-}{\Psi;\Gamma \vdash \mathsf{A}_1\,\mathsf{M} \Rightarrow [\mathsf{M}/\mathsf{x}]\,\mathsf{K}_1}\ \textsc{FAlg-App} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A}_1 \Rightarrow \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 \Rightarrow \mathsf{type}}{\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \Rightarrow \mathsf{type}}\ \textsc{FAlg-Pi}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 \Rightarrow \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 \Rightarrow \mathsf{type}}{\Psi;\Gamma \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \Rightarrow \mathsf{type}}\ \textsc{FAlg-Sigma} \qquad \frac{}{\Psi;\Gamma \vdash 1 \Rightarrow \mathsf{type}}\ \textsc{FAlg-Unit}$$

$$\frac{\Psi(\mathsf{X}_\mathsf{A}) = \Gamma_1 \vdash \mathsf{K} \quad \Psi;\Gamma_2 \vdash \sigma : \Gamma_1}{\Psi;\Gamma_2 \vdash \mathsf{X}_\mathsf{A}[\sigma] \Rightarrow [\sigma]\mathsf{K}}\ \textsc{FAlg-Meta}$$

Figure 6.16: Algorithmic type checking:Families

$$\frac{}{\Psi;\Gamma \vdash \mathsf{type} \Rightarrow \mathsf{kind}}\ \textsc{LAlg-Type} \qquad \frac{\Psi;\Gamma \vdash \mathsf{A} \Rightarrow \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{K} \Rightarrow \mathsf{kind}}{\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K} \Rightarrow \mathsf{kind}}\ \textsc{LAlg-Pi}$$

Figure 6.17: Algorithmic type checking:Kinds

1. *If we have* $\Psi;\Gamma \vdash \mathsf{A} \equiv \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$ *then* $\mathsf{A} \xrightarrow{\text{whr}}{}^{*} \Pi \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{21}$ *such that* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_{11} : \mathsf{type}$ *and* $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 \equiv \mathsf{A}_{21} : \mathsf{type}$.
2. *If we have* $\Psi;\Gamma \vdash \mathsf{A} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$ *then* $\mathsf{A} \xrightarrow{\text{whr}}{}^{*} \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{21}$ *such that* $\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_{11} : \mathsf{type}$ *and* $\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 \equiv \mathsf{A}_{21} : \mathsf{type}$.

Proof

By Erasure Preservation under Equality together with Inversion on Erasure, and using Injectivity.

□

Lemma 6.9.6 (Completeness of Algorithmic Typechecking)

1. *If* $\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}$ *then* $\Psi;\Gamma \vdash \mathsf{M} \Rightarrow \mathsf{A}'$ *and* $\Psi;\Gamma \vdash \mathsf{A} \equiv \mathsf{A}' : \mathsf{type}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}$ *then* $\Psi;\Gamma \vdash \mathsf{A} \Rightarrow \mathsf{K}'$ *and* $\Psi;\Gamma \vdash \mathsf{K} \equiv \mathsf{K}' : \mathsf{kind}$.
3. *If* $\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}$ *then* $\Psi;\Gamma \vdash \mathsf{K} \Rightarrow \mathsf{kind}$.

Proof

By induction on the derivation. We show two significant cases.

Case 1: $\dfrac{\Psi;\Gamma \vdash \mathsf{M}_1 : \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{M}_1\,\mathsf{M}_2 : [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_1}$ O-App

$\Psi;\Gamma \vdash \mathsf{M}_1 \Rightarrow \mathsf{A}_{11}$,	
$\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 : \mathsf{type}$	By induction
$\mathsf{A}_{11} \xrightarrow{\text{whr}}{}^{*} \Pi \mathsf{x}{:}\mathsf{A}_2'.\mathsf{A}_1'$ $\Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_2' : \mathsf{type}$,	
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_2 \vdash \mathsf{A}_1 \equiv \mathsf{A}_1' : \mathsf{type}$	By Inversion on Product Families
$\Psi;\Gamma \vdash \mathsf{M}_2 \Rightarrow \mathsf{A}_{21}$,	
$\Psi;\Gamma \vdash \mathsf{A}_{21} \equiv \mathsf{A}_2 : \mathsf{type}$	By induction
$\Psi;\Gamma \vdash \mathsf{A}_{21} \equiv \mathsf{A}_2' : \mathsf{type}$	By rule (transitivity)
$\Psi^{-};\Gamma^{-} \vdash \mathsf{A}_{21} \Longleftrightarrow \mathsf{A}_2' : \mathsf{type}^{-}$	By Completeness of Algorithmic Equality
$\Psi;\Gamma \vdash \mathsf{M}_1\,\mathsf{M}_2 \Rightarrow [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_1'$	By rule
$\Psi;\Gamma \vdash [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_1' \equiv [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_1 : \mathsf{type}$	By Functionality

Case 2: $\dfrac{\Psi;\Gamma \vdash \mathsf{M} : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_1 \mathsf{M} : \mathsf{A}_1}$ O-Proj1

$\Psi;\Gamma \vdash \mathsf{M} \Rightarrow \mathsf{A}$,	
$\Psi;\Gamma \vdash \mathsf{A} \equiv \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}$	By induction
$\mathsf{A} \xrightarrow{\text{whr}}{}^{*} \Sigma \mathsf{x}{:}\mathsf{A}_1'.\mathsf{A}_2'$,	
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_1' : \mathsf{type}$,	
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 \equiv \mathsf{A}_2' : \mathsf{type}$	By Inversion on Sum Families
$\Psi;\Gamma \vdash \pi_1 \mathsf{M} \Rightarrow \mathsf{A}_1'$	By rule

□

6.10 Conservativity over LF

Using the algorithm for type checking, it is easy to see that it is directed only by the structure of the term. This has the important consequence that the system $\text{LF}^{\Sigma,1+}$ is conservative over LF. This is important because it allows us to import LF signatures defined in Twelf, which checks metatheoretic properties.

The system LF can be obtained from the current system by deleting the Σ types, the unit type and the metavariables from the level of type families, and deleting pairs, projections, unit, and metavariables from the level of objects. The corresponding typing rules are also deleted. We will denote judgments from this smaller system by $\vdash^{LF} \mathcal{J}$. LF does not need or use a metavariable context. Clearly LF is a sublanguage,

that is, valid judgments in LF are also valid judgments in $\mathrm{LF}^{\Sigma,1+}$. The other direction is the conservativity property we want, which says that we do not get new canonical forms at old types by moving from the language LF to the language $\mathrm{LF}^{\Sigma,1+}$.

We will first prove that canonical terms have a typing derivation lying entirely within LF, and then prove that equality between canonical terms also has a definitional equality derivation lying within LF. Putting these together, we get our required result.

Lemma 6.10.1 (Algorithmic equality sound within LF) *Assume that the signature $\mathcal{S}$ is well-typed in LF, that is, $\vdash^{LF} \mathcal{S} : \mathsf{sig}$. Assume further that all terms appearing in the premises lie within the language of LF.*

1. *If $\cdot;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$, $\cdot;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$, and $\cdot;\Gamma^- \vdash \mathsf{M}_1 \Longleftrightarrow \mathsf{M}_2 : (\mathsf{A})^-$, then $\Gamma \vdash^{LF} \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$.*
2. *If $\cdot;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$, $\cdot;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$, and $\cdot;\Gamma^- \vdash \mathsf{M}_1 \longleftrightarrow \mathsf{M}_2 : \tau$, then $\Gamma \vdash^{LF} \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_1$, $\Gamma \vdash^{LF} \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$, and ${\mathsf{A}_1}^- = {\mathsf{A}_2}^- = \tau$.*
3. *If $\cdot;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$, $\cdot;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$, and $\cdot;\Gamma^- \vdash \mathsf{A}_1 \Longleftrightarrow \mathsf{A}_2 : \mathsf{K}^-$, then $\Gamma \vdash^{LF} \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$.*
4. *If $\cdot;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$, $\cdot;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$, and $\cdot;\Gamma^- \vdash \mathsf{A}_1 \longleftrightarrow \mathsf{A}_2 : \kappa$, then $\Gamma \vdash^{LF} \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}_1$, $\Gamma \vdash^{LF} \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$, and ${\mathsf{K}_1}^- = {\mathsf{K}_2}^- = \kappa$.*
5. *If $\cdot;\Gamma \vdash \mathsf{K}_1 : \mathsf{kind}$, $\cdot;\Gamma \vdash \mathsf{K}_2 : \mathsf{kind}$, and $\cdot;\Gamma^- \vdash \mathsf{K}_1 \longleftrightarrow \mathsf{K}_2 : \mathsf{kind}^-$, then $\Gamma \vdash^{LF} \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$.*

Proof

By induction on the algorithmic judgment, since this is structural, the forbidden rules for metavariables, unit and Σ types do not appear. □

Lemma 6.10.2 (Algorithmic typing for canonical terms sound within LF) *Assume that the signature $\mathcal{S}$ is well-typed in LF, that is, $\vdash^{LF} \mathcal{S} : \mathsf{sig}$. Assume further that the canonical (or atomic) object M^{N}, the canonical (or atomic) family A^{N}, the canonical kind K^{N}, and the context Γ belong to the language of LF.*

- *If $\cdot;\Gamma \vdash \mathsf{M}^{\mathsf{N}} \Rightarrow \mathsf{A}$, then $\Gamma \vdash^{LF} \mathsf{M}^{\mathsf{N}} : \mathsf{A}$.*
- *If $\cdot;\Gamma \vdash \mathsf{A}^{\mathsf{N}} \Rightarrow \mathsf{K}$, then $\Gamma \vdash^{LF} \mathsf{A}^{\mathsf{N}} : \mathsf{K}$.*
- *If $\cdot;\Gamma \vdash \mathsf{K}^{\mathsf{N}} \Rightarrow \mathsf{kind}$, then $\Gamma \vdash^{LF} \mathsf{K}^{\mathsf{N}} : \mathsf{kind}$.*

Proof

By induction on the algorithmic judgment. □

Proposition 6.10.3 (Conservativity over LF) *Assume that the signature $\mathcal{S}$ is well-typed in LF, that is, $\vdash^{LF} \mathcal{S} : \mathsf{sig}$. Assume further that the canonical (or atomic) object M^{N}, the canonical (or atomic) type family A^{N}, and the context Γ all belong to the language of LF. If $\cdot;\Gamma \vdash \mathsf{M}^{\mathsf{N}} : \mathsf{A}^{\mathsf{N}}$, then $\Gamma \vdash^{LF} \mathsf{M}^{\mathsf{N}} : \mathsf{A}^{\mathsf{N}}$.*

If $\cdot;\Gamma \vdash \mathcal{J}$ and Γ, $\mathcal{J}$ belong to the language of LF then there exists a derivation of $\Gamma \vdash^{LF} \mathcal{J}$.

Proof

$\cdot;\Gamma \vdash \mathsf{M}^{\mathsf{N}} \Rightarrow \mathsf{A}$, and	
$\cdot;\Gamma \vdash \mathsf{A}^{\mathsf{N}} \equiv \mathsf{A} : \mathsf{type}$	By completeness of algorithmic type checking
$\Gamma \vdash^{LF} \mathsf{M}^{\mathsf{N}} : \mathsf{A}$	By algorithmic typing of canonical terms sound within LF
$\cdot;\Gamma^- \vdash \mathsf{A} \Longleftrightarrow \mathsf{A}^{\mathsf{N}} : \mathsf{type}^-$	By completeness of algorithmic typing
$\cdot;\Gamma \vdash \mathsf{A}^{\mathsf{N}} : \mathsf{type}$,	
$\cdot;\Gamma \vdash \mathsf{A} : \mathsf{type}$	By regularity
$\Gamma \vdash^{LF} \mathsf{A} \equiv \mathsf{A}^{\mathsf{N}} : \mathsf{type}$	By soundness of algorithmic equality within LF
$\Gamma \vdash^{LF} \mathsf{M}^{\mathsf{N}} : \mathsf{A}^{\mathsf{N}}$	By rule

Chapter 7

Unification for $\mathrm{LF}^{\Sigma,1+}$

A key part of LF/ML is reasoning with the assumption that two $\mathrm{LF}^{\Sigma,1+}$ object (or type family) metavariables stand for the same term. This assumption has consequences for type checking. An assumption may be inconsistent, or be consistent and lead to simplification. This points us to the fact that we need to reason about unification for $\mathrm{LF}^{\Sigma,1+}$. Unfortunately, the theory LF (indeed, the simply typed fragment) already has undecidable unification problems.

Fortunately, there is a fragment of the general problem of higher-order unification which occurs often in practice, and is known to be decidable. This fragment was identified as the so-called pattern fragment by Miller [Mil91] and extended to the dependent case by Pfenning [Pfe91b]. We follow and extend the results to the case with Σ types. We also formulate in the style of Pientka and Pfenning [PP03], who gives an account for modal type theory focusing on optimization techniques for logic-programming implementations.

In this chapter we will look at unification within the pattern fragment for $\mathrm{LF}^{\Sigma,1+}$. We will prove that the algorithm sketched out is sound for unification, and is complete for the pattern fragment. This algorithm will be used in the next chapter as part of the type checking phase for LF/ML.

7.1 Defining the extended pattern fragment

We will first need to isolate the pattern fragment for our calculus. The main idea of the usual definition is that metavariables are always applied to distinct bound variables in all terms appearing in equation. Equivalently, we can lower the type of all metavariables so that they are of base type, and insist that they appear under a substitution substituting distinct bound variables only. Further, since we now have product types in the language, we are also allowed to substitute distinct projections of variables. This extension leads us to define an extended pattern fragment, which is similar to Duggan's work on unification for product types [Dug98].

The key concept then is that of an extended pattern substitution. An extended pattern substitution σ is intuitively a substitution for object variables $\mathsf{M}_1/\mathsf{x}_1, \ldots, \mathsf{M}_n/\mathsf{x}_n$ $(0 \leq n)$ such that all the M_i are definitionally equal to pairs of *distinct* paths, where paths are projections of bound variables.

More formally, we use definitions from the following grammar:

$$
\begin{array}{llll}
\text{Paths} & \mathsf{p} & ::= & \mathsf{x} \\
 & & \mid & \pi_i \mathsf{p} \\
\text{Path Pairs} & \mathsf{pp} & ::= & \mathsf{p} \\
 & & \mid & \langle \mathsf{pp}_1, \mathsf{pp}_2 \rangle^{\mathsf{A}}
\end{array}
$$

We first axiomatize the nonoverlapping conditions on paths and path pairs by the judgment forms

$\mathsf{p}_1 \# \mathsf{p}_2$	p_1 and p_2 are disjoint
$\mathsf{p} \# \mathsf{pp}$	p is disjoint from all paths in pp_2
$\mathsf{pp}_1 \# \mathsf{pp}_2$	All paths in pp_1 are disjoint from all paths in pp_2

$\boxed{\mathsf{p}_1 \# \mathsf{p}_2}$

$$\frac{\mathsf{x}_1 \neq \mathsf{x}_2}{\mathsf{x}_1 \# \mathsf{x}_2} \qquad \frac{}{\pi_1 \mathsf{p} \# \pi_2 \mathsf{p}} \qquad \frac{\mathsf{p}_1 \# \mathsf{p}_2}{\mathsf{p}_1 \# \pi_i \mathsf{p}_2} \qquad \frac{\mathsf{p}_1 \# \mathsf{p}_2}{\mathsf{p}_2 \# \mathsf{p}_1}$$

$\boxed{\mathsf{p} \# \mathsf{pp}}$

$$\frac{\mathsf{p}_1 \# \mathsf{p}_2}{\mathsf{p}_1 \# \mathsf{p}_2} \qquad \frac{\mathsf{p} \# \mathsf{p}_1 \quad \mathsf{p} \# \mathsf{p}_2}{\mathsf{p} \# \langle \mathsf{p}_1, \mathsf{p}_2 \rangle^{\mathsf{A}}}$$

$\boxed{\mathsf{pp}_1 \# \mathsf{pp}_2}$

$$\frac{\mathsf{p} \# \mathsf{pp}}{\mathsf{p} \# \mathsf{pp}} \qquad \frac{\mathsf{p}_1 \# \mathsf{p} \quad \mathsf{p}_2 \# \mathsf{p}}{\langle \mathsf{p}_1, \mathsf{p}_2 \rangle^{\mathsf{A}} \# \mathsf{p}}$$

Extended pattern substitutions We can now define what our extended pattern substitutions are. We use the judgment form

$$\Psi;\Gamma \vdash \sigma \; \mathsf{patsub} \quad \sigma \text{ is a pattern substitution}$$

$\boxed{\Psi;\Gamma \vdash \sigma \; \mathsf{patsub}}$ Each element of the substitution must be provably equivalent to a path pair (that is, a product of paths). Further, each of these path pairs must be distinct from each other.

$$\frac{\begin{array}{l}\forall \mathsf{x} \in \mathsf{dom}(\sigma). \exists \mathsf{pp}_{\mathsf{x}}. \Psi;\Gamma \vdash \mathsf{x}[\sigma] \equiv \mathsf{pp}_{\mathsf{x}} : \mathsf{A} \\ \forall \mathsf{x}_1, \mathsf{x}_2 \in \mathsf{dom}(\sigma). \mathsf{x}_1 \neq \mathsf{x}_2 \text{ implies } \mathsf{pp}_{\mathsf{x}_1} \# \mathsf{pp}_{\mathsf{x}_2}\end{array}}{\Psi;\Gamma \vdash \sigma \; \mathsf{patsub}}$$

An object or family is in the pattern fragment iff all metavariables occurring within the term are associated with a pattern substitution. Recall that objects include metavariables under a substitution $\mathsf{X}_{\mathsf{M}}[\sigma]$. We now insist that for all subterms of this form, σ is a pattern substitution. A similar restriction is made on family level metavariables. We also insist that all metavariables are in so-called "lowered" form, that is, they are of base type. This is not a restriction in practice since any metavariable at higher type can be replaced uniformly by metavariables at lower types, but in possibly extended contexts.

7.2 Unification

Unification problems are defined by the following grammar:

$$\begin{array}{rcl}\text{Unification Problems} & ::= & \Psi;\Gamma \vdash \mathsf{M}_1 =^{?} \mathsf{M}_2 \\ & | & \Psi;\Gamma \vdash \mathsf{A}_1 =^{?} \mathsf{A}_2\end{array}$$

Here we insist the objects, families and kinds appearing are in canonical form, and the objects and families are in the pattern fragment. Also, a precondition of the algorithm is that the two objects (respectively families) which the algorithm is applied to are both well typed (kinded) at the same type (kind). This preprocessing step allows us to fail early if for example we seek to unify a pair against an abstraction.

The unification algorithm is given by the judgments:

$$\Psi;\Gamma \vdash \mathsf{M}_1 =^{?} \mathsf{M}_2 \Rightarrow (\rho, \Psi_1)$$
$$\Psi;\Gamma \vdash \mathsf{A}_1 =^{?} \mathsf{A}_2 \Rightarrow (\rho, \Psi_1)$$

These produce a solution metavariable substitution and a new context for metavariables, if possible.

The rules for unification where the head term is not a metavariable is given in figure 7.1. The first few rules for constants, variables and unit occur when the objects are the same, when the solution is just the identity substitution. We have similar rules at the family level too.

For an abstraction (rules Unif-ObjAbs and Unif-FamAbs, recall that we assume the two objects are in canonical form and have the same type. Thus we do not need to unify the two domains of the abstractions.

For pairs (rule Unif-Pair), applications (rules Unif-ObjApp and Unif-FamApp) and projections (rule Unif-Proj), we just break down the terms and continue. The case for Π and Σ types (rules Unif-Pi and Unif-Sigma) are very similar to these, in that we just have to apply unification recursively to the components. However, in these cases we have to extend the context.

Now, let us consider the case of unifying two terms which are the same metavariable, possibly under different pattern substitutions, as in figure 7.2. Consider how the unification might succeed. The substitution on each side might substitute the same term for the same element of the context, in which case any resulting solution may substitute for that element. On the other hand, if different terms are substituted in, no solution can depend on that element of the context. To perform this check, we define an auxiliary notion of intersection of pattern substitutions, given in figure 7.3. We then create a new metavariable depending on only the smaller context where the two substitutions agree.

The last remaining case is that of unifying a metavariable against another term, which is not the same metavariable at the root 7.4. If the metavariable in question occurs within the other term, there can be no solution since any solution will be circular. This is called the *occurs check* for unification. Further, if an ordinary variable occurs within the term that the metavariable does not depend on, then also there can be no solution. Finally, any other metavariables appearing can no longer depend on the variables the current metavariable does not depend on. This is called *pruning* the dependencies. We perform a traversal over the second term doing pruning and occurrence check together, in a new operation of pruning, defined in figure 7.5 for objects and 7.6 for type families.

The pruning operation returns a term which is the result of applying the inverse of the substitution. This operation may be defined even if the substitution is not fully (right) invertible, since it depends only on the terms actually present. For metavariables, the substitutions are pruned. Notice that if we are trying to unify two well-typed objects at the same type, we need never care about the types.

7.3 Correctness of unification

We want to show that the unification algorithm we have sketched out is correct. Recall that a unification problem asks whether two canonical (or atomic) objects at the same type in the same context can be made to be equal. Solutions returned by the algorithm consist of a substitution for the metavariables present within the two objects. We thus have to show soundness, that is, the solution substitution does make the two objects provably equal. We also have a completeness result for the extended pattern fragment, that is, if there exists a solution, the algorithm will return a solution substitution. Further, the returned substitution is a most general unifier, in that any other solution is an extension of the returned substitution.

Proving soundness is done by induction over the unification judgment. In the congruence cases, the inductive hypotheses usually produce the required result. We have to carefully consider extensions to the context in cases such as abstraction. The interesting cases are when one, or both, sides are metavariables. If they are the same metavariable, the algorithm performs a intersection of the two associated substitutions. We thus have to look at the properties of this intersection. If the metavariables are different, the algorithm performs an inverse substitution and pruning. We thus have to prove this step sound.

Completeness proceeds by induction on the structure of the two terms. There are two proof obligations of completeness, one to show that the algorithm produces a solution if possible, and further, to show that the solution is most general. Again, the congruence cases make a direct appeal to the inductive hypothesis. Now suppose that the two terms are the same metavariable, possibly associated with different substitutions. The unification algorithm always succeeds in this case. It remains to prove that the provided solution is most general, which is a property of the intersection of the two substitutions. The last remaining case is

$$\frac{}{\Psi;\Gamma \vdash \langle\rangle =^{?} \langle\rangle \Rightarrow (\mathsf{id}_\Psi, \Psi)}\ \textsc{Unif-ObjUnit} \qquad \frac{}{\Psi;\Gamma \vdash \mathsf{x} =^{?} \mathsf{x} \Rightarrow (\mathsf{id}_\Psi, \Psi)}\ \textsc{Unif-ObjVar}$$

$$\frac{}{\Psi;\Gamma \vdash \mathsf{c} =^{?} \mathsf{c} \Rightarrow (\mathsf{id}_\Psi, \Psi)}\ \textsc{Unif-ObjConst}$$

$$\frac{}{\Psi;\Gamma \vdash 1 =^{?} 1 \Rightarrow (\mathsf{id}_\Psi, \Psi)}\ \textsc{Unif-FamUnit} \qquad \frac{}{\Psi;\Gamma \vdash \mathsf{a} =^{?} \mathsf{a} \Rightarrow (\mathsf{id}_\Psi, \Psi)}\ \textsc{Unif-FamConst}$$

$$\frac{\Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{M}_1 =^{?} \mathsf{M}_2 \Rightarrow (\rho, \Psi_1)}{\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_1 =^{?} \lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_2 \Rightarrow (\rho, \Psi_1)}\ \textsc{Unif-ObjAbs}$$

$$\frac{\Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{A}_1 =^{?} \mathsf{A}_2 \Rightarrow (\rho, \Psi_1)}{\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}.\mathsf{A}_1 =^{?} \lambda\mathsf{x}{:}\mathsf{A}.\mathsf{A}_2 \Rightarrow (\rho, \Psi_1)}\ \textsc{Unif-FamAbs}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_{11} =^{?} \mathsf{M}_{21} \Rightarrow (\rho_1, \Psi_1) \quad \Psi_1;[\rho_1]\Gamma \vdash [\rho_1]\mathsf{M}_{21} =^{?} [\rho_1]\mathsf{M}_{22} \Rightarrow (\rho_2, \Psi_2)}{\Psi;\Gamma \vdash \langle \mathsf{M}_{11}, \mathsf{M}_{12}\rangle^{\mathsf{A}_1} =^{?} \langle \mathsf{M}_{21}, \mathsf{M}_{22}\rangle^{\mathsf{A}_2} \Rightarrow (\rho_2 \circ \rho_1, \Psi_2)}\ \textsc{Unif-Pair}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_{11} =^{?} \mathsf{M}_{21} \Rightarrow (\rho_1, \Psi_1) \quad \Psi_1;[\rho_1]\Gamma \vdash [\rho_1]\mathsf{M}_{21} =^{?} [\rho_1]\mathsf{M}_{22} \Rightarrow (\rho_2, \Psi_2)}{\Psi;\Gamma \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} =^{?} \mathsf{M}_{21}\,\mathsf{M}_{22} \Rightarrow (\rho_2 \circ \rho_1, \Psi_2)}\ \textsc{Unif-ObjApp}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_{11} =^{?} \mathsf{A}_{21} \Rightarrow (\rho_1, \Psi_1) \quad \Psi_1;[\rho_1]\Gamma \vdash [\rho_1]\mathsf{M}_{21} =^{?} [\rho_1]\mathsf{M}_{22} \Rightarrow (\rho_2, \Psi_2)}{\Psi;\Gamma \vdash \mathsf{A}_{11}\,\mathsf{M}_{12} =^{?} \mathsf{A}_{21}\,\mathsf{M}_{22} \Rightarrow (\rho_2 \circ \rho_1, \Psi_2)}\ \textsc{Unif-FamApp}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 =^{?} \mathsf{M}_2 \Rightarrow (\rho, \Psi_1)}{\Psi;\Gamma \vdash \pi_i\mathsf{M}_1 =^{?} \pi_i\mathsf{M}_2 \Rightarrow (\rho, \Psi_1)}\ \textsc{Unif-Proj}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_{11} =^{?} \mathsf{A}_{21} \Rightarrow (\rho_1, \Psi_1) \quad \Psi_1;[\rho_1]\Gamma, \mathsf{x}{:}[\rho_1]\mathsf{A}_{11} \vdash [\rho_1]\mathsf{A}_{12} =^{?} [\rho_1]\mathsf{A}_{22} \Rightarrow (\rho_2, \Psi_2)}{\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} =^{?} \Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} \Rightarrow (\rho_2 \circ \rho_1, \Psi_2)}\ \textsc{Unif-Pi}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_{11} =^{?} \mathsf{A}_{21} \Rightarrow (\rho_1, \Psi_1) \quad \Psi_1;[\rho_1]\Gamma, \mathsf{x}{:}[\rho_1]\mathsf{A}_{11} \vdash [\rho_1]\mathsf{A}_{12} =^{?} [\rho_1]\mathsf{A}_{22} \Rightarrow (\rho_2, \Psi_2)}{\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} =^{?} \Sigma\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} \Rightarrow (\rho_2 \circ \rho_1, \Psi_2)}\ \textsc{Unif-Sigma}$$

Figure 7.1: Unification, part 1

$$\frac{\begin{array}{l}\Psi = \Psi_1, \mathsf{X_M}{:}\Gamma_1 \vdash \mathsf{A}_1, \Psi_2 \\ \Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2 \\ \Psi_2' = \Psi_2[\mathsf{id}_{\Psi_1}, \mathsf{X_{M1}}[\mathsf{id}_{\Gamma_2}]/\mathsf{X_M}] \\ \Psi' = \Psi_1, \mathsf{X_{M1}}{:}(\Gamma_2 \vdash \mathsf{A}_1), \Psi_2' \\ \rho = \mathsf{id}_{\Psi_1}, \mathsf{X_{M1}}[\mathsf{id}_{\Gamma_2}]/\mathsf{X_M}, \mathsf{id}_{\Psi_2'} \\ (\mathsf{X_{M1}} \notin \mathsf{dom}(\Psi))\end{array}}{\Psi;\Gamma \vdash \mathsf{X_M}[\sigma_1] =^? \mathsf{X_M}[\sigma_2] \Rightarrow (\rho, \Psi')}\ \textsc{Unif-ObjSameVar}$$

$$\frac{\begin{array}{l}\Psi = \Psi_1, \mathsf{X_A}{:}(\Gamma_1 \vdash \mathsf{K}_1)\Psi_2 \\ \Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2 \\ \Psi_2' = \Psi_2[\mathsf{id}_{\Psi_1}, \mathsf{X_{A1}}[\mathsf{id}_{\Gamma_2}]/\mathsf{X_A}] \\ \Psi' = \Psi_1, \mathsf{X_{A1}}{:}(\Gamma_2 \vdash \mathsf{K}_1), \Psi_2' \\ \rho = \mathsf{id}_{\Psi_1}, \mathsf{X_{A1}}[\mathsf{id}_{\Gamma_2}]/\mathsf{X_A}, \mathsf{id}_{\Psi_2'} \\ (\mathsf{X_{A1}} \notin \mathsf{dom}(\Psi))\end{array}}{\Psi;\Gamma \vdash \mathsf{X_A}[\sigma_1] =^? \mathsf{X_A}[\sigma_2] \Rightarrow (\mathsf{id}_{\Psi'}, \Psi')}\ \textsc{Unif-FamSameVar}$$

Figure 7.2: Unification, part 2

$$\frac{}{\Psi \vdash \cdot \cap \cdot : \cdot \Rightarrow \cdot}\ \textsc{Intersect-Nil} \qquad \frac{\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2 \quad \Psi;\Gamma_1 \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}}{\Psi \vdash \sigma_1, \mathsf{M}_1/\mathsf{x}_1 \cap \sigma_2, \mathsf{M}_2/\mathsf{x}_1 : \Gamma_1, \mathsf{x}_1{:}\mathsf{A} \Rightarrow \Gamma_2, \mathsf{x}_1{:}\mathsf{A}}\ \textsc{Intersect-Same}$$

$$\frac{\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2 \quad \Psi;\Gamma_1 \vdash \mathsf{M}_1 \not\equiv \mathsf{M}_2 : \mathsf{A}}{\Psi \vdash \sigma_1, \mathsf{M}_1/\mathsf{x}_1 \cap \sigma_2, \mathsf{M}_2/\mathsf{x}_1 : \Gamma_1, \mathsf{x}_1{:}\mathsf{A} \Rightarrow \Gamma_2}\ \textsc{Intersect-Diff}$$

Figure 7.3: Intersection of substitution

$$\frac{\begin{array}{l}\Psi;\Gamma\vdash(\mathsf{M}\backslash\mathsf{X}_{\mathsf{M}})[\sigma]^{-1}\Longrightarrow(\mathsf{M}_1,\rho_1,\Psi_1)\\ \Psi_1=\Psi_{11},\mathsf{X}_{\mathsf{M}}{:}\Gamma_1\vdash\mathsf{A},\Psi_{12}\\ \Psi_{22}=\Psi_{12}[\mathsf{id}_{\Psi_{11}},\mathsf{M}_1/\mathsf{X}_{\mathsf{M}}]\end{array}}{\Psi;\Gamma\vdash\mathsf{X}_{\mathsf{M}}[\sigma]=^{?}\mathsf{M}\Rightarrow((\mathsf{id}_{\Psi_{11}},\mathsf{M}_1/\mathsf{X}_{\mathsf{M}},\mathsf{id}_{\Psi_{22}})\circ\rho_1,(\Psi_{11},\Psi_{22}))}\ \textsc{Unif-ObjDiffVar1}$$

$$\frac{\begin{array}{l}\Psi;\Gamma\vdash(\mathsf{M}\backslash\mathsf{X}_{\mathsf{M}})[\sigma]^{-1}\Longrightarrow(\mathsf{M}_1,\rho_1,\Psi_1)\\ \Psi_1=\Psi_{11},\mathsf{X}_{\mathsf{M}}{:}\Gamma_1\vdash\mathsf{A},\Psi_{12}\\ \Psi_{22}=\Psi_{12}[\mathsf{id}_{\Psi_{11}},\mathsf{M}_1/\mathsf{X}_{\mathsf{M}}]\end{array}}{\Psi;\Gamma\vdash\mathsf{M}=^{?}\mathsf{X}_{\mathsf{M}}[\sigma]\Rightarrow((\mathsf{id}_{\Psi_{11}},\mathsf{M}_1/\mathsf{X}_{\mathsf{M}},\mathsf{id}_{\Psi_{22}})\circ\rho_1,(\Psi_{11},\Psi_{22}))}\ \textsc{Unif-ObjDiffVar2}$$

$$\frac{\begin{array}{l}\Psi;\Gamma\vdash(\mathsf{A}\backslash\mathsf{X}_{\mathsf{A}})[\sigma]^{-1}\Longrightarrow(\mathsf{A}_1,\rho_1,\Psi_1)\\ \Psi_1=\Psi_{11},\mathsf{X}_{\mathsf{A}}{:}\Gamma_1\vdash\mathsf{K},\Psi_{12}\\ \Psi_{22}=\Psi_{12}[\mathsf{id}_{\Psi_{11}},\mathsf{A}_1/\mathsf{X}_{\mathsf{A}}]\end{array}}{\Psi;\Gamma\vdash\mathsf{X}_{\mathsf{A}}[\sigma]=^{?}\mathsf{A}\Rightarrow((\mathsf{id}_{\Psi_{11}},\mathsf{A}_1/\mathsf{X}_{\mathsf{A}},\mathsf{id}_{\Psi_{22}})\circ\rho_1,(\Psi_{11},\Psi_{22}))}\ \textsc{Unif-FamDiffVar1}$$

$$\frac{\begin{array}{l}\Psi;\Gamma\vdash(\mathsf{A}\backslash\mathsf{X}_{\mathsf{A}})[\sigma]^{-1}\Longrightarrow(\mathsf{A}_1,\rho_1,\Psi_1)\\ \Psi_1=\Psi_{11},\mathsf{X}_{\mathsf{A}}{:}\Gamma_1\vdash\mathsf{K},\Psi_{12}\\ \Psi_{22}=\Psi_{12}[\mathsf{id}_{\Psi_{11}},\mathsf{A}_1/\mathsf{X}_{\mathsf{A}}]\end{array}}{\Psi;\Gamma\vdash\mathsf{M}=^{?}\mathsf{X}_{\mathsf{A}}[\sigma]\Rightarrow((\mathsf{id}_{\Psi_{11}},\mathsf{A}_1/\mathsf{X}_{\mathsf{A}},\mathsf{id}_{\Psi_{22}})\circ\rho_1,(\Psi_{11},\Psi_{22}))}\ \textsc{Unif-FamDiffVar2}$$

Figure 7.4: Unification, part 3

$$\frac{}{\Psi;\Gamma\vdash(\mathsf{c}\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{c},\mathsf{id}_\Psi,\Psi)} \qquad \frac{}{\Psi;\Gamma\vdash(\langle\rangle\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\langle\rangle,\mathsf{id}_\Psi,\Psi)}$$

$$\frac{\mathsf{pp}/\mathsf{x}\in\sigma}{\Psi;\Gamma\vdash(\mathsf{pp}\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{x},\mathsf{id}_\Psi,\Psi)} \qquad \frac{\Psi;\Gamma\vdash(\mathsf{M}\backslash\mathsf{X})[\sigma,\mathsf{x}/\mathsf{x}]^{-1}\Longrightarrow(\mathsf{M}_1,\rho_1,\Psi_1)}{\Psi;\Gamma\vdash(\lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_1,\rho_1,\Psi_1)}$$

$$\frac{\Psi;\Gamma\vdash(\mathsf{M}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{M}_1',\rho_1,\Psi_1)\quad\Psi_1;[\rho_1]\Gamma\vdash([\rho_1]\mathsf{M}_2\backslash\mathsf{X})[[\rho_1]\sigma]^{-1}\Longrightarrow(\mathsf{M}_2',\rho_2,\Psi_2)}{\Psi;\Gamma\vdash(\mathsf{M}_1\,\mathsf{M}_2\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow((\mathsf{M}_1'[\rho_2])\,\mathsf{M}_2',\rho_2\circ\rho_1,\Psi_2)}$$

$$\frac{\Psi;\Gamma\vdash(\mathsf{M}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{M}_1',\rho_1,\Psi_1)\quad\Psi_1;[\rho_1]\Gamma\vdash([\rho_1]\mathsf{M}_2\backslash\mathsf{X})[[\rho_1]\sigma]^{-1}\Longrightarrow(\mathsf{M}_2',\rho_2,\Psi_2)}{\Psi;\Gamma\vdash(\langle\mathsf{M}_1,\mathsf{M}_2\rangle^{\mathsf{A}}\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\langle(\mathsf{M}_1'[\rho_2]),\mathsf{M}_2'\rangle^{[\rho_2\circ\rho_1]\mathsf{A}},\rho_2\circ\rho_1,\Psi_2)}$$

$$\frac{\Psi;\Gamma\vdash(\mathsf{M}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{M}_2,\rho_1,\Psi_1)}{\Psi;\Gamma\vdash(\pi_i\mathsf{M}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\pi_i\mathsf{M}_2,\rho_1,\Psi_1)}$$

$$\frac{\begin{array}{c}\mathsf{X}_{\mathsf{M}1}\neq\mathsf{X}_2\quad\Psi=\Psi_1,\mathsf{X}_{\mathsf{M}1}{:}(\Gamma_1\vdash\mathsf{A}),\Psi_2\\ \sigma_1:\Gamma_1|[\sigma_2]^{-1}\Longrightarrow(\Gamma_2,\sigma_1')\\ \mathsf{X}_{\mathsf{M}3}\notin\mathsf{dom}(\Psi)\\ \Psi_2'=[\mathsf{X}_{\mathsf{M}3}[\mathsf{id}_{\Gamma_2}]/\mathsf{X}_{\mathsf{M}1}]\Psi_2\\ \Psi'=\Psi_1,\mathsf{X}_{\mathsf{M}3}{:}(\Gamma_2\vdash\mathsf{A})\\ \rho_1=\mathsf{id}_{\Psi_1},\mathsf{X}_{\mathsf{M}3}[\mathsf{id}_{\Gamma_2}]/\mathsf{X}_{\mathsf{M}3},\mathsf{X}_{\mathsf{M}3}[\mathsf{id}_{\Gamma_2}]/\mathsf{X}_{\mathsf{M}1},\mathsf{id}_{\Psi_2'}\end{array}}{\Psi;\Gamma\vdash(\mathsf{X}_{\mathsf{M}1}[\sigma_1]\backslash\mathsf{X}_2)[\sigma_2]^{-1}\Longrightarrow(\mathsf{X}_{\mathsf{M}3}[\sigma_1'],\rho_1,\Psi')}$$

Figure 7.5: Pruning operation: Objects

$$\frac{}{\Psi;\Gamma\vdash(\mathsf{a}\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{a},\mathsf{id}_\Psi,\Psi)} \qquad \frac{}{\Psi;\Gamma\vdash(1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(1,\mathsf{id}_\Psi,\Psi)}$$

$$\frac{\Psi;\Gamma\vdash(\mathsf{A}_1\backslash\mathsf{X})[\sigma,\mathsf{x}/\mathsf{x}]^{-1}\Longrightarrow(\mathsf{A}_2,\rho_1,\Psi_1)}{\Psi;\Gamma\vdash(\lambda\mathsf{x}{:}\mathsf{A}.\mathsf{A}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\lambda\mathsf{x}{:}\mathsf{A}.\mathsf{A}_2,\rho_1,\Psi_1)} \qquad \frac{\Psi;\Gamma\vdash(\mathsf{A}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{A}_2,\rho_1,\Psi_1)}{\Psi;\Gamma\vdash(\mathsf{A}_1\,\mathsf{M}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{A}_2\,\mathsf{M}_1,\rho_1,\Psi_1)}$$

$$\frac{\Psi;\Gamma\vdash(\mathsf{A}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{A}_1',\rho_1,\Psi_1)\quad\Psi_1;[\rho_1]\Gamma\vdash([\rho_1]\mathsf{A}_2\backslash\mathsf{X})[[\rho_1]\sigma,\mathsf{x}/\mathsf{x}]^{-1}\Longrightarrow(\mathsf{A}_2',\rho_2,\Psi_2)}{\Psi;\Gamma\vdash(\Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\Pi\mathsf{x}{:}\mathsf{A}_1'.\mathsf{A}_2',\rho_2,\Psi_2)}$$

$$\frac{\Psi;\Gamma\vdash(\mathsf{A}_1\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\mathsf{A}_1',\rho_1,\Psi_1)\quad\Psi_1;[\rho_1]\Gamma\vdash([\rho_1]\mathsf{A}_2\backslash\mathsf{X})[[\rho_1]\sigma,\mathsf{x}/\mathsf{x}]^{-1}\Longrightarrow(\mathsf{A}_2',\rho_2,\Psi_2)}{\Psi;\Gamma\vdash(\Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2\backslash\mathsf{X})[\sigma]^{-1}\Longrightarrow(\Sigma\mathsf{x}{:}\mathsf{A}_1'.\mathsf{A}_2',\rho_2,\Psi_2)}$$

$$\frac{\begin{array}{c}\mathsf{X}_{\mathsf{A}1}\neq\mathsf{X}_2\quad\Psi=\Psi_1,\mathsf{X}_{\mathsf{A}1}{:}(\Gamma_1\vdash\mathsf{K}),\Psi_2\\ \sigma_1:\Gamma_1|[\sigma_2]^{-1}\Longrightarrow(\Gamma_2,\sigma_1')\\ \mathsf{X}_{\mathsf{A}3}\notin\mathsf{dom}(\Psi)\\ \Psi_2'=[\mathsf{X}_{\mathsf{A}3}[\mathsf{id}_{\Gamma_2}]/\mathsf{X}_{\mathsf{A}1}]\Psi_2\\ \Psi'=\Psi_1,\mathsf{X}_{\mathsf{A}3}{:}(\Gamma_2\vdash\mathsf{K}),\Psi_2'\\ \rho_1=\mathsf{id}_{\Psi_1},\mathsf{X}_{\mathsf{A}3}[\mathsf{id}_{\Gamma_2}]/\mathsf{X}_{\mathsf{A}3},\mathsf{X}_{\mathsf{A}3}[\mathsf{id}_{\Gamma_2}]/\mathsf{X}_{\mathsf{A}1},\mathsf{id}_{\Psi_2'}\end{array}}{\Psi;\Gamma\vdash(\mathsf{X}_{\mathsf{A}1}[\sigma_1]\backslash\mathsf{X}_2)[\sigma_2]^{-1}\Longrightarrow(\mathsf{X}_{\mathsf{A}3}[\sigma_1'],\rho_1,\Psi')}$$

Figure 7.6: Pruning operation: Families

$\boxed{\sigma_1:\Gamma_1|[\sigma_2]^{-1}\Longrightarrow(\Gamma_2,\sigma_3)}$

$$\frac{}{\cdot:\cdot|[\sigma]^{-1}\Longrightarrow(\cdot,\cdot)} \qquad \frac{\sigma_1:\Gamma_1|[\sigma_2]^{-1}\Longrightarrow(\Gamma_2,\sigma_3)\quad\mathsf{pp}/\mathsf{x}_2\in\sigma_2}{\sigma_1,\mathsf{x}_2/\mathsf{x}_1:\Gamma_1,\mathsf{x}_1{:}\mathsf{A}|[\sigma_2]^{-1}\Longrightarrow(\Gamma_2,\mathsf{x}_1{:}\mathsf{A},\sigma_3,\mathsf{x}_2/\mathsf{x}_1)}$$

$$\frac{\sigma_1:\Gamma_1|[\sigma_2]^{-1}\Longrightarrow(\Gamma_2,\sigma_3)\quad\mathsf{pp}/\mathsf{x}_2\notin\sigma_2}{\sigma_1,\mathsf{x}_2/\mathsf{x}_1:\Gamma_1,\mathsf{x}_1{:}\mathsf{A}|[\sigma_2]^{-1}\Longrightarrow(\Gamma_2,\sigma_3)}$$

Figure 7.7: Pruning operation: substitutions

of a metavariable being unified with a different term. Here the algorithm performs an inverse substitution and pruning step. We show, in two lemmas, the production of a solution if possible, and that the produced solution is most general.

7.3.1 Elementary properties of metavariable substitutions

We start with some elementary properties of metavariable substitutions. Since metavariable substitutions are heavily used by the unification algorithm, their properties are useful in later proofs.

First, we want to show that metavariable substitutions maintain the typing and equivalence judgments.

Lemma 7.3.1 (Substitution property for metavariable substitutions) *If* $\Psi_2 \vdash \rho : \Psi_1$ *and* $\Psi_1; \Gamma \vdash \mathcal{J}$ *then* $\Psi_2; \Gamma[\rho] \vdash \mathcal{J}[\rho]$.

Proof

By induction on the second judgment. □

Next, we show that metavariable substitutions compose properly, over all terms of the language.

Lemma 7.3.2 (Composition of metavariable substitutions) *Assume* $\Psi_2 \vdash \rho_1 : \Psi_1$ *and* $\Psi_3 \vdash \rho_2 : \Psi_2$.

1. $(\sigma[\rho_1])[\rho_2] = \sigma[\rho 2 \circ \rho_1]$.
2. $(\mathsf{M}[\rho_1])[\rho_2] = \mathsf{M}[\rho 2 \circ \rho_1]$.
3. $(\mathsf{A}[\rho_1])[\rho_2] = \mathsf{A}[\rho 2 \circ \rho_1]$.
4. $(\mathsf{K}[\rho_1])[\rho_2] = \mathsf{K}[\rho 2 \circ \rho_1]$.

Proof

By simultaneous induction on the structure of the term or substitution. □

We next prove two useful lemmas about the interaction of ordinary variable substitutions and metavariable substitutions.

Lemma 7.3.3 (Distributing metavariable substitution over ordinary substitutions) *Assume* $\Psi_2 \vdash \rho : \Psi_1$ *and* $\Psi_1; \Gamma_2 \vdash \sigma : \Gamma_1$.

1. $(\sigma_2 \circ \sigma)[\rho] = (\sigma_2[\rho]) \circ (\sigma[\rho])$.
2. $(\mathsf{M}[\sigma])[\rho] = (\mathsf{M}[\rho])[\sigma[\rho]]$.
3. $(\mathsf{A}[\sigma])[\rho] = (\mathsf{A}[\rho])[\sigma[\rho]]$.
4. $(\mathsf{K}[\sigma])[\rho] = (\mathsf{K}[\rho])[\sigma[\rho]]$.

Proof

By induction on the definition of ordinary substitution. □

Lemma 7.3.4 (Pattern substitutions untouched) *If* $\Psi_2 \vdash \rho : \Psi_2$ *and* σ *is a pattern substitution, then* $\sigma[\rho] = \sigma$.

Proof

By the rules for pattern substitution, any metavariable cannot occur within the substituents. □

7.3.2 Properties of intersections of substitutions

We now turn to the intersection of substitutions. Recall that this step is performed when we are trying to unify two terms which are the same metavariable, possibly under different substitutions. For soundness of the algorithm, it is essential that the resulting context of the intersection be well-formed, and invariant under the two substitutions. These are the first two lemmas. The last lemma is used for completeness, ensuring that there always exists an intersection.

Lemma 7.3.5 (Well-formedness of intersection) *If* σ_1 *and* σ_2 *are pattern substitutions,* $\Psi; \Gamma \vdash \sigma_1 : \Gamma_1$, $\Psi; \Gamma \vdash \sigma_2 : \Gamma_1$ *and* $\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2$ *then* $\Psi \vdash \Gamma_2 : \mathsf{ctx}$

Proof

By induction on the derivation of the intersection judgment.

Case 1: $\dfrac{}{\Psi \vdash \cdot \cap \cdot : \cdot \Rightarrow \cdot}$ Intersect-Nil

$\Psi \vdash \cdot : \mathsf{ctx}$ By rule

Case 2:
$$\frac{\begin{array}{c}\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2 \\ \Psi;\Gamma \vdash \mathsf{A}[\sigma_1] \equiv \mathsf{A}[\sigma_2] : \mathsf{type} \\ \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}[\sigma_1]\end{array}}{\Psi \vdash \sigma_1, \mathsf{M}_1/\mathsf{x}_1 \cap \sigma_2, \mathsf{M}_2/\mathsf{x}_1 : \Gamma_1, \mathsf{x}_1{:}\mathsf{A} \Rightarrow \Gamma_2, \mathsf{x}_1{:}\mathsf{A}}\ \textsc{Intersect-Same}$$

$\Psi;\Gamma \vdash \sigma_1, \mathsf{M}_1/\mathsf{x}_1 : \Gamma_1, \mathsf{x}_1{:}\mathsf{A}$ By assumption
$\Psi;\Gamma \vdash \sigma_1 : \Gamma_1$,
$\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}[\sigma_1]$ By typing inversion
$\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}[\sigma_2]$ By type conversion
$\Psi;\Gamma \vdash \sigma_2, \mathsf{M}_2/\mathsf{x}_1 : \Gamma_1, \mathsf{x}_1{:}\mathsf{A}$ By assumption
$\Psi;\Gamma \vdash \sigma_2 : \Gamma_1$,
$\Psi;\Gamma_1 \vdash \mathsf{M}_2 : \mathsf{A}$ By typing inversion
$\Psi \vdash \Gamma_2 : \mathsf{ctx}$ By induction
$\Psi;\Gamma \vdash \mathsf{A}[\sigma_1] \equiv \mathsf{A}[\sigma_2] : \mathsf{type}$ By assumption
No variable where σ_1 and σ_2 differ can appear free within A
$\Psi;\Gamma_2 \vdash \mathsf{A} : \mathsf{type}$ From above
$\Psi \vdash \Gamma_2, \mathsf{x}_1{:}\mathsf{A} : \mathsf{ctx}$ By rule

Case 3:
$$\frac{\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2 \quad \Psi;\Gamma_1 \vdash \mathsf{M}_1 \not\equiv \mathsf{M}_2 : \mathsf{A}}{\Psi \vdash \sigma_1, \mathsf{M}_1/\mathsf{x}_1 \cap \sigma_2, \mathsf{M}_2/\mathsf{x}_1 : \Gamma_1, \mathsf{x}_1{:}\mathsf{A} \Rightarrow \Gamma_2}\ \textsc{Intersect-Diff}$$

$\Psi;\Gamma \vdash \sigma_1, \mathsf{M}_1/\mathsf{x}_1 : \Gamma_1, \mathsf{x}_1{:}\mathsf{A}$ By assumption
$\Psi;\Gamma \vdash \sigma_1 : \Gamma_1$ By typing inversion
$\Psi;\Gamma \vdash \sigma_2, \mathsf{M}_2/\mathsf{x}_1 : \Gamma_1, \mathsf{x}_1{:}\mathsf{A}$ By assumption
$\Psi;\Gamma \vdash \sigma_2 : \Gamma_1$ By typing inversion
$\Psi \vdash \Gamma_2 : \mathsf{ctx}$ By induction

□

Lemma 7.3.6 (Correctness of intersection) *If* $\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2$ *then* $\sigma_1[\mathsf{id}_{\Gamma_2}] = \sigma_2[\mathsf{id}_{\Gamma_2}]$.

Proof

By induction on the derivation of the intersection judgment. □

Lemma 7.3.7 (Completeness of intersection) *If* $\Psi;\Gamma \vdash \sigma_1 : \Gamma_1$, $\Psi;\Gamma \vdash \sigma_2 : \Gamma_1$ *and* σ_1 *and* σ_2 *are pattern substitutions, then there exists a* Γ_2 *such that* $\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2$ *holds.*

Proof

By induction on the structure of the two substitutions. Notice that the two substitutions are well-typed to transfer between the same two contexts, and thus must substitute for the same variables. Also, equality of path pairs (indeed, any two well-typed objects) is decidable. □

7.3.3 Properties of pruning

We now turn to the pruning operation. Recall that this step is performed when a metavariable is being unified with a different term. This operation has to ensure three things. First, there is an occurrence check so that the metavariable in question does not appear deep within the structure of the term. Second, all metavariables must be pruned so that they do not depend on extraneous variables. Third, an inverse substitution must be notionally performed so that the resulting term can appear as a solution.

We first prove that the occurrence check is performed correctly. We then prove that the inverse substitution is performed correctly. Next, we prove two lemmas useful in proving completeness of unification

which say pruning is possible under different preconditions. First says that pruning is always possible with a strong assumption that a similar term can be pruned. Second, pruning is always possible assuming the substitution is apparently invertible, that is, we have a term which is the same after substitution.

Lemma 7.3.8 (Pruning includes occurs check)

1. *If* $\Psi;\Gamma_1 \vdash \sigma : \Gamma_2$, $\Psi;\Gamma_1 \vdash \mathsf{M} : \mathsf{A}$, *and* $\Psi;\Gamma_1 \vdash (\mathsf{M}\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{M}_1, \rho_1, \Psi_1)$ *then* X *does not occur in* M *(or in* M_1*).*
2. *If* $\Psi;\Gamma_1 \vdash \sigma : \Gamma_2$, $\Psi;\Gamma_1 \vdash \mathsf{A} : \mathsf{K}$, *and* $\Psi;\Gamma_1 \vdash (\mathsf{A}\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{A}_1, \rho_1, \Psi_1)$ *then* X *does not occur in* A *(or in* A_1*).*

Proof

By induction on the pruning derivation.

Case for metavariable: In the base case, the pruning rule checks explicitly for disequality of the metavariable in question and the metavariable unified against. Indeed, this rule is the reason the metavariable is passed in as an input argument to the judgment.

□

Lemma 7.3.9 (Soundness of pruning)

1. *If* $\Psi;\Gamma \vdash \sigma : \Gamma_1$, $\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}$, *and* $\Psi;\Gamma \vdash (\mathsf{M}\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{M}_1, \rho_1, \Psi_1)$ *then* $\mathsf{M}[\rho_1] = \mathsf{M}_1[\sigma]$.
2. *If* $\Psi;\Gamma \vdash \sigma : \Gamma_1$, $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}$, *and* $\Psi;\Gamma \vdash (\mathsf{A}\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{A}_1, \rho_1, \Psi_1)$ *then* $\mathsf{A}[\rho_1] = \mathsf{A}_1[\sigma]$.

Proof Sketch

By induction on the pruning judgment. □

Lemma 7.3.10 (Pruning under substitution)

1. *If* M *is canonical (or atomic),* $\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}$, ρ *substitutes atomic terms only,* $\Psi \vdash \rho : \Psi_1$, *and* $\Psi;\Gamma \vdash (\mathsf{M}[\rho]\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{M}'_1, \rho'_1, \Psi'_1)$ *then* $\Psi;\Gamma \vdash (\mathsf{M}\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{M}_1, \rho_1, \Psi_1)$, *and there exists a substitution* ρ_2 *such that* $\rho = \rho_2 \circ \rho_1$.
2. *If* A *is canonical (or atomic),* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}$, ρ *substitutes atomic terms only,* $\Psi \vdash \rho : \Psi_1$, *and* $\Psi;\Gamma \vdash (\mathsf{A}[\rho]\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{A}'_1, \rho'_1, \Psi'_1)$ *then* $\Psi;\Gamma \vdash (\mathsf{A}\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{A}_1, \rho_1, \Psi_1)$, *and there exists a substitution* ρ_2 *such that* $\rho = \rho_2 \circ \rho_1$.

Proof

By induction on the pruning derivation and on the metavariable substitution. Note that all metavariables are of base type, hence the metavariable substitution substitutes atomic terms only, and cannot create new redices. □

Lemma 7.3.11 (Completeness of pruning)

1. *If* M *is canonical (or atomic),* $\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}$, X *does not appear within* M, $\Psi;\Gamma \vdash \sigma : \Gamma_1$, *and there exists a* M'_1 *such that* $\mathsf{M} = \mathsf{M}'_1[\sigma]$, *then* $\Psi;\Gamma \vdash (\mathsf{M}\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{M}_1, \rho_1, \Psi_1)$.
2. *If* A *is canonical (or atomic),* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}$, X_A *does not appear within* A, $\Psi;\Gamma \vdash \sigma : \Gamma_1$, *and there exists a* A'_1 *such that* $\mathsf{A} = \mathsf{A}'_1[\sigma]$, *then* $\Psi;\Gamma \vdash (\mathsf{A}\backslash\mathsf{X})[\sigma]^{-1} \Longrightarrow (\mathsf{A}_1, \rho_1, \Psi_1)$.

Proof

By induction on the structure of the canonical object or family.

□

7.3.4 Correctness of the algorithm

We will prove that the algorithm is correct for all problems in the pattern fragment. First, the algorithm is sound, in that if the algorithm returns an answer substitution, the substitution applied to the two terms indeed produces equal results. Next, the algorithm is complete. This means that for a higher-order pattern unification problem, if there exists any solution, then the algorithm produces a different solution. Further, the produced solution substitution is most general, and can be produce any other solution by composition of substitutions.

Theorem 7.3.12 (Soundness of algorithm)

1. *If* M_1 *and* M_2 *are both canonical (or both atomic) objects,* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$, $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$ *and* $\Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2 \Rightarrow (\rho, \Psi_1)$ *then* $\mathsf{M}_1[\rho] = \mathsf{M}_2[\rho]$.

2. *If* A_1 *and* A_2 *are both canonical (or both atomic) families,* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$, $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$ *and* $\Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2 \Rightarrow (\rho, \Psi_1)$ *then* $\mathsf{A}_1[\rho] = \mathsf{A}_2[\rho]$.

Proof

By induction on the unification judgment. We show a few representative cases.

Case 1:
$$\frac{\Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{M}_1 =^? \mathsf{M}_2 \Rightarrow (\rho, \Psi_1)}{\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_1 =^? \lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_2 \Rightarrow (\rho, \Psi_1)}\ \textsc{Unif-ObjAbs}$$

$\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_1 : \mathsf{A}_1$,	
$\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_2 : \mathsf{A}_1$	By assumption
$\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$,	
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{M}_1 : \mathsf{A}_2$, and	
$\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{A}_2 \equiv \mathsf{A}_1 : \mathsf{type}$	By typing inversion
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{M}_2 : \mathsf{A}_3$, and	
$\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{A}_3 \equiv \mathsf{A}_1 : \mathsf{type}$	By typing inversion
$\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{A}_2 \equiv \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{A}_3 : \mathsf{type}$	By symmetry and transitivity
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A} \vdash \mathsf{A}_2 \equiv \mathsf{A}_3 : \mathsf{type}$	By injectivity
$\mathsf{M}_1[\rho] = \mathsf{M}_2[\rho]$	By induction
$\mathsf{A}[\rho] = \mathsf{A}[\rho]$	Equal elements
$\lambda\mathsf{x}{:}(\mathsf{A}[\rho]).(\mathsf{M}_1[\rho]) = \lambda\mathsf{x}{:}(\mathsf{A}[\rho]).(\mathsf{M}_2[\rho])$	From above
$(\lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_1)[\rho] = (\lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}_2)[\rho]$	By definition of substitution

Case 2:
$$\frac{\begin{array}{c}\Psi;\Gamma \vdash \mathsf{A}_{11} =^? \mathsf{A}_{21} \Rightarrow (\rho_1, \Psi_1) \\ \Psi_1;[\rho_1]\Gamma, \mathsf{x}{:}[\rho_1]\mathsf{A}_{11} \vdash \mathsf{A}_{12}[\rho_1] =^? \mathsf{A}_{22}[\rho_1] \Rightarrow (\rho_2, \Psi_2)\end{array}}{\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} =^? \Sigma\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} \Rightarrow (\rho_2 \circ \rho_1, \Psi_2)}\ \textsc{Unif-Sigma}$$

$\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} : \mathsf{type}$,	
$\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} : \mathsf{type}$	By assumption
$\Psi;\Gamma \vdash \mathsf{A}_{11} : \mathsf{type}$, and	
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{11} \vdash \mathsf{A}_{12} : \mathsf{type}$	By typing inversion
$\Psi;\Gamma \vdash \mathsf{A}_{21} : \mathsf{type}$, and	
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_{21} \vdash \mathsf{A}_{22} : \mathsf{type}$	By typing inversion
$\mathsf{A}_{11}[\rho_1] = \mathsf{A}_{21}[\rho_1]$	By induction
$\Psi_1;\Gamma[\rho_1], \mathsf{x}{:}(\mathsf{A}_{11}[\rho_1]) \vdash \mathsf{A}_{12}[\rho_1] : \mathsf{type}$	By substitution
$\Psi_1;\Gamma[\rho_1], \mathsf{x}{:}(\mathsf{A}_{21}[\rho_1]) \vdash \mathsf{A}_{22}[\rho_1] : \mathsf{type}$	By substitution
$\Psi_1;\Gamma[\rho_1], \mathsf{x}{:}(\mathsf{A}_{11}[\rho_1]) \vdash \mathsf{A}_{22}[\rho_1] : \mathsf{type}$	Replacing by equal element
$(\mathsf{A}_{12}[\rho_1])[\rho_2] = (\mathsf{A}_{22}[\rho_1])[\rho_2]$	By induction
$\mathsf{A}_{12}[\rho_2 \circ \rho_1] = \mathsf{A}_{22}[\rho_2 \circ \rho_1]$	By composing substitutions
$\mathsf{A}_{11}[\rho_2 \circ \rho_1] = \mathsf{A}_{12}[\rho_2 \circ \rho_1]$	By extending substitution
$\Pi\mathsf{x}{:}(\mathsf{A}_{11}[\rho_2 \circ \rho_1]).(\mathsf{A}_{12}[\rho_2 \circ \rho_1]) = \Pi\mathsf{x}{:}(\mathsf{A}_{21}[\rho_2 \circ \rho_1]).(\mathsf{A}_{22}[\rho_2 \circ \rho_1])$	From above
$(\Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12})[\rho_2 \circ \rho_1] = (\Pi\mathsf{x}{:}\mathsf{A21}.\mathsf{A}_{22})[\rho_2 \circ \rho_1]$	By definition of substitutions

Case 3:

$$\frac{\begin{array}{l}\Psi = \Psi_1, \mathsf{X_M}{:}\Gamma_1 \vdash \mathsf{A}_1, \Psi_2 \\ \Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2 \\ \Psi_2' = \Psi_2[\mathsf{id}_{\Psi_1}, \mathsf{X_{M1}}[\mathsf{id}_{\Gamma_2}]/\mathsf{X_M}] \\ \Psi' = \Psi_1, \mathsf{X_{M1}}{:}(\Gamma_2 \vdash \mathsf{A}_1), \Psi_2' \\ \rho = \mathsf{id}_{\Psi_1}, \mathsf{X_{M1}}[\mathsf{id}_{\Gamma_2}]/\mathsf{X_M}, \mathsf{id}_{\Psi_2'} \\ (\mathsf{X_{M1}} \notin \mathsf{dom}(\Psi))\end{array}}{\Psi;\Gamma \vdash \mathsf{X_M}[\sigma_1] =^{?} \mathsf{X_M}[\sigma_2] \Rightarrow (\rho, \Psi')}\ \textsc{Unif-ObjSameVar}$$

$\Psi = \Psi_1, \mathsf{X_M}{:}\Gamma_1 \vdash \mathsf{A}_1, \Psi_2$	By premise
$\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2$	By premise
$\sigma_1[\mathsf{id}_{\Gamma_2}] = \sigma_2[\mathsf{id}_{LFctx_2}]$	By correctness of intersection
$\mathsf{X_{M1}}[\mathsf{id}_{\Gamma_2} \circ \sigma_1] = \mathsf{X_{M1}}[\mathsf{id}_{\Gamma_2} \circ \sigma_2]$	By composition
$(\mathsf{X_{M1}}[\sigma_1])[\mathsf{id}_{\Gamma_2}] = (\mathsf{X_{M1}}[\sigma_2])[\mathsf{id}_{\Gamma_2}]$	By definition of substitution
$(\mathsf{X_M}[\sigma_1])[\rho] = (\mathsf{X_M}[\sigma_2])[\rho]$	By definition of substitution

Case 4:

$$\frac{\begin{array}{l}\Psi;\Gamma \vdash (\mathsf{M} \backslash \mathsf{X_M})[\sigma]^{-1} \Longrightarrow (\mathsf{M}_1, \rho_1, \Psi_1) \\ \Psi_1 = \Psi_{11}, \mathsf{X_M}{:}\Gamma_1 \vdash \mathsf{A}, \Psi_{12} \\ \Psi_{22} = \Psi_{12}[\mathsf{id}_{\Psi_{11}}, (\mathsf{M}_1[\rho_1])/\mathsf{X_M}]\end{array}}{\Psi;\Gamma \vdash \mathsf{X_M}[\sigma] =^{?} \mathsf{M} \Rightarrow ((\mathsf{id}_{\Psi_{11}}, (\mathsf{M}_1[\rho_1])/\mathsf{X_M}, \mathsf{id}_{\Psi_{22}}), (\Psi_{11}, \Psi_{22}))}\ \textsc{Unif-ObjDiffVar1}$$

$(\mathsf{X_M}[\sigma])[\rho] = (\mathsf{X_M}[\rho])[\sigma[\rho]]$	By distributing metavariable substitution
$(\mathsf{X_M}[\sigma])[\rho] = (\mathsf{X_M}[\rho])[\sigma]$	By pattern substitutions untouched
$(\mathsf{X_M}[\sigma])[\rho] = (\mathsf{M}_1[\rho_1])[\sigma]$	By substitution
$\mathsf{M} = (\mathsf{M}_1[\rho_1])[\sigma]$	By soundness of pruning
M does not contain $\mathsf{X_M}$	By properties of pruning
$\mathsf{M}[\rho] = (\mathsf{M}_1[\rho_1])[\sigma]$	By definition of substitution

□

Theorem 7.3.13 (Completeness of algorithm)

1. *Assume that* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$, $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$ *and* M_1 *and* M_2 *are both canonical (or both atomic) objects. Further, assume there is a metavariable substitution* ρ *such that* $\mathsf{M}_1[\rho] = \mathsf{M}_2[\rho]$ *and* $\Psi' \vdash \rho : \Psi$. *Then* $\Psi;\Gamma \vdash \mathsf{M}_1 =^{?} \mathsf{M}_2 \Rightarrow (\rho_1, \Psi_1)$ *such that* $\Psi \vdash \rho_1 : \Psi_1$ *and* $\rho = \rho_2 \circ \rho_1$ *for some* ρ_2.
2. *Assume that* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$, $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$ *and* A_1 *and* A_2 *are both canonical (or both atomic) families. Further, assume there is a metavariable substitution* ρ *such that* $\mathsf{A}_1[\rho] = \mathsf{A}_2[\rho]$. *Then* $\Psi;\Gamma \vdash \mathsf{A}_1 =^{?} \mathsf{A}_2 \Rightarrow (\rho_1, \Psi_1)$ *such that* $\Psi \vdash \rho_1 : \Psi_1$ *and* $\rho = \rho_2 \circ \rho_1$ *for some* ρ_2.

Proof

By induction on the structure of the term. We will show a few illustrative cases.

Case 1: $\mathsf{M}_1 = \lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{11}$ and $\mathsf{M}_2 = \lambda \mathsf{x}{:}\mathsf{A}_2.\mathsf{M}_{21}$

$(\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{11})[\rho] = (\lambda \mathsf{x}{:}\mathsf{A}_2.\mathsf{M}_{21})[\rho]$	By assumption
$\lambda \mathsf{x}{:}(\mathsf{A}_1[\rho]).(\mathsf{M}_{11}[\rho]) = \lambda \mathsf{x}{:}(\mathsf{A}_2[\rho]).(\mathsf{M}_{21}[\rho])$	By definition of substitution
$\mathsf{M}_{11}[\rho] = \mathsf{M}_{21}[\rho]$	By definition of substitution
$\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{11} : \mathsf{A}$	By assumption
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_{11} : \mathsf{A}_{11}$,	
$\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_{11} \equiv \mathsf{A} : \mathsf{type}$	By typing inversion
$\Psi;\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{A}_2.\mathsf{M}_{21} : \mathsf{A}$	By assumption
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_2 \vdash \mathsf{M}_{21} : \mathsf{A}_{21}$,	
$\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_{21} \equiv \mathsf{A} : \mathsf{type}$	By typing inversion
$\Psi;\Gamma \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_{11} \equiv \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_{21} : \mathsf{type}$	By symmetry and transitivity
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}$,	

$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_{11} \equiv \mathsf{A}_{21} : \mathsf{type}$ By injectivity
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_{21} : \mathsf{A}_{21}$ By context conversion
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_{21} : \mathsf{A}_{11}$ By type conversion rule
$\Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_{11} =^? \mathsf{M}_{21} \Rightarrow (\rho_1, \Psi_1)$,
$\Psi \vdash \rho_1 : \Psi_1$,
$\rho = \rho_2 \circ \rho_1$ By induction
$\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{11} =^? \lambda\mathsf{x}{:}\mathsf{A}_2.\mathsf{M}_{21} \Rightarrow (\rho_1, \Psi_1)$ By rule

Case 2: $\mathsf{M}_1 = \mathsf{X}_\mathsf{M}[\sigma_1]$ and $\mathsf{M}_2 = \mathsf{X}_\mathsf{M}[\sigma_2]$

$\Psi;\Gamma \vdash \mathsf{X}_\mathsf{M}[\sigma_1] : \mathsf{A}$ By assumption
$\Psi = \Psi_1, \mathsf{X}_\mathsf{M}{:}(\Gamma_1 \vdash \mathsf{A}_1), \Psi_2$,
$\Psi;\Gamma \vdash \sigma_1 : \Gamma_1$,
$\Psi;\Gamma \vdash \mathsf{A}_1[\sigma_1] \equiv \mathsf{A} : \mathsf{type}$ By typing inversion
$\Psi;\Gamma \vdash \mathsf{X}_\mathsf{M}[\sigma_2] : \mathsf{A}$ By assumption
$\Psi;\Gamma \vdash \sigma_2 : \Gamma_1$,
$\Psi;\Gamma \vdash \mathsf{A}_1[\sigma_2] \equiv \mathsf{A} : \mathsf{type}$ By typing inversion
$\Psi \vdash \sigma_1 \cap \sigma_2 : \Gamma_1 \Rightarrow \Gamma_2$ By completeness of intersection
Create a new metavariable $\mathsf{X}_{\mathsf{M}1} \notin \mathsf{dom}(\Psi)$
$\Psi;\Gamma \vdash \mathsf{X}_\mathsf{M}[\sigma_1] =^? \mathsf{X}_\mathsf{M}[\sigma_2] \Rightarrow (\mathsf{id}_{\Psi_1}, \mathsf{X}_{\mathsf{M}1}[\mathsf{id}_{\Gamma_2}]/\mathsf{X}_\mathsf{M}, \mathsf{id}_{\Psi_2'}, \Psi_1, \mathsf{X}_{\mathsf{M}1}{:}(\Gamma_2 \vdash \mathsf{A}), \Psi_2')$ By rule
$(\mathsf{X}_\mathsf{M}[\sigma_1])[\rho] = (\mathsf{X}_\mathsf{M}[\sigma_2])[\rho]$ By assumption
Required substitution ρ_2 is
  Replace $(\mathsf{X}_\mathsf{M}[\rho])/\mathsf{X}_{\mathsf{M}1}$ in ρ

Case 3: $\mathsf{M}_1 = \mathsf{X}_\mathsf{M}[\sigma]$ and $\mathsf{M}_2 \neq \mathsf{X}_\mathsf{M}[\sigma']$

$(\mathsf{X}_\mathsf{M}[\sigma])[\rho] = \mathsf{M}_2[\rho]$ By assumption
X_M not in M_2 By syntactic substitution property
$\rho = \rho_1, \mathsf{M}/\mathsf{X}_\mathsf{M}, \rho_2$,
such that $\mathsf{M}[\sigma] = \mathsf{M}_2[\rho]$ By definition of substitution
$\Psi;\Gamma \vdash (\mathsf{M}_2[\rho] \backslash \mathsf{X}_\mathsf{M})[\sigma]^{-1} \Longrightarrow (\mathsf{M}, \rho_1, \Psi_1)$ By completeness of pruning
$\Psi = \Psi_1, \mathsf{X}_\mathsf{M}{:}\Gamma_1\mathsf{A}, \Psi_2$ By typing inversion
$\Psi;\Gamma \vdash \mathsf{X}_\mathsf{M}[\sigma] : \mathsf{A}[\sigma]$ By rule
$\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}[\sigma]$ By assumption
$\Psi;\Gamma \vdash (\mathsf{M}_2 \backslash \mathsf{X}_\mathsf{M})[\sigma]^{-1} \Longrightarrow (\mathsf{M}_{21}, \rho_u, \Psi_u)$,
$\exists \rho_3 . \rho = \rho_3 \circ \rho_u$,
$\Psi' \vdash \rho_3 : \Psi_u$ By pruning under substitution
$\Psi_u \vdash \rho_2 : \Psi$ By well-formedness of pruning
$\rho_u = \rho_{u1}, \mathsf{X}_\mathsf{M}'[\mathsf{id}_{\Gamma_1[\rho]}]/\mathsf{X}_\mathsf{M}, \rho_{u2}$,
$\Psi_u = \Psi_{u1}, \mathsf{X}_\mathsf{M}'{:}\Gamma_1[\rho] \vdash \mathsf{A}[\rho], \Psi_{u2}$ By properties of pruning
$\Psi;\Gamma \vdash \mathsf{X}_\mathsf{M}[\sigma] =^? \mathsf{M}_2 \Rightarrow ((\mathsf{id}_{\Psi_{u1}}, (\mathsf{M}_{21}[\rho_u])/\mathsf{X}_\mathsf{M}, \mathsf{id}_{\Psi_{u2}[\mathsf{M}_{21}/\mathsf{X}_\mathsf{M}]}), (\Psi_{u1}, \Psi_{u2}[(\mathsf{M}_{21}[\rho_u])/\mathsf{X}_\mathsf{M}]))$ By rule
$\rho_3 = \rho_{31}, \mathsf{M}_{21}[\rho_3 \circ \rho_u]/\mathsf{X}_\mathsf{M}, \rho_{32}$ By definition of substitution
$\rho_3 = \rho_{31}, \mathsf{M}_{21}[\rho_u]/\mathsf{X}_\mathsf{M}, \rho_{32}$ Since $\mathsf{X}_\mathsf{M} \notin \mathsf{M}_{21}$
Required substitution is then ρ_{31}, ρ_{32} By composition of substitution

□

7.4 A constraint formulation

Based on the pattern unification algorithm, we now give a constraint formulation of unification. The language of constraints is as follows:

Constraints	$\mathcal{C} ::=$	true	True Constraint
	$\mid$	$\mathcal{A} \wedge \mathcal{C}$	Conjunction
Atomic Constraints	$\mathcal{A} ::=$	$\mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A}$	Object Equality
	$\mid$	$\mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{K}$	Family Equality

Further, the result of solving constraints is either failure, or a (possibly partial) solution with possibly some postponed constraints.

Solutions	$\mathcal{S} ::=$	false	Failure
	$\mid$	$(\rho, \Psi, \mathcal{C})$	(Partial) Success

The solution is given as follows:

$\boxed{\Psi;\Gamma \vdash \mathcal{C} \Longrightarrow \mathcal{S}}$ We start with the base case of true, in which case we are done. Next, if the constraint lies in the pattern fragment, we apply the algorithm, since this fragment is decidable. If we succeed with an answer substitution, we apply this substitution and proceed. If the unification fails, we can issue an immediate failure.

$$\frac{}{\Psi;\Gamma \vdash \mathsf{true} \Longrightarrow (\mathsf{id}_\Psi, \Psi, \mathsf{true})} \qquad \frac{\begin{array}{l}\mathsf{M}_1, \mathsf{M}_2 \in \text{ pattern fragment} \\ \Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2 \Rightarrow (\rho_1, \Psi_1) \\ \Psi_1;\Gamma[\rho_1] \vdash \mathcal{C}[\rho_1] \Longrightarrow \mathcal{S}\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A} \wedge \mathcal{C} \Longrightarrow \mathcal{S}} \qquad \frac{\begin{array}{l}\mathsf{A}_1, \mathsf{A}_2 \in \text{ pattern fragment} \\ \Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2 \Rightarrow (\rho_1, \Psi_1) \\ \Psi_1;\Gamma[\rho_1] \vdash \mathcal{C}[\rho_1] \Longrightarrow \mathcal{S}\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{K} \wedge \mathcal{C} \Longrightarrow \mathcal{S}}$$

$$\frac{\begin{array}{l}\mathsf{M}_1, \mathsf{M}_2 \in \text{ pattern fragment} \\ \text{Unification algorithm fails with } \Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A} \wedge \mathcal{C} \Longrightarrow \mathsf{false}} \qquad \frac{\begin{array}{l}\mathsf{A}_1, \mathsf{A}_2 \in \text{ pattern fragment} \\ \text{Unification algorithm fails with } \Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{K} \wedge \mathcal{C} \Longrightarrow \mathsf{false}}$$

Now we turn to the case where the constraint is not in the pattern fragment. If there are no other constraints, we postpone this constraint.

$$\frac{\mathsf{M}_1, \mathsf{M}_2 \notin \text{ pattern fragment}}{\Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A} \wedge \mathsf{true} \Longrightarrow (\mathsf{id}_\Psi, \Psi, \mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A} \wedge \mathsf{true})}$$

$$\frac{\mathsf{A}_1, \mathsf{A}_2 \notin \text{ pattern fragment}}{\Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{K} \wedge \mathsf{true} \Longrightarrow (\mathsf{id}_\Psi, \Psi, \mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{K} \wedge \mathsf{true})}$$

If there are other nontrivial constraints, we try to solve them first. If there is a failure, we can also issue a failure. If not, we can retry the constraint with the new solution.

$$\frac{\begin{array}{l}\mathsf{M}_1, \mathsf{M}_2 \notin \text{ pattern fragment} \\ \mathcal{C} \neq \mathsf{true} \\ \Psi;\Gamma \vdash \mathcal{C} \Longrightarrow \mathsf{false}\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A} \wedge \mathcal{C} \Longrightarrow \mathsf{false}} \qquad \frac{\begin{array}{l}\mathsf{A}_1, \mathsf{A}_2 \notin \text{ pattern fragment} \\ \mathcal{C} \neq \mathsf{true} \\ \Psi;\Gamma \vdash \mathcal{C} \Longrightarrow \mathsf{false}\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{A} \wedge \mathcal{C} \Longrightarrow \mathsf{false}}$$

$$\frac{\begin{array}{l}\mathsf{M}_1, \mathsf{M}_2 \notin \text{ pattern fragment} \\ \mathcal{C} \neq \mathsf{true} \\ \Psi;\Gamma \vdash \mathcal{C} \Longrightarrow (\rho_1, \Psi_1, \mathcal{C}_1) \\ \mathcal{C}_1 \neq \mathcal{C} \\ \Psi_1;\Gamma[\rho_1] \vdash \mathsf{M}_1[\rho_1] =^? \mathsf{M}_2[\rho_1] : \mathsf{A}[\rho_1] \wedge \mathcal{C}_1 \Longrightarrow \mathcal{S}\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A} \wedge \mathcal{C} \Longrightarrow \mathcal{S}} \qquad \frac{\begin{array}{l}\mathsf{A}_1, \mathsf{A}_2 \notin \text{ pattern fragment} \\ \mathcal{C} \neq \mathsf{true} \\ \Psi;\Gamma \vdash \mathcal{C} \Longrightarrow (\rho_1, \Psi_1, \mathcal{C}_1) \\ \mathcal{C}_1 \neq \mathcal{C} \\ \Psi_1;\Gamma[\rho_1] \vdash \mathsf{A}_1[\rho_1] =^? \mathsf{A}_2[\rho_1] : \mathsf{K}[\rho_1] \wedge \mathcal{C}_1 \Longrightarrow \mathcal{S}\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{K} \wedge \mathcal{C} \Longrightarrow \mathcal{S}}$$

$$\frac{\begin{array}{l}\mathsf{M}_1, \mathsf{M}_2 \notin \text{ pattern fragment} \\ \mathcal{C} \neq \mathsf{true} \\ \Psi;\Gamma \vdash \mathcal{C} \Longrightarrow (\mathsf{id}_\Psi, \Psi, \mathcal{C})\end{array}}{\Psi;\Gamma \vdash \mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A} \wedge \mathcal{C} \Longrightarrow (\mathsf{id}_\Psi, \Psi, \mathsf{M}_1 =^? \mathsf{M}_2 : \mathsf{A} \wedge \mathcal{C})}$$

$$\frac{\begin{array}{l}\mathsf{A}_1, \mathsf{A}_2 \notin \text{ pattern fragment} \\ \mathcal{C} \neq \mathsf{true} \\ \Psi;\Gamma \vdash \mathcal{C} \Longrightarrow (\mathsf{id}_\Psi, \Psi, \mathcal{C})\end{array}}{\Psi;\Gamma \vdash \mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{K} \wedge \mathcal{C} \Longrightarrow (\mathsf{id}_\Psi, \Psi, \mathsf{A}_1 =^? \mathsf{A}_2 : \mathsf{K} \wedge \mathcal{C})}$$

The important theorem about the constraint solving system is that it is correct. If we already know a solution, then the constraint system will not fail, and further, will discover the solution or a part thereof. The partial nature of the solution comes from the fact that the posed constraint may not lie within the pattern fragment.

To state the theorem, we introduce some notation. We let the metavariable $\mathcal{P}$ range over LF terms M and A, and the metavariable $\mathcal{Q}$ range over LF classifiers A and K. Assume $\mathcal{C} = \mathcal{P}_{11} =^? \mathcal{P}_{12} : \mathcal{Q}_1 \wedge \ldots \wedge \mathsf{true}$. We say $\Psi_1;\Gamma_1 \vdash \mathcal{C}[\rho] = \mathsf{true}$ if for every equation in $\mathcal{C}$, $\Psi_1;\Gamma_1 \vdash \mathcal{P}_{i1}[\rho] \equiv \mathcal{P}_{i2}[\rho] : \mathcal{Q}_i[\rho]$.

Theorem 7.4.1 (Correctness of constraint solving) *Suppose we have* $\vdash \Psi$, $\Psi \vdash \Gamma : \mathsf{ctx}$. *Suppose* $\mathcal{C} = \mathcal{P}_{11} =^? \mathcal{P}_{12} : \mathcal{Q}_1 \wedge \ldots \wedge \mathsf{true}$, *where for every* i, $\Psi;\Gamma \vdash \mathcal{P}_{i1} : \mathcal{Q}_i$, $\Psi;\Gamma \vdash \mathcal{P}_{i2} : \mathcal{Q}_i$, *Suppose further that we already know* Ψ_1, Γ_1, *and* ρ *such that* $\Psi_1;\Gamma_1 \vdash \mathcal{C}[\rho] = \mathsf{true}$. *Then*

1. *It is not the case that* $\Psi;\Gamma \vdash \mathcal{C} \Longrightarrow \mathsf{false}$.
2. *If* $\Psi;\Gamma \vdash \mathcal{C} \Longrightarrow (\rho_2, \Psi_2, \mathcal{C}_2)$, *then there exists a substitution* ρ_2 *such that* $\rho = \rho_2 \circ \rho_1$, $\Psi_1 \vdash \rho_2 : \Psi_2$, *and* $\Psi_1;\Gamma_2 \vdash \mathcal{C}_1[\rho_2] = \mathsf{true}$.

Proof

By induction on the structure of the constraint. We decompose the constraint into atomic constraints and consider them left to right.

Case 1: true

Then we have nothing left to do.

Case 2: $\mathcal{P}_1 =^? \mathcal{P}_2 : \mathcal{Q}$ is in pattern fragment

In this case, the completeness of the unification algorithm gives us that the unification algorithm does not fail. Further, the required substitution for the metavariables in this equation is obtained from unification. We proceed inductively on the rest of the constraint.

Case 3: $\mathcal{P}_1 =^? \mathcal{P}_2 : \mathcal{Q}$ is not in the pattern fragment

We proceed based on whether there are other non-trivial constraints.

Subcase 3.1: There are no more non-trivial constraints

From rule, constraint solving does not fail, but exits with the non-pattern constraint. We know from premises that there is a solution. We can output this solution itself.

Subcase 3.2: There exists at least one more non-trivial constraint

We proceed inductively on the smaller constraint. We inductively get that the constraint solving does not fail. If we get the same answer back, we output the solution we get as input. If not, then inductively we have part of a valid solution. We can retry to see whether the constraint in question falls in the pattern fragment. By completeness, it cannot fail. In either case, we can augment the inner solution with the valid solution that we have as input.

□

Chapter 8

The programming language LF/ML

We are now in a position to embed the calculus $\mathrm{LF}^{\Sigma,1+}$ in the previous two chapters into a functional language similar to ML. This language, called LF/ML, is the calculus we use to write provably correct checking programs. Thus, apart from desirable properties like safety, the language should also provide a means of proving correctness properties of programs statically. We will use a restricted form of dependent types, and use the canonical forms property to argue about correctness. The restrictions on dependent types is inspired by Dependent ML [XP99], and allows us to have effective type checking. It also cleanly separates the representation language $\mathrm{LF}^{\Sigma,1+}$, ensuring it is purely static and has no dynamic significance. Indeed, we can have an erasure semantics of LF/ML to core ML.

In the following, we wish to ignore the issue of type inference. The practical implementation does have to perform inference, which is done by a two stage process, giving ML types by Hindley-Milner inference and then performing more precise analysis. For studying the properties of the language, we prefer to work with an explicitly typed internal language, which we will call $\mathrm{LF}/\mathrm{F}^{\omega}$. This is an extension of the F^{ω} language with dependent function and pair types, and a notion of datatypes. The dependencies are allowed only over closed $\mathrm{LF}^{\Sigma,1+}$ objects and type families. This will be the means of introducing metavariables, which will be constrained to stand for closed LF objects and type families.

8.1 Type Structure of $\mathrm{LF}/\mathrm{F}^{\omega}$

The language will be defined in two stages. First, we talk about the type structure. Using the previous results, we come up with an effective algorithm to decide equivalence at the level of type constructors, and as a corollary get consistency of the type structure.

8.1.1 Abstract Syntax

We let the metavariable $\mathcal{P}$ range over LF terms M and A, and the metavariable $\mathcal{Q}$ range over LF classifiers A and K. The metavariable w ranges over object and family level metavariables X_{M} and X_{A}. The abstract syntax of kinds and type constructors is presented in figure 8.1.

Type constructors are classified by kinds. The type constructors belonging to the kind $\mathbb{T}$ are called types, and are ranged over by the metavariable τ. As is usual, we have a base type Unit, arrows, products and forall types. The new features over a standard presentation of F^{ω} is the presence of Dependent Product and Sum Types, and declared datatypes. Datatypes are assigned signatures by a signature $\mathcal{S}$. Dependent product and sum types are indexed over empty sorts. That is, they assume a new LF metavariable which is well-formed in the empty context.

Type constructor substitutions substitute constructors for constructor variables.

Constructor Substitutions	$\sigma ::=$	$\cdot$	Nil
	$\mid$	$\sigma, \mathsf{c}/\alpha$	Cons

Indices	$\mathcal{P}$ ::=	M	
	|	A	
Sorts	$\mathcal{Q}$ ::=	A	
	|	K	
Kinds	κ ::=	$\mathbb{T}$	Kind of Types
	|	$\kappa_1 \rightarrow \kappa_2$	Function Kind
Type Constructors	c, τ ::=	Unit	Unit Type
	|	$\tau_1 \rightarrow \tau_2$	Arrow Type
	|	$\tau_1 \times \tau_2$	Product Type
	|	$\Pi w{:}(\cdot \vdash \mathcal{Q}).\tau$	Universal Dependent Types
	|	$\Sigma w{:}(\cdot \vdash \mathcal{Q}).\tau$	Existential Dependent Types
	|	$\mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)$	Datatypes
	|	$\forall(\alpha{:}\kappa_1).\tau$	Forall Type
	|	α	Constructor Variables
	|	$\lambda(\alpha{:}\kappa_1).\mathsf{c}_2$	Constructor Level Function
	|	$\mathsf{c}_1\ \mathsf{c}_2$	Constructor Level Application
Signatures	$\mathcal{S}$ ::=	$\cdot$	
	|	$\mathcal{S}, \mathsf{D}{:}\Pi w_1{:}(\cdot \vdash \mathcal{Q}_1). \ldots \Pi w_n{:}(\cdot \vdash \mathcal{Q}_n).\kappa$	
Contexts	Δ ::=	$\cdot$	
	|	$\Delta, \alpha{:}\kappa$	

Figure 8.1: Abstract Syntax: Constructors and kinds

As is usual, we denote the identity substitution on the kinding context Δ by id_Δ.

8.1.2 Static Semantics

The static semantics is defined in terms of the following judgments.

$\vdash \mathcal{S}$	$\mathcal{S}$ is a valid signature
$\Psi;\Delta \vdash \mathsf{c} : \kappa$	c is a valid type constructor
$\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$	c_1 and c_2 are equal

We will now present the static semantics of the constructor levels. We assume given a signature which provides datatype (and later, constant) definitions. The signature is well-formed exactly when all datatypes are defined to depend on LF classifiers that are well-formed in the closed context. Further, the LF classifiers can only contain metavariables from the list of indices of the datatype. This is shown in figure 8.2.

Type contexts are always well-formed since all kinds that can be written are well-formed.

Checking of constructors is done within two contexts, a metavariable context recording LF metavariable assumptions, and a type context recording kinding assumptions on constructor variables. This is shown in figure 8.3. Most cases such as abstraction, application and base types are standard for a F^ω-like calculus. The new cases are for the dependent types, which add to the LF metavariable context, and the datatypes, which performs checking.

We also have a notion of constructor equality, built by a congruence closure of β and η conversion rules. We define this in the form of a definitional equality rule. Because of the presence of extensionality, it is convenient to present equality in a well-kinded form. This is given in figures 8.4 and 8.5. Notice that in the dependent cases, we allow conversion of equal terms from the $\mathrm{LF}^{\Sigma,1+}$ theory.

Well Typed Substitutions The notation for typing substitutions is explained below.

Empty

$$\frac{}{\vdash \cdot : \mathsf{ok}}$$

Datatype Declarations

$$\frac{\vdash \mathcal{S} : \mathsf{ok} \qquad \forall 1 \leq i \leq n. \begin{cases} w_1{:}(\cdot \vdash \mathcal{Q}_1), \ldots, w_{i-1}{:}(\cdot \vdash \mathcal{Q}_{i-1}); \cdot \vdash \mathsf{A} : \mathsf{type} & \text{if } \mathcal{Q}_i = \mathsf{A} \\ w_1{:}(\cdot \vdash \mathcal{Q}_1), \ldots, w_{i-1}{:}(\cdot \vdash \mathcal{Q}_{i-1}); \cdot \vdash \mathsf{K} : \mathsf{kind} & \text{if } \mathcal{Q}_i = \mathsf{K} \end{cases}}{\vdash \mathcal{S}, \mathsf{D}{:}\Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\kappa}$$

Figure 8.2: Well formed signatures

Definition 8.1.1 *The judgment* $\Psi; \Delta_2 \vdash \sigma : \Gamma_1$ *holds iff* $\forall \alpha \in \mathsf{dom}(\Delta_1).\Psi; \Delta_2 \vdash \sigma(\alpha) : \sigma(\Delta_1(\alpha))$.

Definition 8.1.2 *The judgment* $\Psi; \Delta_2 \vdash \sigma_1 \equiv \sigma_2 : \Delta_1$ *holds iff*

- $\Psi; \Delta_2 \vdash \sigma_1 : \Delta_1$,
- $\Psi; \Delta_2 \vdash \sigma_2 : \Delta_1$, *and*
- $\forall \alpha \in \mathsf{dom}(\Delta_1).\Psi; \Delta_2 \vdash \sigma_1(\alpha) \equiv \sigma_2(\alpha) : \sigma_1(\Delta_1(\alpha))$.

8.1.3 Proof of consistency

The principal result required of the constructor level of the language is of consistency. For example, we must not be able to treat something of function type as if it was a product, or confuse two dependent functions. This requires us to compare two type constructors for equality. We use a similar proof to that in $\mathrm{LF}^{\Sigma,1+}$, but simpler in that this structure is not fully dependent. We therefore present a kind-directed algorithm, and prove it sound and complete. To check dependent types, we have to call the equality algorithm of chapter 6. As before, the proof of completeness is done by means of a Kripke logical relation to encode, in the higher kind case, the strengthened inductive hypothesis required for extensionality.

We begin by proving some structural properties of the kinding system, such as substitution, regularity and kinding inversion. We then produce a kind-directed algorithm for checking constructor equality. We perform a logical relations style completeness argument by induction on the kinds at which constructors are compared. The soundness proof is standard induction on the algorithm. With soundness and completeness proved, a variety of consistency results can be proved.

8.1.4 Structural Properties

We begin by proving some elementary structural properties of the kinding system. The proofs for the most part are by an easy structural induction on the derivations.

Lemma 8.1.3 (Weakening) *For* $\mathcal{J} \in \{\mathsf{c} : \kappa, \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa\}$, *if* $\Psi_1; \Delta_1 \vdash \mathcal{J}$ *and* $\Delta_1 \subseteq \Delta_2$, $\Psi_1 \subseteq \Psi_2$ *then* $\Psi_2; \Delta_2 \vdash \mathcal{J}$.

Proof

By induction on the structure of the derivation.

□

Lemma 8.1.4 (Free Variable Containment) *For* $\mathcal{J} \in \{\mathsf{c} : \kappa, \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa\}$, *if* $\vdash \Psi$, $\Psi \vdash \Delta$ *and* $\Psi; \Delta \vdash \mathcal{J}$ *then*
$FV(\mathcal{J}) \in \mathsf{dom}(\Delta) \cup \mathsf{dom}(\Psi)$.

Proof

By induction on the structure of the derivation.

□

Unit

$$\frac{}{\Psi;\Delta \vdash \mathsf{Unit} : \mathbb{T}}$$

Arrow

$$\frac{\Psi;\Delta \vdash \tau_1 : \mathbb{T} \quad \Psi;\Delta \vdash \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \tau_1 \rightarrow \tau_2 : \mathbb{T}}$$

Product

$$\frac{\Psi;\Delta \vdash \tau_1 : \mathbb{T} \quad \Psi;\Delta \vdash \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \tau_1 \times \tau_2 : \mathbb{T}}$$

Dependent Universals

$$\frac{\Psi;\cdot \vdash \mathsf{A} : \mathsf{type} \quad \Psi, \mathsf{X_M}{:}(\cdot \vdash \mathsf{A});\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{X_M}{:}(\cdot \vdash \mathsf{A}).\tau : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind} \quad \Psi, \mathsf{X_A}{:}(\cdot \vdash \mathsf{K});\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{X_A}{:}(\cdot \vdash \mathsf{K}).\tau : \mathbb{T}}$$

Dependent Existentials

$$\frac{\Psi;\cdot \vdash \mathsf{A} : \mathsf{type} \quad \Psi, \mathsf{X_M}{:}(\cdot \vdash \mathsf{A});\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{X_M}{:}(\cdot \vdash \mathsf{A}).\tau : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind} \quad \Psi, \mathsf{X_A}{:}(\cdot \vdash \mathsf{K});\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{X_A}{:}(\cdot \vdash \mathsf{K}).\tau : \mathbb{T}}$$

Datatypes

$$\frac{\mathcal{S}(\mathsf{D}) = \Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\mathbb{T} \quad \forall 1 \leq i \leq n.\,(\Psi;\cdot \vdash \mathcal{P}_i : \mathcal{Q}_i)}{\Psi;\Delta \vdash \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n) : \mathbb{T}}$$

Universals

$$\frac{\Psi;\Delta, \alpha{:}\kappa \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \forall(\alpha{:}\kappa).\tau : \mathbb{T}}$$

Variables

$$\frac{\Delta(\alpha) = \kappa}{\Psi;\Delta \vdash \alpha : \kappa}$$

Abstractions

$$\frac{\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c} : \kappa_2}{\Psi;\Delta \vdash \lambda(\alpha{:}\kappa_1).\mathsf{c} : \kappa_1 \rightarrow \kappa_2}$$

Applications

$$\frac{\Psi;\Delta \vdash \mathsf{c}_1 : \kappa_1 \rightarrow \kappa_2 \quad \Psi;\Delta \vdash \mathsf{c}_2 : \kappa_1}{\Psi;\Delta \vdash \mathsf{c}_1\ \mathsf{c}_2 : \kappa_2}$$

Figure 8.3: Well-formed constructors

Unit

$$\frac{}{\Psi;\Delta \vdash \mathsf{Unit} \equiv \mathsf{Unit} : \mathbb{T}}$$

Arrows

$$\frac{\Psi;\Delta \vdash \tau_{11} \equiv \tau_{21} : \mathbb{T} \quad \Psi;\Delta \vdash \tau_{12} \equiv \tau_{22} : \mathbb{T}}{\Psi;\Delta \vdash \tau_{11} \to \tau_{12} \equiv \tau_{21} \to \tau_{22} : \mathbb{T}}$$

Products

$$\frac{\Psi;\Delta \vdash \tau_{11} \equiv \tau_{21} : \mathbb{T} \quad \Psi;\Delta \vdash \tau_{12} \equiv \tau_{22} : \mathbb{T}}{\Psi;\Delta \vdash \tau_{11} \times \tau_{12} \equiv \tau_{21} \times \tau_{22} : \mathbb{T}}$$

Dependent Universals

$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \quad \Psi, \mathsf{X}_\mathsf{M}{:}(\cdot \vdash \mathsf{A}_1);\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{X}_\mathsf{M}{:}(\cdot \vdash \mathsf{A}_1).\tau_1 \equiv \Pi \mathsf{X}_\mathsf{M}{:}(\cdot \vdash \mathsf{A}_2).\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \quad \Psi, \mathsf{X}_\mathsf{M}{:}(\cdot \vdash \mathsf{K}_1);\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{X}_\mathsf{A}{:}(\cdot \vdash \mathsf{K}_1).\tau_1 \equiv \Pi \mathsf{X}_\mathsf{A}{:}(\cdot \vdash \mathsf{K}_2).\tau_2 : \mathbb{T}}$$

Dependent Existentials

$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \quad \Psi, \mathsf{X}_\mathsf{M}{:}(\cdot \vdash \mathsf{A}_1);\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{X}_\mathsf{M}{:}(\cdot \vdash \mathsf{A}_1).\tau_1 \equiv \Sigma \mathsf{X}_\mathsf{M}{:}(\cdot \vdash \mathsf{A}_2).\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \quad \Psi, \mathsf{X}_\mathsf{M}{:}(\cdot \vdash \mathsf{K}_1);\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{X}_\mathsf{A}{:}(\cdot \vdash \mathsf{K}_1).\tau_1 \equiv \Sigma \mathsf{X}_\mathsf{A}{:}(\cdot \vdash \mathsf{K}_2).\tau_2 : \mathbb{T}}$$

Datatypes

$$\frac{\begin{array}{l}\mathcal{S}(\mathsf{D}) = \Pi w_1{:}(\cdot \vdash \mathcal{Q}_1). \ldots \Pi w_n{:}(\cdot \vdash \mathcal{Q}_n).\mathbb{T} \\ \forall 1 \le i \le n.\ (\Psi;\cdot \vdash \mathcal{P}_{1i} \equiv \mathcal{P}_{2i} : \mathcal{Q}_i[\mathcal{P}_{11}/w_1, \ldots, \mathcal{P}_{1i-1}/w_{i-1}]\,)\end{array}}{\Psi;\Delta \vdash \mathsf{D}(\mathcal{P}_{11} \ldots \mathcal{P}_{1n}) \equiv \mathsf{D}(\mathcal{P}_{21} \ldots \mathcal{P}_{2n}) : \mathbb{T}}$$

Universals

$$\frac{\Psi;\Delta, \alpha{:}\kappa \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \forall(\alpha{:}\kappa).\tau_1 \equiv \forall(\alpha{:}\kappa).\tau_2 : \mathbb{T}}$$

Variables

$$\frac{\Delta(\alpha) = \kappa}{\Psi;\Delta \vdash \alpha \equiv \alpha : \kappa}$$

Applications

$$\frac{\Psi;\Delta \vdash \mathsf{c}_{11} \equiv \mathsf{c}_{21} : \kappa_2 \to \kappa_1 \quad \Psi;\Delta \vdash \mathsf{c}_{12} \equiv \mathsf{c}_{22} : \kappa_2}{\Psi;\Delta \vdash \mathsf{c}_{11}\,\mathsf{c}_{12} \equiv \mathsf{c}_{21}\,\mathsf{c}_{22} : \kappa_1}$$

Abstractions

$$\frac{\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa_2}{\Psi;\Delta \vdash \lambda(\alpha{:}\kappa_1).\mathsf{c}_1 \equiv \lambda(\alpha{:}\kappa_1).\mathsf{c}_2 : \kappa_1 \to \kappa_2}$$

Symmetry

$$\frac{\Psi;\Delta \vdash \mathsf{c}_2 \equiv \mathsf{c}_1 : \kappa}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa}$$

Transitivity

$$\frac{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa \quad \Psi;\Delta \vdash \mathsf{c}_2 \equiv \mathsf{c}_3 : \kappa}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_3 : \kappa}$$

Figure 8.4: Constructor Equality: Congruence

Parallel β-Conversion

$$\frac{\Psi;\Delta,\alpha{:}\kappa_1 \vdash \mathsf{c}_{12} \equiv \mathsf{c}_{22} : \kappa_2 \qquad \Psi;\Delta \vdash \mathsf{c}_{11} \equiv \mathsf{c}_{21} : \kappa_1}{\Psi;\Delta \vdash (\lambda(\alpha{:}\kappa_1).\mathsf{c}_{12})\,\mathsf{c}_{11} \equiv [\mathsf{c}_{21}/\alpha]\,\mathsf{c}_{22} : \kappa_2}$$

Extensionality

$$\frac{\Psi;\Delta \vdash \mathsf{c}_1 : \kappa_1 \to \kappa_2 \quad \Psi;\Delta \vdash \mathsf{c}_2 : \kappa_1 \to \kappa_2 \quad \Psi;\Delta,\alpha{:}\kappa_1 \vdash \mathsf{c}_1\,\alpha \equiv \mathsf{c}_2\,\alpha : \kappa_2}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa_1 \to \kappa_2}$$

Figure 8.5: Constructor Equality: Conversion

Next, we want to show that well-formed constructors are equivalent to themselves. This shows that definitional equality is a equivalence relation, with symmetry and transitivity being explicit rules.

Lemma 8.1.5 (Reflexivity) *If* $\Psi;\Delta \vdash \mathsf{c} : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c} \equiv \mathsf{c} : \kappa$.

Proof

By induction on the structure of the kinding judgment.

□

The important property that substitution is admissible is proved next. This requires us to show some properties of constructor substitutions first. First we need a simple lemma saying that we can always create a well-typed identity substitution equivalent to itself. Then we need that a substitution can be extended by a single equivalent point. This is required in the case of substitution for abstraction, where the context is extended.

Lemma 8.1.6 (Identity Substitutions) *If* $\vdash \Psi$ *and* $\Psi \vdash \Delta$ *then* $\Psi;\Delta \vdash \mathsf{id}_\Delta \equiv \mathsf{id}_\Delta : \Delta$.

Proof

By induction on the construction of the context.

□

Lemma 8.1.7 (Extending Substitutions)

1. *If* $\Psi;\Delta_1 \vdash \sigma_1 : \Delta$ *and* $\alpha \notin \mathsf{dom}(\Delta) \cup \mathsf{dom}(\Delta_1)$ *then* $\Psi;\Delta_1,\alpha{:}\kappa \vdash \sigma_1,\alpha/\alpha : \Delta,\alpha{:}\kappa$.
2. *If* $\Psi;\Delta_1 \vdash \sigma_1 \equiv \sigma_2 : \Delta$ *and* $\alpha \notin \mathsf{dom}(\Delta) \cup \mathsf{dom}(\Delta_1)$ *then* $\Psi;\Delta_1,\alpha{:}\kappa \vdash \sigma_1,\alpha/\alpha \equiv \sigma_2,\alpha/\alpha : \Delta,\alpha{:}\kappa$.

Proof

Directly, using weakening and the definition of substitution typing.

□

Lemma 8.1.8 (Substitution) *For* $\mathcal{J} \in \{\mathsf{c} : \kappa, \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa\}$,

1. *If* $\Psi_1 \vdash \mathcal{J}$ *and* $\Psi_2 \vdash \rho : \Psi_1$, *then* $\Psi \vdash [\rho]\mathcal{J}$.
2. *If* $\Psi;\Delta_1 \vdash \mathcal{J}$ *and* $\Psi;\Delta_2 \vdash \sigma : \Delta_1$, *then* $\Psi;\Delta_2 \vdash [\sigma]\mathcal{J}$.

Proof

By induction on the structure of the judgment.

□

We can now prove that substitutions are functional. That is, given equal elements, they produce equal elements.

Lemma 8.1.9 (Functionality) *Assume* $\Psi_1 \vdash \rho : \Psi$.

1. *If* $\Psi;\Delta \vdash \mathsf{c} : \kappa$ *then* $\Psi_1;\Delta[\rho] \vdash \mathsf{c}[\rho] : \kappa$.
2. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$ *then* $\Psi_1;\Delta[\rho] \vdash \mathsf{c}_1[\rho] \equiv \mathsf{c}_2[\rho] : \kappa$.

Now assume $\Psi;\Delta_1 \vdash \sigma : \Delta$.

3. *If* $\Psi;\Delta \vdash \mathsf{c} : \kappa$ *then* $\Psi;\Delta_1 \vdash \mathsf{c}[\sigma] : \kappa$.
4. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta_1 \vdash \mathsf{c}_1[\sigma] \equiv \mathsf{c}_2[\sigma] : \kappa$.

Proof

By structural induction on the derivation of the judgment.

□

We are now in a position to prove the regularity property, which says that constructors judged to be equal at some kind are also well-formed at that kind.

Lemma 8.1.10 (Regularity) *If* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_1 : \kappa$ *and* $\Psi;\Delta \vdash \mathsf{c}_2 : \kappa$.

Proof

By induction on the derivation of the equality judgment.

□

One last property we will prove in this section is inversion on the kinding judgment, to show that kinds have the expected shape, and components of constructors are themselves well-formed.

Lemma 8.1.11 (Kinding Inversion)

1. *If* $\Psi;\Delta \vdash \mathsf{Unit} : \kappa$ *then* $\kappa = \mathbb{T}$.
2. *If* $\Psi;\Delta \vdash \tau_1 \to \tau_2 : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\Delta \vdash \tau_1 : \mathbb{T}$ *and* $\Psi;\Delta \vdash \tau_2 : \mathbb{T}$.
3. *If* $\Psi;\Delta \vdash \tau_1 \times \tau_2 : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\Delta \vdash \tau_1 : \mathbb{T}$ *and* $\Psi;\Delta \vdash \tau_2 : \mathbb{T}$.
4. *If* $\Psi;\Delta \vdash \Pi \mathsf{X_M}{:}\mathsf{A}.\tau : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\cdot \vdash \mathsf{A} : \mathsf{type}$ *and* $\Psi, \mathsf{X_M}{:}\mathsf{A};\Delta \vdash \tau : \mathbb{T}$.
5. *If* $\Psi;\Delta \vdash \Pi \mathsf{X_A}{:}\mathsf{K}.\tau : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind}$ *and* $\Psi, \mathsf{X_A}{:}\mathsf{K};\Delta \vdash \tau : \mathbb{T}$.
6. *If* $\Psi;\Delta \vdash \Sigma \mathsf{X_M}{:}\mathsf{A}.\tau : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\cdot \vdash \mathsf{A} : \mathsf{type}$ *and* $\Psi, \mathsf{X_M}{:}\mathsf{A};\Delta \vdash \tau : \mathbb{T}$.
7. *If* $\Psi;\Delta \vdash \Sigma \mathsf{X_A}{:}\mathsf{K}.\tau : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind}$ *and* $\Psi, \mathsf{X_A}{:}\mathsf{K};\Delta \vdash \tau : \mathbb{T}$.
8. *If* $\Psi;\Delta \vdash \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n) : \kappa$ *then* $\kappa = \mathbb{T}$, $\mathcal{S}(\mathsf{D}) = \Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\mathbb{T}$ *and for all* $1 \le i \le n$, $\Psi;\cdot \vdash \mathcal{P}_i : \mathcal{Q}_i$.
9. *If* $\Psi;\Delta \vdash \forall(\alpha{:}\kappa).\tau : \kappa$, *then* $\kappa = \mathbb{T}$ *and* $\Psi;\Delta, \alpha{:}\kappa \vdash \tau : \mathbb{T}$.
10. *If* $\Psi;\Delta \vdash \alpha : \kappa$ *then* $\Delta(\alpha) = \kappa$.
11. *If* $\Psi;\Delta \vdash \lambda(\alpha{:}\kappa_1).\mathsf{c} : \kappa$ *then* $\kappa = \kappa_1 \to \kappa_2$ *and* $\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c} : \kappa_2$.
12. *If* $\Psi;\Delta \vdash \mathsf{c}_1\,\mathsf{c}_2 : \kappa_2$ *then* $\Psi;\Delta \vdash \mathsf{c}_1 : \kappa_1 \to \kappa_2$ *and* $\Psi;\Delta \vdash \mathsf{c}_2 : \kappa_1$

Proof

By structural induction on the kinding judgment.

□

8.1.5 Type Equivalence Algorithm

The idea behind checking equivalence of constructors is to weak-head normalize both sides, and compare the normal forms. However, there is also the extensionality rule which can be used to judge constructors equal. We follow the familiar strategy by making our algorithm be directed by the kinds at which constructors are compared. The above scheme of weak-head normalize and compare will be used at the base kind $\mathbb{T}$, and at higher kinds, extensionality is used.

We first give a weak-head reduction scheme for constructors.

$\boxed{\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2}$

$$\frac{}{(\lambda(\alpha{:}\kappa_1).\mathsf{c}_2)\,\mathsf{c}_1 \xrightarrow{\text{whr}} [\mathsf{c}_1/\alpha]\,\mathsf{c}_2} \qquad \frac{\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2}{\mathsf{c}_1\,\mathsf{c} \xrightarrow{\text{whr}} \mathsf{c}_2\,\mathsf{c}}$$

It is easy to see that the weak head reduction relation is deterministic, in the following sense:

Lemma 8.1.12 (Determinacy) *If* $\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2$ *and* $\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_3$ *then* $\mathsf{c}_2 = \mathsf{c}_3$.

Proof

By inspection of the rules.

□

Now we are in a position to define the algorithm, which is given in terms of the following two judgments.

$\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa$ Kind directed algorithmic equality
$\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa$ Structural algorithmic equality

$\boxed{\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa}$

In the kind directed phase of the algorithm, extensionality is applied repeatedly until the kind at which comparison is done is the base kind $\mathbb{T}$. At that point, we weak-head normalize both sides and switch to the structural part of the algorithm.

$$\frac{\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2 \quad \Psi;\Delta \vdash \mathsf{c}_2 \Longleftrightarrow \mathsf{c} : \mathbb{T}}{\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c} : \mathbb{T}} \qquad \frac{\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2 \quad \Psi;\Delta \vdash \mathsf{c} \Longleftrightarrow \mathsf{c}_2 : \mathbb{T}}{\Psi;\Delta \vdash \mathsf{c} \Longleftrightarrow \mathsf{c}_1 : \mathbb{T}}$$

$$\frac{\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \mathbb{T}}{\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \mathbb{T}} \qquad \frac{\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c}_1\,\alpha \Longleftrightarrow \mathsf{c}_2\,\alpha : \kappa_2}{\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa_1 \to \kappa_2}$$

$\boxed{\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa}$

In the structural part of the algorithm, we proceed by induction on the structure of the two constructors, which are known to be weak head normal forms. The kind can be looked on as output of the algorithm. In the application case, we would have to switch to the kind-directed phase of the algorithm to compare the applicands. Similarly, in the basic types like arrows and products, we have to compare at the kind $\mathbb{T}$. Notice that in the dependent cases, we appeal to the $\mathrm{LF}^{\Sigma,1+}$ equality algorithm given before.

$$\frac{\Delta(\alpha) = \kappa}{\Psi;\Delta \vdash \alpha \longleftrightarrow \alpha : \kappa} \qquad \frac{\Psi;\Delta \vdash \mathsf{c}_{11} \longleftrightarrow \mathsf{c}_{12} : \kappa_1 \to \kappa_2 \quad \Psi;\Delta \vdash \mathsf{c}_{21} \Longleftrightarrow \mathsf{c}_{22} : \kappa_1}{\Psi;\Delta \vdash \mathsf{c}_{11}\,\mathsf{c}_{12} \longleftrightarrow \mathsf{c}_{21}\,\mathsf{c}_{22} : \kappa_2}$$

$$\frac{}{\Psi;\Delta \vdash \mathsf{Unit} \longleftrightarrow \mathsf{Unit} : \mathbb{T}} \qquad \frac{\Psi;\Delta \vdash \tau_{11} \Longleftrightarrow \tau_{21} : \mathbb{T} \quad \Psi;\Delta \vdash \tau_{12} \Longleftrightarrow \tau_{22} : \mathbb{T}}{\Psi;\Delta \vdash \tau_{11} \to \tau_{12} \longleftrightarrow \tau_{21} \to \tau_{22} : \mathbb{T}}$$

$$\frac{\Psi;\Delta \vdash \tau_{11} \Longleftrightarrow \tau_{21} : \mathbb{T} \quad \Psi;\Delta \vdash \tau_{12} \Longleftrightarrow \tau_{22} : \mathbb{T}}{\Psi;\Delta \vdash \tau_{11} \times \tau_{12} \longleftrightarrow \tau_{21} \times \tau_{22} : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}^- \quad \Psi, \mathsf{X}_\mathsf{M}{:}\mathsf{A}_1;\Delta \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{X}_\mathsf{M}{:}\mathsf{A}_1.\tau_1 \longleftrightarrow \Pi \mathsf{X}_\mathsf{M}{:}\mathsf{A}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}^- \quad \Psi, \mathsf{X}_\mathsf{M}{:}\mathsf{A}_1;\Delta \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{X}_\mathsf{M}{:}\mathsf{A}_1.\tau_1 \longleftrightarrow \Sigma \mathsf{X}_\mathsf{M}{:}\mathsf{A}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \quad \Psi, \mathsf{X}_\mathsf{A}{:}\mathsf{K}_1;\Delta \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{X}_\mathsf{A}{:}\mathsf{K}_1.\tau_1 \longleftrightarrow \Pi \mathsf{X}_\mathsf{A}{:}\mathsf{K}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \quad \Psi, \mathsf{X}_\mathsf{A}{:}\mathsf{K}_1;\Delta \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{X}_\mathsf{A}{:}\mathsf{K}_1.\tau_1 \longleftrightarrow \Sigma \mathsf{X}_\mathsf{A}{:}\mathsf{K}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\mathcal{S}(\mathsf{D}) = \Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\kappa \qquad \forall 1 \leq i \leq n.\ \begin{cases} \Psi;\cdot \vdash \mathcal{P}_{1i} \equiv \mathcal{P}_{2i} : \mathsf{A}^- & \text{if } \mathcal{Q}_i = \mathsf{A} \\ \Psi;\cdot \vdash \mathcal{P}_{1i} \equiv \mathcal{P}_{2i} : \mathsf{K}^- & \text{if } \mathcal{Q}_i = \mathsf{K} \end{cases}}{\Psi;\Delta \vdash \mathsf{D}(\mathcal{P}_{11} \ldots \mathcal{P}_{1n}) \longleftrightarrow \mathsf{D}(\mathcal{P}_{21} \ldots \mathcal{P}_{2n}) : \kappa}$$

$$\frac{\Psi;\Delta, \alpha{:}\kappa \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \forall(\alpha{:}\kappa).\tau_1 \longleftrightarrow \forall(\alpha{:}\kappa).\tau_2 : \mathbb{T}}$$

We can directly prove some structural properties of the algorithm, such that it works the same if we weaken the context, and that it is symmetric and transitive. These turn out to be important in proving correctness of the algorithm, since the corresponding properties of course hold for the declarative system.

Lemma 8.1.13 (Weakening)

1. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa$ *and* $\Psi \subseteq \Psi^+$, $\Delta \subseteq \Delta^+$ *then* $\Psi^+;\Delta^+ \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa$.
2. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa$ *and* $\Psi \subseteq \Psi^+$, $\Delta \subseteq \Delta^+$ *then* $\Psi^+;\Delta^+ \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa$.

Proof

By structural induction on the derivation of the judgment.

□

Lemma 8.1.14 (Symmetry)

1. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_2 \Longleftrightarrow \mathsf{c}_1 : \kappa$.
2. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_2 \longleftrightarrow \mathsf{c}_1 : \kappa$.

Proof

By structural induction on the derivation of the judgment.

□

Lemma 8.1.15 (Transitivity)

1. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa$ *and* $\Psi;\Delta \vdash \mathsf{c}_2 \Longleftrightarrow \mathsf{c}_3 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_3 : \kappa$.
2. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa$ *and* $\Psi;\Delta \vdash \mathsf{c}_2 \longleftrightarrow \mathsf{c}_3 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_3 : \kappa$.

Proof

By structural induction on the derivation of the two judgments.

□

8.1.6 Completeness

We will prove completeness of the algorithm with respect to the definitional equality judgment given previously. The problem is the presence of the extensionality rule, which means that the shape of the judgment changes, and simple induction is too weak. There are various well-known techniques to handle this, such as performing $\beta\eta$ normalization eagerly and then comparing. We choose a different approach, and strengthen the inductive hypothesis. This method, also used for the $\mathrm{LF}^{\Sigma,1+}$ is the method of Kripke logical relations, and is more robust to extensions of the type theory. This method is a standard one for proving type or kind-directed equivalence algorithms complete [Cra05]. The logical relation we use is indexed by the kind, and at higher kinds provides the stronger conditions needed for the induction to work. The relation is defined by the following rules.

1. $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\mathbb{T}]\!]$ iff $\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \mathbb{T}$.
2. $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa_1 \to \kappa_2]\!]$ iff for every context Δ_1 such that $\Delta \subseteq \Delta_1$ and every pair of constructors c'_1 and c'_2 such that $\Psi;\Delta_1 \vdash \mathsf{c}'_1$ **is** c'_2 **in** $[\![\kappa_1]\!]$ we have $\Psi;\Delta_1 \vdash \mathsf{c}_1\,\mathsf{c}'_1$ **is** $\mathsf{c}_2\,\mathsf{c}'_2$ **in** $[\![\kappa_2]\!]$.
3. $\Psi;\Delta \vdash \sigma_1$ **is** σ_2 **in** $[\![\cdot]\!]$ iff $\sigma_1 = \sigma_2 = \cdot$.

4. $\Psi;\Delta_1 \vdash \sigma_1, \mathsf{c}_1/\alpha$ **is** $\sigma_2, \mathsf{c}_2/\alpha$ **in** $[\![\Delta, \alpha{:}\kappa]\!]$ iff $\Psi;\Delta_1 \vdash \sigma_1$ **is** σ_2 **in** $[\![\Delta]\!]$ and $\Psi;\Delta_1 \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa]\!]$.

Relying on the corresponding properties of the algorithm, we can lift the structural properties of weakening, symmetry and transitivity in the previous section to the logical relation.

Lemma 8.1.16 (Weakening)

1. *If* $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa]\!]$ *and* $\Delta \subseteq \Delta_1$ *then* $\Psi;\Delta_1 \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa]\!]$.
2. *If* $\Psi;\Delta \vdash \sigma_1$ **is** σ_2 **in** $[\![\Delta']\!]$ *and* $\Delta \subseteq \Delta_1$ *then* $\Psi;\Delta_1 \vdash \sigma_1$ **is** σ_2 **in** $[\![\Delta']\!]$.

Proof

By structural induction on the kind or context indexing the relation.

□

Lemma 8.1.17 (Symmetry)

1. *If* $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa]\!]$ *then* $\Psi;\Delta \vdash \mathsf{c}_2$ **is** c_1 **in** $[\![\kappa]\!]$.
2. *If* $\Psi;\Delta \vdash \sigma_1$ **is** σ_2 **in** $[\![\Delta']\!]$ *then* $\Psi;\Delta \vdash \sigma_2$ **is** σ_1 **in** $[\![\Delta']\!]$.

Proof

By structural induction on the kind or context indexing the relation, relying on symmetry of the algorithm at kind $\mathbb{T}$.

□

Lemma 8.1.18 (Transitivity)

1. *If* $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa]\!]$ *and* $\Psi;\Delta \vdash \mathsf{c}_2$ **is** c_3 **in** $[\![\kappa]\!]$ *then* $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_3 **in** $[\![\kappa]\!]$.
2. *If* $\Psi;\Delta \vdash \sigma_1$ **is** σ_2 **in** $[\![\Delta']\!]$ *and* $\Psi;\Delta \vdash \sigma_2$ **is** σ_3 **in** $[\![\Delta']\!]$ *then* $\Psi;\Delta \vdash \sigma_1$ **is** σ_3 **in** $[\![\Delta']\!]$.

Proof

By structural induction on the kind or context indexing the relation, relying on transitivity of the algorithm at kind $\mathbb{T}$.

□

With these preliminaries over, we now begin proving completeness by the method of logical relations. The method hinges on two steps. First, the fundamental lemma says that all logically related type constructors are judged equal by the algorithm. The second result, often called the main lemma of logical relations, says that if two constructors are equal in the definitional system, they are logically related under all related substitutions. This has the immediate consequence that all definitionally equal constructors are logically related, and thence by the fundamental lemma to the fact that they are judged equal by the algorithm.

The fundamental lemma of logical relations proves two facts simultaneously by mutual induction. First, logical relatedness imply that a kind-directed equivalence judgment exists. Second, structural equivalence implies logical relatedness.

Lemma 8.1.19 (Fundamental Lemma)

1. *If* $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa]\!]$ *then* $\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa$.
2. *If* $\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa]\!]$.

Proof

By induction on the kind.

Case 1: Part (1) $\kappa = \mathbb{T}$

$\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \mathbb{T}$	By definition of relation

Case 2: Part (1) $\kappa = \kappa_1 \to \kappa_2$

$\Psi;\Delta, \alpha{:}\kappa_1 \vdash \alpha \longleftrightarrow \alpha : \kappa_1$	By rule
$\Psi;\Delta, \alpha{:}\kappa_1 \vdash \alpha$ **is** α **in** $[\![\kappa_1]\!]$	By induction (part 2)
$\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c}_1\,\alpha$ **is** $\mathsf{c}_2\,\alpha$ **in** $[\![\kappa_2]\!]$	By definition of relation
$\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c}_1\,\alpha \Longleftrightarrow \mathsf{c}_2\,\alpha : \kappa_2$	By induction on the smaller kind κ_2
$\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa_1 \to \kappa_2$	By rule

Case 3: Part (2) $\kappa = \mathbb{T}$

$\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \mathbb{T}$	By rule
$\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\mathbb{T}]\!]$	By definition of relation

Case 4: Part (2) $\kappa = \kappa_1 \rightarrow \kappa_2$

$\Delta \subseteq \Delta_1$ for arbitrary Δ_1	New assumption
$\Psi;\Delta_1 \vdash \mathsf{c}_1'$ **is** c_2' **in** $[\![\kappa_1]\!]$ for arbitrary c_1', c_2'	New assumption
$\Psi;\Delta_1 \vdash \mathsf{c}_1' \Longleftrightarrow \mathsf{c}_2' : \kappa_2$	By induction (part 1)
$\Psi;\Delta_1 \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa_1 \rightarrow \kappa_2$	By weakening
$\Psi;\Delta_1 \vdash \mathsf{c}_1\,\mathsf{c}_1' \longleftrightarrow \mathsf{c}_2\,\mathsf{c}_2' : \kappa_2$	By rule
$\Psi;\Delta_1 \vdash \mathsf{c}_1\,\mathsf{c}_1'$ **is** $\mathsf{c}_2\,\mathsf{c}_2'$ **in** $[\![\kappa_2]\!]$	By induction on the smaller kind $kind_2$
$\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa_1 \rightarrow \kappa_2]\!]$	By definition of relation

□

The other part of the argument of logical relations is to show that constructors judged equal are logically related to each other under all possible related substitutions.

We first prove a technical fact about the logical relation, that it is closed under weak-head expansion. This is important in proving the main lemma for the beta-conversion case, since beta-reduction is carried out by the algorithm at the head of the term.

Lemma 8.1.20 (Closure under Head Expansion) *If* $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c **in** $[\![\kappa]\!]$ *and* $\mathsf{c}_2 \xrightarrow{\text{whr}} \mathsf{c}_1$ *then* $\Psi;\Delta \vdash \mathsf{c}_2$ **is** c **in** $[\![\kappa]\!]$.

Proof

By structural induction on the derivation of the first judgment.

□

Lemma 8.1.21 (Main Lemma) *If* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$ *and* $\Psi;\Delta \vdash \sigma_1$ **is** σ_2 **in** $[\![\Delta]\!]$ *then* $\Psi;\Delta \vdash [\sigma_1]\mathsf{c}_1$ **is** $[\sigma_2]\mathsf{c}_2$ **in** $[\![\kappa]\!]$.

Proof

By structural induction on the derivation of the judgment. We show a few representative cases.

Case 1:
$$\frac{\Psi;\Delta \vdash \tau_{11} \equiv \tau_{21} : \mathbb{T} \quad \Psi;\Delta \vdash \tau_{12} \equiv \tau_{22} : \mathbb{T}}{\Psi;\Delta \vdash \tau_{11} \rightarrow \tau_{12} \equiv \tau_{21} \rightarrow \tau_{22} : \mathbb{T}}$$

$\Psi;\Delta \vdash [\sigma_1]\tau_{11}$ **is** $[\sigma_2]\tau_{21}$ **in** $[\![\mathbb{T}]\!]$	By induction
$\Psi;\Delta \vdash [\sigma_1]\tau_{12}$ **is** $[\sigma_2]\tau_{22}$ **in** $[\![\mathbb{T}]\!]$	By induction
$\Psi;\Delta \vdash [\sigma_1](\tau_{11} \rightarrow \tau_{12})$ **is** $[\sigma_2](\tau_{21} \rightarrow \tau_{22})$ **in** $[\![\mathbb{T}]\!]$	By rule

Case 2:
$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \quad \Psi, \mathsf{X}_\mathsf{M}{:}\mathsf{A}_1;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi\mathsf{X}_\mathsf{M}{:}\mathsf{A}_1.\tau_1 \equiv \Pi\mathsf{X}_\mathsf{M}{:}\mathsf{A}_2.\tau_2 : \mathbb{T}}$$

$\Psi, \mathsf{X}_\mathsf{M}{:}\mathsf{A}_1;\Delta \vdash [\sigma_1]\tau_1 \equiv [\sigma_2]\tau_2 : \mathbb{T}$	By induction
$\Psi;\Delta \vdash [\sigma_1](\Pi\mathsf{X}_\mathsf{M}{:}\mathsf{A}_1.\tau_1) \equiv [\sigma_2](\Pi\mathsf{X}_\mathsf{M}{:}\mathsf{A}_2.\tau_2) : \mathbb{T}$	By rule

Case 3:
$$\frac{\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c}_{12} \equiv \mathsf{c}_{22} : \kappa_2 \qquad \Psi;\Delta \vdash \mathsf{c}_{11} \equiv \mathsf{c}_{21} : \kappa_1}{\Psi;\Delta \vdash (\lambda(\alpha{:}\kappa_1).\mathsf{c}_{12})\,\mathsf{c}_{11} \equiv [\mathsf{c}_{21}/\alpha]\,\mathsf{c}_{22} : \kappa_2}$$

$\Psi;\Delta \vdash \sigma_1$ **is** σ_2 **in** $[\![\Delta]\!]$	By assumption
$\Psi;\Delta \vdash [\sigma_1]\mathsf{c}_{11}$ **is** $[\sigma_2]\mathsf{c}_{21}$ **in** $[\![\kappa_1]\!]$	By induction
$\Psi;\Delta, \alpha{:}\kappa_1 \vdash \sigma_1, [\sigma_1]\mathsf{c}_{11}/\alpha$ **is** $\sigma_2, [\sigma_2]\mathsf{c}_{21}/\alpha$ **in** $[\![\Delta, \alpha{:}\kappa_1]\!]$	By definition of relation

$\Psi;\Delta \vdash [\sigma_1, [\sigma_1]\mathsf{c}_{11}/\alpha]\mathsf{c}_{12}$ **is** $[\sigma_2, [\sigma_2]\mathsf{c}_{21}/\alpha]\mathsf{c}_{22}$ **in** $[\![\kappa_2]\!]$ — By induction
$\Psi;\Delta \vdash [\sigma_1]([\alpha/\mathsf{c}_{12}]\,\mathsf{c}_{11})$ **is** $[\sigma_2]([\alpha/\mathsf{c}_{22}]\,\mathsf{c}_{21})$ **in** $[\![\kappa_2]\!]$ — By definition of substitution
$\Psi;\Delta \vdash [\sigma_1]\,((\lambda(\alpha{:}\kappa_1).\mathsf{c}_{12})\,\mathsf{c}_{11})$ **is** $[\sigma_2]\,([\mathsf{c}_{21}/\alpha]\,\mathsf{c}_{22})$ **in** $[\![\kappa_2]\!]$ — By closure under weak head expansion

Case 4: $\dfrac{\Psi;\Delta \vdash \mathsf{c}_2 \equiv \mathsf{c}_1 : \kappa}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa}$

$\Psi;\Delta \vdash \sigma_2$ **is** σ_1 **in** $[\![\Delta]\!]$ — By symmetry of relation
$\Psi;\Delta \vdash [\sigma_2]\mathsf{c}_2$ **is** $[\sigma_1]\mathsf{c}_1$ **in** $[\![\kappa]\!]$ — By induction
$\Psi;\Delta \vdash [\sigma_1]\mathsf{c}_1$ **is** $[\sigma_2]\mathsf{c}_2$ **in** $[\![\kappa]\!]$ — By symmetry of the relation

Case 5: $\dfrac{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa \quad \Psi;\Delta \vdash \mathsf{c}_2 \equiv \mathsf{c}_3 : \kappa}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_3 : \kappa}$

$\Psi;\Delta \vdash \sigma_1$ **is** σ_2 **in** $[\![\Delta]\!]$ — By assumption
$\Psi;\Delta \vdash \sigma_2$ **is** σ_1 **in** $[\![\Delta]\!]$ — By symmetry of the relation
$\Psi;\Delta \vdash \sigma_2$ **is** σ_2 **in** $[\![\Delta]\!]$ — By transitivity of the relation
$\Psi;\Delta \vdash [\sigma_2]\mathsf{c}_2$ **is** $[\sigma_2]\mathsf{c}_3$ **in** $[\![\kappa]\!]$ — By induction
$\Psi;\Delta \vdash [\sigma_1]\mathsf{c}_1$ **is** $[\sigma_2]\mathsf{c}_2$ **in** $[\![\kappa]\!]$ — By induction
$\Psi;\Delta \vdash [\sigma_1]\mathsf{c}_1$ **is** $[\sigma_2]\mathsf{c}_3$ **in** $[\![\kappa]\!]$ — By transitivity of relation

□

Now we need the simple fact that an identity substitution is related to itself.

Lemma 8.1.22 (Identity Substitution Related) $\Psi;\Delta \vdash \mathsf{id}_\Delta$ **is** id_Δ **in** $[\![\Delta]\!]$.

Proof

By induction on the construction of the context.

□

Lemma 8.1.23 *If* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_1$ **is** c_2 **in** $[\![\kappa]\!]$.

Proof

Direct, by using lemmas 8.1.21 and 8.1.22.

□

Using the foregoing, completeness of the algorithmic system is an immediate consequence.

Theorem 8.1.24 (Completeness) *If* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa$.

Proof

Direct, by the previous lemma and lemma 8.1.19.

□

8.1.7 Soundness

Soundness of the algorithm with respect to the definitional equality judgment is relatively easier to prove. The proof is by direct induction on the derivation of algorithmic equality. A preliminary result of subject reduction needs to be proved to show the head-reduction step of the algorithm sound.

Lemma 8.1.25 (Subject Reduction) *If* $\Psi;\Delta \vdash \mathsf{c}_1 : \kappa$ *and* $\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2$ *then* $\Psi;\Delta \vdash \mathsf{c}_2 : \kappa$ *and* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$.

Proof

By induction on the kinding derivation.

□

Theorem 8.1.26 (Soundness)

1. *If* $\Psi;\Delta \vdash \mathsf{c}_1 : \kappa$, $\Psi;\Delta \vdash \mathsf{c}_2 : \kappa$ *and* $\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$.
2. *If* $\Psi;\Delta \vdash \mathsf{c}_1 : \kappa_1$, $\Psi;\Delta \vdash \mathsf{c}_2 : \kappa_2$ *and* $\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa$ *then* $\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa$ *and* $\kappa_1 = \kappa_2 = \kappa$.

Proof

By an easy structural induction on the algorithmic derivation. At kind $\mathbb{T}$ the algorithm weak-head normalizes both constructors. We use subject reduction to produce a derivation of definitional equality, and the transitivity of definitional equality to put the derivations together.

□

8.1.8 Consistency

With the proof of soundness and completeness of the algorithm for equality, a variety of consistency results can be proved for the equality judgment by the easy consistency of algorithmic equality. We show two representative results.

Lemma 8.1.27 *If* $\vdash \Psi$, $\Psi \vdash \Delta$ *then it is not the case that* $\Psi;\Delta \vdash \mathsf{Unit} \equiv \tau_1 \to \tau_2 : \mathbb{T}$.

Proof

Since Unit and $\tau_1 \to \tau_2$ are not algorithmically equal at kind $\mathbb{T}$, by soundness and completeness of the algorithmic equality, they are not definitionally equal either.

□

Lemma 8.1.28 *If* $\vdash \Psi$, $\Psi \vdash \Delta$ *then it is not the case that* $\Psi;\Delta \vdash \tau_1 \times \tau_2 \equiv \tau_1' \to \tau_2' : \mathbb{T}$.

Proof

As for the previous lemma.

□

8.2 Term Level of LF/F$^\omega$

Next, we give the term structure and equip the language with a call-by-value small-step semantics. We prove type safety by the use of the standard syntactic method, proving progress and preservation of typing under evaluation(also known as subject reduction). The subject reduction property is essential to our methodology, independent of the type safety result.

8.2.1 Abstract Syntax

We present the abstract syntax of the term level is given in figure 8.6. This is explicitly typed, since it is intended to be an internal language. The term structure includes such common features as recursive functions, applications, products and projections. There are type abstractions and applications as is usual in F$^\omega$. The new features are functions taking index arguments and applications to indices, which are constructors and destructors of the Π types, and packages and unpackaging constructs for the Σ types. Also, the notion of datatype constructors is extended to possibly take index arguments in addition to the term level arguments.

Signatures are extended to record assumptions on datatype constructors. These assumptions can be of higher kind as well as take index arguments.

The substitutions for LF variables and constructor variables are extended to the term level. We need a new concept of substitutions for term variables. This is defined in the obvious way.

$$\begin{array}{lrll} \text{Term Substitutions} & \sigma ::= & \cdot & \text{empty} \\ & \mid & \sigma, e/x & \text{cons with term} \end{array}$$

We write id_Υ for the identity on the context Υ, and also define $e[\sigma]$.

Matches	$ms ::=$	$\cdot$	Nil
	$\mid$	$\mathsf{C}\,[w_1 \ldots w_n, x] \Rightarrow e \mid ms$	Cons
Terms	$e ::=$	unit	Unit
	$\mid$	$\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e$	Functions
	$\mid$	$e_1\, e_2$	Applications
	$\mid$	$\langle e_1, e_2 \rangle$	Pairs
	$\mid$	$\pi_i\, e$	Projection from Pair
	$\mid$	$\Lambda\alpha{:}\kappa.e$	Constructor Abstraction
	$\mid$	$e\ [\mathsf{c}]$	Constructor Application
	$\mid$	$\mathsf{Fun}\, f(w{:}(\cdot \vdash \mathcal{Q})){:}\tau.e$	Functions taking LF arguments
	$\mid$	$e\ [\mathcal{P}]$	LF term Application
	$\mid$	$\mathsf{pack}\, \langle \mathcal{P}, e \rangle$	Package of LF term and Expression
	$\mid$	$\mathsf{let\ pack}\, \langle w, x \rangle = e_1\, \mathsf{in}\, e_2\, \mathsf{end}$	Unpacking a Pair
	$\mid$	$\mathsf{C}\,[\mathcal{P}_1 \ldots \mathcal{P}_n, e]$	Datatype Constructors
	$\mid$	$\mathsf{case}^{\tau}\, e_1\, \mathsf{of}\, ms\, \mathsf{end}$	Case Expression
	$\mid$	$\mathsf{let\ val}\, x = e_1\, \mathsf{in}\, e_2\, \mathsf{end}$	Let Expresion
	$\mid$	error	Error
Signatures	$\mathcal{S} ::=$	$\ldots$	As before
	$\mid$	$\mathcal{S}, \mathsf{C}{:}\forall(\alpha_1{:}\kappa_1) \ldots \forall(\alpha_m{:}\kappa_m).$ $(\Pi w_1{:}(\cdot \vdash \mathcal{Q}_1) \ldots \Pi w_n{:}(\cdot \vdash \mathcal{Q}_n).\tau_1 \to \tau_2)$	
Contexts	$\Upsilon ::=$	$\cdot$	Nil
	$\mid$	$\Upsilon, x{:}\tau$	Cons

Figure 8.6: Abstract Syntax: Terms

8.2.2 Static Semantics

The static semantics at the term level are defined by the new judgment forms.

$\Psi;\Delta \vdash \Upsilon$	Υ is a valid context
$\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau$	e is well-typed at τ
$\Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$	ms takes $\mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m$ to τ

The semantics of terms assume that a well-formed signature is given. This signature records assumptions on both datatypes and datatype constructors. Datatype constructors are well-formed when the abstracted sorts are valid $\mathrm{LF}^{\Sigma,1+}$ classifiers in the empty context, that is, they are closed families or kinds, as shown in figure 8.7.

Term variable contexts are well-formed when the types of the variables are well-formed, as is shown in figure 8.8.

The judgments for terms and matches take four different sets of assumptions, on the metavariables of $\mathrm{LF}^{\Sigma,1+}$, the constructor variables, the term variables, and a set of constraints. The constraints postulate equality of closed $\mathrm{LF}^{\Sigma,1+}$ terms.

Terms are given types by the judgments in figures 8.9 and 8.10. The first part is entirely standard for a F^{ω} like language. For the new parts, indices must be checked within LF to be closed and well-formed.

The interesting rule is for checking matches within case analysis 8.11. Within the body of a match, we have more information about the index assumptions. Specifically, the type matched against can be unified with the result value of constructors. The datatype constructors in the language are nonuniform with respect to the $\mathrm{LF}^{\Sigma,1+}$ domain. We eagerly try to unify as much as we can. If some equation fails, we do not have to check this branch since it is unreachable. If some equation is not solvable yet, it is postponed. Later

$\boxed{\vdash \mathcal{S}}$

$$\frac{\begin{array}{c}\vdash \mathcal{S} : \mathsf{ok} \\ \forall 1 \leq i \leq n. \left\{ \begin{array}{ll} w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1}; \cdot \vdash \mathsf{A} : \mathsf{type} & \text{if } \mathcal{Q}_i = \mathsf{A} \\ w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1}; \cdot \vdash \mathsf{K} : \mathsf{kind} & \text{if } \mathcal{Q}_i = \mathsf{K} \end{array} \right. \\ w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n; \cdot \vdash \tau : \mathbb{T} \\ \\ \vdash_S \mathsf{D} : \Pi w'_1{:}\mathcal{Q}'_1 \ldots \Pi w'_m{:}\mathcal{Q}'_m.\kappa \\ \kappa = \kappa_1 \rightarrow \ldots \kappa_p \rightarrow \mathbb{T} \\ \forall 1 \leq j \leq m. w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n; \cdot \vdash \mathcal{P}_j : \mathcal{Q}'_j \end{array}}{\vdash \mathcal{S}, \mathsf{C}{:}\forall(\alpha_1{:}\kappa_1) \ldots \forall(\alpha_p{:}\kappa_p).(\Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\tau \rightarrow \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_m)\, \alpha_1 \ldots \alpha_p)}$$

Figure 8.7: Well-formed Signature (constants)

$\boxed{\Psi;\Delta \vdash \Upsilon}$

Empty
$$\frac{}{\Psi;\Delta \vdash \cdot}$$

Cons
$$\frac{\Psi;\Delta \vdash \Upsilon \quad \Psi;\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Upsilon, x{:}\tau}$$

Figure 8.8: Well-formed term variable contexts

information might resolve the equations. If we come to a base case where we have postponed equations, we fail since we cannot safely know if the equation is solvable.

Well Typed Substitutions The notation for typing substitutions is defined in a familiar way in that every element of the domain must get a valid type.

Definition 8.2.1 *The judgment* $\Psi; \Delta; \Upsilon_2; \mathcal{C} \vdash \sigma : \Upsilon_1$ *holds iff*
$\forall x \in \mathsf{dom}(\Upsilon_1).\Psi;\Delta;\Upsilon_2;\mathcal{C} \vdash \sigma(x) : \sigma(\Upsilon_1(x))$.

8.2.3 Structural Properties

We will now prove some simple structural properties of the static semantics.

Lemma 8.2.2 (Weakening)

1. *If* $\Psi;\Delta \vdash \Upsilon$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau$ *and* $\Upsilon \subseteq \Upsilon_1$, *then* $\Psi;\Delta;\Upsilon_1;\mathcal{C} \vdash e : \tau$.
2. *If* $\Psi;\Delta \vdash \Upsilon$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau$ *and* $\Delta \subseteq \Delta_1$, *then* $\Psi;\Delta_1;\Upsilon;\mathcal{C} \vdash e : \tau$.
3. *If* $\Psi;\Delta \vdash \Upsilon$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau$ *and* $\Psi \subseteq \Psi_1$, *then* $\Psi_1;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau$.
4. *If* $\Psi;\Delta \vdash \Upsilon$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \rightarrow \tau$ *and* $\Upsilon \subseteq \Upsilon_1$, *then* $\Psi;\Delta;\Upsilon_1;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \rightarrow \tau$.
5. *If* $\Psi;\Delta \vdash \Upsilon$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \rightarrow \tau$ *and* $\Delta \subseteq \Delta_1$, *then* $\Psi;\Delta_1;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \rightarrow \tau$.
6. *If* $\Psi;\Delta \vdash \Upsilon$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \rightarrow \tau$ *and* $\Psi \subseteq \Psi_1$, *then* $\Psi_1;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \rightarrow \tau$.

$\boxed{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau}$

Type Conversion

$$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau_2 \quad \Psi;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau_1}$$

Variables

$$\frac{\Upsilon(x) = \tau}{\Psi;\Delta;\Upsilon;\mathsf{true} \vdash x : \tau}$$

Constants

$$\frac{}{\Psi;\Delta;\Upsilon;\mathsf{true} \vdash \mathsf{unit} : \mathsf{Unit}}$$

Functions

$$\frac{\Psi;\Delta \vdash \tau_1 \to \tau_2 : \mathbb{T} \quad \Psi;\Delta;\Upsilon, f{:}\tau_1 \to \tau_2, x{:}\tau_1;\mathcal{C} \vdash e : \tau_2}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2}$$

Applications

$$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \tau_1 \to \tau_2 \quad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_2 : \tau_1}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1\, e_2 : \tau_2}$$

Pairs

$$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \tau_1 \quad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_2 : \tau_2}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

Projections

$$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau_1 \times \tau_2}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \pi_i\, e : \tau_i}$$

Constructor Abstractions

$$\frac{\Psi;\Delta, \alpha{:}\kappa;\Upsilon;\mathcal{C} \vdash e : \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \Lambda\alpha{:}\kappa.e : \forall(\alpha{:}\kappa).\tau}$$

Constructor Applications

$$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \forall(\alpha{:}\kappa).\tau \quad \Psi;\Delta \vdash \mathsf{c} : \kappa}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e\ [\mathsf{c}] : [\mathsf{c}/\alpha]\,\tau}$$

Error

$$\frac{\Psi;\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{error} : \tau}$$

Let form

$$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \tau_1 \quad \Psi;\Delta;\Upsilon, x{:}\tau_1;\mathcal{C} \vdash e_2 : \tau_2}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{let\ val}\, x = e_1\, \mathsf{in}\, e_2\, \mathsf{end} : \tau_2}$$

Figure 8.9: Well-formed terms, part 1

Function taking LF object

$$\frac{\Psi;\cdot \vdash \mathsf{A} : \mathsf{type} \quad \Psi, \mathsf{X_M}{:}\mathsf{A};\Delta \vdash \tau : \mathbb{T} \quad \Psi, \mathsf{X_M}{:}\mathsf{A};\Delta;\Upsilon, f{:}(\Pi \mathsf{X_M}{:}\mathsf{A}.\tau);\mathcal{C} \vdash e : \mathsf{c}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{Fun}\, f(\mathsf{X_M}{:}\mathsf{A}){:}\tau.e : \Pi \mathsf{X_M}{:}\mathsf{A}.\mathsf{c}}$$

Function taking LF family

$$\frac{\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind} \quad \Psi, \mathsf{X_A}{:}\mathsf{K};\Delta \vdash \tau : \mathbb{T} \quad \Psi, \mathsf{X_A}{:}\mathsf{K};\Delta;\Upsilon, f{:}(\Pi \mathsf{X_A}{:}\mathsf{K}.\tau);\mathcal{C} \vdash e : \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{Fun}\, f(\mathsf{X_A}{:}\mathsf{K}){:}\tau.e : \Pi \mathsf{X_A}{:}\mathsf{K}.\tau}$$

Application to LF term

$$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \Pi w{:}\mathcal{Q}.\tau \quad \Psi;\cdot \vdash \mathcal{P} : \mathcal{Q}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e\ [\mathcal{P}] : [\mathcal{P}/w]\,\tau}$$

Package with LF term

$$\frac{\Psi;\cdot \vdash \mathcal{P} : \mathcal{Q} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : [\mathcal{P}/w]\,\tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{pack}\,\langle \mathcal{P}, e\rangle : \Sigma w{:}\mathcal{Q}.\tau}$$

Unpack form

$$\frac{\begin{array}{c}\Psi;\Delta \vdash \tau : \mathbb{T} \\ \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \Sigma w{:}\mathcal{Q}.\tau_1 \quad \Psi, w{:}\mathcal{Q};\Delta;\Upsilon, x{:}\tau_1;\mathcal{C} \vdash e_2 : \tau\end{array}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{let\ pack}\,\langle w, x\rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : \tau}$$

Datatype Constructors

$$\frac{\begin{array}{c}\mathcal{S}(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1).\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1.\ldots\Pi w_n{:}\mathcal{Q}_n. \\ \qquad \tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p \\ \forall 1 \le i \le n. \quad \Psi, w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathcal{P}_{1i} : \mathcal{Q}_i \\ \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \left[\begin{array}{c|c} \mathsf{c}_1, \ldots, \mathsf{c}_p, & \alpha_1, \ldots, \alpha_p, \\ \mathcal{P}_{11}, \ldots, \mathcal{P}_{1n} & w_1, \ldots, w_n \end{array}\right] \tau_1\end{array}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{C}\left[\begin{array}{c} \mathsf{c}_1\ldots\mathsf{c}_p, \\ \mathcal{P}_{11}\ldots\mathcal{P}_{1n} \end{array}, e\right] : \mathsf{D}\left(\begin{array}{c} [\mathcal{P}_{11}, \ldots, \mathcal{P}_{1n}/w_1, \ldots, w_n]\,\mathcal{P}_{21}\ldots \\ [\mathcal{P}_{11}, \ldots, \mathcal{P}_{1n}/w_1, \ldots, w_n]\,\mathcal{P}_{2m} \\ \mathsf{c}_1\ldots\mathsf{c}_p \end{array}\right)}$$

Case

$$\frac{\begin{array}{c}\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \quad \Psi;\Delta \vdash \tau : \mathbb{T} \\ \Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \to \tau\end{array}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{case}^{\tau}\ e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau}$$

Figure 8.10: Well-formed terms, part 2

$\boxed{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\,\mathsf{c}_1 \ldots \mathsf{c}_m \to \tau}$

Nil

$$\frac{}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \cdot : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\,\mathsf{c}_1 \ldots \mathsf{c}_m \to \tau}$$

Success

$$\frac{\begin{array}{l}\mathcal{S}(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p \\ \mathcal{S}(\mathsf{D}) = \Pi w'_1{:}\mathcal{Q}'_1\ldots\Pi w'_m{:}\mathcal{Q}'_m.\mathbb{T} \\ \tau'_1 = [\mathsf{c}_1\ldots\mathsf{c}_p/\alpha_1\ldots\alpha_p]\,\tau_1 \\ \Psi, w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n;\cdot \vdash \mathcal{P}_{21} \doteq \mathcal{P}_1 : \mathcal{Q}'_1 \wedge \ldots \wedge \mathcal{P}_{2m} \doteq \mathcal{P}_m : \mathcal{Q}'_m \wedge \mathcal{C} \Longrightarrow (\rho, \Psi_1, \mathcal{C}_1) \\ \Psi_1;\Delta[\rho];\Upsilon[\rho], x{:}\tau'_1[\rho];\mathcal{C}_1 \vdash e[\rho] : \tau \\ \Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_m)\,\mathsf{c}_1 \ldots \mathsf{c}_p \to \tau\end{array}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{C}\,[w_1 \ldots w_n, x] \Rightarrow e | ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_m)\,\mathsf{c}_1 \ldots \mathsf{c}_p \to \tau}$$

Failure

$$\frac{\begin{array}{l}\mathcal{S}(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p \\ \mathcal{S}(\mathsf{D}) = \Pi w'_1{:}\mathcal{Q}'_1\ldots\Pi w'_m{:}\mathcal{Q}'_m.\mathbb{T} \\ \tau'_1 = [\mathsf{c}_1\ldots\mathsf{c}_p/\alpha_1\ldots\alpha_p]\,\tau \\ \Psi, w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n;\cdot \vdash \mathcal{P}_{21} \doteq \mathcal{P}_1 : \mathcal{Q}'_1 \wedge \ldots \wedge \mathcal{P}_{2m} \doteq \mathcal{P}_m : \mathcal{Q}'_m \wedge \mathcal{C} \Longrightarrow \mathsf{false} \\ \Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_m)\,\mathsf{c}_1 \ldots \mathsf{c}_p \to \tau\end{array}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{C}\,[w_1 \ldots w_n, x] \Rightarrow e | ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_m)\,\mathsf{c}_1 \ldots \mathsf{c}_p \to \tau}$$

Figure 8.11: Well-formed matches

Proof

By an easy induction over the structure of the typing derivation. We need weakening of judgments for $\mathrm{LF}^{\Sigma,1+}$ and constructors, which have already been proved. □

As is usual for a declarative system, we want to show substitution is admissible. We need to show that the identity substitution is always well-typed, and that we can extend substitutions with variable for variable substitutions.

Lemma 8.2.3 (Identity Substitution) *If* $\Psi;\Delta \vdash \Upsilon$ *then* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{id}_\Upsilon : \Upsilon$.

Proof

By an easy induction on the construction of the context. □

Lemma 8.2.4 (Extending Substitutions) *If* $\Psi;\Delta;\Upsilon_1;\mathcal{C} \vdash \sigma : \Upsilon$, $\Psi;\Delta \vdash \tau : \mathbb{T}$ *and* $x \notin \mathsf{dom}(\Upsilon_1) \cup \mathsf{dom}(\Upsilon)$ *then*
$\Psi;\Delta;\Upsilon_1, x{:}\tau;\mathcal{C} \vdash \sigma, x/x : \Upsilon, x{:}\tau$.

Proof

Directly, by definition of typing substitutions and weakening. □

Lemma 8.2.5 (Substitution) *In the following,* $\mathcal{J} \in \{\vdash e : \tau, \vdash ms : \mathsf{D}(\mathcal{P}) \to \tau\}$.

1. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathcal{J}$ *and* $\Psi_1 \vdash \rho : \Psi$ *then* $\Psi_1;[\rho]\Delta;[\rho]\Upsilon;[\rho]\mathcal{C} \vdash [\rho]\mathcal{J}$.
2. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathcal{J}$ *and* $\Psi;\Delta_1 \vdash \sigma : \Delta$ *then* $\Psi;\Delta_1;[\sigma]\Upsilon;\mathcal{C} \vdash [\sigma]\mathcal{J}$.
3. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathcal{J}$ *and* $\Psi;\Delta;\Upsilon_1;\mathcal{C} \vdash \sigma : \Upsilon$ *then* $\Psi;\Delta;\Upsilon_1;\mathcal{C} \vdash [\sigma]\mathcal{J}$.

Proof

By induction on the given derivation of $\mathcal{J}$. We need substitution for $\mathrm{LF}^{\Sigma,1+}$ and constructors. Again, these have been proved in earlier sections. To handle the abstraction cases, we need the lemma about extending substitutions proved above. □

We can now prove two important results about the system. The first is regularity, which says that all types of terms are themselves well-formed. The second is a typing inversion property, which talks about the shapes that the type of a term must have, assuming the shape of the term.

Lemma 8.2.6 (Regularity) *If* $\vdash \Psi$, $\Psi \vdash \Delta$, $\Psi;\Delta \vdash \Upsilon$ *and* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau$, *then* $\Psi;\Delta \vdash \tau : \mathbb{T}$.

Proof

By a structural induction on the typing derivation. □

Lemma 8.2.7 (Typing Inversion) *Assume* $\vdash \Psi$, $\Psi \vdash \Delta$, $\Psi;\Delta \vdash \Upsilon$ *all hold.*

1. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{unit} : \tau$ *then* $\Psi;\Delta \vdash \tau \equiv \mathsf{Unit} : \mathbb{T}$.
2. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e : \tau$ *then* $\Psi;\Delta \vdash \tau_1 \to \tau_2 : \mathbb{T}$, $\Psi;\Delta;\Upsilon, f{:}\tau_1 \to \tau_2, x{:}\tau_1;\mathcal{C} \vdash e : \tau_2$ *and* $\Psi;\Delta \vdash \tau \equiv \tau_1 \to \tau_2 : \mathbb{T}$.
3. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1\, e_2 : \tau$ *then* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \tau_1 \to \tau_2$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_2 : \tau_1$ *and* $\Psi;\Delta \vdash \tau \equiv \tau_2 : \mathbb{T}$.
4. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \langle e_1, e_2\rangle : \tau$ *then* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \tau_1$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_2 : \tau_2$, *and* $\Psi;\Delta \vdash \tau \equiv \tau_1 \times \tau_2 : \mathbb{T}$.
5. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \pi_i\, e : \tau$ *then* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau_1 \times \tau_2$ *and* $\Psi;\Delta \vdash \tau \equiv \tau_i : \mathbb{T}$.
6. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \Lambda\alpha{:}\kappa.e : \tau$ *then* $\Psi;\Delta, \alpha{:}\kappa;\Upsilon;\mathcal{C} \vdash e : \tau_1$ *and* $\Psi;\Delta \vdash \tau \equiv \forall(\alpha{:}\kappa).\tau_1 : \mathbb{T}$.
7. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e\ [\mathsf{c}] : \tau$ *then* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \forall(\alpha{:}\kappa).\tau_1$, $\Psi;\Delta \vdash \mathsf{c} : \kappa$ *and* $\Psi;\Delta \vdash \tau \equiv [\mathsf{c}/\alpha]\,\tau_1 : \mathbb{T}$.
8. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e : \tau_1$ *then* $\Psi \vdash \mathcal{Q} : \mathsf{sort}$, $\Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \kappa$, $\Psi, w{:}\mathcal{Q};\Delta;\Upsilon, f{:}(\Pi w{:}\mathcal{Q}.\tau);\mathcal{C} \vdash e : \tau$ *and* $\Psi;\Delta \vdash \tau_1 \equiv \Pi w{:}\mathcal{Q}.\tau : \mathbb{T}$.
9. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e\ [\mathcal{P}] : \tau$ *then* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \Pi w{:}\mathcal{Q}.\tau_1$, $\Psi \vdash \mathcal{P} : \mathcal{Q}$ *and* $\Psi;\Delta \vdash \tau \equiv [\mathcal{P}/w]\,\tau_1 : \mathbb{T}$.
10. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{pack}\,\langle \mathcal{P}, e\rangle : \tau$ *then* $\Psi \vdash \mathcal{P} : \mathcal{Q}$, $\Psi, w{:}\mathcal{Q};\Delta \vdash \tau_1 : \mathbb{T}$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : [\mathcal{P}/w]\,\tau_1$ *and* $\Psi;\Delta \vdash \tau \equiv \Sigma w{:}\mathcal{Q}.\tau_1 : \mathbb{T}$.
11. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{let\ pack}\,\langle w, x\rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : \tau$ *then* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \Sigma w{:}\mathcal{Q}.\tau_1$, $\Psi, w{:}\mathcal{Q};\Delta;\Upsilon, x{:}\tau_1;\mathcal{C} \vdash e_2 : \tau_2$ *and* $\Psi;\Delta \vdash \tau \equiv \tau_2 : \mathbb{T}$.
12. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{C}\,[\mathsf{c}_1 \ldots \mathsf{c}_p\, \mathcal{P}_1 \ldots \mathcal{P}_n, e] : \tau$ *then*
$\mathcal{S}(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1).\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1.\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \mathsf{D}(\mathcal{P}'_1 \ldots \mathcal{P}'_m)\,\alpha_1 \ldots \alpha_p$, *for all* $1 \le i \le n$,
$\Psi, w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathcal{P}_i : \mathcal{Q}_i$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : [\mathsf{c}_1 \ldots \mathsf{c}_p, \mathcal{P}_1 \ldots \mathcal{P}_n/\alpha_1 \ldots \alpha_p, w_1 \ldots w_n]\,\tau_1$ *and*
$\Psi;\Delta \vdash \tau \equiv \mathsf{D}([\mathcal{P}_1, \ldots, \mathcal{P}_n/w_1, \ldots, w_n]\,\mathcal{P}'_1 \ldots [\mathcal{P}_1, \ldots, \mathcal{P}_n/w_1, \ldots, w_n]\,\mathcal{P}'_m)\,\mathsf{c}_1 \ldots \mathsf{c}_p : \mathbb{T}$.

13. *If* $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{case}^{\tau}\ e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau_1$ *then*
$\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\,\mathsf{c}_1 \ldots \mathsf{c}_m$, $\Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\,\mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$ *and*
$\Psi;\Delta \vdash \tau_1 \equiv \tau : \mathbb{T}$.

Proof

By structural induction on the typing derivation. The premises of the rules give the required facts directly for structural rules. The only other rule is the type conversion rule, where we need to apply induction on the typing premise, and apply transitivity of type conversion to the inductive hypothesis and the premise. □

8.2.4 Canonical Forms

A subset of terms are judged to be values. This purely syntactic notion is defined by the grammar below.

Values	$v ::=$	unit	Unit
	$\mid$	$\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e$	Functions
	$\mid$	$\langle v_1, v_2\rangle$	Pairs
	$\mid$	$\Lambda\alpha{:}\kappa.e$	Constructor Abstraction
	$\mid$	$\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e$	Recursive functions taking index arguments
	$\mid$	$\mathsf{pack}\,\langle\mathcal{P},v\rangle$	Package of Index and Expression
	$\mid$	$\mathsf{C}\,[\mathcal{P}_1 \ldots \mathcal{P}_n, v]$	Datatype Constructors

Intuitively, values represent the result of successful executions. They have the important property that their shape is determined by their type, a property called the canonical forms property. Thus, values are the canonical forms of the respective types.

Lemma 8.2.8 (Canonical Forms)

1. *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash v : \mathsf{Unit}$ *then* $v = \mathsf{unit}$.
2. *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash v : \tau_1 \to \tau_2$ *then* $v = \mathsf{fun}\, f(x{:}\tau_1'){:}\tau_2'.e$.
3. *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash v : \tau_1 \times \tau_2$ *then* $v = \langle v_1, v_2\rangle$.
4. *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash v : \forall(\alpha{:}\kappa).\tau$ *then* $v = \Lambda\alpha{:}\kappa.e$.
5. *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash v : \Pi w{:}\mathcal{Q}.\tau$ *then* $v = \mathsf{Fun}\, f(w{:}\mathcal{Q}_1){:}\tau_1.e$.
6. *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash v : \Sigma w{:}\mathcal{Q}.\tau$ *then* $v = \mathsf{pack}\,\langle\mathcal{P},v'\rangle$.
7. *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash v : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_m)\,\mathsf{c}_1 \ldots \mathsf{c}_p$ *then* $v = \mathsf{C}\left[\mathsf{c}_1' \ldots \mathsf{c}_p'\mathcal{P}_1' \ldots \mathcal{P}_n', v'\right]$.

Proof

By an easy case analysis on the form of the value in question. A contradiction is derived for all values not of the right form by the use of typing inversion, and consistency for type (constructors). □

8.3 Dynamic Semantics and Type Safety

We now equip our language by a standard call-by-value dynamic semantics. Thus, in application terms, the function position is evaluated first, followed by the argument, and then the reduction step is carried out. A left-to-right order is fixed for pairs. A call-by-value system is useful in anticipation of extending the system with effects such as references.

The evaluation relation $e_1 \mapsto e_2$ is defined by the following rules:

$$\frac{}{(\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e)\, v \mapsto [\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e, v/f, x]\, e} \qquad \frac{e_1 \mapsto e_2}{e_1\, e \mapsto e_2\, e} \qquad \frac{e_1 \mapsto e_2}{v\, e_1 \mapsto v\, e_2}$$

$$\frac{}{\mathsf{error}\, e \mapsto \mathsf{error}} \qquad \frac{}{v\, \mathsf{error} \mapsto \mathsf{error}}$$

$$\frac{}{(\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e)\ [\mathcal{P}] \mapsto [\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e, \mathcal{P}/f, w]\, e} \qquad \frac{e_1 \mapsto e_2}{e_1\ [\mathcal{P}] \mapsto e_2\, \mathcal{P}} \qquad \frac{}{\mathsf{error}\ [\mathcal{P}] \mapsto \mathsf{error}}$$

$$\frac{}{(\Lambda\alpha{:}\kappa.e)\ [\mathsf{c}] \mapsto [\mathsf{c}/\alpha]\, e} \qquad \frac{e_1 \mapsto e_2}{e_1\ [\mathsf{c}] \mapsto e_2\ [\mathsf{c}]} \qquad \frac{}{\mathsf{error}\ [\mathsf{c}] \mapsto \mathsf{error}}$$

$$\frac{e_1 \mapsto e_2}{\langle e_1, e\rangle \mapsto \langle e_2, e\rangle} \qquad \frac{e_1 \mapsto e_2}{\langle v, e_1\rangle \mapsto \langle v, e_2\rangle} \qquad \frac{}{\langle \mathsf{error}, e\rangle \mapsto \mathsf{error}} \qquad \frac{}{\langle v, \mathsf{error}\rangle \mapsto \mathsf{error}}$$

$$\frac{}{\pi_i\, \langle v_1, v_2\rangle \mapsto v_i} \qquad \frac{e_1 \mapsto e_2}{\pi_i\, e_1 \mapsto \pi_i\, e_2} \qquad \frac{}{\pi_i\, \mathsf{error} \mapsto \mathsf{error}} \qquad \frac{e_1 \mapsto e_2}{\mathsf{pack}\, \langle \mathcal{P}, e_1\rangle \mapsto \mathsf{pack}\, \langle \mathcal{P}, e_2\rangle}$$

$$\frac{}{\mathsf{pack}\, \langle \mathcal{P}, \mathsf{error}\rangle \mapsto \mathsf{error}} \qquad \frac{}{\mathsf{let\ pack}\, \langle w, x\rangle = \mathsf{pack}\, \langle \mathcal{P}, v\rangle\ \mathsf{in}\ e\ \mathsf{end} \mapsto [\mathcal{P}, v/w, x]\, e}$$

$$\frac{e_1 \mapsto e_2}{\mathsf{let\ pack}\, \langle w, x\rangle = e_1\ \mathsf{in}\ e\ \mathsf{end} \mapsto \mathsf{let\ pack}\, \langle w, x\rangle = e_2\ \mathsf{in}\ e\ \mathsf{end}} \qquad \frac{}{\mathsf{let\ pack}\, \langle w, x\rangle = \mathsf{error}\ \mathsf{in}\ e\ \mathsf{end} \mapsto \mathsf{error}}$$

$$\frac{e_1 \mapsto e_2}{\mathsf{C}\, [\mathsf{c}_1 \ldots \mathsf{c}_m \mathcal{P}_1 \ldots \mathcal{P}_n, e_1] \mapsto \mathsf{C}\, [\mathsf{c}_1 \ldots \mathsf{c}_m \mathcal{P}_1 \ldots \mathcal{P}_n, e_2]} \qquad \frac{}{\mathsf{C}\, [\mathsf{c}_1 \ldots \mathsf{c}_m \mathcal{P}_1 \ldots \mathcal{P}_n, \mathsf{error}] \mapsto \mathsf{error}}$$

$$\frac{}{\mathsf{case}^{\tau}\ \mathsf{C}\, [\mathsf{c}_1 \ldots \mathsf{c}_m \mathcal{P}_1 \ldots \mathcal{P}_n, v]\ \mathsf{of}\ \ldots |\mathsf{C}\, [w_1 \ldots w_n, x] \Rightarrow e| \ldots \mathsf{end} \mapsto [\mathcal{P}_1 \ldots \mathcal{P}_n, v/w_1 \ldots w_n, x]\, e}$$

$$\frac{e_1 \mapsto e_2}{\mathsf{case}^{\tau}\ e_1\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{case}^{\tau}\ e_2\ \mathsf{of}\ ms\ \mathsf{end}} \qquad \frac{}{\mathsf{case}^{\tau}\ \mathsf{error}\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{error}}$$

We can now prove the standard type soundness results using the so-called syntactic method. This depends on proving progress and preservation. Progress says that well-typed terms can always perform a transition, unless they have been fully evaluated successfully to a value or unsuccessfully to the special term error. This is of course an useful property of the system, but arguably, the more important property is of type preservation, also known as the subject reduction property. This says that the type of a term is preserved under a single step of the evaluation relation. This is important because we wish to talk about partial correctness of programs, that is, we want to say something about the results of execution.

8.3.1 Progress

The property of progress is proved by a simple structural induction.

Theorem 8.3.1 (Progress) *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash e : \tau$ *then either* e *is a value, or* e *is* error*, or there exists a* e_1 *such that* $e \mapsto e_1$*.*

Proof

By structural induction on the typing derivation. We show a few representative cases.

Case 1: $\dfrac{\Upsilon(x) = \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash x : \tau}$

Impossible, since $\Upsilon = \cdot$.

Case 2: $\dfrac{}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{unit} : \mathsf{Unit}}$

Directly, since unit is a value.

Case 3: $\dfrac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau_2 \quad \Psi;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau_1}$

e is a value, or e is error, or evaluates to some e_1 — By inductive hypotheses.

Case 4: $\dfrac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \tau_1 \to \tau_2 \quad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_2 : \tau_1}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1\, e_2 : \tau_2}$

e_1 is a value, or e_1 is error, or e_1 evaluates to e_1' — By induction

Subcase 4.1: e_1 is a value v_1

$e_1 = \mathsf{fun}\, f(x{:}\tau_1'){:}\tau_2'.e$ — By canonical forms

e_2 is a value, or e_2 is error, or e_2 evaluates to e_2' — By induction

Subcase 4.1.1: e_2 is a value v_2

$\mathsf{fun}\, f(x{:}\tau_1'){:}\tau_2'.e\, v_2 \mapsto [\mathsf{fun}\, f(x{:}\tau_1'){:}\tau_2'.e, v_2/f, x]\, e$ — By rule

Subcase 4.1.2: e_2 is error

$\mathsf{fun}\, f(x{:}\tau_1'){:}\tau_2'.e\, \mathsf{error} \mapsto \mathsf{error}$ — By rule

Subcase 4.1.3: $e_2 \mapsto e_2'$

$v_1\, e_2 \mapsto v_1\, e_2'$ — By rule

Subcase 4.2: e_1 is error

$\mathsf{error}\, e_2 \mapsto \mathsf{error}$ — By rule

Subcase 4.3: $e_1 \mapsto e_1'$

$e_1\, e_2 \mapsto e_1'\, e_2$ — By rule

Case 5: $\dfrac{\Psi \vdash \mathcal{Q} : \mathsf{sort} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi, w{:}\mathcal{Q};\Delta;\Upsilon, f{:}(\Pi w{:}\mathcal{Q}.\tau);\mathcal{C} \vdash e : \mathsf{c}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e : \Pi w{:}\mathcal{Q}.\mathsf{c}}$

Directly, since $\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e$ is a value.

Case 6: $\dfrac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \Pi w{:}\mathcal{Q}.\tau \quad \Psi \vdash \Delta : \mathcal{P}\mathcal{Q}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e\ [\mathcal{P}] : [\mathcal{P}/w]\, \tau}$

e is a value, or e is error, or e evaluates to some e' — By induction

Subcase 6.1: e is a value v

$e = \mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e_1$ — By canonical forms

$(\mathsf{fun}\, f(w{:}\mathcal{Q}){:}\tau.e_1)\ [\mathcal{P}] \mapsto [\mathsf{fun}\, f(w{:}\mathcal{Q}){:}\tau.e_1, \mathcal{P}/f, w]\, e_1$ — By rule

Subcase 6.2: e is error

$\mathsf{error}\ [\mathcal{P}] \mapsto \mathsf{error}$ — By rule

Subcase 6.3: $e \mapsto e'$

$e\ [\mathcal{P}] \mapsto e'\ [\mathcal{P}]$ — By rule

Case 7: $\dfrac{\Psi \vdash \mathcal{P} : \mathcal{Q} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : [\mathcal{P}/w]\, \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{pack}\, \langle \mathcal{P}, e \rangle : \Sigma w{:}\mathcal{Q}.\tau}$

e is a value, or e is error, or e evaluates to some e' — By induction

Subcase 7.1: e is a value v

$\mathsf{pack}\, \langle \mathcal{P}, v \rangle$ is a value

Subcase 7.2: e is error

$\mathsf{pack}\,\langle\mathcal{P},\mathsf{error}\rangle \mapsto \mathsf{error}$ By rule

Subcase 7.3: $e \mapsto e'$

$\mathsf{pack}\,\langle\mathcal{P},e\rangle \mapsto \mathsf{pack}\,\langle\mathcal{P},e'\rangle$ By rule

Case 8: $$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \Sigma w{:}\mathcal{Q}.\tau_1 \quad \Psi, w{:}\mathcal{Q};\Delta;\Upsilon, x{:}\tau_1;\mathcal{C} \vdash e_2 : \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{let\ pack}\,\langle w,x\rangle = e_1\,\mathsf{in}\,e_2\,\mathsf{end} : \tau}$$

e_1 is a value, or e_1 is error, or e_1 evaluates to some e_1' By induction

Subcase 8.1: e_1 is a value v

$e_1 = \mathsf{pack}\,\langle\mathcal{P},v_1\rangle$ By canonical forms

$\mathsf{let\ pack}\,\langle w,x\rangle = \mathsf{pack}\,\langle\mathcal{P},v_1\rangle\,\mathsf{in}\,e_2\,\mathsf{end} \mapsto [\mathcal{P}, v_1/w, x]\,e_2$ By rule

Subcase 8.2: e_1 is error

$\mathsf{let\ pack}\,\langle w,x\rangle = \mathsf{error}\,\mathsf{in}\,e_2\,\mathsf{end}\tau \mapsto \mathsf{error}$ By rule

Subcase 8.3: $e_1 \mapsto e_1'$

$\mathsf{let\ pack}\,\langle w,x\rangle = e_1\,\mathsf{in}\,e_2\,\mathsf{end} \mapsto \mathsf{let\ pack}\,\langle w,x\rangle = e_1'\,\mathsf{in}\,e_2\,\mathsf{end}$ By rule

Case 9: $$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\,\mathsf{c}_1 \ldots \mathsf{c}_m \quad \Psi;\Delta \vdash \tau : \mathbb{T} \quad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\,\mathsf{c}_1 \ldots \mathsf{c}_m \to \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{case}^\tau\ e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau}$$

e_1 is a value, or e_1 is error, or e_1 evaluates to some e_1'

Subcase 9.1: e_1 is a value v_1

$e_1 = \mathsf{C}\,[\mathsf{c}_1 \ldots \mathsf{c}_p\mathcal{P}_1 \ldots \mathcal{P}_m, v_1']$ By canonical forms

$ms = \ldots|\mathsf{C}\,[w_1 \ldots w_m, x] \Rightarrow e_2|ms$ Since matches are exhaustive

$\mathsf{case}^\tau\ \mathsf{C}\,[\mathsf{c}_1 \ldots \mathsf{c}_p\mathcal{P}_1 \ldots \mathcal{P}_m, v_1']\ \mathsf{of}\ \ldots|\mathsf{C}\,[w_1 \ldots w_m, x] \Rightarrow e_2|\ldots\,\mathsf{end} \mapsto [\mathcal{P}_1 \ldots \mathcal{P}_m, v_1'/w_1 \ldots w_m, x]\,e_2$ By rule

Subcase 9.2: e_1 is error

$\mathsf{case}^\tau\ \mathsf{error}\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{error}$ By rule

Subcase 9.3: $e_1 \mapsto e_1'$

$\mathsf{case}^\tau\ e_1\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{case}^\tau\ e_1'\ \mathsf{of}\ ms\ \mathsf{end}$ By rule

□

8.3.2 Preservation

Preservation is also easy to prove, by a structural induction on the typing derivation and an inner case split over the evaluation derivation.

Theorem 8.3.2 (Preservation) *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash e_1 : \tau$ *and* $\cdot \mapsto e_2$*, then* $\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau$*.*

Proof

By induction on the structure of the typing judgment. We show some cases.

Case 1: $$\frac{\Upsilon(x) = \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash x : \tau}$$

Impossible, since $\Upsilon = \cdot$

Case 2: $$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau_2 \quad \Psi;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \tau_1}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau_2$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau_2$ By type conversion rule

Case 3: $$\frac{\Psi \vdash \mathcal{Q} : \mathsf{sort} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi, w{:}\mathcal{Q};\Delta;\Upsilon, f{:}(\Pi w{:}\mathcal{Q}.\tau);\mathcal{C} \vdash e : \mathsf{c}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e : \Pi w{:}\mathcal{Q}.\mathsf{c}}$$

Impossible, since no evaluation rule applies

Case 4: $$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \Pi w{:}\mathcal{Q}.\tau \quad \Psi;\cdot \vdash \mathcal{P} : \mathcal{Q}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e\ [\mathcal{P}] : [\mathcal{P}/w]\,\tau}$$

We case analyze based on the evaluation rule applied

Subcase 4.1: $$\frac{}{(\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e)\ [\mathcal{P}] \mapsto [\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e, \mathcal{P}/f, w]\, e}$$

$w{:}\mathcal{Q};\cdot;f{:}(\Pi w{:}\mathcal{Q}.\tau);\mathsf{true} \vdash e : \tau$ By inversion

$\cdot;\cdot;\cdot;\mathsf{true} \vdash [\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e, \mathcal{P}/f, w]\, e : [\mathcal{P}/w]\,\tau$ By substitution

Subcase 4.2: $$\frac{e_1 \mapsto e_2}{e_1\ [\mathcal{P}] \mapsto e_2\ \mathcal{P}}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \Pi w{:}\mathcal{Q}.\tau$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2\ [\mathcal{P}] : [\mathcal{P}/w]\,\tau$ By rule

Subcase 4.3: $$\frac{}{\mathsf{error}\ [\mathcal{P}] \mapsto \mathsf{error}}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{error} : \tau$ By rule

Case 5: $$\frac{\Psi;\cdot \vdash \mathcal{P} : \mathcal{Q} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : [\mathcal{P}/w]\,\tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{pack}\,\langle \mathcal{P}, e\rangle : \Sigma w{:}\mathcal{Q}.\tau}$$

We case analyze on the evaluation rule applied

Subcase 5.1: $$\frac{e_1 \mapsto e_2}{\mathsf{pack}\,\langle \mathcal{P}, e_1\rangle \mapsto \mathsf{pack}\,\langle \mathcal{P}, e_2\rangle}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : [\mathcal{P}/w]\,\tau$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{pack}\,\langle \mathcal{P}, e\rangle : \Sigma w{:}\mathcal{Q}.\tau$ By rule

Subcase 5.2: $$\frac{}{\mathsf{pack}\,\langle \mathcal{P}, \mathsf{error}\rangle \mathcal{P} \mapsto \mathsf{error}}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{error} : \tau$ By rule

Case 6: $$\frac{\Psi;\Delta \vdash \tau : \mathbb{T} \qquad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \Sigma w{:}\mathcal{Q}.\tau_1 \quad \Psi, w{:}\mathcal{Q};\Delta;\Upsilon, x{:}\tau_1;\mathcal{C} \vdash e_2 : \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{let\ pack}\,\langle w, x\rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : \tau}$$

We case analyze based on the evaluation rule applied

Subcase 6.1: $$\frac{}{\mathsf{let\ pack}\,\langle w, x\rangle = \mathsf{pack}\,\langle \mathcal{P}, v\rangle\ \mathsf{in}\ e\ \mathsf{end} \mapsto [\mathcal{P}, v/w, x]\, e}$$

$\cdot;\cdot \vdash \mathcal{P} : \mathcal{Q}$,

$\cdot;\cdot;\cdot;\mathsf{true} \vdash v : [\mathcal{P}/w]\,\tau_1$ By inversion

$\cdot;\cdot;x{:}[\mathcal{P}/w]\,\tau_1;\mathsf{true} \vdash [\mathcal{P}/w]\, e : \tau$ By substitution

$\cdot;\cdot;\cdot;\mathsf{true} \vdash [\mathcal{P}, v/w, x]\, e : \tau$ By substitution

Subcase 6.2: $$\frac{e_1 \mapsto e_2}{\mathsf{let\ pack}\,\langle w, x\rangle = e_1\ \mathsf{in}\ e\ \mathsf{end} \mapsto \mathsf{let\ pack}\,\langle w, x\rangle = e_2\ \mathsf{in}\ e\ \mathsf{end}}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \Sigma w{:}\mathcal{Q}.\tau_1$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{let\ pack}\,\langle w, x\rangle = e_2\ \mathsf{in}\ e\ \mathsf{end} : \tau$ By rule

Subcase 6.3: $$\frac{}{\mathsf{let\ pack}\,\langle w, x\rangle = \mathsf{error}\ \mathsf{in}\ e\ \mathsf{end} \mapsto \mathsf{error}}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{error} : \tau$ By rule

Case 7:

$$\frac{\begin{array}{c}\mathcal{S}(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1).\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1.\ldots\Pi w_n{:}\mathcal{Q}_n.\\ \tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p\\ \forall 1 \le i \le n.\quad \Psi, w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathcal{P}_{1i} : \mathcal{Q}_i\\ \Psi;\Delta;\Upsilon;\mathcal{C} \vdash e : \left[\begin{array}{c|c}\mathsf{c}_1,\ldots,\mathsf{c}_p, & \alpha_1,\ldots,\alpha_p,\\ \mathcal{P}_{11},\ldots,\mathcal{P}_{1n} & w_1,\ldots,w_n\end{array}\right]\tau_1\end{array}}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{C}\left[\begin{array}{c|c}\mathsf{c}_1\ldots\mathsf{c}_p, & \\ \mathcal{P}_{11}\ldots\mathcal{P}_{1n} & , e\end{array}\right] : \mathsf{D}(\begin{array}{l}[\mathcal{P}_{11},\ldots,\mathcal{P}_{1n}/w_1,\ldots,w_n]\,\mathcal{P}_{21}\ldots\\ [\mathcal{P}_{11},\ldots,\mathcal{P}_{1n}/w_1,\ldots,w_n]\,\mathcal{P}_{2m}\\ \mathsf{c}_1\ldots\mathsf{c}_p\end{array})}$$

We case analyze based on the evaluation rule applied

Subcase 7.1: $\dfrac{e_1 \mapsto e_2}{\mathsf{C}\,[\mathsf{c}_1\ldots\mathsf{c}_m\mathcal{P}_1\ldots\mathcal{P}_n, e_1] \mapsto \mathsf{C}\,[\mathsf{c}_1\ldots\mathsf{c}_m\mathcal{P}_1\ldots\mathcal{P}_n, e_2]}$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \left[\begin{array}{c|c}\mathsf{c}_1,\ldots,\mathsf{c}_p, & \alpha_1,\ldots,\alpha_p,\\ \mathcal{P}_{11},\ldots,\mathcal{P}_{1n} & w_1,\ldots,w_n\end{array}\right]\tau_1$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{C}\left[\begin{array}{c|c}\mathsf{c}_1\ldots\mathsf{c}_p, & \\ \mathcal{P}_{11}\ldots\mathcal{P}_{1n} & , e_2\end{array}\right] : \mathsf{D}(\begin{array}{l}[\mathcal{P}_{11},\ldots,\mathcal{P}_{1n}/w_1,\ldots,w_n]\,\mathcal{P}_{21}\ldots\\ [\mathcal{P}_{11},\ldots,\mathcal{P}_{1n}/w_1,\ldots,w_n]\,\mathcal{P}_{2m}\\ \mathsf{c}_1\ldots\mathsf{c}_p\end{array})$ By rule

Subcase 7.2: $\dfrac{}{\mathsf{C}\,[\mathsf{c}_1\ldots\mathsf{c}_m\mathcal{P}_1\ldots\mathcal{P}_n, \mathsf{error}] \mapsto \mathsf{error}}$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{error} : \tau$ By rule

Case 8:

$$\frac{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash e_1 : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \quad \Psi;\Delta \vdash \tau : \mathbb{T} \qquad \Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \to \tau}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{case}^{\tau}\ e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau}$$

We case analyze based on the evaluation rule applied

Subcase 8.1:

$$\frac{}{\mathsf{case}^{\tau}\ \mathsf{C}\left[\mathsf{c}'_1\ldots\mathsf{c}'_m\mathcal{P}'_1\ldots\mathcal{P}'_p, v\right]\ \mathsf{of}\ \ldots|\mathsf{C}\,[w_1\ldots w_p, x] \Rightarrow e|\ldots\mathsf{end} \mapsto \left[\mathcal{P}'_1\ldots\mathcal{P}'_p, v/w_1\ldots w_p, x\right]e}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{C}\left[\mathsf{c}'_1\ldots\mathsf{c}'_m\mathcal{P}'_1\ldots\mathcal{P}'_p, v\right] : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m$ By premise
$\mathcal{S}(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1).\ldots\forall(\alpha_m{:}\kappa_m).\Pi w_1{:}\mathcal{Q}_1.\ldots\Pi w_p{:}\mathcal{Q}_p.\tau_1 \to \mathsf{D}(\mathcal{P}''_1\ldots\mathcal{P}''_{n'})\,\alpha_1\ldots\alpha_m,$
for all $1 \le i \le p$, $\Psi, w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathcal{P}'_i : \mathcal{Q}_i$,
$\cdot;\cdot;\cdot;\mathsf{true} \vdash v : \left[\mathsf{c}'_1\ldots\mathsf{c}'_m, \mathcal{P}'_1\ldots\mathcal{P}'_p/\alpha_1\ldots\alpha_p, w_1\ldots w_p\right]\tau_1$,
$\cdot;\cdot \vdash \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \equiv \mathsf{D}([\mathcal{P}'_1,\ldots,\mathcal{P}'_n/w_1,\ldots,w_n]\,\mathcal{P}''_1\ldots[\mathcal{P}'_1,\ldots,\mathcal{P}'_n/w_1,\ldots,w_n]\,\mathcal{P}''_{n'})\,\mathsf{c}_1\ldots\mathsf{c}_m : \mathbb{T}$
By inversion
$n = n'$,
for all $1 \le i \le n$, $\cdot;\cdot \vdash \mathcal{P}_i \equiv [\mathcal{P}'_1,\ldots,\mathcal{P}'_n/w_1,\ldots,w_n]\,\mathcal{P}''_i : \mathcal{Q}_i$ By inversion
$\cdot;\cdot \vdash \mathcal{P}_i : \mathcal{Q}_i$ By regularity
Call $\rho = [\mathcal{P}'_1,\ldots,\mathcal{P}'_n/w_1,\ldots,w_n]$, and $\Psi = w_1{:}\mathcal{Q}_1,\ldots,w_n{:}\mathcal{Q}_n$
$\cdot;\cdot \vdash [\rho]\mathcal{P}_i \equiv [\rho]\mathcal{P}''_i : \mathcal{Q}_i$ From previous
We now case analyze on the matches judgment

Subcase 8.1.1: $\dfrac{}{\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \cdot : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \to \tau}$

Impossible

Subcase 8.1.2: Match success

$\mathcal{S}(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)..\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1..\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p$
$\mathcal{S}(\mathsf{D}) = \Pi w'_1{:}\mathcal{Q}'_1..\ldots\Pi w'_m{:}\mathcal{Q}'_m.\mathbb{T}$
$\tau'_1 = [\mathsf{c}_1\ldots\mathsf{c}_p/\alpha_1\ldots\alpha_p]\,\tau_1$
$\Psi, w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n;\cdot \vdash \mathcal{P}_{21} \doteq \mathcal{P}_1 : \mathcal{Q}'_1 \wedge \ldots \wedge \mathcal{P}_{2m} \doteq \mathcal{P}_m : \mathcal{Q}'_m \wedge \mathcal{C} \Longrightarrow (\rho, \Psi_1, \mathcal{C}_1)$
$\Psi_1;\Delta[\rho];\Upsilon[\rho], x{:}\tau'_1[\rho];\mathcal{C}_1 \vdash e[\rho] : \tau$
$\Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_m)\,\mathsf{c}_1\ldots\mathsf{c}_p \to \tau$

$$\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{C}\,[w_1\ldots w_n, x] \Rightarrow e | ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_m)\,\mathsf{c}_1\ldots\mathsf{c}_p \to \tau$$

$\Psi;\cdot \vdash \mathcal{P}''_1 \doteq \mathcal{P}_1 : \mathcal{Q}'_1 \wedge \ldots \wedge \mathcal{P}''_n \doteq \mathcal{P}_n : \mathcal{Q}'_n \Longrightarrow (\rho_1, \Psi_1, \mathcal{C}_1)$ By premise
There exists ρ_2 such that
$\cdot \vdash \rho_2 : \Psi_1$,
$\rho = \rho_2 \circ \rho_1$,
$\cdot;\cdot \vdash \mathcal{C}_1[\rho_2] = \mathsf{true}$ By correctness of constraint solving
$\Psi_1;\cdot;x{:}\tau'_1[\rho_1];\mathcal{C}_1 \vdash e[\rho_1] : \tau$ By premise
$\Psi_1[\rho_2];\cdot;x{:}\tau'_1[\rho_1][\rho_2];\mathcal{C}_1[\rho_2] \vdash e[\rho_1][\rho_2] : \tau$ By substitution
$\cdot;\cdot;x{:}\tau'_1[\rho];\mathsf{true} \vdash e[\rho] : \tau$ By substitution
$\Psi_1;\cdot;\cdot;\mathcal{C}_1 \vdash [\mathcal{P}'_1, \ldots, \mathcal{P}'_n, v/w_1, \ldots, w_n, x]\,e : \tau$ By substitution

Subcase 8.1.3: Match Failure
$\mathcal{S}(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)..\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1..\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p$
$\mathcal{S}(\mathsf{D}) = \Pi w'_1{:}\mathcal{Q}'_1..\ldots\Pi w'_m{:}\mathcal{Q}'_m.\mathbb{T}$
$\tau'_1 = [\mathsf{c}_1\ldots\mathsf{c}_p/\alpha_1\ldots\alpha_p]\,\tau$
$\Psi, w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n;\cdot \vdash \mathcal{P}_{21} \doteq \mathcal{P}_1 : \mathcal{Q}'_1 \wedge \ldots \wedge \mathcal{P}_{2m} \doteq \mathcal{P}_m : \mathcal{Q}'_m \wedge \mathcal{C} \Longrightarrow \mathsf{false}$
$\Psi;\Delta;\Upsilon;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_m)\,\mathsf{c}_1\ldots\mathsf{c}_p \to \tau$

$$\Psi;\Delta;\Upsilon;\mathcal{C} \vdash \mathsf{C}\,[w_1\ldots w_n, x] \Rightarrow e | ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_m)\,\mathsf{c}_1\ldots\mathsf{c}_p \to \tau$$

Contradiction, by correctness of constraint solving, since we know the constraint problem has a solution

Subcase 8.2: $$\frac{e_1 \mapsto e_2}{\mathsf{case}^\tau\ e_1\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{case}^\tau\ e_2\ \mathsf{of}\ ms\ \mathsf{end}}$$
$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m$ By induction
$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{case}^\tau\ e_2\ \mathsf{of}\ ms\ \mathsf{end} : \tau$ By rule

Subcase 8.3: $$\frac{}{\mathsf{case}^\tau\ \mathsf{error}\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{error}}$$
$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{error} : \tau$ By rule

□

Chapter 9

An XTALT typechecker in LF/ML

We will now use the language LF/ML described in chapter 8 to write certifiably correct checkers. Recall that we have proved the soundness of the XTALT type system within Twelf. We now produce a checker for XTALT written in LF/ML. This will use the formal definition of XTALT within LF. The checker will be written such that its type guarantees that when it returns with success, there exists a derivation in the formal XTALT type system for the program in question.

The language LF/ML is implemented as a type checker and a source-to-source compiler written in the New Jersey implementation of SML. After type checking, the compiler translates LF/ML programs to SML programs. We start with a brief description of the implementation and an illustration of how LF/ML programs are checked.

9.1 LF/ML implementation

The prototype implementation of LF/ML has been written within the SML of New Jersey system. A checker for the XTALT system has been written within this implementation. At the top level, we build a parser and a type checker for the LF/ML language. A final component of the system translates well-typed LF/ML programs to programs in SML by erasing the index annotations from the LF level.

9.1.1 Configuration files

The input to the LF/ML system is in terms of a *configuration*. Configurations are a simple list of three different kinds of files for the system. The first kind is that of a LF source file. These are written using Twelf concrete syntax. In this way, we can reuse the definition of a system written in LF. The second kind of file is that of LF/ML programs, which may use the LF definitions in the LF files. The third kind is the name of an output file, which will contain the ML program obtained by erasing annotations from an LF/ML program. The three kinds of files can be interspersed, and are read (or written to, in the case of the ML file) in the sequence written. Each filename is preceded by the kind of the file. An example configuration file is given in figure 9.1.

9.1.2 $\mathbf{LF}^{\Sigma,1+}$ files

The LF input files are written in the concrete syntax of Twelf. Similar to Twelf, we support the writing of constant types with implicit arguments. If there are free variables whose type can be discovered, they are abstracted over and implicitly quantified at the root of the term. Further uses of such constants with implicit arguments leads to inserting implicit arguments in the structure of the term, and an attempt to discover them by type checking. Type checking can require unification, and standard pattern unification is attempted. Equations not in the pattern fragment are postponed as constraints, and remaining constraints at the end

```
LFS: fomega-tp.elf
LFML: fomega-tp.lfml

LFS: fomega-tm.elf
LFML: fomega-tm.lfml

MLout: fomega.ml
```

Figure 9.1: Sample LF/ML configuration file

of type checking indicates failure. This situation can usually be rectified by adding typing annotations to restrict the set of types considered.

We have preliminary support for writing definitions in the LF source files in the style of Twelf. This makes the LF files more convenient to write and easier to read. We make no attempt to check strictness of the definition's use of arguments, and hence any use of definitions is automatically fully expanded. We have not included standard optimizations of type checking, such as are included in Twelf.

We do not support the full range of Twelf concrete syntax, even the fragment restricted to LF (that is, excluding the proof search and metalogical directives). In particular, there is currently no support for specifying or changing the fixity or precedence of defined predicates.

9.1.3 LF/ML program files

Turning now to LF/ML, the syntax is chosen to be similar to that of SML. The new features are a notation for dependent functions and products, and a notation for indexed datatypes and datatype constructors.

Dependent function types (Π types) are written with the argument in braces, as in {x:A} B for $\Pi x{:}A.B$. Dependent functions are written as Fun, taking both index arguments and normal term level arguments. The index arguments are enclosed within square brackets. Multiple arguments can be written at once. Correspondingly, an application to an index argument is written using square brackets. This is shown in the fragments of code below:

```
(* Function type *)
checkEquiv : {k1:kind} {k2:kind} kind * kind -> ()

(* Declaring a dependent function *)
Fun checkEquiv [k1, k2] (knd1, knd2) = ...

(* Applying a dependent function *)
let val d = checkEquiv [k1] [k2] (knd1, knd2)
```

Dependent product types (Σ types) are written with the first component in angle brackets, as in <x:A> B for $\Sigma x{:}A.B$. The package form is a comma separated list within square brackets, and a unpack form is folded into a let val declaration. This is shown below.

```
(* Package type *)
examplePackage : <e:exp> int

(* Creating a package *)
val a = [u, 5]

(* Unpacking a package *)
let val [x, i] = a in ...
```

Datatypes are declared together with their indexing $\mathrm{LF}^{\Sigma,1+}$ classifier, a kind or a family as follows:

```
datatype Kind of [k:kind]
```

Correspondingly, datatype constructors mention explicitly the $\mathrm{LF}^{\Sigma,1+}$ index they abstract over, and the part of the datatype family they target, as follows:

```
const KArrow  : {k1:kind, k2:kind}
                  Kind [k1] * Kind [k2]
                    => Kind [karrow k1 k2]
```

It is possible in writing programs to omit typing annotations. The implementation will try to perform inference to reconstruct omitted types and indices. Type inference proceeds in two phases. In the first phase, an approximate type is reconstructed for the program. This phase is a standard ML-style type inference, and the approximate types inferred are ML types. In the second part, we try to reconstruct the index information for dependently-typed terms. This will in general require us to unify types and also perform $\mathrm{LF}^{\Sigma,1+}$ unification. $\mathrm{LF}^{\Sigma,1+}$ unification is crucial to check case branches, even in the presence of fully annotated programs.

For case branches, we refine the $\mathrm{LF}^{\Sigma,1+}$ indices to include information from the target of the constant. This requires us to perform higher-order unification. We adopt a constraint-based solution, and solve all constraints in the extended pattern fragment and postpone the rest. Constraints remaining at the end of typechecking indicate type checking failure.

Postponed constraints due to missing annotations can usually be removed by giving more information in the form of annotations. Postponed constraints may however be inherent in the case branches. In practice, we have found that checkers can be written easily without asking the system to go beyond the pattern fragment we have defined. This is because the declarations of constants is usually written to mirror, and sometimes simplify the LF definitions. More complex definitions are rarely written, so the kinds of index variables used are simple. It has already been noted that most practical problems lie within the pattern fragment, and we observe the same behavior.

Notice also that different branches of case analysis have to perform independent unification steps. We thus have to be able to retract instantiation decisions once we have finished checking a particular branch. We thus use a trail to record solutions of unification and use it to rollback decisions on instantiating metavariables.

9.2 The structure of XTALT

XTALT is an explicitly annotated version of TALT. It is a type system for low-level code, designed to be very close to assembly language. In this section, we will describe it in a very broad fashion, enough to understand the structure of the checker, without going into details and justification of the XTALT system itself.

The abstract syntax of XTALT has a top-level type of program, denoting complete programs. A program is a sequence of blocks. Blocks are classified by block types, a subset of the types of TALT, which are represented by the LF type tp.

There are various kinds of blocks, but we will be particularly concerned with two of them. A block can be a code block, which is a sequence of instructions. These instructions represent a basic block, and end with a jump or a special (fake) instruction known as fallthru. Code blocks must have code types, described shortly. Another kind of block represents a global data value placed into memory. Values must have a TALT type of boxed values, since they are data in memory.

Code pieces are assumed never to return, so code types are types of the form $\Gamma \rightarrow 0$. Here Γ ranges over the preconditions in the form of assumptions on the state of registers and stack. Checking a program at the top level then involves checking that it has a code type expecting a standard initial register set and stack (a null stack). This involves checking each block in turn. Code blocks must satisfy code types according to their preconditions. All exits from the block (including the fall-through case) must satisfy the target's preconditions. Value blocks have to have the declared type. This requires us to first extract a store of assumptions of each block's type, and verify each one in turn.

We will not go into great detail on the type and kind structure of TALT. This is described in detail in Crary's work [Cra03]. One issue to highlight is that a low-level system needs to maintain size information on some of its types. This is size measured in number of bytes. For values on stack, Crary [Cra03] describes a mechanism whereby the size has to be determined, even if not explicitly mentioned. Size information is

written as a predicate on types. In addition to the logical connectives of implication and conjunction, the predicate structure includes atomic arithmetic primitives of addition and multiplication. leads us to perform arithmetic while type checking. Checking XTALT thus has to include a simple proof system for arithmetic on natural numbers.

9.3 Programming XTALT in LF/ML

We will describe some interesting issues that arise as we check XTALT. We will ignore trivial details and will not go into great detail about XTALT itself, while trying to illustrate the central programming challenges.

9.3.1 Representing contexts

During checking XTALT, a variety of assumptions have to be made. These include kinding assumptions on type constructor variables, typing assumptions on term variables, and hypothetical proofs of arithmetic. Further, in typing case branches, we often have more precise characterization of term assumptions, and a restricted form of singleton types is present which allows recording so-called replacement assumptions (roughly, a variable can be replaced by a particular value). There is thus a wide variety of dynamic assumptions made.

The LF representation is written so as to place all dynamic assumptions implicitly into the LF context. In chapter 5, we described our method of representing contexts as a product of types. However, the simply typed calculus has a simple context structure. In a more complex calculus, we do not want to make the context a product of every element of the context. To understand the problem, consider the following LF code for typechecking an abstraction in F^ω.

```
expof_lam       : expof (lam T1 E) (arrow T1 T2)
                    <- tyconof T1 ktype
                    <- ({x:exp} expof x T1 -> expof (E x) T2).
```

This says that an abstraction is well-typed if the domain type T1 is a valid type, and on adding to the context a new term variable and a typing assumption, the body is well-typed at the type T2. Now consider a type checking algorithm that outputs the type. This should be possible since all terms are fully annotated. However, if we have one context, the output argument type T2 can depend on the term assumption (and possibly the typing assumption).

The solution in Twelf is to perform a sophisticated subordination analysis to understand that types cannot have embedded terms (or typing assumptions). This is the most principled solution. We however take a simpler approach, and look at the outcome of the subordination analysis. Conceptually, the single LF context can be split after the analysis into multiple contexts, each binding a different kind of type. We then represent each different context as a separate product, packaging them into one datatype.

This solution asks the programmer to perform the analysis that Twelf automatically provides. This is still all right since the programmer of this system is the person who created the LF representation, so this can be looked on as system specific information. More to the point, this allows passing only the particular information to a function instead of a massive all-in-one context, and thus improves separation of concerns.

A further issue in representing these multiple contexts is that one sort of contexts may depend on another, but not vice versa. This issue of dependency can be managed by explicitly abstracting over the domain of dependence, as is done for ordinary terms.

To illustrate, the context for checking F^ω records four different sorts of assumptions, each represented by a different LF type family. There are type constructor assumptions, kinding assumptions on type constructor variables, term assumptions, and typing assumptions on term variables. We split the context into four pieces, one for each of these. We call the constructor variable context ctvar, the kinding judgment context ctkind, the term variable context cevar, and the typing judgment context cetp. Kinding assumptions must mention the constructor variables, so they depend on the ambient constructor context. Term variables and constructor variables do not have any other types embedded. Finally, typing assumptions must mention the term variables, and also possibly the constructor variables. Thus we have the datatype declaration below.

Note that we have chosen only one datatype of contexts. A different implementation might split them into constructor and term contexts.

```
datatype Context of [ctvar:type, ctkind:ctvar -> type,
                     cevar:type, cetp:ctvar -> cexp -> type]
```

Apart from the nil context (not shown here), there are two constructs for creating contexts, one that adds a type constructor and kinding assumption, and one that adds a term variable and typing assumption. These two are shown below. Notice how the different parts of the context are added to separately.

```
const TyCons  : {ct:type, ck:ct -> type, ce:type,
                 cet:ct -> ce -> type, k:kind}
                 Context [ct, ck, ce, cet] * Kind [k] =>
                  Context [ct * tycon,
                           [gt] (ck (#1 gt)) * (tyconof (#2 gt) k),
                           ce,
                           [gt][ge] cet (#1 gt) ge]
const ExpCons : {ct:type, ck:ct -> type, ce:type,
                 cet:ct -> ce -> type, t:ct -> tycon}
                 Context [ct, ck, ce, cet] * Tp [ct, t] =>
                  Context [ct, ck, ce * exp,
                           [gt][ge] (cet gt (#1 ge)) * (expof (#2 ge) (t gt))]
```

In XTALT, we do not have abstraction over values, but we do have abstraction over type constructors. Further, there are universal kinds. During type checking, we thus have to split the context into at least six different parts, for kinds, type constructors, hypothetical proofs, kinding assumptions for constructors, predicate assumptions on constructors, and replacements for constructor variables (singleton assumptions).

9.3.2 Substitutions

A feature of an advanced type system is that the semantics uses the notion of substitutions of various kinds. This is inevitable with higher types and kinds present in the language. There is however a mismatch between the LF representations and ML representations, which we have to bridge.

Consider for example the F^ω calculus again, where the type application rule says that the result type involves a substitution of the argument type.

$$\frac{\Gamma \vdash E : \forall x{:}K.\tau_1 \quad \Gamma \vdash \tau : K}{\Gamma \vdash E[\tau] : \tau_1[\tau/x]}$$

The use of higher-order abstract syntax techniques in LF means that this substitution is conveniently represented as an application, such as:

```
expof_tapp      : expof (tapp E T1) (T2 T1)
                    <- expof E (forall K1 T2)
                    <- tyconof T1 K2
                    <- kequiv K1 K2.
```

Notice that the result type is T2 applied to T1. Let us now try to represent this in our LF/ML program. Assume that the datatype of type constructors is given as follows:

```
datatype Tycon of [ctx:type, t:ctx -> tycon]
```

We have to write a substitution function because of the first order representation (deBruijn representation) that we are using. Then the substitution function to substitute the second argument in the first for the top variable must satisfy the following specification:

```
substTop : {ctx:type} {t1:ctx * tycon -> tycon} {t2:ctx -> tycon}
              Tycon [ctx * tycon, t1] * Tycon [ctx, t2] ->
                  Tycon [ctx, [gamma] t1 (gamma, t2 gamma)]
```

The problem is that the substitution has to work by induction on the structure of t1, and in particular, descend into binders. It is well-known that we need a more general substitution function that can substitute for an arbitrary variable, not just the top one. The challenge is to give a type to the intermediate results for checking purposes.

The method we use is based on explicit substitutions [ACCL91]. This enables talking about the intermediate state of the function in a precise manner. Specifically, we can write a substitution function in the syntax of $\mathrm{LF}^{\Sigma,1+}$ which converts from one context to the other. The datatype of substitution can then be indexed by such a function. We demonstrate using type constructor substitutions for F^{ω}.

```
datatype Tysub of [ctxold:type, ctxnew:type, sub:ctxnew -> ctxold]
```

The constants belonging to this datatype present various cases for substitutions. One important substitution is the Dot operation, which substitutes a specified constructor for the innermost variable (deBruijn index of zero) and an explicit substitution for the remaining variables. Thus it gets rid of the last constructor of a context.

```
const Dot     : {ctxold:type, ctxnew:type,
                 t:ctxnew -> tycon, sub:ctxnew -> ctxold}
                  Tysub [ctxold, ctxnew, sub] * Tp [ctxnew, t] =>
                    Tysub [ctxold * tycon, ctxnew, [gamma] (sub gamma, t gamma)]
```

Another important substitution is the Shift operation, which shifts all variable indices up by one, thus moving terms living in a context to terms living in a context bigger by one element.

```
const Shift   : {ctx:type} unit => Tysub [ctx, ctx * tycon, [gamma] (#1 gamma)]
```

We also have an identity substitution and a compose operation on substitutions.

```
const Id      : {ctx:type} unit => Tysub [ctx, ctx, [gamma] gamma]
const Comp    : {ctx1:type, ctx2:type, ctx3:type,
                 sub1:ctx2 -> ctx1, sub2:ctx3 -> ctx2}
                Tysub [ctx1, ctx2, sub1] * Tysub [ctx2, ctx3, sub2] =>
                Tysub [ctx1, ctx3, [gamma] sub1 (sub2 gamma)]
```

We will now write a program to carry out substitutions in general on the first order deBruijn representation, using the explicit substitutions just defined. The function must have the following type:

```
substTy : {ctx1:type, ctx2:type, tprev:ctx1 -> tycon, sub:ctx2 -> ctx1}
          Tysub [ctx1, ctx2, sub] * Tycon [ctx1, tprev] ->
            Tycon [ctx2, [gamma] tprev (sub gamma)]
```

The function works by recursion on the structure of the type constructor. For example, in the case of the arrow type, we recursively call the substitution function on the components, and return by composing the results under the constructor for Arrow.

```
 case tin of
      Arrow [_, [t1, [t2, (T1, T2)]]] =>
             let
                val T1n = substTy [ctx1] [ctx2] [t1] [sub] (Sub, T1)
                val T2n = substTy [ctx1] [ctx2] [t2] [sub] (Sub, T2)
             in
                Arrow [ctx2] [[gam] t1 (sub gam)] [[g] t2 (sub gam)]
                      (T1n, T2n)
             end
```

More interesting is the case where a constructor variable is added to the context. In this case, the substitution applied recursively to the body must carry the substitution on the preexisting variables, and leave untouched the new constructor variable. Thus the new constructor variable is substituted for itself, and the rest of the substitution is shifted by one position to account for the larger context.

```
      | Forall [_, [tbody, [k, (K, T)]]] =>
             let
```

```
      val Tn = substTy [ctx1 * tycon] [ctx2 * tycon]
                       [[g] tbody (#1 g) (#2 g)]
                       [[g] (sub (#1 g), (#2 g))]
                       ((Dot [ctx1] [ctx2 * tycon]
                             [[g] #2 g] [[g] sub (#1 g)]
                         ((Comp [ctx1] [ctx2] [ctx2 * tycon]
                                [sub] [[g] #1 g]
                                (Sub, Shift [cout] ())),
                           (Tyvar [ctx2 * tycon] [[g] (#2 g)]
                            (Ztp [ctx2] ()))))), T)
   in
      Forall [cout] [[g][t] tbody (sub g) t] [k] (K, Tn)
   end
```

Finally, we have to do the work of performing the substitution when we get to a variable. In this case we defer to a function that works on constructor variables, named substTyvar, with the following signature.

```
substTyvar : {ctx1:type, ctx2:type, tprev:ctx1 -> tycon, sub:ctx2 -> ctx1}
              Tysub [ctx1, ctx2, sub] * Tyvar [ctx1, tprev] ->
                Tycon [ctx2, [gamma] tprev (sub gamma)]
```

This function has to analyze the substitution. In the case of a Shift, the variable is shifted up by one position.

```
   case (Sub, Var) of
      (Shift [ctx, _], Var) => Tyvar [ctx * tycon] [[g] tprev (#1 g)]
                                     (SuccTp [c] [tin] Var)
```

In the case of a Dot form, we look at the variable. If it is the top variable, it must get the substituting term. If it is not, we must recursively call the rest of the substitution on the variable decremented by one, to account for the reduced context.

```
    | (Dot [_, [_, [_, [_, (_, T)]]]], ZeroTp _) => T

    | (Dot [ctx1, [ctx2, [_, [sub, (Sub, _)]]]],
           SuccTp [_, [tindex, TyIndex]]) =>
               subTyvar [ctx1] [ctx2] [tindex] [sub] (Sub, TyIndex)
```

9.3.3 Representing equality

In LF representations, a common idiom is to use syntactically equal existential variables to represent terms that must be equal. For example, the usual representation of the typing rule for application in the simply typed calculus uses the same variable T1 to represent the type both of the domain of the function and of the argument.

```
of_app  : of (app M1 M2) T2
           <- of M1 (arrow T1 T2)
           <- of M2 T1.
```

However, if we consider a type checking function that takes the term as input and produces the type as output, naively using this representation means that we are constraining the output. Thus there is a problem with the flow of information from recursive calls to the function, since we are claiming that we already know the output.

The same issue would arise if we took the logic programming interpretation of the LF signature. This interpretation is implemented within Twelf, where this issue is handled by making the above variables unification (also known as existential, or logic) variables. The information flow problem alluded to above is now contained within the unification algorithm.

Within a functional paradigm, we do not have available this solution. What we have to do is to check the result afterwards to be of the right form. But now we have to specify (as an LF/ML type) this checking step.

We solve this problem at the LF/ML level by creating a new datatype to represent the equality. This has just one constructor, a reflexivity constructor that relates a type to itself.

```
datatype EqTp of [ctp:type, t1:ctp -> tp, t2:ctp -> tp]

const TpReflex  : {ctp:type, t:ctp -> tp}
                    unit =>
                    EqTp [ctp, t, t]
```

The important point about the datatype constructor is that it targets only the datatype relating equal elements. This information can be used in a trivial analysis step to know that the target types are equal, and thus refine information for example in a typing derivation .

```
refineDeriv : {ctx:type, t1:ctx -> tp, t2:ctx -> tp,
               e:ctx -> tm, d1:{g:ctx} of (e g) (t1 g)}
              EqTp [ctx, t1, t2] => <d2:{g:ctx} of (e g) (t2 g)> unit


Fun refineDeriv [ctx] [t1] [t2] [e] [d1] (eqderiv) =
    case eqderiv of TpReflex _ => [d2, ()]
```

9.3.4 Eliminating dependencies

As we have noted, advanced type systems such as TALT have a lot of dependencies in the type and kind structure. Thus, for example, type constructors may potentially depend on both kinds and type constructor assumptions in the context. Certain rules of the semantics may, however, insist that there not be an actual dependency.

For example, there may often be a requirement of a nondependent function type in a calculus that also has dependent function types, as follows:

```
tp  : type.

pi     : tp -> (tp -> tp) -> tp.
arrow  : tp -> tp -> tp.


of/arrow   : of (lam T1 E) (arrow T1 T2)
               <- ({x:tm} of x T1
                      -> of (E x) T2).
```

Again assuming that the type argument is output from a type checking function, we notice that the result type of the recursive call on the body does not depend on the term assumption, even if types potentially can depend on terms. Thus here is a case of a potential dependency which must be eliminated.

We again use our datatype mechanism within LF/ML to encode this transfer between two contexts. As with explicit substitutions, we will need to descend within binders to eliminate a variable which is not necessarily the last variable.

The datatype of unshifting is given as follows:

```
datatype UnshiftTp of [ctp1:type, ctp2:type, f:ctp1 -> ctp2]
```

The constructor for removing the top variable is:

```
const UnshiftTop  : {c:type} unit
                       => UnshiftTp [c * tp, c, [gamma] #1 gamma]
```

We also have a constructor for leaving the last variable untouched, and performing an underlying unshift operation on the other variables.

```
const UnshiftNext : {c1:type, c2:type, f:c1 -> c2}
                      UnshiftTp [c1, c2, f] =>
                      UnshiftTp [c1 * tp, c2 * tp, [g] (f (#1 g), #2 g)]
```

We will now use these unshifting terms together with the equality representations in the previous subsection to implement a function that checks that dependencies are eliminated.

```
unshiftTp : {c1:type, c2:type, f:c1 -> c2, t1:c1 -> tp}
                UnshiftTp [c1, c2, f] * Tp [c1, t1]
                 -> <t2:c2 -> tp> Tp [c2, t2] *
                                  EqTp [c1, t1, [g] t2 (f g)]
```

Of course, there is a possibility that a forbidden variable does occur dynamically in the structure of the term. In this case, we will raise an exception.

```
exception Omitted of unit
```

We now depict some of the cases of the unshift function to exhibit how the function works. For a case such as the arrow type, a recursive call to the components suffices. Notice that we have to perform a trivial case analysis on the equality derivation to exhibit to the LF/ML checker that the input type constructors are equal.

```
      case tin of
         Arrow [_, [t1, [t2, (T1, T2)]]]) =>
              let
                  val [t1n, (T1n, T1r)] = unshiftTp [c1] [c2] [f] [t1]
                                                    (unshift, T1)
                  val [t2n, (T2n, T2r)] = unshiftTp [c1] [c2] [f] [t2]
                                                    (unshift, T2)
              in
                  case (T1r, T2r) of
                      (TpReflex _, TpReflex _) =>
                           [[g] arrow (t1n g) (t2n g),
                            (Arrow [c2] [t1n] [t2n] (T1n, T2n),
                             TpReflex [c1] [[g] arrow (t1 g) (t2 g)] ())]
                  end
              end
```

Sometimes, we have to extend the context before we make the recursive call. This occurs when we descend into the scope of a binder, such as in the case for a universal type. In this case, we package up the unshift argument to not touch the top variable.

```
          Forall [_, [tbody, Tbody]]) =>
              let
                  val [tbodyn, (Tbodyn, Tr)] = unshiftTp
                                               [c1 * tp] [c2 * tp]
                                               [[g] (f (#1 g), #2 g)]
                                               [[g] tbody (#1 g) (#2 g)]
                                               (UnshiftNext [c1] [c2] [f] unshift,
                                                Tbody)
               in
                  case Tr of
                      TpReflex _ =>
                         [[g] forall ([tin] tbodyn (g, tin)),
                          ((Forall [c2] [[g] [tin] tbodyn (g, tin)] Tbodyn),
                          TpReflex [c1] [[g] forall ([tin] tbody g tin)] ())]
```

```
        end
    end
```

The main work of the unshifting function of course is in the variable case. We defer to a helper function `unshiftTyvar` with the following signature:

```
unshiftTyVar : {c1:type, c2:type, f:c1 -> c2, t1:c1 -> tp}
                   UnshiftTp [c1, c2, f] * Tyvar [c1, t1]  ->
                      <t2:c2 -> tp> Tyvar [c2, t2] *
                            EqTp [c1, t1, [g] t2 (f g)]
```

First let us consider the case that we have a top-level unshift. This means that the last variable, or deBruijn index zero, cannot occur. If we have the top variable, of index zero, we raise an exception. If we have any other variable index, we return the variable index decremented by one. Notice that if we strip off the successor, the inner variable has the right index. It also is well-formed in a context smaller by one element, which is what we want.

```
case (unshift, tvar) of
    (UnshiftTop _, ZeroTp _) => raise Omitted ()
  | (UnshiftTop _, SuccTp [_, [t, T]]) =>
        [t, (T, TpReflex [c1] [[g] t (#1 g)] ())]
```

The remaining case is when we have a unshift operation which leaves the top variable untouched. In this case, if the variable index is zero, it can be returned itself. If not, we have to unshift the variable by the packaged unshift operation. The result must be incremented by one. Notice again the trivial case analysis of the equality term to satisfy typing.

```
  | (UnshiftNext [c1, [c2, _]], ZeroTp _) =>
        [[g] #2 g, (ZeroTp [c2] (), TpReflex [c1 * tp] [[g] #2 g] ())]
  | (UnshiftNext [c1, [c2, [f, unshift]]], SuccTp [_, [t1, T1]]) =>
        let
            val [t2, (T2, Tr)] = unshiftTyVar [c1] [c2] [f] [t1]
                                                  (unshift, T1)
        in
            case Tr of
                TpReflex _ =>
                    [[g] t2 (#1 g),
                     (SuccTp [c2] [t2] T2,
                      TpReflex [c1 * tp] [[g] t1 (#1 g)] ())]
            end
        end
```

9.4 Soundness of the checker

The checker for XTALT is written in LF/ML, written above definitions of datatypes and constructors corresponding to the LF representations. The top level function has the signature given as:

```
checkProgram : {p:program} Prog [p] -> <dprogok:check_program p> unit
```

Assume then that we have a XTALT program, whose LF representation is `prog` and correspondingly, a LF/ML representation `MLProg` of type `Prog [prog]`. Thus, by the use of the typing rules, we have a derivation of the following:

$$\cdot;\cdot;\cdot;\cdot \vdash \texttt{checkProgram [prog] (MLProg)} : \texttt{<dprogok:check\_program prog> unit}$$

Suppose the checker terminates in success, that is, terminates with a value `v`.

Then, under those assumptions, we must have that `prog` is well-typed within XTALT.

Proposition 9.4.1 *Under the conditions of the last two paragraphs, the XTALT program* `prog` *is well-typed within XTALT.*

Proof

$\cdot;\cdot;\cdot;\cdot \vdash$ `v : <d:check_program prog> unit`	By subject reduction
`v = [d, ()]` with $\cdot;\cdot \vdash$ `d : check_program prog`	By canonical forms (LF/ML)
There exists a canonical d^N such that $\cdot;\cdot \vdash d^N \equiv$ `d : check_program prog`	By canonical forms ($\mathrm{LF}^{\Sigma,1+}$)
$\cdot;\cdot \vdash d^N$: `check_program prog`	By regularity ($\mathrm{LF}^{\Sigma,1+}$)
$\cdot \vdash^{LF} d^N$: `check_program prog`	By conservativity over LF
There is a derivation of typing of `prog` in XTALT	By adequacy of XTALT encoding

□

Thus, we have a certifying checker for XTALT.

Chapter 10

Conclusion

The problem of generating trust in an untrusted setting is a general problem, which is of increasing interest. The solution of certified code provides an elegant solution, which can be used by consumers of code to get high assurance of the code they run, before the code has started executing. Engineering such a system is a significant challenge, since the system must not impose undue burdens on its users, while being provably safe.

This thesis presents the design and implementation of a practical certified code system. For high assurance, we also want this system to be foundational, in the sense that proofs of safety of execution is relative to a concrete machine architecture. In a now-common approach, we factored the work of proving into two parts, one showing a broad class of programs to be sound, and then checking that the program belongs to the class. We will now survey and discuss our key contributions in each of these parts.

10.1 Mechanized Proof of Safety

The first part of the work involved a mechanized proof of safety for TALT. This used the Twelf metalogical framework heavily for representing and verifying the proof. The logical framework LF provides an elegant encoding of formal systems, including our specification of what it means to be safe, and also technical devices such as the TALT type system to help us check programs. Built on top of LF, the Twelf metalogical system provides a means of encoding inductive proofs, and can automatically verify such proofs. Our proof of safety is expressed and checked within Twelf.

Twelf was convenient in expressing inductions on derivations, since derivations within logical systems are themselves expressed explicitly within LF. The structure of the proof was in close correspondence with what an informal approach would have been. This helped us prove the safety theorem in a relatively short amount of time.

Twelf of course is a checking program for metatheorems. The mode we used it in was quite interactive, however. The use of LF types for stating and proving theorems meant that Twelf can give the expected types of holes in incomplete proofs. After some experience reading error messages, this made developing proofs interactively very pleasant. Typically, one provides a skeleton of the proof with some holes. Twelf then performs type inference to assign expected types to the holes, and one tries to produce ground terms of those types. The process goes on in an interactive loop till all holes are filled, and Twelf returns the joyous message `%% OK %%`.

There are some costs of the Twelf approach as well. The most serious for a foundational system is the fact that checking metatheorems is complex, requiring a relatively large code base. Twelf is quite well-engineered, but of course there is no formal guarantee that it is correct. There might be legitimate concerns with including this large complex code into the trusted computing base. In mitigation of this, Twelf has solid theoretical backing, and also a growing user base. With use in varied applications, we can gain confidence in its correctness. We have gained in flexibility and extensibility by moving to a foundational setting. We have also gained in requiring trust in fewer components, but not in smaller components.

A more technical issue is that Twelf is currently restricted to the Π_2 class of theorems. This has proved sufficient for our purposes, but of course one can imagine theorems and proof techniques lying outside this class. Work is ongoing to increase the class of expressible proofs, but is not available yet.

In terms of user experience, the most significant shortcoming is the lack of a module system and namespaces. This necessitates the use of ad-hoc naming conventions, and requires a certain degree of care in separating different components. A module system for LF is on the drawing-board, and when implemented will remove this current problem.

10.2 Certified programming

The second part of the work involved verifying a type checker to be correct relative to an LF specification. We designed a programming language, LF/ML, to be able to express such properties of functional programs. This used dependent types crucially to express the connection with the LF specification. Indeed, types are indexed over an extension of LF. The representation layer allows the well-understood method of representing logics within LF. It also went further and allowed representing reified contexts. The programming language types were indexed over this representation layer, and thus could express partial correctness statically. Also, since there is a clear phase separation between representation and computation, an erasure interpretation is possible, leading to programs within ML.

LF/ML provides an elaboration of the idea that the type of a program is a specification of what the program computes. As we increase the expressiveness of the type structure, we can make more precise specifications. The LCF project demonstrated the use of static typing to ensure provers produced valid proofs. We show that using dependent, or indexed, types, we can improve the guarantee so that the proof is of a specified theorem. This is very close to the notion of reflection where programs talk about represented logics. By maintaining a clear phase separation between programs and representations of logics, we provide effective type checking.

A desirable extension of the current work is to integrate with SML's module system. This is crucial to large-scale programming. The language has been designed to make this extension simpler by having a clear phase separation. The types are always indexed over purely static terms. The theory of the interaction of our type structure with singleton kinds present in signatures remains to be worked out.

Harper and Licata are working out a general theory of indexed types, including user-defined extensions of the index structure. They propose additional, static, computation at the index level. When completed, this line of research may subsume LF/ML, by making the representation language $\mathrm{LF}^{\Sigma,1+}$ a particular instance of an indexing language.

A second desirable extension is the incorporation of effects such as state. The system has been designed to have a call-by-value semantics in the anticipation of this extension. State however brings complications on the type system. What type to assign a given location in memory becomes a key issue. Giving it an explicit indexed type would mean that it cannot thereby change in a significant manner. On the other hand, allowing the index of the type of a location to change would require the use of linearity to manage assumptions. A system such as that of Mandelbaum, Harper and Walker [MWH03] would be a good starting point for such an effort.

A different proposed direction of work would be the translation of indexing by LF to lower-level languages. Xi and Harper show preliminary work in a dependently typed assembly language DTAL. Extending this language with indexing by a representational logical framework will allow stating precise properties of low-level code. More speculatively, maintaining our kind of indexed types through compilation will open up new opportunities in optimizations and proofs of their correctness. Certifying compilers for LF/ML like languages, if built, can provide some very strong guarantees indeed.

Bibliography

[ABF+05] Brian Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Proceedings of the Eighteenth Theorem Proving in Higher Order Logics*, 2005.

[ACCL91] Martin Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 243–253, Boston, January 2000.

[AF04] Andrew W. Appel and Amy P. Felty. Dependent types ensure partial correctness of theorem provers. *Journal of Functional Programming*, 14(1):3–19, January 2004.

[AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Draft, 2005.

[AMSV03] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. *Journal of Automated Reasoning*, 31:231–260, 2003.

[App01] Andrew W. Appel. Foundational proof carrying code. In *Sixteenth IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[BL02] Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. In *Eighteenth International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Denmark, July 2002. Springer-Verlag.

[CAB+86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[CCNS05] Bor-Yuh Evan Chang, Adam Chlipala, George C Necula, and Robert Schneck. The open verifier framework for foundational verifiers. In *Second Types in Language Design and Implementation*, 2005.

[CDX05] Sa Cui, Kevin Donnelly, and Hongwei Xi. ATS: A language that combines programming with theorem proving. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems*, pages 310–320, Vienna, Austria, September 2005.

[CLN+00] Christopher Colby, Peter Lee, George C Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[Con01] ConCert. `http://www.cs.cmu.edu/~concert`, September 2001.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *Thirtieth ACM Symposium on*

Principles of Programming Languages, pages 198–212, New Orleans, Louisiana, January 2003.

[Cra05] Karl Crary. Logical relations and a case study in equivalence checking. In Benjamin C Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2005.

[CWAF03] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–219, June 2003.

[CX03] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *Eighth ACM International Conference on Functional Programming*, pages 275–286, Uppsala, Sweden, August 2003.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proceedings of the TenthACM International Conference on Functional Programming*, pages 66–77, Tallinn, Estonia, September 2005.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Eleventh IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Del04] Delphin. `http://cs-www.cs.yale.edu/homes/carsten/delphin/`, September 2004.

[Dis04] Distributed.Net. `http://distributed.net`, June 2004.

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 38(3):555–604, May 2001.

[DP03] Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In A.D. Gordon, editor, *Sixth Foundations of Software Systems and Computation Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 250–266, Warsaw, Poland, April 2003. Springer-Verlag.

[DP04] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In Xavier Leroy, editor, *Thirty-First ACM Symposium on Principles of Programming Languages*, pages 281–292, Venice, Italy, January 2004.

[Dug98] Dominic Duggan. Unification with extended patterns. *Theoretical Computer Science*, 206:1–50, 1998.

[Dun02] Joshua Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University, School of Computer Science, September 2002.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[GMW79] Michael J C Gordon, Robin Milner, and Christopher P Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Har94] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, 1994. Follow-up note in *Information Processing Letters*, 57(1), 1996.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[HL06] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework, 2006. Submitted for publication.

[HP00] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Carnegie Mellon University, School of Computer Science, July 2000.

[HST+02] Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Seventeenth IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[JWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: Type inference for generalised algebraic data types. Draft, July 2004.

[Kno87] Todd Knoblock. *Metamathematical Extensibility in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, 1987.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Ler06] Xavier Leroy. Formal certification of a compiler back–end. In *Thirty-Third ACM Symposium on Principles of Programming Languages*. ACM Press, 2006.

[Let03] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, Berg en Dal, The Netherlands, April 24-28, 2002, 2003. Springer-Verlag.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, May 1999.

[MCGW02] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[Mil91] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *Eighth International Conference on Logic Programming*, pages 225–269. The MIT Press, 1991.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):3–10, 1996.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004.

[MTBS99] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In S. D. Swierstra, editor, *Eighth European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[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):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[MWH03] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Eighth ACM International Conference on Functional Programming*, pages 213–226, September 2003.

[Nec97] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.

[Nec98] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, September 1998.

[NL96] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second*

Symposium on Operating Systems Design and Implementation, pages 229–243, Seattle, October 1996.

[NL98] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *1998 SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, June 1998.

[NPP05] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. Draft, in submission, September 2005.

[NR01] George Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 142–154, London, January 2001.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[Pfe91a] 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.

[Pfe91b] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Sixth IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, July 1991.

[Pfe01] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. The MIT Press and Elsevier Science, 2001.

[PM89] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system coq—rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[PP00] Brigitte Pientka and Frank Pfenning. Termination and reduction checking in the logical framework. In *Workshop of Automation of Proofs by Mathematical Induction*, June 2000.

[PP03] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *Nineteenth International Conference on Automated Deduction*, 2003.

[PR92] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In *Eleventh International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logic framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

[PS02] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide, Version 1.3R4*, 2002. Available electronically at `http://www.cs.cmu.edu/~twelf`.

[PTS02] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *ACM International Conference on Functional Programming*, Pittsburgh, Pennsylvania, January 2002.

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag.

[Sch00a] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50,

2000.

[Sch00b] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, October 2000.

[SET00] SETI@Home. `http://setiathome.ssl.berkeley.edu`, November 2000.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Fifteenth International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, Lindau, Germany, July 1998. Springer-Verlag.

[SPC05] Susmit Sarkar, Brigitte Pientka, and Karl Crary. Small proof witnesses for the logical framework LF. Draft, January 2005.

[SPS04] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus : Functional programming with higher-order encodings. In Preparation, September 2004.

[TBS98] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. In *International Colloquium on Automata, Languages, and Programming*, pages 918–929, July 1998.

[The07] The Twelf wiki. `http://twelf.plparty.org`, April 2007.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 203–217, June 1997.

[VC02] Joseph C. Vanderwaart and Karl Crary. A simplified account of the metatheory of linear LF. Technical Report CMU-CS-01-154, Carnegie Mellon University, School of Computer Science, 2002. A short version appeared in 2002 International Workshop on Logical Frameworks and Meta-Languages.

[WAS03] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 264–274, August 2003.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[WLPD98] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), September 1998.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, 1998.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *1998 SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999.