A Language for Certified Computation

Susmit Sarkar and Karl Crary

November 22, 2005

CMU-CS-05-180

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We would like to verify partial correctness for a program that implements a logical system, such as a type system. Further, this verification should be possible at compile-time. We define an expressive language for writing programs together with annotations to help verify such computation. Our language is a dependently-typed functional programming language, that extends a familiar type structure with dependent sums and products over closed terms from the LF language [11]. This lets us reuse LF as our representation language for logics. We demonstrate our ideas by proving a type checker for the simply typed lambda calculus correct.

This material is based on work supported by NSF grant CCR-0204248. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

Keywords: Type Theory, Programming Languages, Certified Code

1 Introduction

A familiar situation is when code written by unknown or untrusted "code producers" is to be executed on trusted machines by "code consumers". Examples include such issues as applets over the web, new drivers for operating systems, or distributed computing projects over the internet. The code consumer can be justifiably wary of executing code of unknown provenance, or even known but not fully trustable provenance. The technology of certified code, pioneered by Proof Carrying Code [18] followed by Typed Assembly Language [16], was developed to resolve such situations, without the necessity of setting up a pre-existing trust relationship. Certified code systems ask the code producer to package a proof of safety of the code together with the code. The code consumer can then verify the safety of the code independently. Automated deduction techniques can then be used to automate the process.

The standard way of proving pieces of code safe to execute is to use the technology of type systems. In this setup, a logical system known as a type system is defined to isolate a set of programs. A good type system for this purpose has the following two important properties. First, a theorem known as "type safety" assures us that any program lying within the set is safe to execute on the machine. Second, it must be possible to check easily whether a program belongs in the set of well-typed programs. An algorithm to do this is called a type checking algorithm.

Unfortunately, the requirement of being able to type check programs statically automatically ensures that the type system must be limited in its expressiveness. That is, some perfectly good programs must be inevitably rejected by a static type system. The issue of what is a type system expressive enough to permit the programs we are interested in will thus change from situation to situation. For this reason, we are interested in building a so-called "foundational certified code system" [1], in which the code producer is allowed to define his own type system for the programs he writes. Such a system is generic enough to accomodate (possibly multiple) untrusted type systems.

In such a system, a key question is how to trust the implementation of the type checking algorithm. Recall that the type system is provided by untrusted sources, and so the type checker must inevitably be provided by the same untrusted source. Whether an implementation of an algorithm correctly implements the purported type system is thus a central question.

In this paper, we design a new programming language, which we call LF/F$^\omega$, built to write programs for "certified computation". This is a programming language designed to express not just an algorithm, but in addition the fact that the algorithm implements a specified logic. The overarching philosophy is that the proof of correctness with respect to the logic is purely static in nature. This means that it can be verified before ever running the program, and has no run-time effect whatsoever. Further, we are interested in partial correctness proofs, that is, we prove that the program returns the right answer if it terminates. We do not deal with the proof obligation of proving programs terminating, so that we do not restrict the programs we want to write.

Let us now talk about proof representation issues. Clearly, our language needs to represent logics and derivations within logics, and reason about derivations. The logical framework LF [11] is widely used for this purpose. We will use LF for representing logics and derivations. The methodology used to represent logics in LF is summarized by the slogan "judgments as types" with its corollary "derivations as terms". Since we want to reason about derivations, and build derivation objects, we want to manipulate terms from the LF language. To be able to manipulate terms easily, we will actually need to extend LF from its original definition. We talk about the reasons and the extensions in the next part, coming up with an extension of LF we call LF$^{\Sigma,1+}$. This language was studied in previous work [23].

The key idea behind being able to statically verify programs is the technology of dependent types. Appel and Felty [2] have already noticed that a dependently-typed language can certify partial correctness. Dependent types essentially are types indexed by objects from some domain of indices. A term belonging to such a dependent type is an object which is statically represented by the corresponding index. The static reasoning can thus be done by reasoning about the indices appearing in types of terms.

Pure dependent type theories include the terms of the language themselves as the index domain referred to above. This leads quickly to intractable problems in type checking in the presence of such term-level constructs as unbounded recursion and effects. The work of Xi and others [28, 7] has shown however that it is possible to have languages which can be checked statically if the index domain is a simpler one. We will let our types be indexed by closed terms from the LF language.

Many compilers for ML-like languages work by elaborating the source-language level into an explicitly typed variant of F^{ω}. Our intended compilers will work the same way, except that we need to enrich the target language to have types dependent on terms from the $LF^{\Sigma,1+}$ language. In the rest of this paper, we will define the language LF/F^{ω}, a language that will serve as an internal language for a compiler for LF/ML. We will also illustrate the language by writing a type checker for the simply-typed lambda calculus and showing how it can be verified.

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

As discussed in the introduction, we want to use the LF logical framework for representing logics and derivations. Judgments and derivations are most naturally represented in LF by the use of so-called "higher-order abstract syntax" [22]. This means that object language variables are represented by framework (LF) language variables, object language abstractions are mapped to the function space of the framework's terms, and issues of alpha-equivalence classes and capture-avoiding substitutions of object language terms are dealt with automatically by the use of similar concepts in the framework. Logic specifications can then be concise, dealing with the constructs of the logic proper, and avoid the need to write boilerplate for capture-avoiding substitutions anew for every new logic.

In our work, we need to reason about derivations and their structure. At a first glance, if we wish to use higher-order abstract syntax, it seems that we need to reason about open terms. There are many difficulties with such reasoning, to do with checking inductive arguments on such terms, though the Delphin project [6] is one approach to solve them. In our work, we wish to simplify, and reason about closed LF terms only. This is at odds with our desire to permit the usual representation techniques.

The solution we adopt is to reify contexts of ordinary LF within the language, to be able to reason about them. A context is a list of bindings of variable at types. We reify such a context as a product of the corresponding types. This idea is an old one, called "telescopes" by deBruijn [5]. Open terms can then be represented as abstractions from the reified context to the terms. Variables are represented as the appropriate projections from the reified context.

This solution thus calls for us to extend LF with dependent product types. Since types in the context can depend on previously declared variables, we need the product to be dependent. Further, the empty context is reified as a unit type. This type has only one canonical inhabitant. The study of the LF language by Harper and Pfenning [12] deleted family-level abstractions which were present in the original proposal for LF. Since open terms are encoded as above, we need the family-level abstractions.

The metatheory of this extension of LF, which we may call $LF^{\Sigma,1}$ was studied in a previous technical report [23]. That work ensured the presence of canonical forms, and also proved the extension conservative over LF. Conservativity here is meant to state that the pre-existing types have all the canonical forms they had before, and no more. Thus, there is no *junk*, or exotic terms, assuming the same signature as before. We have also shown the existence of efficient type checking algorithms for this language.

Source language terms will postulate the existence of certain terms from the above language. As explained above, we will enforce that these postulates always refer to closed terms. The variables that encode these postulates are not variables of the LF language itself. Instead, these variables are a different kind of variable, which bear similarities to the notion of metavariables studied by Nanevski, Pfenning and Pientka [17]. We follow them in introducing metavariables which can depend on an arbitrary context, which is actually more general than we need. This generalization will be important when we study unification within this language, which is not treated in the current work. We slightly extend their work to introduce metavariables at the type family level in addition to that at object level, as both can be postulated by our language.

This language $LF^{\Sigma,1+}$ for LF with dependent products, unit and metavariables is described next.

2.1 Abstract Syntax

The language $LF^{\Sigma,1+}$ is a dependently typed lambda calculus. We have families of types A, classified by kinds K. The type families belonging to the kind type are called types, and may classify objects M. A context Γ assigns types to object language variables. We may declare constants at both type family and object levels. A signature Σ assigns kinds and types respectively to these constants. We also have metavariables, which stand for variables instantiable by objects or families. These are declared together with the (ordinary) context that instantiating terms must live in. Substitutions mediate between the context that metavariables are declared in and the context that they are used. The abstract syntax is generated by the following grammar.

Kinds	$\mathsf{K} ::=$	type	kind of types
	$\mid$	$\Pi\mathsf{x}{:}\mathsf{A}.\mathsf{K}$	dependent product kind
Families	$\mathsf{A} ::=$	a	family constants
	$\mid$	$\mathsf{b}[\sigma]$	family level metavariable
	$\mid$	$\lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	family level abstraction
	$\mid$	$\mathsf{A}\,\mathsf{M}$	family application
	$\mid$	$\Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	family of functions
	$\mid$	$\Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	family of products
	$\mid$	$\mathsf{1}$	unit type
Objects	$\mathsf{M} ::=$	c	object constants
	$\mid$	$\mathsf{u}[\sigma]$	object level metavariable
	$\mid$	x	object variables
	$\mid$	$\lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}$	object functions
	$\mid$	$\mathsf{M}_1\,\mathsf{M}_2$	object level application
	$\mid$	$\langle\mathsf{M}_1, \mathsf{M}_2\rangle^{\mathsf{A}}$	pairs of objects
	$\mid$	$\pi_i\mathsf{M} \quad (i = 1, 2)$	projections from pairs
	$\mid$	$\langle\rangle$	unit object
Substitutions	$\sigma ::=$	$\cdot$	empty
	$\mid$	$\sigma, \mathsf{M}/\mathsf{x}$	cons
Signatures	$\Sigma ::=$	$\cdot$	empty
	$\mid$	$\Sigma, \mathsf{a}{:}\mathsf{K}$	extension by family level constant
	$\mid$	$\Sigma, \mathsf{c}{:}\mathsf{A}$	extension by object level constant
Contexts	$\Gamma ::=$	$\cdot$	empty
	$\mid$	$\Gamma, \mathsf{x}{:}\mathsf{A}$	context extension
Metavariable Contexts	$\Psi ::=$	$\cdot$	empty
	$\mid$	$\Psi, \mathsf{b}{:}(\Gamma \vdash \mathsf{K})$	context extension with family metavariable
	$\mid$	$\Psi, \mathsf{u}{:}(\Gamma \vdash \mathsf{A})$	context extension with object metavariable

Contexts and metavariable contexts are assumed to always bind fresh (disjoint) variables. All variables appearing in judgments will also be assumed to be distinct. The meta-operation $\mathsf{Dom}()$ is defined to return the set of variables bound by an ordinary or metavariable context. We define a partial order on contexts syntactically. $\Gamma_1 \subseteq \Gamma_2$ holds if $\Gamma_1(\mathsf{x}) = \Gamma_2(\mathsf{x})$ for every $\mathsf{x} \in \mathsf{Dom}(\Gamma_1)$. 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. Analogously, the partial order $\Psi_1 \subseteq \Psi_2$ is defined as $\Psi_1(\mathsf{u}) = \Psi_2(\mathsf{u})$ for every $\mathsf{u} \in \mathsf{Dom}(\Psi_1)$ and $\Psi_1(\mathsf{b}) = \Psi_2(\mathsf{b})$ for every $\mathsf{b} \in \mathsf{Dom}(\Psi_1)$.

LF substitutions are finite maps from object variables to objects. They are defined as simultaneously substituting for all variables in their domain. We assume all object variables occurring in substitutions are distinct. We write id_Γ for the substitution which is identity on all variables in the domain of Γ. The result of applying substitutions on objects, families, kinds, indices and sorts is written as $\mathsf{M}[\sigma]$, $\mathsf{A}[\sigma]$, $\mathsf{K}[\sigma]$, and this notation is extended to all judgments $\mathcal{J}$ of the theory.

We also need a notion of metavariable substitutions, which substitute for metavariables.

Metavariable Substitutions	$\rho ::=$	$\cdot$	empty
	$\mid$	$\rho, \mathsf{A}/\mathsf{b}$	cons with family
	$\mid$	$\rho, \mathsf{M}/\mathsf{u}$	cons with object

We write id_Ψ for the identity, and also define $\mathsf{M}[\rho]$ *etc.*

2.2 Static Semantics

2.2.1 Judgment Forms

The static semantics is presented in the form of eleven mutually recursive judgments, whose meanings are explained below.

$\vdash \Psi$	Ψ is a valid context
$\vdash \Sigma : \mathsf{sig}$	Σ is a valid signature
$\Psi \vdash \Gamma : \mathsf{ctx}$	Γ is a valid context
$\Psi;\Gamma_1 \vdash \sigma : \Gamma_2$	σ matches Γ_2 to Γ_1
$\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}$	M has type A
$\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}$	A has kind K
$\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}$	K is a valid kind
$\Psi;\Gamma_1 \vdash \sigma_1 \equiv \sigma_2 : \Gamma_2$	σ_1 equals σ_2 matching Γ_2 to Γ_1
$\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}$	M_1 equals M_2 at type A
$\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}$	A_1 equals A_2 at kind K
$\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$	K_1 equals K_2

The last four judgments define a typed definitional equality judgment. These equate terms at a particular type, families at particular kinds, and kinds, as also equal substitutions.

2.2.2 Inference Rules

Recall that signatures assign types and kinds to constants. The first judgment ensures that such types and kinds are well-formed. Notice that such types and kinds have to be closed as well.

$\boxed{\vdash \Sigma : \mathsf{sig}}$

$$\frac{}{\vdash \cdot : \mathsf{sig}} \qquad \frac{\vdash \Sigma : \mathsf{sig} \qquad \cdot \vdash_\Sigma \mathsf{K} : \mathsf{kind}}{\vdash \Sigma, \mathsf{a}{:}\mathsf{K} : \mathsf{sig}} \qquad \frac{\vdash \Sigma : \mathsf{sig} \qquad \cdot \vdash_\Sigma \mathsf{A} : \mathsf{type}}{\vdash \Sigma, \mathsf{c}{:}\mathsf{A} : \mathsf{sig}}$$

From now on we assume fixed a valid signature Σ and omit it from the judgments. All further judgments assume this signature to be fixed.

Next, we discuss well-formed metavariable contexts. These declare ordinary contexts which instantiating terms for the metavariables must inhabit.

$\boxed{\vdash \Psi}$

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Psi \quad \Psi \vdash \Gamma : \mathsf{ctx} \quad \Psi;\Gamma \vdash \mathsf{K}_b : \mathsf{kind}}{\vdash \Psi, \mathsf{b}{:}(\Gamma \vdash \mathsf{K}_b)} \qquad \frac{\vdash \Psi \quad \Psi \vdash \Gamma : \mathsf{ctx} \quad \Psi;\Gamma \vdash \mathsf{A}_b : \mathsf{type}}{\vdash \Psi, \mathsf{u}{:}(\Gamma \vdash \mathsf{A}_b)}$$

Next, ordinary contexts are typed. These merely ensure that the types declared for ordinary object variables are well-formed in the ambient context. Notice that they may contain occurrences of metavariables.

$\boxed{\Psi \vdash \Gamma : \mathsf{ctx}}$

$$\frac{}{\Psi \vdash \cdot : \mathsf{ctx}} \qquad \frac{\Psi \vdash \Gamma : \mathsf{ctx} \qquad \Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}}{\Psi \vdash \Gamma, \mathsf{x}{:}\mathsf{A} : \mathsf{ctx}}$$

Now we give rules for typing substitutions. The empty substitution matches the empty context, and a substitution extended at one variable must match a context extended at that point.

$\boxed{\Psi;\Gamma_1 \vdash \sigma : \Gamma_2}$

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

Next, we must give types for objects.

$\boxed{\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}}$

Variables

$$\frac{\Gamma(\mathsf{x}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{x} : \mathsf{A}}$$

Constants

$$\frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{c} : \mathsf{A}}$$

Applications

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 : \Pi\mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{M}_1\,\mathsf{M}_2 : [\mathsf{M}_2/\mathsf{x}]\,\mathsf{A}_1}$$

Abstractions

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_2 : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

Pairs

$$\frac{\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type} \qquad \Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_2 : [\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2}{\Psi;\Gamma \vdash \langle \mathsf{M}_1, \mathsf{M}_2\rangle^{\Sigma\mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

Projections

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

Unit

$$\frac{}{\Psi;\Gamma \vdash \langle\rangle : 1}$$

Metavariables

$$\frac{\Psi(\mathsf{u}) = \Gamma_1 \vdash \mathsf{A} \quad \Psi;\Gamma_2 \vdash \sigma : \Gamma_1}{\Psi;\Gamma_2 \vdash \mathsf{u}[\sigma] : [\sigma]\mathsf{A}}$$

Type Conversion

$$\frac{\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}}{\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}_2}$$

The kinding judgment for type families is defined next.

$\boxed{\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}}$

Constants

$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{a} : \mathsf{K}}$$

Abstractions

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \quad \Psi;\Gamma, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{K}}{\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{K}}$$

Applications

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \Pi\mathsf{x}{:}\mathsf{A}_2.\mathsf{K} \qquad \Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{A}_1\,\mathsf{M} : [\mathsf{M}/\mathsf{x}]\,\mathsf{K}}$$

Products

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}}{\Psi;\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}}$$

Sums

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}}{\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}}$$

Unit

$$\frac{}{\Psi;\Gamma \vdash 1 : \mathsf{type}}$$

Metavariables

$$\frac{\Psi(\mathsf{b}) = \Gamma_1 \vdash \mathsf{K} \quad \Psi;\Gamma_2 \vdash \sigma : \Gamma_1}{\Psi;\Gamma_2 \vdash \mathsf{b}[\sigma] : [\sigma]\mathsf{K}}$$

Kind Conversion

$$\frac{\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}_1 \qquad \Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}}{\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}_2}$$

$\boxed{\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}}$

Type

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

Dependent Products

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

Definitional Equality is presented in the form of the four judgments detailed below. These axiomatize equality between substitutions, objects, type families and kinds.

$\boxed{\Psi;\Gamma_1 \vdash \sigma_1 \equiv \sigma_2 : \Gamma_2}$

Nil

$$\frac{}{\Psi;\Gamma \vdash \cdot \equiv \cdot : \cdot}$$

Cons

$$\frac{\Psi;\Gamma \vdash \sigma_1 \equiv \sigma_2 : \Gamma_1 \quad \Psi;\Gamma \vdash \mathsf{A} \equiv \mathsf{A} : \mathsf{type} \quad \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : [\sigma_1]\mathsf{A}}{\Psi;\Gamma \vdash (\sigma_1, \mathsf{M}_1/\mathsf{x}) \equiv (\sigma_2, \mathsf{M}_2/\mathsf{x}) : (\Gamma_1, \mathsf{x}{:}\mathsf{A})}$$

$\boxed{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}}$

Variables

$$\frac{\Gamma(\mathsf{x}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{x} \equiv \mathsf{x} : \mathsf{A}}$$

Constants

$$\frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{c} \equiv \mathsf{c} : \mathsf{A}}$$

Applications

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \Pi\mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} \equiv \mathsf{M}_{21}\,\mathsf{M}_{22} : [\mathsf{M}_{12}/\mathsf{x}]\,\mathsf{A}_1}$$

Abstractions

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_{11} \equiv \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma \vdash \mathsf{A}_{12} \equiv \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{M}_1 \equiv \lambda\mathsf{x}{:}\mathsf{A}_{12}.\mathsf{M}_2 : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

Extensionality
(Product Type)

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \quad \Psi;\Gamma \vdash \mathsf{M}_1 : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \quad \Psi;\Gamma \vdash \mathsf{M}_2 : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \quad \Psi;\Gamma,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1\,\mathsf{x} \equiv \mathsf{M}_2\,\mathsf{x} : \mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

Parallel β-Conversion
(at Product Type)

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Psi;\Gamma,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : \mathsf{A}_2 \qquad \Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \mathsf{A}_1}{\Psi;\Gamma \vdash (\lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{12})\,\mathsf{M}_{11} \equiv [\mathsf{M}_{21}/\mathsf{x}]\,\mathsf{M}_{22} : [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2}$$

Pairs

$$\frac{\Psi;\Gamma \vdash \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type} \quad \Psi;\Gamma \vdash \mathsf{M}_{11} \equiv \mathsf{M}_{21} : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_{12} \equiv \mathsf{M}_{22} : [\mathsf{M}_{11}/\mathsf{x}]\,\mathsf{A}_2}{\Psi;\Gamma \vdash \langle \mathsf{M}_{11},\mathsf{M}_{12}\rangle^{\Sigma\mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} \equiv \langle \mathsf{M}_{21},\mathsf{M}_{22}\rangle^{\Sigma\mathsf{x}:\mathsf{A}_1.\mathsf{A}_2} : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

Projections

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Psi;\Gamma \vdash \pi_1\mathsf{M}_1 \equiv \pi_1\mathsf{M}_2 : \mathsf{A}_1}$$

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

Extensionality
(Unit Type)

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

Parallel Conversion
(at Sum Type)

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_3 : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Psi;\Gamma \vdash \pi_1\langle \mathsf{M}_1,\mathsf{M}_2\rangle^{\mathsf{A}} \equiv \mathsf{M}_3 : \mathsf{A}_1}$$

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}_1 \qquad \Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_3 : \mathsf{A}_2}{\Psi;\Gamma \vdash \pi_2\langle \mathsf{M}_1,\mathsf{M}_2\rangle^{\mathsf{A}} \equiv \mathsf{M}_3 : \mathsf{A}_2}$$

Extensionality
(Sum Type)

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \quad \Psi;\Gamma \vdash \mathsf{M}_2 : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \quad \Psi;\Gamma \vdash \pi_1\mathsf{M}_1 \equiv \pi_1\mathsf{M}_2 : \mathsf{A}_1 \quad \Psi;\Gamma \vdash \pi_2\mathsf{M}_1 \equiv \pi_2\mathsf{M}_2 : [\pi_1\mathsf{M}_1/\mathsf{x}]\,\mathsf{A}_2}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

Metavariables

$$\frac{\Psi(\mathsf{u}) = \Gamma_1 \vdash \mathsf{A} \quad \Psi;\Gamma_2 \vdash \sigma_1 \equiv \sigma_2 : \Gamma_1}{\Psi;\Gamma_2 \vdash \mathsf{u}[\sigma_1] \equiv \mathsf{u}[\sigma_2] : [\sigma_1]\mathsf{A}}$$

Symmetry

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_1 : \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}}$$

Transitivity

$$\frac{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A} \qquad \Psi;\Gamma \vdash \mathsf{M}_2 \equiv \mathsf{M}_3 : \mathsf{A}}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_3 : \mathsf{A}}$$

Type Conversion

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \qquad \Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_1}{\Psi;\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}_2}$$

$\boxed{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}}$

Constants

$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{a} \equiv \mathsf{a} : \mathsf{K}}$$

Abstractions

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

Applications

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

Extensionality

$$\frac{\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type} \quad \Psi;\Gamma \vdash \mathsf{A}_1 : \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K} \quad \Psi;\Gamma \vdash \mathsf{A}_2 : \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K} \quad \Psi;\Gamma,\mathsf{x}{:}\mathsf{A} \vdash \mathsf{A}_1\,\mathsf{x} \equiv \mathsf{A}_2\,\mathsf{x} : \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K}}$$

Parallel Conversion

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

Products

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

Sums

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

Unit

$$\frac{}{\Psi;\Gamma \vdash 1 \equiv 1 : \mathsf{type}}$$

Metavariables

$$\frac{\Psi(\mathsf{b}) = \Gamma_1 \vdash \mathsf{K} \quad \Psi;\Gamma_2 \vdash \sigma_1 \equiv \sigma_2 : \Gamma_1}{\Psi;\Gamma_2 \vdash \mathsf{b}[\sigma_1] \equiv \mathsf{b}[\sigma_2] : [\sigma_1]\mathsf{K}}$$

Symmetry

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_1 : \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}}$$

Transitivity

$$\frac{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K} \qquad \Psi;\Gamma \vdash \mathsf{A}_2 \equiv \mathsf{A}_3 : \mathsf{K}}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_3 : \mathsf{K}}$$

Kind Conversion

$$\frac{\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \qquad \Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}_1}{\Psi;\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}_2}$$

$\boxed{\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}}$

Type

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

Products

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

Symmetry

$$\frac{\Psi;\Gamma \vdash \mathsf{K}_2 \equiv \mathsf{K}_1 : \mathsf{kind}}{\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}}$$

Transitivity

$$\frac{\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \qquad \Psi;\Gamma \vdash \mathsf{K}_2 \equiv \mathsf{K}_3 : \mathsf{kind}}{\Psi;\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_3 : \mathsf{kind}}$$

Well Typed Metavariable Substitutions Finally, we introduce notation for typing metavariable substitutions below.

Definition 2.1 *The judgment* $\Psi_2 \vdash \rho : \Psi_1$ *holds iff* $\forall \mathsf{u} \in \mathsf{Dom}(\Psi_1)$. *if* $\Psi(\mathsf{u}) = \Gamma \vdash \mathsf{A}$ *then* $\Psi_2;\rho(\Gamma) \vdash \rho(\mathsf{u}) : \rho(\mathsf{A})$ *and* $\forall \mathsf{b} \in \mathsf{Dom}(\Psi_1)$. *if* $\Psi(\mathsf{b}) = \Gamma \vdash \mathsf{K}$ *then* $\Psi_2;\rho(\Gamma) \vdash \rho(\mathsf{b}) : \rho(\mathsf{K})$.

Definition 2.2 *The judgment* $\Psi_2 \vdash \rho_1 = \rho_2 : \Psi_1$ *holds iff*

- $\Psi_2 \vdash \rho_1 : \Psi_1$,
- $\Psi_2 \vdash \rho_2 : \Psi_1$, *and*
- $\forall \mathsf{u} \in \mathsf{Dom}(\Psi_1)$. *if* $\Psi(\mathsf{u}) = \Gamma \vdash \mathsf{A}$ *then* $\Psi_2;\rho_1(\Gamma) \vdash \rho_1(\mathsf{u}) \equiv \rho_2(\mathsf{u}) : \rho_1(\mathsf{A})$
- $\forall \mathsf{b} \in \mathsf{Dom}(\Psi_1)$. *if* $\Psi(\mathsf{b}) = \Gamma \vdash \mathsf{K}$ *then* $\Psi_2;\rho_1(\Gamma) \vdash \rho_1(\mathsf{b}) \equiv \rho_2(\mathsf{b}) : \rho_1(\mathsf{K})$

2.3 Properties of $\mathbf{LF}^{\Sigma,1+}$

A closely related version of this language was studied in a previous tech report [23]. That study did not have metavariables, but was otherwise identical to the version in this paper. A straightforward extension to those results give us decidability of type checking and canonical forms for this language.

Type checking for this language involves checking equivalence of types, which in turn depends on checking equivalence of objects. An algorithm which weak head normalizes and compares weak-head normal forms structurally is shown to be sound and complete for checking equivalences. A straightforward algorithm using this equivalence algorithm can then be shown to decide all judgments of the theory above.

Theorem 2.3 (Decidability of Judgments)

1. *If* $\Psi;\Gamma \vdash \mathsf{M}_1 : \mathsf{A}$ *and* $\Psi;\Gamma \vdash \mathsf{M}_2 : \mathsf{A}$, *then it is decidable whether* $\Psi;\Psi^{-}\Gamma \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}^{-}$.
2. *If* $\Psi;\Gamma \vdash \mathsf{A}_1 : \mathsf{K}$ *and* $\Psi;\Gamma \vdash \mathsf{A}_2 : \mathsf{K}$, *then it is decidable whether* $\Psi;\Psi^{-}\Gamma \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}^{-}$.
3. *If* $\Psi;\Gamma \vdash \mathsf{K}_1 : \mathsf{kind}$ *and* $\Psi;\Gamma \vdash \mathsf{K}_2 : \mathsf{kind}$, *then it is decidable whether* $\Psi;\Psi^{-}\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind}$.
4. *Given* Ψ, Γ *and* A *such that* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{type}$ *and a* M, *it is decidable whether* $\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}$.
5. *Given* Ψ, Γ *and* K *such that* $\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}$ *and a* A, *it is decidable whether* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}$.
6. *Given a* Ψ *and* Γ *and a* K, *it is decidable whether* $\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}$.

Proof

In previous work [23]. □

The equivalence judgment can be instrumented to produce a canonical term which are equivalent to the two equal terms. This lets us prove the important canonical forms property. To state this property, we need to define canonical and atomic forms. The set of canonical and atomic objects, families and kinds, are syntactically subsets of the corresponding terms, and are defined by the following grammar.

Canonical Kinds	$\bar{\mathsf{K}}$ ::=	type	kind of types
	|	$\Pi \mathsf{x}{:}\bar{\bar{\mathsf{A}}}.\bar{\mathsf{K}}$	dependent product kind
Atomic Families	$\bar{\mathsf{A}}$ ::=	a	family constants
	|	$\bar{\mathsf{A}}\,\bar{\bar{\mathsf{M}}}$	family application
	|	$\Pi \mathsf{x}{:}\bar{\bar{\mathsf{A}}}_1.\bar{\bar{\mathsf{A}}}_2$	family of functions
	|	$\Sigma \mathsf{x}{:}\bar{\bar{\mathsf{A}}}_1.\bar{\bar{\mathsf{A}}}_2$	family of products
	|	$\mathsf{1}$	unit type
Canonical Families	$\bar{\bar{\mathsf{A}}}$ ::=	$\bar{\mathsf{A}}$	atomic families
	|	$\lambda \mathsf{x}{:}\mathsf{A}_1.\bar{\bar{\mathsf{A}}}_2$	family level abstraction
Atomic Objects	$\bar{\mathsf{M}}$ ::=	c	object constants
	|	x	object variables
	|	$\bar{\mathsf{M}}_1\,\bar{\bar{\mathsf{M}}}_2$	object level application
	|	$\pi_i \bar{\mathsf{M}} \quad (i = 1, 2)$	projections from pairs
Canonical Objects	$\bar{\bar{\mathsf{M}}}$ ::=	$\bar{\mathsf{M}}$	atomic objects
	|	$\langle \bar{\bar{\mathsf{M}}}_1, \bar{\bar{\mathsf{M}}}_2 \rangle^{\mathsf{A}}$	pairs of objects
	|	$\langle\rangle$	unit object
	|	$\lambda \mathsf{x}{:}\mathsf{A}.\bar{\bar{\mathsf{M}}}$	object functions

Notice that these are different from the original notion of canonical forms. A better term might be quasi-canonical forms, or almost canonical forms, but that term is already used in Harper and Pfenning [12] for something else. 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. This is the same in spirit to the quasi-canonical form of Harper and Pfenning [12], but those forms elide type annotations on abstractions, so that the canonical forms do not belong syntactically to the language of LF.

With this definition of canonical forms, we can now state the canonical forms theorem.

Theorem 2.4 (Canonical Forms)

1. *If* $\Psi;\Gamma \vdash \mathsf{M} : \mathsf{A}$ *then there exists a canonical object* $\bar{\bar{\mathsf{M}}}$ *such that* $\Psi;\Gamma \vdash \bar{\bar{\mathsf{M}}} : \mathsf{A}$ *and* $\Psi;\Gamma \vdash \mathsf{M} \equiv \bar{\bar{\mathsf{M}}} : \mathsf{A}$.

2. *If* $\Psi;\Gamma \vdash \mathsf{A} : \mathsf{K}$ *then there exists a canonical family* $\bar{\bar{\mathsf{A}}}$ *such that* $\Psi;\Gamma \vdash \bar{\bar{\mathsf{A}}} : \mathsf{K}$ *and* $\Psi;\Gamma \vdash \mathsf{A} \equiv \bar{\bar{\mathsf{A}}} : \mathsf{K}$.

3. *If* $\Psi;\Gamma \vdash \mathsf{K} : \mathsf{kind}$ *then there exists a canonical kind* $\bar{\mathsf{K}}$ *such that* $\Psi;\Gamma \vdash \bar{\mathsf{K}} : \mathsf{kind}$ *and* $\Psi;\Gamma \vdash \mathsf{K} \equiv \bar{\mathsf{K}} : \mathsf{kind}$.

Proof

In previous work [23]. □

3 Constraint Solving with terms in LF$^{\Sigma,1+}$

We wish to solve particular constraints to typecheck programs in our language. These constraints will be over closed terms of the language LF$^{\Sigma,1+}$ already described. The language of constraints is defined below.

Constraints	$\mathcal{C}$ ::=	$\mathcal{A}$	Atomic Constraint
	|	$\mathcal{A} \wedge \mathcal{C}$	Conjunction
Atomic Constraints	$\mathcal{A}$ ::=	true	True
	|	false	False
	|	$\mathsf{M}_1 =^? \mathsf{M}_2$	Object Equality
	|	$\mathsf{A}_1 =^? \mathsf{A}_2$	Family Equality

The set of solutions to constraint problems is written as either a set of simplified constraints together with some substitution and a metavariable context, or false.

$$\text{Solutions} \quad \mathcal{S} ::= \begin{array}{l} (\mathcal{C}, \rho, \Psi) \\ \mid \quad \mathsf{false} \end{array}$$

The constraint solution judgment can then be written as

$$\Psi \vdash \mathcal{C} \Rightarrow \mathcal{S}$$

When writing constraints, we ensure that the objects and families appearing in them are well-formed, and have the same type (and kind respectively). The semantics we would like to have for this judgment is that $\Psi \vdash \mathcal{C} \Rightarrow (\mathsf{true}, \rho, \Psi_1)$ holds iff the unification problem $\Psi; \cdot \vdash \mathcal{C}$ has the most general solution substitution ρ, with $\Psi \vdash \rho : \Psi_1$, and $\Psi \vdash \mathcal{C} \Rightarrow \mathsf{false}$ holds iff there is no solution for the unification problem. Unfortunately, the system is known to be undecidable, even for the simply typed case. Thus we cannot hope to have a system with such a property. We will give an approximation in this section.

The first set of rules simplifies conjunctions.

$$\frac{\Psi \vdash \mathcal{A} \Rightarrow \mathsf{false}}{\Psi \vdash \mathcal{A} \wedge \mathcal{C} \Rightarrow \mathsf{false}} \qquad \frac{\Psi \vdash \mathcal{A} \Rightarrow (\mathsf{true}, \rho, \Psi_1) \qquad \Psi_1 \vdash [\rho]\mathcal{C} \Rightarrow \mathcal{S}}{\Psi \vdash \mathcal{A} \wedge \mathcal{C} \Rightarrow \mathcal{S}}$$

Now, we have to deal with atomic constraints. The simplest case is the constraints which are true or false syntactically.

$$\frac{}{\Psi \vdash \mathsf{true} \Rightarrow (\mathsf{true}, \mathsf{id}_\Psi, \Psi)} \qquad \frac{}{\Psi \vdash \mathsf{false} \Rightarrow \mathsf{false}}$$

Next, when the constraint is between equal terms, the unifier is obvious.

$$\frac{\Psi; \cdot \vdash \mathsf{M}_1 \equiv \mathsf{M}_2 : \mathsf{A}}{\Psi \vdash \mathsf{M}_1 =^? \mathsf{M}_2 \Rightarrow (\mathsf{true}, \mathsf{id}_\Psi, \Psi)} \qquad \frac{\Psi; \cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{K}}{\Psi \vdash \mathsf{A}_1 =^? \mathsf{A}_2 \Rightarrow (\mathsf{true}, \mathsf{id}_\Psi, \Psi)}$$

Since any well-typed term is equal to a canonical term (at object or family levels), we can assume that the terms in constraints have been brought to canonical form. We can then analyze the form of the two terms.

$$\frac{\Psi; \Gamma \vdash \mathsf{M}_1 \equiv \bar{\bar{\mathsf{M}}}_1 : \mathsf{A} \qquad \Psi; \Gamma \vdash \mathsf{M}_2 \equiv \bar{\bar{\mathsf{M}}}_2 : \mathsf{A} \qquad \Psi \vdash \bar{\bar{\mathsf{M}}}_1 =^? \bar{\bar{\mathsf{M}}}_2 \Rightarrow \mathcal{S}}{\Psi \vdash \mathsf{M}_1 =^? \mathsf{M}_2 \Rightarrow \mathcal{S}}$$

$$\frac{\Psi; \Gamma \vdash \mathsf{A}_1 \equiv \bar{\bar{\mathsf{A}}}_1 : \mathsf{K} \qquad \Psi; \Gamma \vdash \mathsf{A}_2 \equiv \bar{\bar{\mathsf{A}}}_2 : \mathsf{K} \qquad \Psi \vdash \bar{\bar{\mathsf{A}}}_1 =^? \bar{\bar{\mathsf{A}}}_2 \Rightarrow \mathcal{S}}{\Psi \vdash \mathsf{A}_1 =^? \mathsf{A}_2 \Rightarrow \mathcal{S}}$$

$$\frac{\mathsf{A} \text{ is of the form } \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2, \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \text{ or } \mathsf{a}\,\mathsf{M}_1 \ldots \mathsf{M}_m}{\Psi \vdash 1 =^? \mathsf{A} \Rightarrow \mathsf{false}} \qquad \frac{\mathsf{A} \text{ is of the form } 1, \Pi \mathsf{x}{:}\mathsf{A}_1'.\mathsf{A}_2' \text{ or } \mathsf{a}\,\mathsf{M}_1 \ldots \mathsf{M}_m}{\Psi \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 =^? \mathsf{A} \Rightarrow \mathsf{false}}$$

$$\frac{\mathsf{A} \text{ is of the form } 1, \Sigma \mathsf{x}{:}\mathsf{A}_1'.\mathsf{A}_2' \text{ or } \mathsf{a}\,\mathsf{M}_1 \ldots \mathsf{M}_m}{\Psi \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 =^? \mathsf{A} \Rightarrow \mathsf{false}} \qquad \frac{\mathsf{A} \text{ is of the form } 1, \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 \text{ or } \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Psi \vdash \mathsf{a}\,\mathsf{M}_1 \ldots \mathsf{M}_n =^? \mathsf{A} \Rightarrow \mathsf{false}}$$

$$\frac{\Psi \vdash \mathsf{A}_{11} =^? \mathsf{A}_{21} \wedge \mathsf{A}_{12} =^? \mathsf{A}_{22} \Rightarrow \mathcal{S}}{\Psi \vdash \Sigma \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} =^? \Sigma \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} \Rightarrow \mathcal{S}} \qquad \frac{\Psi \vdash \mathsf{A}_{11} =^? \mathsf{A}_{21} \wedge \mathsf{A}_{12} =^? \mathsf{A}_{22} \Rightarrow \mathcal{S}}{\Psi \vdash \Pi \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12} =^? \Pi \mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22} \Rightarrow \mathcal{S}}$$

$$\frac{\mathsf{a}_1 \neq \mathsf{a}_2}{\Psi \vdash \mathsf{a}_1\, \mathsf{M}_{11} \ldots \mathsf{M}_{1n} =^? \mathsf{a}_2\, \mathsf{M}_{21} \ldots \mathsf{M}_{2m} \Rightarrow \mathsf{false}} \qquad \frac{\Psi \vdash \mathsf{M}_{11} =^? \mathsf{M}_{21} \wedge \ldots \wedge \mathsf{M}_{1n} =^? \mathsf{M}_{2n} \Rightarrow \mathcal{S}}{\Psi \vdash \mathsf{a}\, \mathsf{M}_{11} \ldots \mathsf{M}_{1n} =^? \mathsf{a}\, \mathsf{M}_{21} \ldots \mathsf{M}_{2n} \Rightarrow \mathcal{S}}$$

$$\frac{\mathsf{c}_1 \neq \mathsf{c}_2}{\Psi \vdash \mathsf{c}_1\, \mathsf{M}_{11} \ldots \mathsf{M}_{1n} =^? \mathsf{c}_2\, \mathsf{M}_{21} \ldots \mathsf{M}_{2m} \Rightarrow \mathsf{false}} \qquad \frac{\mathsf{x}_1 \neq \mathsf{x}_2}{\Psi \vdash \mathsf{x}_1\, \mathsf{M}_{11} \ldots \mathsf{M}_{1n} =^? \mathsf{x}_2\, \mathsf{M}_{21} \ldots \mathsf{M}_{2m} \Rightarrow \mathsf{false}}$$

$$\frac{\Psi \vdash \mathsf{M}_{11} =^? \mathsf{M}_{21} \wedge \ldots \wedge \mathsf{M}_{1n} =^? \mathsf{M}_{2n} \Rightarrow \mathcal{S}}{\Psi \vdash \mathsf{c}\, \mathsf{M}_{11} \ldots \mathsf{M}_{1n} =^? \mathsf{c}\, \mathsf{M}_{21} \ldots \mathsf{M}_{2n} \Rightarrow \mathcal{S}} \qquad \frac{\Psi \vdash \mathsf{M}_{11} =^? \mathsf{M}_{21} \wedge \ldots \wedge \mathsf{M}_{1n} =^? \mathsf{M}_{2n} \Rightarrow \mathcal{S}}{\Psi \vdash \mathsf{x}\, \mathsf{M}_{11} \ldots \mathsf{M}_{1n} =^? \mathsf{x}\, \mathsf{M}_{21} \ldots \mathsf{M}_{2n} \Rightarrow \mathcal{S}}$$

$$\frac{\Psi \vdash \mathsf{M}_{11} =^? \mathsf{M}_{21} \wedge \mathsf{M}_{12} =^? \mathsf{M}_{22} \Rightarrow \mathcal{S}}{\Psi \vdash \langle \mathsf{M}_{11}, \mathsf{M}_{12} \rangle^{\mathsf{A}_{11}} =^? \langle \mathsf{M}_{21}, \mathsf{M}_{22} \rangle^{\mathsf{A}_{21}} \Rightarrow \mathcal{S}} \qquad \frac{\Psi \vdash \mathsf{M}_1 =^? \mathsf{M}_2 \Rightarrow \mathcal{S}}{\Psi \vdash \pi_i \mathsf{M}_1 =^? \pi_i \mathsf{M}_2 \Rightarrow \mathcal{S}}$$

Pattern Unification A class of unification problems in the simply typed case was shown to be decidable by Miller [15]. This class has come to be known as the pattern fragment. Pfenning [21] extended this result to the corresponding case for dependently typed systems. This fragment is the case where for every metavariables in the problem, the associated substitution is a "pattern substitution", that is, substitutes only distinct variables for variables. Many practically appearing problems fall within this class. We can state a general rule for problems in this class, since we know these are decidable.

$$\frac{\begin{array}{c}\Psi;\cdot \vdash \mathcal{C} \text{ is a pattern unification problem,} \\ \text{and has the most general solution substitution } \rho \qquad \Psi \vdash \rho : \Psi_1\end{array}}{\Psi \vdash \mathcal{C} \Rightarrow (\mathsf{true}, \rho, \Psi_1)}$$

$$\frac{\Psi;\cdot \vdash \mathcal{C} \text{ is a pattern unification problem, and has no solution}}{\Psi \vdash \mathcal{C} \Rightarrow \mathsf{false}}$$

Theorem 3.1 (Soundness of Constraint Solving) *Assume there is some Γ_i and A_i such that $\Psi;\Gamma_i \vdash \mathsf{M}_{i1} : \mathsf{A}_i$ and $\Psi;\Gamma_i \vdash \mathsf{M}_{i2} : \mathsf{A}_i$ for all $\mathsf{M}_{i1} =^? \mathsf{M}_{i2}$ appearing in $\mathcal{C}$, and similarly there is some Γ_i and K_i for each pair of A_{i1} and A_{i2}.*

1. *If $\Psi \vdash \mathcal{C} \Rightarrow \mathsf{false}$ then there is no substitution ρ such that $\Psi;[\rho]\Gamma_i \vdash [\rho]\mathsf{M}_{i1} \equiv [\rho]\mathsf{M}_{i2} : [\rho]\mathsf{A}_i$ and the corresponding for all the families hold together.*
2. *If $\Psi \vdash \mathcal{C} \Rightarrow (\mathsf{true}, \rho, \Psi_1)$ then all the $\Psi_1;[\rho]\Gamma_i \vdash [\rho]\mathsf{M}_{i1} \equiv [\rho]\mathsf{M}_{i2} : [\rho]\mathsf{A}_i$ and the corresponding for all the families hold together.*

Proof

By induction on the derivation of the constraint solving judgment. □

Since higher-order unification is undecidable already for the simply typed case, we cannot hope for an analogous completeness result for a effective judgment.

4 Type Structure of LF/F$^\omega$

Next, we start with defining an explicitly typed internal language, which we will call LF/F$^\omega$. This is an extension of the F$^\omega$ language with dependent function and pair types, and a notion of datatypes. The dependencies are allowed only over closed 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.

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.

4.1 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 u and b.

$$
\begin{array}{llcll}
\text{Kinds} & \kappa & ::= & \mathbb{T} & \text{Kind of Types} \\
 & & \mid & \kappa_1 \to \kappa_2 & \text{Function Kind} \\
\text{Type Constructors} & \mathsf{c}, \tau & ::= & \mathsf{Unit} & \text{Unit Type} \\
 & & \mid & \tau_1 \to \tau_2 & \text{Arrow Type} \\
 & & \mid & \tau_1 \times \tau_2 & \text{Product Type} \\
 & & \mid & \Pi w{:}\mathcal{Q}.\tau & \text{Universal Dependent Types} \\
 & & \mid & \Sigma w{:}\mathcal{Q}.\tau & \text{Existential Dependent Types} \\
 & & \mid & \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n) & \text{Datatypes} \\
 & & \mid & \forall(\alpha{:}\kappa_1).\tau & \text{Forall Type} \\
 & & \mid & \alpha & \text{Constructor Variables} \\
 & & \mid & \lambda(\alpha{:}\kappa_1).\mathsf{c}_2 & \text{Constructor Level Function} \\
 & & \mid & \mathsf{c}_1\, \mathsf{c}_2 & \text{Constructor Level Application} \\
\text{Signatures} & \Sigma & ::= & \cdot & \\
 & & \mid & \Sigma, \mathsf{D}{:}\Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\kappa & \\
\text{Contexts} & \Delta & ::= & \cdot & \\
 & & \mid & \Delta, \alpha{:}\kappa &
\end{array}
$$

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 Σ.

Type constructor substitutions substitute constructors for constructor variables.

$$
\begin{array}{llcll}
\text{Constructor Substitutions} & \sigma & ::= & \cdot & \text{Nil} \\
 & & \mid & \sigma, \mathsf{c}/\alpha & \text{Cons}
\end{array}
$$

As is usual, we denote the identity substitution on the kinding context Δ by id_Δ.

4.2 Static Semantics

The static semantics is defined in terms of the following judgments.

$$
\begin{array}{ll}
\vdash \Sigma & \Sigma \text{ is a valid signature} \\
\Psi \vdash \Delta & \Delta \text{ is a valid context} \\
\Psi;\Delta \vdash \mathsf{c} : \kappa & \mathsf{c} \text{ is a valid type constructor} \\
\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa & \mathsf{c}_1 \text{ and } \mathsf{c}_2 \text{ are equal}
\end{array}
$$

We now give the rules below.

$\boxed{\vdash \Sigma}$

Empty

$$\frac{}{\vdash \cdot : \mathsf{ok}}$$

Datatype Declarations

$$\frac{\vdash \Sigma : \mathsf{ok} \qquad \forall 1 \le i \le n. \begin{cases} w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathsf{A} : \mathsf{type} & \text{if } \mathcal{Q}_i = \mathsf{A} \\ w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathsf{K} : \mathsf{kind} & \text{if } \mathcal{Q}_i = \mathsf{K} \end{cases}}{\vdash \Sigma, \mathsf{D}{:}\Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\kappa}$$

$\boxed{\Psi \vdash \Delta}$

Nil

$$\frac{}{\Psi \vdash \cdot}$$

Cons

$$\frac{\Psi \vdash \Delta}{\Psi \vdash \Delta, \alpha{:}\kappa}$$

$\boxed{\Psi;\Delta \vdash \mathsf{c} : \kappa}$

Unit

$$\frac{}{\Psi;\Delta \vdash \mathsf{Unit} : \mathbb{T}}$$

Arrow

$$\frac{\Psi;\Delta \vdash \tau_1 : \mathbb{T} \quad \Psi;\Delta \vdash \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \tau_1 \to \tau_2 : \mathbb{T}}$$

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

$$\frac{\Psi;\cdot \vdash \mathsf{A} : \mathsf{type} \quad \Psi, \mathsf{u}{:}\mathsf{A};\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{u}{:}\mathsf{A}.\tau : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind} \quad \Psi, \mathsf{b}{:}\mathsf{K};\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{b}{:}\mathsf{K}.\tau : \mathbb{T}}$$

Dependent Existentials

$$\frac{\Psi;\cdot \vdash \mathsf{A} : \mathsf{type} \quad \Psi, \mathsf{u}{:}\mathsf{A};\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{u}{:}\mathsf{A}.\tau : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind} \quad \Psi, \mathsf{b}{:}\mathsf{K};\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{b}{:}\mathsf{K}.\tau : \mathbb{T}}$$

Datatypes

$$\frac{\mathit{\Sigma}(\mathsf{D}) = \Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\mathbb{T} \quad \forall 1 \le i \le n.\, (\Psi;\cdot \vdash \mathcal{P}_i : \mathcal{Q}_i\,)}{\Psi;\Delta \vdash \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n) : \mathbb{T}}$$

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

$$\frac{\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c} : \kappa_2}{\Psi;\Delta \vdash \lambda(\alpha{:}\kappa_1).\mathsf{c} : \kappa_1 \to \kappa_2}$$

Applications

$$\frac{\Psi;\Delta \vdash \mathsf{c}_1 : \kappa_1 \to \kappa_2 \quad \Psi;\Delta \vdash \mathsf{c}_2 : \kappa_1}{\Psi;\Delta \vdash \mathsf{c}_1\, \mathsf{c}_2 : \kappa_2}$$

We give the rules defining definitional equality of constructors below. For the F^ω part, we axiomatize $\beta\eta$ equality. For dependent function and pair types, we defer to definitional equality for $\mathrm{LF}^{\Sigma,1+}$ terms.

Similarly, the rule for equality of two datatypes is that the head terms must be syntactically the same, while the indexing $\mathrm{LF}^{\Sigma,1+}$ terms are definitionally equal.

$\boxed{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa}$

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

$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \quad \Psi, \mathsf{u}{:}\mathsf{A}_1;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{u}{:}\mathsf{A}_1.\tau_1 \equiv \Pi \mathsf{u}{:}\mathsf{A}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \quad \Psi, \mathsf{u}{:}\mathsf{K}_1;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{b}{:}\mathsf{K}_1.\tau_1 \equiv \Pi \mathsf{b}{:}\mathsf{K}_2.\tau_2 : \mathbb{T}}$$

Dependent Existentials

$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \quad \Psi, \mathsf{u}{:}\mathsf{A}_1;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{u}{:}\mathsf{A}_1.\tau_1 \equiv \Sigma \mathsf{u}{:}\mathsf{A}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \quad \Psi, \mathsf{u}{:}\mathsf{K}_1;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{b}{:}\mathsf{K}_1.\tau_1 \equiv \Sigma \mathsf{b}{:}\mathsf{K}_2.\tau_2 : \mathbb{T}}$$

Datatypes

$$\frac{\mathit{\Sigma}(\mathsf{D}) = \Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\mathbb{T} \quad \forall 1 \leq i \leq n.\ (\Psi;\cdot \vdash \mathcal{P}_{1i} \equiv \mathcal{P}_{2i} : \mathcal{Q}_i\,)}{\Psi;\Delta \vdash \mathsf{D}(\mathcal{P}_{11} \ldots \mathcal{P}_{1n}) \equiv \mathsf{D}(\mathcal{P}_{21} \ldots \mathcal{P}_{2n}) : \mathbb{T}}$$

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

$$\frac{\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa_2}{\Psi;\Delta \vdash \lambda(\alpha{:}\kappa_1).\mathsf{c}_1 \equiv \lambda(\alpha{:}\kappa_1).\mathsf{c}_2 : \kappa_1 \to \kappa_2}$$

Extensionality

$$\frac{\Psi;\Delta \vdash \mathsf{c}_1 : \kappa_1 \to \kappa_2 \quad \Psi;\Delta \vdash \mathsf{c}_2 : \kappa_1 \to \kappa_2 \quad \Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c}_1\,\alpha \equiv \mathsf{c}_2\,\alpha : \kappa_2}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa_1 \to \kappa_2}$$

Parallel β-Conversion

$$\frac{\Psi;\Delta,\alpha{:}\kappa_1 \vdash \mathsf{c}_{12} \equiv \mathsf{c}_{22} : \kappa_2 \qquad \Psi;\Delta \vdash \mathsf{c}_{11} \equiv \mathsf{c}_{21} : \kappa_1}{\Psi;\Delta \vdash (\lambda(\alpha{:}\kappa_1).\mathsf{c}_{12})\,\mathsf{c}_{11} \equiv [\mathsf{c}_{21}/\alpha]\,\mathsf{c}_{22} : \kappa_2}$$

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

$$\frac{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa \quad \Psi;\Delta \vdash \mathsf{c}_2 \equiv \mathsf{c}_3 : \kappa}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_3 : \kappa}$$

Well Typed Substitutions The notation for typing substitutions is explained below.

Definition 4.1 *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 4.2 *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*
- $\forall\alpha \in \mathsf{Dom}(\Delta_1).\Psi;\Delta_2 \vdash \sigma_1(\alpha) \equiv \sigma_2(\alpha) : \sigma_1(\Delta_1(\alpha))$.

4.3 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 4.3 (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 4.4 (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.

□

Next, we want to show that well-formed constructors are equivalent to themselves.

Lemma 4.5 (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.

Lemma 4.6 (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 4.7 (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 4.8 (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 need the fact that functionality holds for substitutions. This means that, given equal elements, they produce equal elements.

Lemma 4.9 (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 can now prove the regularity property, which says that constructors judged to be equal at some kind are also well-formed at that kind.

Lemma 4.10 (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 4.11 (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* $\Psi;\Delta \vdash \Pi \mathsf{u}{:}\mathsf{A}.\tau : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\cdot \vdash \mathsf{A} : \mathsf{type}$ *and* $\Psi, \mathsf{u}{:}\mathsf{A};\Delta \vdash \tau : \mathbb{T}$.

5. *If* $\Psi;\Delta \vdash \Pi \mathsf{b}{:}\mathsf{K}.\tau : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind}$ *and* $\Psi, \mathsf{b}{:}\mathsf{K};\Delta \vdash \tau : \mathbb{T}$.

6. *If* $\Psi;\Delta \vdash \Sigma \mathsf{u}{:}\mathsf{A}.\tau : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\cdot \vdash \mathsf{A} : \mathsf{type}$ *and* $\Psi, \mathsf{u}{:}\mathsf{A};\Delta \vdash \tau : \mathbb{T}$.

7. *If* $\Psi;\Delta \vdash \Sigma \mathsf{b}{:}\mathsf{K}.\tau : \kappa$ *then* $\kappa = \mathbb{T}$, $\Psi;\cdot \vdash \mathsf{K} : \mathsf{kind}$ *and* $\Psi, \mathsf{b}{:}\mathsf{K};\Delta \vdash \tau : \mathbb{T}$.

8. *If* $\Psi;\Delta \vdash \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n) : \kappa$ *then* $\kappa = \mathbb{T}$, $\Sigma(\mathsf{D}) = \Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\mathbb{T}$ *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.

□

4.4 Type Equivalence Algorithm

The idea behind checking equivalence of constructors is to weak-head normalize both sides, and compare the normal forms. There is also the extensionality rules, which can be used to judge constructors equal. Thus our algorithm will 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.

$\boxed{\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2}$

$$\frac{}{(\lambda(\alpha{:}\kappa_1).\mathsf{c}_2)\, \mathsf{c}_1 \xrightarrow{\text{whr}} [\mathsf{c}_1/\alpha]\, \mathsf{c}_2} \qquad \frac{\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2}{\mathsf{c}_1\, \mathsf{c} \xrightarrow{\text{whr}} \mathsf{c}_2\, \mathsf{c}}$$

Lemma 4.12 (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

$\boxed{\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa}$

$$\frac{\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2 \quad \Psi;\Delta \vdash \mathsf{c}_2 \Longleftrightarrow \mathsf{c} : \mathbb{T}}{\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c} : \mathbb{T}} \qquad \frac{\mathsf{c}_1 \xrightarrow{\text{whr}} \mathsf{c}_2 \quad \Psi;\Delta \vdash \mathsf{c} \Longleftrightarrow \mathsf{c}_2 : \mathbb{T}}{\Psi;\Delta \vdash \mathsf{c} \Longleftrightarrow \mathsf{c}_1 : \mathbb{T}}$$

$$\frac{\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \mathbb{T}}{\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \mathbb{T}} \qquad \frac{\Psi;\Delta, \alpha{:}\kappa_1 \vdash \mathsf{c}_1\, \alpha \Longleftrightarrow \mathsf{c}_2\, \alpha : \kappa_2}{\Psi;\Delta \vdash \mathsf{c}_1 \Longleftrightarrow \mathsf{c}_2 : \kappa_1 \to \kappa_2}$$

$\boxed{\Psi;\Delta \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa}$

$$\frac{\Delta(\alpha) = \kappa}{\Psi;\Delta \vdash \alpha \longleftrightarrow \alpha : \kappa} \qquad \frac{\Psi;\Delta \vdash \mathsf{c}_{11} \longleftrightarrow \mathsf{c}_{12} : \kappa_1 \to \kappa_2 \quad \Psi;\Delta \vdash \mathsf{c}_{21} \Longleftrightarrow \mathsf{c}_{22} : \kappa_1}{\Psi;\Delta \vdash \mathsf{c}_{11}\, \mathsf{c}_{12} \longleftrightarrow \mathsf{c}_{21}\, \mathsf{c}_{22} : \kappa_2}$$

$$\frac{}{\Psi;\Delta \vdash \mathsf{Unit} \longleftrightarrow \mathsf{Unit} : \mathbb{T}} \qquad \frac{\Psi;\Delta \vdash \tau_{11} \Longleftrightarrow \tau_{21} : \mathbb{T} \quad \Psi;\Delta \vdash \tau_{12} \Longleftrightarrow \tau_{22} : \mathbb{T}}{\Psi;\Delta \vdash \tau_{11} \to \tau_{12} \longleftrightarrow \tau_{21} \to \tau_{22} : \mathbb{T}}$$

$$\frac{\Psi;\Delta \vdash \tau_{11} \Longleftrightarrow \tau_{21} : \mathbb{T} \quad \Psi;\Delta \vdash \tau_{12} \Longleftrightarrow \tau_{22} : \mathbb{T}}{\Psi;\Delta \vdash \tau_{11} \times \tau_{12} \longleftrightarrow \tau_{21} \times \tau_{22} : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}^- \quad \Psi, \mathsf{u}{:}\mathsf{A}_1;\Delta \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{u}{:}\mathsf{A}_1.\tau_1 \longleftrightarrow \Pi \mathsf{u}{:}\mathsf{A}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type}^- \quad \Psi, \mathsf{u}{:}\mathsf{A}_1;\Delta \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{u}{:}\mathsf{A}_1.\tau_1 \longleftrightarrow \Sigma \mathsf{u}{:}\mathsf{A}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \quad \Psi, \mathsf{b}{:}\mathsf{K}_1;\Delta \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{b}{:}\mathsf{K}_1.\tau_1 \longleftrightarrow \Pi \mathsf{b}{:}\mathsf{K}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\Psi;\cdot \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 : \mathsf{kind} \quad \Psi, \mathsf{b}{:}\mathsf{K}_1;\Delta \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Sigma \mathsf{b}{:}\mathsf{K}_1.\tau_1 \longleftrightarrow \Sigma \mathsf{b}{:}\mathsf{K}_2.\tau_2 : \mathbb{T}}$$

$$\frac{\begin{array}{c}\Sigma(\mathsf{D}) = \Pi w_1{:}\mathcal{Q}_1 \ldots \Pi w_n{:}\mathcal{Q}_n.\kappa \\ \forall 1 \le i \le n. \quad \left\{\begin{array}{ll} \Psi;\cdot \vdash \mathcal{P}_{1i} \equiv \mathcal{P}_{2i} : \mathsf{A}^- & \text{if } \mathcal{Q}_i = \mathsf{A} \\ \Psi;\cdot \vdash \mathcal{P}_{1i} \equiv \mathcal{P}_{2i} : \mathsf{K}^- & \text{if } \mathcal{Q}_i = \mathsf{K} \end{array}\right.\end{array}}{\Psi;\Delta \vdash \mathsf{D}(\mathcal{P}_{11} \ldots \mathcal{P}_{1n}) \longleftrightarrow \mathsf{D}(\mathcal{P}_{21} \ldots \mathcal{P}_{2n}) : \kappa}$$

$$\frac{\Psi;\Delta, \alpha{:}\kappa \vdash \tau_1 \Longleftrightarrow \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \forall(\alpha{:}\kappa).\tau_1 \longleftrightarrow \forall(\alpha{:}\kappa).\tau_2 : \mathbb{T}}$$

Notice that the algorithm relies on the presence of algorithms to compare objects, type families and kinds from the $\mathrm{LF}^{\Sigma,1+}$ language. These algorithms are given in our previous technical report.

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.

Lemma 4.13 (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 4.14 (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 4.15 (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.

□

4.5 Completeness

We will prove completeness of the algorithm with respect to the definitional equality judgment given previously by the method of Kripke logical relations. This method is the standard one for proving type or kind-directed equivalence algorithms complete [4]. The logical relation we use is indexed by the kind, and 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 in the previous section to the logical relation.

Lemma 4.16 (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 4.17 (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 4.18 (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 $\mathbb{T}$.

□

We now begin proving completeness by the method of logical relations. The abstraction case will require us to use closure of the logical relation under weak-head expansion.

Lemma 4.19 (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.

□

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 4.20 (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) $\kappa = \kappa_1 \rightarrow \kappa_2$

$\Psi;\Delta, \alpha{:}\kappa_1 \vdash \alpha \longleftrightarrow \alpha : \kappa_1$ — 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) $\kappa = \kappa_1 \to \kappa_2$

$\Delta \subseteq \Delta_1$ 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)
$\Psi;\Delta_1 \vdash \mathsf{c}_1 \longleftrightarrow \mathsf{c}_2 : \kappa_1 \to \kappa_2$ — 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** $[\![\kappa_1 \to \kappa_2]\!]$ — 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.

Lemma 4.21 (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: $$\frac{\Psi;\Delta \vdash \tau_{11} \equiv \tau_{21} : \mathbb{T} \quad \Psi;\Delta \vdash \tau_{12} \equiv \tau_{22} : \mathbb{T}}{\Psi;\Delta \vdash \tau_{11} \to \tau_{12} \equiv \tau_{21} \to \tau_{22} : \mathbb{T}}$$

$\Psi;\Delta \vdash [\sigma_1]\tau_{11}$ **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
$\Psi;\Delta \vdash [\sigma_1](\tau_{11} \to \tau_{12})$ **is** $[\sigma_2](\tau_{21} \to \tau_{22})$ **in** $[\![\mathbb{T}]\!]$ — By rule

Case 2: $$\frac{\Psi;\cdot \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 : \mathsf{type} \quad \Psi, \mathsf{u}{:}\mathsf{A}_1;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta \vdash \Pi \mathsf{u}{:}\mathsf{A}_1.\tau_1 \equiv \Pi \mathsf{u}{:}\mathsf{A}_2.\tau_2 : \mathbb{T}}$$

$\Psi, \mathsf{u}{:}\mathsf{A}_1;\Delta \vdash [\sigma_1]\tau_1 \equiv [\sigma_2]\tau_2 : \mathbb{T}$ — By induction
$\Psi;\Delta \vdash [\sigma_1](\Pi \mathsf{u}{:}\mathsf{A}_1.contp_1) \equiv [\sigma_2](\Pi \mathsf{u}{:}\mathsf{A}_2.\tau_2) : \mathbb{T}$ — 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: $$\frac{\Psi;\Delta \vdash \mathsf{c}_2 \equiv \mathsf{c}_1 : \kappa}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa}$$

$\Psi;\Delta \vdash \sigma_2$ **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: $$\frac{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_2 : \kappa \quad \Psi;\Delta \vdash \mathsf{c}_2 \equiv \mathsf{c}_3 : \kappa}{\Psi;\Delta \vdash \mathsf{c}_1 \equiv \mathsf{c}_3 : \kappa}$$

$\Psi;\Delta \vdash \sigma_1$ **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 fact that an identity substitution is related to itself.

Lemma 4.22 (Identity Substitution Related) $\Psi;\Delta \vdash \mathsf{id}_\Delta$ **is** id_Δ **in** $[\![\Delta]\!]$.

Proof

By induction on the construction of the context.

□

Lemma 4.23 *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 4.21 and 4.22.

□

Theorem 4.24 (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 4.20.

□

4.6 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.

Lemma 4.25 (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 4.26 (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.

□

4.7 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 4.27 *If* $\vdash \Psi$, $\Psi \vdash \Delta$ *then it is not the case that* $\Psi;\Delta \vdash \mathsf{Unit} \equiv \tau_1 \rightarrow \tau_2 : \mathbb{T}$.

Proof

Since Unit and $\tau_1 \rightarrow \tau_2$ are not algorithmically equal at kind $\mathbb{T}$, by soundness and completeness of the algorithmic equality, they are not definitionally equal either.

□

Lemma 4.28 *If* $\vdash \Psi$, $\Psi \vdash \Delta$ *then it is not the case that* $\Psi;\Delta \vdash \tau_1 \times \tau_2 \equiv \tau_1' \rightarrow \tau_2' : \mathbb{T}$.

Proof

As for the previous lemma.

□

5 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.

5.1 Abstract Syntax

The syntax of the term level is given next. This is explicitly typed, since it is intended to be an internal language. We do not treat issues of type inference in this paper.

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
	$\mid$	$\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e$	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
Signatures	$\Sigma ::=$	$\ldots$	As before
	$\mid$	$\Sigma, \mathsf{C}{:}\forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_m{:}\kappa_m).$ $(\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \tau_2)$	
Contexts	$\Gamma ::=$	$\cdot$	Nil
	$\mid$	$\Gamma, x{:}\tau$	Cons

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.

Term Substitutions	$\sigma ::=$	$\cdot$	empty
	$\mid$	$\sigma, e/x$	cons with term

We write id_Γ for the identity on the context Γ, and also define $e[\sigma]$.

5.2 Static Semantics

The static semantics at the term level are defined by the new judgment forms.

$\Psi;\Delta \vdash \Gamma$	Γ is a valid context
$\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau$	e is well-typed at τ
$\Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$	ms takes $\mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m$ to τ

Notice that terms are typed under constraints postulating equality of closed $\mathrm{LF}^{\Sigma,1+}$ terms.

$\boxed{\vdash \Sigma}$

$$\frac{\begin{array}{c} \vdash \Sigma : \mathsf{ok} \\ \forall 1 \le i \le n. \left\{ \begin{array}{ll} w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathsf{A} : \mathsf{type} & \text{if } \mathcal{Q}_i = \mathsf{A} \\ w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathsf{K} : \mathsf{kind} & \text{if } \mathcal{Q}_i = \mathsf{K} \end{array} \right. \\ w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n;\cdot \vdash \tau : \mathbb{T} \\ \\ \vdash_S \mathsf{D} : \Pi w'_1{:}\mathcal{Q}'_1\ldots\Pi w'_m{:}\mathcal{Q}'_m.\kappa \\ \kappa = \kappa_1 \to \ldots \kappa_p \to \mathbb{T} \\ \forall 1 \le j \le m. w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n;\cdot \vdash \mathcal{P}_j : \mathcal{Q}'_j \end{array}}{\vdash \Sigma, \mathsf{C}{:}\forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).(\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n.\tau \to \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_m)\, \alpha_1 \ldots \alpha_p)}$$

$\boxed{\Psi;\Delta \vdash \Gamma}$

Empty

$$\frac{}{\Psi;\Delta \vdash \cdot}$$

Cons

$$\frac{\Psi;\Delta \vdash \Gamma \quad \Psi;\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta \vdash \Gamma, x{:}\tau}$$

$\boxed{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau}$

Contradiction

$$\frac{\Psi \vdash \mathcal{C} \Rightarrow \mathsf{false} \qquad \Psi;\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau}$$

Type Conversion

$$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_2 \quad \Psi;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_1}$$

Variables

$$\frac{\Gamma(x) = \tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash x : \tau}$$

Constants

$$\frac{}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{unit} : \mathsf{Unit}}$$

Functions

$$\frac{\Psi;\Delta \vdash \tau_1 \rightarrow \tau_2 : \mathbb{T} \quad \Psi;\Delta;\Gamma, f{:}\tau_1 \rightarrow \tau_2, x{:}\tau_1;\mathcal{C} \vdash e : \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e : \tau_1 \rightarrow \tau_2}$$

Applications

$$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash e_2 : \tau_1}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1\, e_2 : \tau_2}$$

Pairs

$$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \tau_1 \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash e_2 : \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

Projections

$$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_1 \times \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \pi_i\, e : \tau_i}$$

Constructor Abstractions

$$\frac{\Psi;\Delta, \alpha{:}\kappa;\Gamma;\mathcal{C} \vdash e : \tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \Lambda\alpha{:}\kappa.e : \forall(\alpha{:}\kappa).\tau}$$

Constructor Applications

$$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \forall(\alpha{:}\kappa).\tau \quad \Psi;\Delta \vdash \mathsf{c} : \kappa}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e\ [\mathsf{c}] : [\mathsf{c}/\alpha]\,\tau}$$

Function taking LF object

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

Function taking LF family

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

Application to LF term

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

Package with LF term

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

Unpack form

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

Datatype Constructors

$$\frac{\begin{array}{c}\Sigma(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n. \\ \qquad\qquad \tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p \\ \forall 1 \le i \le n. \quad \Psi, w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathcal{P}_{1i} : \mathcal{Q}_i \\ \Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \left[\begin{array}{c}\mathsf{c}_1, \ldots, \mathsf{c}_p, \\ \mathcal{P}_{11}, \ldots, \mathcal{P}_{1n}\end{array} / \begin{array}{c}\alpha_1, \ldots, \alpha_p, \\ w_1, \ldots, w_n\end{array}\right]\tau_1\end{array}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{C}\left[\begin{array}{c}\mathsf{c}_1\ldots\mathsf{c}_p, \\ \mathcal{P}_{11}\ldots\mathcal{P}_{1n}\end{array}, e\right] : \mathsf{D}\left(\begin{array}{l}[\mathcal{P}_{11}, \ldots, \mathcal{P}_{1n}/w_1, \ldots, w_n]\,\mathcal{P}_{21}\ldots \\ [\mathcal{P}_{11}, \ldots, \mathcal{P}_{1n}/w_1, \ldots, w_n]\,\mathcal{P}_{2m} \\ \mathsf{c}_1\ldots\mathsf{c}_p\end{array}\right)}$$

Case

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \quad \Psi;\Delta \vdash \tau : \mathbb{T} \\ \Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \to \tau\end{array}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{case}^{\tau}\ e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau}$$

For analyzing case expressions, we get more information while type checking each branch. This is because datatype constructors target a specified indexed set of the datatype. Thus, we get to assume equality of various $\mathrm{LF}^{\Sigma,1+}$ terms. We formalize this notion in the form of constraints already discussed before. The constraint solving judgment sketched out previously is used to simplify the constraints. Notice also that we can statically check that a branch is unreachable if we can notice that it leads to contradictory assumptions.

$\boxed{\Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \to \tau}$

$$\frac{}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \cdot : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_n)\,\mathsf{c}_1\ldots\mathsf{c}_m \to \tau}$$

$$\frac{\begin{array}{c}\Sigma(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p \\ \Sigma(\mathsf{D}) = \Pi w_1'{:}\mathcal{Q}_1'\ldots\Pi w_m'{:}\mathcal{Q}_m'.\mathbb{T} \\ \tau_1' = [\mathsf{c}_1\ldots\mathsf{c}_p/\alpha_1\ldots\alpha_p]\,\tau \\ \Psi, w_1{:}\mathcal{Q}_1, \ldots, w_n{:}\mathcal{Q}_n;\Delta;\Gamma, x{:}\tau_1';\mathcal{P}_{21} \doteq \mathcal{P}_1 : \mathcal{Q}_1' \wedge \ldots \wedge \mathcal{P}_{2m} \doteq \mathcal{P}_m : \mathcal{Q}_m' \wedge \mathcal{C} \vdash e : \tau \\ \Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_m)\,\mathsf{c}_1\ldots\mathsf{c}_p \to \tau\end{array}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{C}\,[w_1\ldots w_n, x] \Rightarrow e | ms : \mathsf{D}(\mathcal{P}_1\ldots\mathcal{P}_m)\,\mathsf{c}_1\ldots\mathsf{c}_p \to \tau}$$

Well Typed Substitutions The notation for typing substitutions is defined in a familiar way.

Definition 5.1 *The judgment* $\Psi;\Delta;\Gamma_2;\mathcal{C} \vdash \sigma : \Gamma_1$ *holds iff* $\forall x \in \mathsf{Dom}(\Gamma_1).\Psi;\Delta;\Gamma_2;\mathcal{C} \vdash \sigma(x) : \sigma(\Gamma_1(x))$.

5.3 Structural Properties

We will now prove some simple structural properties of the static semantics.

Lemma 5.2 (Weakening)

1. *If* $\Psi;\Delta \vdash \Gamma$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau$ *and* $\Gamma \subseteq \Gamma_1$, *then* $\Psi;\Delta;\Gamma_1;\mathcal{C} \vdash e : \tau$.

2. *If* $\Psi;\Delta \vdash \Gamma$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau$ *and* $\Delta \subseteq \Delta_1$, *then* $\Psi;\Delta_1;\Gamma;\mathcal{C} \vdash e : \tau$.

3. *If* $\Psi;\Delta \vdash \Gamma$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau$ *and* $\Psi \subseteq \Psi_1$, *then* $\Psi_1;\Delta;\Gamma;\mathcal{C} \vdash e : \tau$.

4. *If* $\Psi;\Delta \vdash \Gamma$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$ *and* $\Gamma \subseteq \Gamma_1$, *then* $\Psi;\Delta;\Gamma_1;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$.

5. *If* $\Psi;\Delta \vdash \Gamma$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$ *and* $\Delta \subseteq \Delta_1$, *then* $\Psi;\Delta_1;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$.

6. *If* $\Psi;\Delta \vdash \Gamma$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$ *and* $\Psi \subseteq \Psi_1$, *then* $\Psi_1;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$.

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 have show that the identity substitution is always well-typed, and that we can extend substitutions with variable for variable substitutions.

Lemma 5.3 (Identity Substitution) *If* $\Psi;\Delta \vdash \Gamma$ *then* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{id}_\Gamma : \Gamma$.

Proof

By an easy induction on the construction of the context. □

Lemma 5.4 (Extending Substitutions) *If* $\Psi;\Delta;\Gamma_1;\mathcal{C} \vdash \sigma : \Gamma$, $\Psi;\Delta \vdash \tau : \mathbb{T}$ *and* $x \notin \mathsf{Dom}(\Gamma_1) \cup \mathsf{Dom}(\Gamma)$ *then* $\Psi;\Delta;\Gamma_1, x{:}\tau;\mathcal{C} \vdash \sigma, x/x : \Gamma, x{:}\tau$.

Proof

Directly, by definition of typing substitutions and weakening. □

Lemma 5.5 (Substitution) *In the following,* $\mathcal{J} \in \{\vdash e : \tau, \vdash ms : \mathsf{D}(\mathcal{P}) \to \tau\}$.

1. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathcal{J}$ *and* $\Psi_1 \vdash \rho : \Psi$ *then* $\Psi_1;[\rho]\Delta;[\rho]\Gamma;[\rho]\mathcal{C} \vdash [\rho]\mathcal{J}$.

2. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathcal{J}$ *and* $\Psi;\Delta_1 \vdash \sigma : \Delta$ *then* $\Psi;\Delta_1;[\sigma]\Gamma;\mathcal{C} \vdash [\sigma]\mathcal{J}$.

3. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathcal{J}$ *and* $\Psi;\Delta;\Gamma_1;\mathcal{C} \vdash \sigma : \Gamma$ *then* $\Psi;\Delta;\Gamma_1;\mathcal{C} \vdash [\sigma] judgment$.

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. □

With the substitution property in hand, we can prove that types are preserved under constraint solving.

Lemma 5.6 (Constraint Solving) *Assume* $\vdash \Psi$, $\Psi \vdash \Delta$, $\Psi;\Delta \vdash \Gamma$, *there is some* A_i *such that* $\Psi;\Gamma \vdash \mathsf{M}_{i1} : \mathsf{A}_i$ *and* $\Psi;\Gamma \vdash \mathsf{M}_{i2} : \mathsf{A}_i$ *for all* $\mathsf{M}_{i1} =^? \mathsf{M}_{i2}$ *appearing in* $\mathcal{C}$, *and similarly there is some* K_i *for each pair of* A_{i1} *and* A_{i2}. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau$ *and* $\Psi \vdash \mathcal{C} \Rightarrow (\mathsf{true}, \rho, \Psi_1)$, *then* $\Psi_1;[\rho]\Delta;[\rho]\Gamma;\mathsf{true} \vdash [\rho]e : [\rho]\tau$.

Proof

By soundness of constraint solving, $\Psi_1;[\rho]\Gamma \vdash [\rho]\mathsf{M}_{i1} \equiv [\rho]\mathsf{M}_{i2} : [\rho]\mathsf{A}_i$ and correspondingly for the type families. We now apply structural induction on the typing judgment, using substitution properties. □

Lemma 5.7 (Regularity) *If* $\vdash \Psi$, $\Psi \vdash \Delta$, $\Psi;\Delta \vdash \Gamma$ *and* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau$ *then* $\Psi;\Delta \vdash \tau : \mathbb{T}$.

Proof

By a structural induction on the typing derivation. □

Lemma 5.8 (Typing Inversion) *Assume* $\vdash \Psi$, $\Psi \vdash \Delta$, $\Psi;\Delta \vdash \Gamma$ *all hold and* $\Psi \vdash \mathcal{C} \Rightarrow \mathsf{false}$ *does not hold.*

1. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{unit} : \tau$ *then* $\Psi;\Delta \vdash \tau \equiv \mathsf{Unit} : \mathbb{T}$.
2. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e : \tau$ *then* $\Psi;\Delta \vdash \tau_1 \to \tau_2 : \mathbb{T}$, $\Psi;\Delta;\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1;\mathcal{C} \vdash e : \tau_2$ *and* $\Psi;\Delta \vdash \tau \equiv \tau_1 \to \tau_2 : \mathbb{T}$.
3. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1\, e_2 : \tau$ *then* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \tau_1 \to \tau_2$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_2 : \tau_1$ *and* $\Psi;\Delta \vdash \tau \equiv \tau_2 : \mathbb{T}$.
4. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \langle e_1, e_2\rangle : \tau$ *then* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \tau_1$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_2 : \tau_2$, *and* $\Psi;\Delta \vdash \tau \equiv \tau_1 \times \tau_2 : \mathbb{T}$.
5. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \pi_i\, e : \tau$ *then* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_1 \times \tau_2$ *and* $\Psi;\Delta \vdash \tau \equiv \tau_i : \mathbb{T}$.
6. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \Lambda\alpha{:}\kappa.e : \tau$ *then* $\Psi;\Delta, \alpha{:}\kappa;\Gamma;\mathcal{C} \vdash e : \tau_1$ *and* $\Psi;\Delta \vdash \tau \equiv \forall(\alpha{:}\kappa).\tau_1 : \mathbb{T}$.
7. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e\ [\mathsf{c}] : \tau$ *then* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \forall(\alpha{:}\kappa).\tau_1$, $\Psi;\Delta \vdash \mathsf{c} : \kappa$ *and* $\Psi;\Delta \vdash \tau \equiv [\mathsf{c}/\alpha]\,\tau_1 : \mathbb{T}$.
8. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e : \tau_1$ *then* $\Psi \vdash \mathcal{Q} : \mathsf{sort}$, $\Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \kappa$, $\Psi, w{:}\mathcal{Q};\Delta;\Gamma, f{:}(\Pi w{:}\mathcal{Q}.\tau);\mathcal{C} \vdash e : \tau$ *and* $\Psi;\Delta \vdash \tau_1 \equiv \Pi w{:}\mathcal{Q}.\tau : \mathbb{T}$.
9. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e\ [\mathcal{P}] : \tau$ *then* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \Pi w{:}\mathcal{Q}.\tau_1$, $\Psi \vdash \mathcal{P} : \mathcal{Q}$ *and* $\Psi;\Delta \vdash \tau \equiv [\mathcal{P}/w]\,\tau_1 : \mathbb{T}$.
10. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{pack}\,\langle \mathcal{P}, e\rangle : \tau$ *then* $\Psi \vdash \mathcal{P} : \mathcal{Q}$, $\Psi, w{:}\mathcal{Q};\Delta \vdash \tau_1 : \mathbb{T}$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : [\mathcal{P}/w]\,\tau_1$ *and* $\Psi;\Delta \vdash \tau \equiv \Sigma w{:}\mathcal{Q}.\tau_1 : \mathbb{T}$.
11. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{let\,pack}\,\langle w,x\rangle = e_1\, \mathsf{in}\, e_2\, \mathsf{end} : \tau$ *then* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \Sigma w{:}\mathcal{Q}.\tau_1$, $\Psi, w{:}\mathcal{Q};\Delta;\Gamma, x{:}\tau_1;\mathcal{C} \vdash e_2 : \tau_2$ *and* $\Psi;\Delta \vdash \tau \equiv \tau_2 : \mathbb{T}$.
12. *If* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{C}\,[\mathsf{c}_1 \ldots \mathsf{c}_p\, \mathcal{P}_1 \ldots \mathcal{P}_n, e] : \tau$ *then* $\Sigma(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \mathsf{D}(\mathcal{P}'_1 \ldots \mathcal{P}'_m)\,\alpha_1 \ldots \alpha_p$, *for all* $1 \leq i \leq n$, $\Psi, w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathcal{P}_i : \mathcal{Q}_i$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : [\mathsf{c}_1 \ldots \mathsf{c}_p, \mathcal{P}_1 \ldots \mathcal{P}_n/\alpha_1 \ldots \alpha_p, w_1 \ldots w_n]\,\tau$ *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* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{case}^{\tau}\, e_1\, \mathsf{of}\, ms\, \mathsf{end} : \tau_1$ *then* $\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\,\mathsf{c}_1 \ldots \mathsf{c}_m$, $\Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\,\mathsf{c}_1 \ldots \mathsf{c}_m \to \tau$ *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. □

5.4 Canonical Forms

A subset of terms are judged to be values. This purely syntactic notion is defined by the grammar below.

$$
\begin{array}{lrll}
\text{Values} & v ::= & \mathsf{unit} & \text{Unit} \\
& | & \mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e & \text{Functions} \\
& | & \langle v_1, v_2 \rangle & \text{Pairs} \\
& | & \Lambda\alpha{:}\kappa.e & \text{Constructor Abstraction} \\
& | & \mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e & \text{Recursive functions taking index arguments} \\
& | & \mathsf{pack}\,\langle \mathcal{P},v \rangle & \text{Package of Index and Expression} \\
& | & \mathsf{C}\,[\mathcal{P}_1 \ldots \mathcal{P}_n, v] & \text{Datatype Constructors}
\end{array}
$$

Lemma 5.9 (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 unique weak-head normal forms for type (constructors). □

6 Dynamic Semantics and Type Safety

The evaluation relation $e_1 \mapsto e_2$ is defined by the following rules:

$$\frac{}{(\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e)\, v \mapsto [\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e, v/f, x]\, e} \qquad \frac{e_1 \mapsto e_2}{e_1\, e \mapsto e_2\, e} \qquad \frac{e_1 \mapsto e_2}{v\, e_1 \mapsto v\, e_2}$$

$$\frac{}{(\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e)\ [\mathcal{P}] \mapsto [\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e, \mathcal{P}/f, w]\, e} \qquad \frac{e_1 \mapsto e_2}{e_1\ [\mathcal{P}] \mapsto e_2\, \mathcal{P}}$$

$$\frac{}{(\Lambda\alpha{:}\kappa.e)\ [\mathsf{c}] \mapsto [\mathsf{c}/\alpha]\, e} \qquad \frac{e_1 \mapsto e_2}{e_1\ [\mathsf{c}] \mapsto e_2\ [\mathsf{c}]} \qquad \frac{e_1 \mapsto e_2}{\langle e_1, e \rangle \mapsto \langle e_2, e \rangle} \qquad \frac{e_1 \mapsto e_2}{\langle v, e_1 \rangle \mapsto \langle v, e_2 \rangle}$$

$$\frac{}{\pi_i\, \langle v_1, v_2 \rangle \mapsto v_i} \qquad \frac{e_1 \mapsto e_2}{\pi_i\, e_1 \mapsto \pi_i\, e_2}$$

$$\frac{e_1 \mapsto e_2}{\mathsf{pack}\,\langle \mathcal{P},e_1 \rangle \mapsto \mathsf{pack}\,\langle \mathcal{P},e_2 \rangle} \qquad \frac{}{\mathsf{let\ pack}\,\langle w,x \rangle = \mathsf{pack}\,\langle \mathcal{P},v \rangle\ \mathsf{in}\ e\ \mathsf{end} \mapsto [\mathcal{P}, v/w, x]\, e}$$

$$\frac{e_1 \mapsto e_2}{\mathsf{let\ pack}\,\langle w,x \rangle = e_1\ \mathsf{in}\ e\ \mathsf{end} \mapsto \mathsf{let\ pack}\,\langle w,x \rangle = e_2\ \mathsf{in}\ e\ \mathsf{end}}$$

$$\frac{e_1 \mapsto e_2}{\mathsf{C}\,[\mathsf{c}_1 \ldots \mathsf{c}_m \mathcal{P}_1 \ldots \mathcal{P}_n, e_1] \mapsto \mathsf{C}\,[\mathsf{c}_1 \ldots \mathsf{c}_m \mathcal{P}_1 \ldots \mathcal{P}_n, e_2]}$$

$$\frac{}{\mathsf{case}^{\tau}\ \mathsf{C}\,[\mathsf{c}_1 \ldots \mathsf{c}_m \mathcal{P}_1 \ldots \mathcal{P}_n, v]\ \mathsf{of}\ \ldots |\mathsf{C}\,[w_1 \ldots w_n, x] \Rightarrow e| \ldots \mathsf{end} \mapsto [\mathcal{P}_1 \ldots \mathcal{P}_n, v/w_1 \ldots w_n, x]\, e}$$

$$\frac{e_1 \mapsto e_2}{\mathsf{case}^{\tau}\ e_1\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{case}^{\tau}\ e_2\ \mathsf{of}\ ms\ \mathsf{end}}$$

6.1 Progress

Theorem 6.1 (Progress) *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash e : \tau$ *then either* e *is a value, or there exists a* e_1 *such that* $e \mapsto e_1$.

Proof

By structural induction on the typing derivation.

Case 1: $$\frac{\Psi \vdash \mathcal{C} \Rightarrow \mathsf{false} \qquad \Psi;\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau}$$

Impossible, since $\Psi \vdash \mathsf{true} \Rightarrow \mathsf{false}$ does not hold.

Case 2: $$\frac{\Gamma(x) = \tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash x : \tau}$$

Impossible, since $\Gamma = \cdot$.

Case 3: $$\frac{}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{unit} : \mathsf{Unit}}$$

Directly, since unit is a value.

Case 4: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_2 \quad \Psi;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_1}$$

e is a value, or evaluates to some e_1 By inductive hypotheses.

Case 5: $$\frac{\Psi;\Delta \vdash \tau_1 \to \tau_2 : \mathbb{T} \quad \Psi;\Delta;\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1;\mathcal{C} \vdash e : \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2}$$

Directly, since $\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e$ is a value.

Case 6: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \tau_1 \to \tau_2 \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash e_2 : \tau_1}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1\, e_2 : \tau_2}$$

e_1 is a value, or evaluates to e_1' — By induction

Subcase 6.1: 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 evaluates to e_2' — By induction

Subcase 6.1.1: 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 6.1.2: $e_2 \mapsto e_2'$

$v_1\, e_2 \mapsto v_1\, e_2'$ — By rule

Subcase 6.2: $e_1 \mapsto e_1'$

$e_1\, e_2 \mapsto e_1'\, e_2$ — By rule

Case 7: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \tau_1 \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash e_2 : \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

e_1 is either a value, or evaluates to some e_1'. — By induction

Subcase 7.1: e_1 is a value v_1

e_2 is a value, or evaluates to some e_2'. — By induction

Subcase 7.1.1: e_2 is a value v_2

$\langle v_1, v_2 \rangle$ is a value

Subcase 7.1.2: $e_2 \mapsto e_2'$

$\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle$ — By rule

Subcase 7.2: $e_1 \mapsto e_1'$

$\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle$ — By rule

Case 8: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_1 \times \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \pi_i\, e : \tau_i}$$

e is a value, or evaluates to some e'. — By induction

Subcase 8.1: e is a value v.

$e = \langle v_1, v_2 \rangle$ — By canonical forms

$\pi_i\, \langle v_1, v_2 \rangle \mapsto v_i$ — By rule

Subcase 8.2: $e \mapsto e'$

$\pi_i\, e \mapsto \pi_i\, e'$ — By rule

Case 9: $$\frac{\Psi;\Delta, \alpha{:}\kappa;\Gamma;\mathcal{C} \vdash e : \tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \Lambda\alpha{:}\kappa.e : \forall(\alpha{:}\kappa).\tau}$$

Directly, since $\Lambda\alpha{:}\kappa.e$ is a value.

Case 10: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \forall(\alpha{:}\kappa).\tau \quad \Psi;\Delta \vdash \mathsf{c} : \kappa}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e\ [\mathsf{c}] : [\mathsf{c}/\alpha]\, \tau}$$

e is a value, or evaluates to some e'. — By induction

Subcase 10.1: e is a value v

$e = \Lambda\alpha{:}\kappa.e_1$ — By canonical forms

$(\Lambda\alpha{:}\kappa.e_1)\ [\mathsf{c}] \mapsto [\mathsf{c}/\alpha]\, e_1$ — By rule

Subcase 10.2: $e \mapsto e'$

$e\ [\mathsf{c}] \mapsto e'\ [\mathsf{c}]$ — By rule

Case 11: $$\frac{\Psi \vdash \mathcal{Q} : \mathsf{sort} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi, w{:}\mathcal{Q};\Delta;\Gamma, f{:}(\Pi w{:}\mathcal{Q}.\tau);\mathcal{C} \vdash e : \mathsf{c}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e : \Pi w{:}\mathcal{Q}.\mathsf{c}}$$

Directly, since $\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e$ is a value.

Case 12: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \Pi w{:}\mathcal{Q}.\tau \quad \Psi \vdash \Delta : \mathcal{P}\mathcal{Q}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e\ [\mathcal{P}] : [\mathcal{P}/w]\,\tau}$$

e is a value, or evaluates to some e' — By induction

Subcase 12.1: 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 12.2: $e \mapsto e'$

$e\ [\mathcal{P}] \mapsto e'\ [\mathcal{P}]$ — By rule

Case 13: $$\frac{\Psi \vdash \mathcal{P} : \mathcal{Q} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash e : [\mathcal{P}/w]\,\tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{pack}\,\langle \mathcal{P}, e\rangle : \Sigma w{:}\mathcal{Q}.\tau}$$

e is a value, or evaluates to some e' — By induction

Subcase 13.1: e is a value v

$\mathsf{pack}\,\langle \mathcal{P}, v\rangle$ is a value

Subcase 13.2: $e \mapsto e'$

$\mathsf{pack}\,\langle \mathcal{P}, e\rangle \mapsto \mathsf{pack}\,\langle \mathcal{P}, e'\rangle$ — By rule

Case 14: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \Sigma w{:}\mathcal{Q}.\tau_1 \quad \Psi, w{:}\mathcal{Q};\Delta;\Gamma, x{:}\tau_1;\mathcal{C} \vdash e_2 : \tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{let\ pack}\,\langle w, x\rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : \tau}$$

e_1 is a value, or evaluates to some e_1' — By induction

Subcase 14.1: 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 14.2: $e_1 \mapsto e_1'$

$\mathsf{let\ pack}\,\langle w, x\rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \mapsto \mathsf{let\ pack}\,\langle w, x\rangle = e_1'\ \mathsf{in}\ e_2\ \mathsf{end}$ — By rule

Case 15: $$\frac{\begin{array}{c} \Sigma(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n. \\ \tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p \\ \forall 1 \le i \le n. \quad \Psi, w_1{:}\mathcal{Q}_1, \ldots, w_{i-1}{:}\mathcal{Q}_{i-1};\cdot \vdash \mathcal{P}_{1i} : \mathcal{Q}_i \\ \Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \left[\begin{array}{c} \mathsf{c}_1, \ldots, \mathsf{c}_p, \\ \mathcal{P}_{11}, \ldots, \mathcal{P}_{1n} \end{array} / \begin{array}{c} \alpha_1, \ldots, \alpha_p, \\ w_1, \ldots, w_n \end{array}\right]\tau_1 \end{array}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{C}\left[\begin{array}{c} \mathsf{c}_1\ldots\mathsf{c}_p, \\ \mathcal{P}_{11}\ldots\mathcal{P}_{1n} \end{array}, e\right] : \mathsf{D}(\begin{array}{l} [\mathcal{P}_{11}, \ldots, \mathcal{P}_{1n}/w_1, \ldots, w_n]\,\mathcal{P}_{21}\ldots \\ [\mathcal{P}_{11}, \ldots, \mathcal{P}_{1n}/w_1, \ldots, w_n]\,\mathcal{P}_{2m} \\ \mathsf{c}_1\ldots\mathsf{c}_p \end{array})}$$

e is a value, or evaluates to some e' — By induction

Subcase 15.1: e is a value v

$\mathsf{C}\,[\mathsf{c}_1\ldots\mathsf{c}_p\mathcal{P}_{11}\ldots\mathcal{P}_{1n}, v]$ is a value

Subcase 15.2: $e \mapsto e'$

$\mathsf{C}\,[\mathcal{P}_1, e] \mapsto \mathsf{C}\,[\mathsf{c}_1\ldots\mathsf{c}_p\mathcal{P}_{11}\ldots\mathcal{P}_{1n}, e']$ — By rule

Case 16:
$$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \quad \Psi;\Delta \vdash \tau : \mathbb{T} \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash ms : \mathsf{D}(\mathcal{P}_1 \ldots \mathcal{P}_n)\, \mathsf{c}_1 \ldots \mathsf{c}_m \to \tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{case}^\tau\ e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau}$$

e_1 is a value, or evaluates to some e_1'

Subcase 16.1: 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 16.2: $e_1 \mapsto e_1'$

$\mathsf{case}^\tau\ e_1\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{case}^\tau\ e_1'\ \mathsf{of}\ ms\ \mathsf{end}$ By rule

□

6.2 Preservation

Theorem 6.2 (Preservation) *If* $\cdot;\cdot;\cdot;\mathsf{true} \vdash e_1 : \tau$ *and* $e_1 \mapsto e_2$ *then* $\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau$.

Proof

By induction on the structure of the typing judgment.

Case 1:
$$\frac{\Psi \vdash \mathcal{C} \Rightarrow \mathsf{false} \qquad \Psi;\Delta \vdash \tau : \mathbb{T}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau}$$

Impossible, since $\cdot \vdash \mathsf{true} \Rightarrow \mathsf{false}$ does not hold.

Case 2:
$$\frac{\Gamma(x) = \tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash x : \tau}$$

Impossible, since $\Gamma = \cdot$

Case 3:
$$\frac{}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{unit} : \mathsf{Unit}}$$

Impossible, since no evaluation rule applies

Case 4:
$$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_2 \quad \Psi;\Delta \vdash \tau_1 \equiv \tau_2 : \mathbb{T}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_1}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau_2$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau_2$ By type conversion rule

Case 5:
$$\frac{\Psi;\Delta \vdash \tau_1 \to \tau_2 : \mathbb{T} \quad \Psi;\Delta;\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1;\mathcal{C} \vdash e : \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{fun}\ f(x{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2}$$

Impossible, since no evaluation rule applies

Case 6: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash e_2 : \tau_1}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1\, e_2 : \tau_2}$$

We case analyze based on the evaluation rule applied

Subcase 6.1: $$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_1' : \tau_1 \rightarrow \tau_2$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_1'\, e_2 : \tau_2$ By rule

Subcase 6.2: $$\frac{e_1 \mapsto e_2}{v\, e_1 \mapsto v\, e_2}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau_2$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash v\, e_2 : \tau_2$ By rule

Subcase 6.3: $$\frac{}{(\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e)\, v \mapsto [\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e, v/f, x]\, e}$$

$\cdot;\cdot;f{:}\tau_1 \rightarrow \tau_2, x{:}\tau_1;\mathsf{true} \vdash e : \tau_2$ By inversion

$\cdot;\cdot;\cdot;\mathsf{true} \vdash [\mathsf{fun}\, f(x{:}\tau_1){:}\tau_2.e, v/f, x]\, e : \tau_2$ By substitution

Case 7: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \tau_1 \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash e_2 : \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}$$

We case analyze based on the evaluation rule applied

Subcase 7.1: $$\frac{e_1 \mapsto e_2}{\langle e_1, e\rangle \mapsto \langle e_2, e\rangle}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau_1$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \langle e_2, e\rangle : \tau_1 \times \tau_2$ By rule

Subcase 7.2: $$\frac{e_1 \mapsto e_2}{\langle v, e_1\rangle \mapsto \langle v, e_2\rangle}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau_2$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \langle v, e_2\rangle : \tau_1 \times \tau_2$ By rule

Case 8: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \tau_1 \times \tau_2}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \pi_i\, e : \tau_i}$$

We case analyze based on the evaluation rule applied

Subcase 8.1: $$\frac{}{\pi_i\, \langle v_1, v_2\rangle \mapsto v_i}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash v_i : \tau_i$ By inversion

Subcase 8.2: $$\frac{e_1 \mapsto e_2}{\pi_i\, e_1 \mapsto \pi_i\, e_2}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \tau_1 \times \tau_2$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \pi_i\, e_2 : \tau_i$ By rule

Case 9: $$\frac{\Psi;\Delta, \alpha{:}\kappa;\Gamma;\mathcal{C} \vdash e : \tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \Lambda\alpha{:}\kappa.e : \forall(\alpha{:}\kappa).\tau}$$

Impossible, since no evaluation rule applies

Case 10: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \forall(\alpha{:}\kappa).\tau \quad \Psi;\Delta \vdash \mathsf{c} : \kappa}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e\ [\mathsf{c}] : [\mathsf{c}/\alpha]\,\tau}$$

We case analyze based on the evaluation rule applied

Subcase 10.1: $$\frac{}{(\Lambda\alpha{:}\kappa.e)\ [\mathsf{c}] \mapsto [\mathsf{c}/\alpha]\,e}$$

$\cdot;\alpha{:}\kappa;\cdot;\mathsf{true} \vdash e : \tau$ By inversion

$\cdot;\cdot;\cdot;\mathsf{true} \vdash [\mathsf{c}/\alpha]\,e : [\mathsf{c}/\alpha]\,\tau$ By substitution

Subcase 10.2: $$\frac{e_1 \mapsto e_2}{e_1\ [\mathsf{c}] \mapsto e_2\ [\mathsf{c}]}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \forall(\alpha{:}\kappa).\tau$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2\ [\mathsf{c}] : [\mathsf{c}/\alpha]\,\tau$ By rule

Case 11: $$\frac{\Psi \vdash \mathcal{Q} : \mathsf{sort} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi, w{:}\mathcal{Q};\Delta;\Gamma, f{:}(\Pi w{:}\mathcal{Q}.\tau);\mathcal{C} \vdash e : \mathsf{c}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e : \Pi w{:}\mathcal{Q}.\mathsf{c}}$$

Impossible, since no evaluation rule applies

Case 12: $$\frac{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e : \Pi w{:}\mathcal{Q}.\tau \quad \Psi;\cdot \vdash \mathcal{P} : \mathcal{Q}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash e\ [\mathcal{P}] : [\mathcal{P}/w]\,\tau}$$

We case analyze based on the evaluation rule applied

Subcase 12.1: $$\frac{}{(\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e)\ [\mathcal{P}] \mapsto [\mathsf{Fun}\, f(w{:}\mathcal{Q}){:}\tau.e, \mathcal{P}/f, w]\,e}$$

$w{:}\mathcal{Q};\cdot;f{:}(\Pi w{:}\mathcal{Q}.\tau);\mathsf{true} \vdash e : \tau$ 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 12.2: $$\frac{e_1 \mapsto e_2}{e_1\ [\mathcal{P}] \mapsto e_2\,\mathcal{P}}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \Pi w{:}\mathcal{Q}.\tau$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2\ [\mathcal{P}] : [\mathcal{P}/w]\,\tau$ By rule

Case 13: $$\frac{\Psi;\cdot \vdash \mathcal{P} : \mathcal{Q} \quad \Psi, w{:}\mathcal{Q};\Delta \vdash \tau : \mathbb{T} \quad \Psi;\Delta;\Gamma;\mathcal{C} \vdash e : [\mathcal{P}/w]\,\tau}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{pack}\,\langle \mathcal{P}, e\rangle : \Sigma w{:}\mathcal{Q}.\tau}$$

The only evaluation rule that can apply is $$\frac{e_1 \mapsto e_2}{\mathsf{pack}\,\langle \mathcal{P}, e_1\rangle \mapsto \mathsf{pack}\,\langle \mathcal{P}, e_2\rangle}$$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : [\mathcal{P}/w]\,\tau$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{pack}\,\langle \mathcal{P}, e\rangle : \Sigma w{:}\mathcal{Q}.\tau$ By rule

Case 14: $$\frac{\begin{array}{c}\Psi;\Delta \vdash \tau : \mathbb{T} \\ \Psi;\Delta;\Gamma;\mathcal{C} \vdash e_1 : \Sigma w{:}\mathcal{Q}.\tau_1 \quad \Psi, w{:}\mathcal{Q};\Delta;\Gamma, x{:}\tau_1;\mathcal{C} \vdash e_2 : \tau\end{array}}{\Psi;\Delta;\Gamma;\mathcal{C} \vdash \mathsf{let\ pack}\,\langle w, x\rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : \tau}$$

We case analyze based on the evaluation rule applied

Subcase 14.1: $$\frac{}{\mathsf{let\ pack}\,\langle w, x\rangle = \mathsf{pack}\,\langle \mathcal{P}, v\rangle\ \mathsf{in}\ e\ \mathsf{end} \mapsto [\mathcal{P}, v/w, x]\,e}$$

$\cdot;\cdot \vdash \mathcal{P} : \mathcal{Q}$,

$\cdot;\cdot;\cdot;\mathsf{true} \vdash v : [\mathcal{P}/w]\,\tau_1$ 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 14.2: $\dfrac{e_1 \mapsto e_2}{\mathsf{let\ pack}\ \langle w,x\rangle = e_1\ \mathsf{in}\ e\ \mathsf{end} \mapsto \mathsf{let\ pack}\ \langle w,x\rangle = e_2\ \mathsf{in}\ e\ \mathsf{end}}$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \Sigma w{:}\mathcal{Q}.\tau_1$ 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

Case 15:

$$\dfrac{\begin{array}{c}\Sigma(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n.\\ \tau_1 \to \mathsf{D}(\mathcal{P}_{21}\ldots\mathcal{P}_{2m})\,\alpha_1\ldots\alpha_p\\ \forall 1\le i\le n.\quad \Psi, w_1{:}\mathcal{Q}_1,\ldots,w_{i-1}{:}\mathcal{Q}_{i-1};\cdot\vdash \mathcal{P}_{1i} : \mathcal{Q}_i\\ \Psi;\Delta;\Gamma;\mathcal{C}\vdash e : \left[\begin{array}{c}\mathsf{c}_1,\ldots,\mathsf{c}_p,\\ \mathcal{P}_{11},\ldots,\mathcal{P}_{1n}\end{array} \Big/ \begin{array}{c}\alpha_1,\ldots,\alpha_p,\\ w_1,\ldots,w_n\end{array}\right]\tau_1\end{array}}{\Psi;\Delta;\Gamma;\mathcal{C}\vdash \mathsf{C}\left[\begin{array}{c}\mathsf{c}_1\ldots\mathsf{c}_p,\\ \mathcal{P}_{11}\ldots\mathcal{P}_{1n}\end{array}, e\right] : \mathsf{D}\left(\begin{array}{l}[\mathcal{P}_{11},\ldots,\mathcal{P}_{1n}/w_1,\ldots,w_n]\,\mathcal{P}_{21}\ldots\\ [\mathcal{P}_{11},\ldots,\mathcal{P}_{1n}/w_1,\ldots,w_n]\,\mathcal{P}_{2m}\\ \mathsf{c}_1\ldots\mathsf{c}_p\end{array}\right)}$$

The only evaluation rule that can apply is $\dfrac{e_1 \mapsto e_2}{\mathsf{C}\,[\mathsf{c}_1\ldots\mathsf{c}_m\mathcal{P}_1\ldots\mathcal{P}_n, e_1] \mapsto \mathsf{C}\,[\mathsf{c}_1\ldots\mathsf{c}_m\mathcal{P}_1\ldots\mathcal{P}_n, e_2]}$

$\cdot;\cdot;\cdot;\mathsf{true} \vdash e_2 : \left[\begin{array}{c}\mathsf{c}_1,\ldots,\mathsf{c}_p,\\ \mathcal{P}_{11},\ldots,\mathcal{P}_{1n}\end{array} \Big/ \begin{array}{c}\alpha_1,\ldots,\alpha_p,\\ w_1,\ldots,w_n\end{array}\right]\tau_1$ By induction

$\cdot;\cdot;\cdot;\mathsf{true} \vdash \mathsf{C}\left[\begin{array}{c}\mathsf{c}_1\ldots\mathsf{c}_p,\\ \mathcal{P}_{11}\ldots\mathcal{P}_{1n}\end{array}, e_2\right] : \mathsf{D}\left(\begin{array}{l}[\mathcal{P}_{11},\ldots,\mathcal{P}_{1n}/w_1,\ldots,w_n]\,\mathcal{P}_{21}\ldots\\ [\mathcal{P}_{11},\ldots,\mathcal{P}_{1n}/w_1,\ldots,w_n]\,\mathcal{P}_{2m}\\ \mathsf{c}_1\ldots\mathsf{c}_p\end{array}\right)$ By rule

Case 16:

$$\dfrac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{C}\vdash e_1 : \mathsf{D}(\mathcal{P}''_1\ldots\mathcal{P}''_m)\,\mathsf{c}''_1\ldots\mathsf{c}''_p \quad \Psi;\Delta\vdash\tau:\mathbb{T}\\ \Psi;\Delta;\Gamma;\mathcal{C}\vdash ms : \mathsf{D}(\mathcal{P}'_1\ldots\mathcal{P}'_n)\,\mathsf{c}'_1\ldots\mathsf{c}'_m \to \tau\end{array}}{\Psi;\Delta;\Gamma;\mathcal{C}\vdash \mathsf{case}^\tau\ e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau}$$

We case analyze based on the evaluation rule applied

Subcase 16.1: $\dfrac{}{\mathsf{case}^\tau\ \mathsf{C}\,[\mathsf{c}_1\ldots\mathsf{c}_m\mathcal{P}_1\ldots\mathcal{P}_n, v]\ \mathsf{of}\ \ldots|\mathsf{C}\,[w_1\ldots w_n, x] \Rightarrow e|\ldots\mathsf{end} \mapsto [\mathcal{P}_1\ldots\mathcal{P}_n, v/w_1\ldots w_n, x]\,e}$

$\Sigma(\mathsf{C}) = \forall(\alpha_1{:}\kappa_1)\ldots\forall(\alpha_p{:}\kappa_p).\Pi w_1{:}\mathcal{Q}_1\ldots\Pi w_n{:}\mathcal{Q}_n.\tau_1 \to \mathsf{D}(\mathcal{P}'_1\ldots\mathcal{P}'_m)\,\alpha_1\ldots\alpha_p$,
for all $1\le i\le n$, $\Psi, w_1{:}\mathcal{Q}_1,\ldots,w_{i-1}{:}\mathcal{Q}_{i-1};\cdot\vdash\mathcal{P}_i : \mathcal{Q}_i$,
for all $1\le i\le m$, $\Psi;\alpha_1{:}\kappa_1,\ldots,\alpha_{i-1}{:}\kappa_{i-1}\vdash\mathsf{c}_i : \kappa_i$,
$\Psi;\Delta;\Gamma;\mathcal{C}\vdash v : [\mathsf{c}_1\ldots\mathsf{c}_p, \mathcal{P}_1\ldots\mathcal{P}_n/\alpha_1\ldots\alpha_p, w_1\ldots w_n]\,\tau_1$ and
$\Psi;\Delta\vdash \mathsf{D}(\mathcal{P}''_1,\ldots,\mathcal{P}''_m)\,\mathsf{c}'_1\ldots\mathsf{c}'_p \equiv \mathsf{D}([\mathcal{P}_1,\ldots,\mathcal{P}_n/w_1,\ldots,w_n]\,\mathcal{P}'_1\ldots[\mathcal{P}_1,\ldots,\mathcal{P}_n/w_1,\ldots,w_n]\,\mathcal{P}'_m)\,\mathsf{c}_1\ldots\mathsf{c}_p : \mathbb{T}$
By inversion
$\cdot;\cdot\vdash\mathcal{P}''_i \equiv [\mathcal{P}_1,\ldots,\mathcal{P}_n/w_1,\ldots,w_n]\,\mathcal{P}'_i : \mathcal{Q}_i$ By inversion on equality of constructors
$w_1{:}\mathcal{Q}_1,\ldots,w_n{:}\mathcal{Q}_n;\cdot;x{:}[\mathsf{c}_1\ldots\mathsf{c}_n/\alpha_i\ldots\mathsf{c}_n]\,\tau_1;\mathcal{P}''_1 =^? [\mathcal{P}_1,\ldots,\mathcal{P}_n/w_1,\ldots,w_n]\,\mathcal{P}'_1\wedge\ldots\vdash e : \tau$
By inversion on the match rule
$\cdot;\cdot;\cdot;\mathsf{true}\vdash[\mathcal{P}_1\ldots\mathcal{P}_n, v/w_1\ldots w_n, x]\,e : \tau$ By substitution and constraint solving

Subcase 16.2: $\dfrac{e_1 \mapsto e_2}{\mathsf{case}^\tau\ e_1\ \mathsf{of}\ ms\ \mathsf{end} \mapsto \mathsf{case}^\tau\ e_2\ \mathsf{of}\ ms\ \mathsf{end}}$

$\cdot;\cdot;\cdot;\mathsf{true}\vdash e_2 : \mathsf{D}(\mathcal{P}''_1\ldots\mathcal{P}''_m)\,\mathsf{c}'_1\ldots\mathsf{c}'_p$ By induction
$\cdot;\cdot;\cdot;\mathsf{true}\vdash\mathsf{case}^\tau\ e_2\ \mathsf{of}\ ms\ \mathsf{end} : \tau$ By rule

□

7 An Extended Example: Type Checking Simply Typed Lambda

We will now illustrate the system given before by an extended example. We wish to show that a type checker for the simply typed lambda calculus can be given such that its correctness can be checked in the static type system already presented. We work with the explicitly typed variant of the calculus for simplicity, which means that abstractions are annotated by their domain types.

The simply typed lambda calculus is the language defined below.

$$\begin{array}{lll} \text{Types} & T ::= & \mathsf{UnitType} \mid T_1 \to T_2 \\ \text{Terms} & e ::= & \mathsf{UnitTerm} \mid x \mid \lambda x{:}T.e \mid e_1\, e_2 \end{array}$$

The type system is defined by the following rules.

$$\frac{}{\Gamma \vdash \mathsf{UnitTerm} : \mathsf{UnitType}} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \lambda x{:}T_1.e : T_1 \to T_2}$$

$$\frac{\Gamma \vdash e_1 : T_{11} \to T_2 \quad \Gamma \vdash e_2 : T_{12} \quad T_{11} \equiv T_{12}}{\Gamma \vdash e_1\, e_2 : T_2}$$

$$\frac{}{\mathsf{UnitType} \equiv \mathsf{UnitType}} \qquad \frac{T_{11} \equiv T_{21} \qquad T_{12} \equiv T_{22}}{T_{11} \to T_{12} \equiv T_{21} \to T_{22}}$$

In the following, we will use some concrete syntax for purposes of explanation. In every case, it should be easy to see how to convert these to the official syntax given before. We use the abbreviation $\mathsf{P} \to \mathsf{Q}$ for the non-dependent product type $\Pi\_ : \mathsf{P}.\ \mathsf{Q}$. Similarly, we will use $\mathsf{P} \times \mathsf{Q}$ for the non-dependent sum type $\Sigma\_ : \mathsf{P}.\ \mathsf{Q}$. We will elide some typing annotations that can be easily inferred from the context, replacing them by $\bullet$. We will use ML-like pattern matching notation to define functions. We also assume that we have option datatype, with the constructors SOME and NONE.

7.1 LF definitions

The first order of business is to represent the simply typed lambda calculus. This is done by adding to the $\mathrm{LF}^{\Sigma,1+}$ signature a series of constants that represent the types and terms of the language.

The tp family represents the types of the calculus, with unitType being the base type and arrow being the arrow constructor.

```
tp         :  type.

unitType   :  tp.
arrow      :  tp → tp → tp.
```

The exp family represents the terms of the calculus, with unitTerm representing the term constant, lam representing the lambda abstraction (with a rigid type), and app representing the application.

```
exp        :  type.

unitTerm   :  exp.
lam        :  tp → (exp → exp) → exp.
app        :  exp → exp → exp.
```

The important fact about the representation is that the representation is adequate. This is stated formally by the so-called Adequacy Theorem.

Theorem 7.1 (Adequacy for Syntax)

1. *There is a compositional bijection between the types of the simply typed lambda calculus and canonical objects of the LF$^{\Sigma,1+}$ type* `tp` *in the empty context and the signature above.*

2. *For every context of the simply typed lambda calculus X, we can produce a LF$^{\Sigma,1+}$ context Γ_X such that there is a compositional bijection between terms a term e with free variables in Γ and canonical objects of the LF$^{\Sigma,1+}$ type* `tm` *in the context Γ_X and the signature above.*

Proof Sketch

We can produce the required bijections by induction on the structure. We will denote the bijection in the forward direction by $\ulcorner \_ \urcorner$.

$\ulcorner \mathsf{UnitType} \urcorner = \mathsf{unitType}$

$\ulcorner T_1 \rightarrow T_2 \urcorner = \mathsf{arrow} \ulcorner T_1 \urcorner \ulcorner T_2 \urcorner$.

Similarly we can produce a bijection for terms. This requires us to transform a context X to a LF$^{\Sigma,1+}$ context Γ_X, in which we assume variables of type exp for every variable of the simply typed lambda calculus. □

Now we have to represent the static semantics of the calculus. This is done again by adding constants to the signature. The family eqtp represents structural equality between two types, and of represents typing a exp at a type tp. We do not have any implicit types in this formalism (such as in Twelf) for concreteness, and all parameters are explicit.

```
eqtp        : tp → tp → type.

eqtp_unit   : eqtp unitType unitType.
eqtp_arrow  : Π tp11:tp. Π tp12:tp. Π tp21:tp. Π tp22:tp.
              eqtp tp11 tp12
              → eqtp tp12 tp22
              → eqtp (arrow tp11 tp12) (arrow tp21 tp22).

of       : exp → tp → type.

of_unit  : of unitTerm unitType.
of_app   : Π tp1:tp. Π tp1′:tp. Π tp2:tp. Π e1:exp. Π e2:exp.
           eqtp tp1 tp1′
           → of e2 tp1′
           → of e1 (arrow tp1 tp2)
           → of (app e1 e2) tp2.
of_lam   : Π tp1:tp. Π tp2:tp. Π e:exp → exp.
           (Π x:exp. of x tp1
                 → of (e x) tp2)
           → of (lam tp1 e) (arrow tp1 tp2).
```

Theorem 7.2 (Adequacy for Semantics)

1. *We can produce a bijection between derivations of equivalence of types $t1$ and $t2$ and canonical objects of type* `eqtp` *$\ulcorner t1 \urcorner \ulcorner t2 \urcorner$ in the empty context.*

2. *For all contexts of the simply typed lambda calculus X we can produce a LF$^{\Sigma,1+}$ context Γ_X such that there is a bijection between derivations $X \vdash e : T$ and canonical objects of type* `of` *$\ulcorner e \urcorner \ulcorner t \urcorner$ in the context Γ_X.*

The importance of the adequacy theorems is that we now know that the representation is adequate. Thus, if we find a canonical object of the required type, the corresponding judgment must be derivable within the simply-typed lambda calculus type system.

7.2 Datatype and Constructor definitions

Now we present datatypes and constructor definitions for the core language. The Tp datatype represents the types of the calculus. This is indexed by a term of the LF type tp, representing the type in LF.

$$\mathsf{Tp} \quad : \quad \mathsf{tp} \to \mathbb{T}$$

$$\begin{array}{lll} \mathsf{UnitType} & : & \mathsf{Tp}(\mathsf{unitType}) \\ \mathsf{Arrow} & : & \Pi\,\mathsf{tp1{:}tp}.\ \Pi\,\mathsf{tp2{:}tp}.\ \mathsf{Tp}\ (\mathsf{tp1}) \to \mathsf{Tp}\ (\mathsf{tp2}) \\ & & \to \mathsf{Tp}\ (\mathsf{arrow}\ (\mathsf{tp1}[\cdot])\ (\mathsf{tp2}[\cdot])) \end{array}$$

The $\mathsf{Context}$ datatype represents the context used for typechecking. It is indexed by a representation of the context terms (and derivations) live in. We represent the context as a product (within $\mathrm{LF}^{\Sigma,1+}$) of "regular blocks" which are composed of a term assumption and a typing assumption. The empty context is represented by the unit type.

$$\mathsf{Context} \quad : \quad \mathsf{type} \to \mathbb{T}$$

$$\begin{array}{lll} \mathsf{Nil} & : & \mathsf{Context}\ (1) \\ \mathsf{Cons} & : & \Pi\,\mathsf{c{:}type}.\ \Pi\,\mathsf{t{:}tp}.\ \mathsf{Context}\ (\mathsf{c}) \to \mathsf{Tp}\ (\mathsf{t}) \\ & & \to \mathsf{Context}\ (\mathsf{c}[\cdot] \times \Sigma\,\mathsf{e{:}exp}.\ \mathsf{of}\ \ \mathsf{e}\ \ (\mathsf{t}[\cdot])) \end{array}$$

The Exp datatype represents terms of the calculus. It is indexed by a context and a $\mathrm{LF}^{\Sigma,1+}$ abstraction which produces a exp in that context.

$$\mathsf{Exp} \quad : \quad \Pi\,\mathsf{c{:}type}.\ (\mathsf{c}[\cdot] \to \mathsf{exp}) \to \mathbb{T}$$

The simplest case is for representing unit, which is easily represented in $\mathrm{LF}^{\Sigma,1+}$ by a closed term.

$$\mathsf{UnitTerm} \quad : \quad \Pi\,\mathsf{c{:}type}.\ \mathsf{Unit} \to \mathsf{Exp}\ \ (\mathsf{c}, (\lambda\_{:}\mathsf{c}.\ \mathsf{unitTerm}))$$

Applications take two pieces which belong in the same context.

$$\begin{array}{lll} \mathsf{App} & : & \Pi\,\mathsf{c{:}type}.\ \Pi\,\mathsf{e1{:}}(\mathsf{c}[\cdot] \to \mathsf{exp}).\ \Pi\,\mathsf{e2{:}}(\mathsf{c}[\cdot] \to \mathsf{exp}). \\ & & \quad \mathsf{Exp}(\mathsf{c},\mathsf{e1}) \times \mathsf{Exp}(\mathsf{c},\mathsf{e2}) \to \mathsf{Exp}(\mathsf{c}, (\lambda\gamma{:}\mathsf{c}[\cdot].\ \mathsf{app}\ \ (\mathsf{e1}\ \gamma)\ \ (\mathsf{e2}\ \gamma))) \end{array}$$

For abstractions, the definition is slightly complicated. Since lam is expressed using higher-order abstract syntax, the argument depends on an extended context. Further, a term-level representation of the simple type of the domain must be an argument, in the form of a member of the Tp datatype.

$$\begin{array}{lll} \mathsf{Lam} & : & \Pi\,\mathsf{c{:}type}.\ \Pi\,\mathsf{t{:}tp}.\ \Pi\,\mathsf{efunc{:}}(\mathsf{c}[\cdot] \times (\mathsf{exp} \to \mathsf{exp})). \\ & & \quad \mathsf{Exp}(\ (\mathsf{c}[\cdot] \times \Sigma\,\mathsf{e{:}exp}.\ \mathsf{of}\ \ \mathsf{e}\ \ \mathsf{t}[\cdot]), (\lambda\gamma{:}\bullet.\ \mathsf{efunc}\,\langle \pi_1\gamma, \pi_{12}\gamma\rangle)\,)) \times \mathsf{Tp}((\mathsf{t})) \\ & & \quad \to \mathsf{Exp}\ \ (\mathsf{c}, (\lambda\gamma{:}\mathsf{c}[\cdot].\ \mathsf{lam}\ \ (\mathsf{t}[\cdot])\ \ (\lambda\,\mathsf{e{:}exp}.\ \mathsf{efunc}\ \ \langle\gamma,\mathsf{e}\rangle))) \end{array}$$

The last case is for variables. We represent variables in deBruijn notation, with zero the variable bound closest in the context, and successor to discard the top of the context. We defer to a new datatype Index which indexes the context.

$$\mathsf{Var} \quad : \quad \Pi\,\mathsf{c{:}type}.\ \Pi\,\mathsf{e{:}}(\mathsf{c}[\cdot] \to \mathsf{exp}).\ \mathsf{Index}\ \ (\mathsf{c},\mathsf{e}) \to \mathsf{Exp}\ \ (\mathsf{c},\mathsf{e})$$

The Index datatype indexes into the current context to pick out a term level variable from the context. We assume variables are represented in a deBruijn notation, so that the Z constructor pulls out the top term from the context, and the S constructor discards the top term and typing assumption and looks in the rest of the context. The context is a product of the blocks mentioned already.

Index	:	Π c:type. (c[$\cdot$] $\rightarrow$ exp) $\rightarrow \mathbb{T}$
Z	:	Π c:type. Π t:tp. Unit $\rightarrow$ Index ((c[$\cdot$] $\times\ \Sigma$ e:exp. of e t[$\cdot$]), $(\lambda\,\gamma{:}\bullet.\pi_{12}\gamma)$)
S	:	Π c:type. Π t:tp. Π e:(c[$\cdot$] $\rightarrow$ exp). Index (c, e) $\rightarrow$ Index ((c $\times\ \Sigma$ e:exp. of e t[$\cdot$]), $(\lambda\,\gamma{:}\bullet.$ e $(\pi_1\gamma)$))

7.3 Main Type Checking Function

We now begin to give the type checker. First, the type we want it to have is as follows:

```
typecheck : Πc:type. Πe:(c[·] -> exp).
                    Context (c) * Exp (c, e)
                              -> (Σt:tp. Σd:(Πγ:c[·]. of (e[·] γ) (t[·])). Tp [t]) option
```

This function takes a (static) LF representations of a context and a exp in that context, as well as (dynamic) term-language representations of the context and the term. It returns both a LF representation and a term-language representation of the type, but importantly, also returns a LF representation of the derivation.

For the unit case, we can come up with the required pieces immediately.

```
Fun  typecheck [c] [_] (_, (UnitTerm _ _)) =
       SOME (pack (unitType ,
                 pack ( λ_:c.of_unit ,
                      UnitTerm 1 unit)))
```

For the application case, we must make two recursive calls. If the term in function position does not have an arrow type, type checking fails. Otherwise, the domain type must match the type of the argument, which is checked by an auxiliary function checkEqTp.

```
...
 |  typecheck [c] [_] (context, (App ⟨c, 'e1, 'e2⟩,(e1, e2))) =
      case (typecheck [c] ['e1] (context, e1)) of
         SOME (pack ('tp12, d1, tp12)) =>
            case (typecheck [c] ['e2] (context, e2)) of
               SOME (pack ('tp2, d2, tp2)) =>
                  case tp12 of
                     Arrow ⟨'tp11,'tp12⟩ (tp11,tp12) =>
                        case (checkEqTp ['tp11] ['tp2] (tp11, tp2)) of
                           SOME (pack (d3, unit)) =>
                              SOME (pack ('tp12,
                                       pack (
                                          λγ:c[·].of_app 'tp11[·] 'tp2[·] 'tp12[·] ,
                                             ('e1[·] γ) ('e2[·] γ)
                                             d3[·] (d2[·] γ) (d1[·] γ)
                                       tp12)))
                         | NONE => NONE
                   | UnitType _ _ => error
             | NONE => NONE
       | NONE => NONE
```

For the abstraction case, we have to check the body in an extended context. We extend the context with a new exp assumption as well as an assumption that this variable has the specified type. We then check the body in the extended context, and package the results back. Notice that the returned derivation has to be in the ambient context, not the extended one.

```
...
 |  typecheck [c] [_] (context, Lam ⟨_,'tp1,'e⟩(tp1, e)) =
        case (typecheck [c[·] × Σe1:exp.of e1 'tp1[·]]
                                                    [λγ:•.'e[·] ⟨π₁γ,π₁₂γ⟩]
                                             ((Cons ⟨c,'tp1⟩ (context,tp1)), e)) of
           SOME (pack ('tp2,d1,tp2)) =>
                SOME (pack (arrow 'tp1[·] 'tp2[·] ,
                         pack (                                                ,
                             λc1:c[·]. of_lam 'tp1[·] 'tp2[·]
                                        (λe1:exp. 'e[·] (e1))
                                        (λe1:exp. λd2:(of e1 'tp1[·]).
                                                     d1[·]⟨c1,e1,d2⟩)
                         Arrow ⟨'tp1,'tp2⟩ (tp1, tp2) )))
         | NONE => NONE
```

For the variable case, we defer to an auxiliary function that crawls through the context.

```
...
 | typecheck [c] [e] (context, (Var _ ind)) = getTypeCtx [c] [e] (context, ind)
```

7.4 Auxiliary Functions

The getTypeCtx function takes a context and a index into that context, and returns a type with typing derivation of the corresponding variable.

```
getTypeCtx : Πc:type. Πe:(c[·] -> exp).
                 Context (c) * Index (c, e)
                       -> Σt:tp. Σd:(Πγ:c[·]. of (e[·] γ) (t[·])). Tp [t]
```

Because of the construction of the Index datatype, this function can never be passed an empty context. We have to case analyze the Context argument. In the Nil case, case analyzing the Index datatype leads to a contradiction, so any term in the body will be well-typed. We pick unit for concreteness.

```
Fun getTypeCtx [_] [_] (Nil , Z _) = unit
 |  getTypeCtx [_] [_] (Nil , (S _ _)) = unit
```

In the Cons case, we case analyze the Index argument to know whether we should return the typing assumption of the closest bound variable in the context or we have to pick through the rest of the context.

```
...
 |  getTypeCtx [_] [_] ((Cons ⟨c, t⟩ (context,tp)), (Z _)) =
         pack (t,
              pack (λγ:(c[·]  ×  Σe:exp.of e (t1[·]). π₂₂γ,
                   tp))
 |  getTypeCtx [_] [_] ((Cons ⟨c, t⟩ (context, tp)) , (S ⟨c1, t1, e⟩ index)) =
         let
            pack (t_out, d_out, tp_out) = getTypeCtx ⟨c,e⟩ (context, index)
         in
              pack (t_out,
                   pack (λγ:(c[·]  ×  Σe:exp.of e (t1[·]).d_out (π₁γ),
                        tp_out))
         end
```

For completeness, we now give the checkEqTp function, which performs the structural equality test on two types passed to it. If the equality test fails, it returns NONE. This is used in the application case of the main type checker.

```
checkEqTp : Πtp1:tp. Πtp2:tp.
                Tp (tp1) * Tp (tp2) -> (Σd:eqtp (tp1[·]) (tp2[·]. Unit) option

Fun checkEqTp [tp1] [tp2] ((UnitType _ _), (UnitType _ _)) =
     SOME (pack (eqtp_unit , unit))
 |  checkEqTp [tp1] [tp2] ((UnitType _ _), (Arrow _ _)) =
     NONE
 |  checkEqTp [tp1] [tp2] ((Arrow _ _), (UnitType _ _)) =
     NONE
 |  checkEqTp [tp1] [tp2] ((Arrow ⟨tp11, tp12⟩ (Tp11, Tp12)),
                              (Arrow ⟨tp21, tp22⟩ (Tp21, Tp22))) =
       case checkEqTp [tp11, tp21] (Tp11, Tp21) of
           SOME (pack (d1, _)) =>
               case checkEqTp [tp12, tp22] (Tp12, Tp22) of
                   SOME (pack (d2, _)) =>
                       SOME (pack (eqtp_arrow (t11[·]) (t12[·]) (t21[·]) (t22[·])
                                                          (d1[·]) (d2[·],
                     unit)))
                 | NONE => NONE
         | NONE => NONE
```

7.5 Typing Derivation (excerpt)

We now show that the type checker function has the required type. We will show the case for abstraction. That is, we are in the case

```
fun typecheck [c] ['e] (context, e) =
   case e of
     ...
     | Lam ⟨c','tp1,'ein⟩(tp1, ein)) =>
       case (typecheck [c[·] × Σe1:exp.of e1 'tp1[·]]
                                                   [λγ:•.'ein[·] ⟨π₁γ, π₁₂γ⟩]
                                    ((Cons ⟨c,'tp1⟩ (context,tp1)), ein)) of
           SOME (pack ('tp2,d1,tp2)) =>
                 SOME (pack (arrow 'tp1[·] 'tp2[·] ,
                             pack (
                                   λc1:c[·]. of_lam 'tp1[·] 'tp2[·]                   ,
                                               (λe1:exp. 'ein[·] (e1))
                                               (λe1:exp. λd2:(of e1 'tp1[·]).
                                                                d1[·]⟨c1,e1,d2⟩)
                             Arrow ⟨'tp1,'tp2⟩ (tp1, tp2) )))
         | NONE => NONE
```

We add to the metavariable context Ψ the declarations c:type and the declaration $\mathsf{e}:\mathsf{c}[\cdot] \to \mathsf{exp}$. To the term level context Γ we add the declarations context:Context(c) and e:Exp(c, e). We perform a case analysis on e. Since we are typechecking the Lam constructor, we get to assume c':type, $'$tp1:tp and $'$ein:(c[·] × (exp → exp)). Further, we have the new equations $\mathsf{c}' =^? \mathsf{c}$ and $\mathsf{e} =^? (\lambda\gamma{:}\mathsf{c}[\cdot].\ \mathsf{lam}\ ('\mathsf{tp1}[\cdot])\ (\lambda\,\mathsf{e{:}exp}.\ '\mathsf{ein}\ \langle\gamma, \mathsf{e}\rangle))$. Finally, we have the term-language assumptions ein:Exp($(\mathsf{c}[\cdot] \times \Sigma\,\mathsf{e{:}exp.\ of\ e}\ '\mathsf{tp1}[\cdot]), (\lambda\gamma{:}\bullet.\ '\mathsf{ein}\,\langle\pi_1\gamma, \pi_{12}\gamma\rangle)$) and tp1:Tp((t)).

We now have to argue that the recursive call is well-typed. From the assumption of the type of the typecheck function, we need to pass in an extended context for the $'$ein argument. The context representation in LF$^{\Sigma,1+}$ required is $(\mathsf{c}'[\cdot] \times \Sigma\,\mathsf{e{:}exp.\ of\ e}\ '\mathsf{tp1}[\cdot])$. This is what is passed, if $\mathsf{c}'[\cdot] =^? \mathsf{c}[\cdot]$ results in unifying the metavariables c' and c. Since this is in the pattern fragment, this will be discovered by the constraint algorithm. The LF$^{\Sigma,1+}$ representation of the term required is $(\lambda\gamma{:}\bullet.\ '\mathsf{ein}\,\langle\pi_1\gamma, \pi_{12}\gamma\rangle)$, exactly as specified.

The result of the recursive call is an option type. The less interesting case is for NONE, in which case the returned value NONE is well-typed. The more interesting case is when the recursive call succeeded. Then we unpack the results to get 'tp2:tp, d1:($\Pi\gamma$:(c'[$\cdot$] $\times$ Σ e:exp. of e 'tp1[$\cdot$]). of ('ein γ) 'tp2[$\cdot$]), and tp2:Tp('tp2).

Now we check the returned value in this case. The returned type is an arrow, with the $LF^{\Sigma,1+}$ representation and the term-level representation well-formed according to the rules for the constructors arrow and Arrow respectively. For the derivation argument, we need to show that all the arguments to of_lam are well-formed. Looking at the arguments actually passed in, we have to do some work in the derivation argument (the last argument to of_lam). In this case, we have to use the assumption on the argument d1 to know that its application is well-formed.

7.6 Correctness of the type checker

We now will prove the type checker partially correct, in that given a closed term, if it terminates in success, (returning SOME_), then there exists a typing derivation in the simply typed lambda calculus type system.

Recall that our type checker has the type

```
typecheck : Πc:type. Πe:(c[·] -> exp).
                    Context (c) * Exp (c, e)
                             -> (Σt:tp. Σd:(Πγ:c[·]. of (e[·] γ) (t[·])). Tp [t]) option
```

For a closed term p of the calculus, the adequacy theorem assures us that we have a corresponding LF representation prog. We also have a dynamic representation Prog belonging to the Exp datatype. Given this, we can apply this checker to the nil context (static representation 1 and dynamic representation Nil) and the above representations prog and Prog. Assume typecheck [1] [prog] (Nil, Prog) evaluates to some val. By subject reduction, val has the type (Σt:tp. Σd:($\Pi\gamma$:1. of (prog γ) (t[$\cdot$])). Tp [t]) option.

Now further assume that the program terminated with SOME package. Then package has the type Σt:tp. Σd:($\Pi\gamma$:1. of (prog γ) (t[$\cdot$])). Tp [t] By canonical forms, it must be of the form pack (t, (pack (d, tp))), with t belonging to the $LF^{\Sigma,1+}$ type tp and d belonging to the $LF^{\Sigma,1+}$ type $\Pi\gamma$:1.of (prog γ) (t[$\cdot$]). Canonical forms for $LF^{\Sigma,1+}$ now gives us that d must be of the form $\lambda\gamma$:1.of (prog γ) t[$\cdot$]. Applying this function to $\langle\rangle$ gives us a LF term of prog t[$\cdot$]. Notice that this is canonical. Applying adequacy again, we get that there must exist a typing derivation in the calculus.

8 Related Work

The idea of certified computation has attracted many approaches. The most closely related work is that of Appel and Felty [2]. The authors investigate the use of dependent types in proving programs correct at compile-time. The key insight is that given a strongly-typed and dependently-typed system, program properties can be analyzed before executing the code. They work in the LF language itself. They use the operational interpetation of LF signatures as a logic-programming language [20] to execute their programs. They program a simple theorem prover in this setting, and show that the tactics are correct. Our work builds on the idea, but differs in that we define a functional language to do our programming. Thus our programming language differs from our representation language (LF). This leads to a gain in expressivity, in that algorithms can be more naturally specified in a functional language.

Using dependent type theories as programming languages is not a new idea, of course. Our approach owes much to the work of Xi *et al.* [28, 27]. Xi's Dependent ML system (or DML) integrated a form of dependent types into a core-ML like language. The key insight was to respect the phase-distinction between compile-time objects and run-time objects, and make types dependent on a static index domain. This helps with effective type checking algorithms. The original system [27] had the objects indexing types be natural numbers with linear equalities being the equality theory. In principle, other index domains can be integrated into the system. Our work can be looked upon as the extension of this system with $LF^{\Sigma,1+}$ terms as the index domain. This has been extended by Dunfield and Pfenning [7, 8], who elides writing index domain constructs in the source code. This requires them to produce new type inference and type checking algorithm [9].

Particular work in dependently typed programming languages has been done in the theorem-proving community, in the NuPRL [3] project, the Coq system [26], and more recently the Epigram project [10].

Within the Coq system, Pauline-Mohring [19], and more recently Letouzey [13], have shown how to extract functional programs from proofs in Coq. This relies on a notion of erasing proof terms which are not computationally significant. In the Epigram system, based on Luo's Unified Type Theory [14], functional programs that can be verified statically are also being written, including a certified type checker for the simply typed lambda calculus. All these systems unify the proof representation metalogic and the computation language. This creates problems for checking programs. Either we have to restrict ourselves to a class of provably terminating programs, as in Coq or Epigram, or we have to accept non-termination of the checking algorithms, as in NuPRL. We separate the concerns of representation and computation, thus allowing a more expressive language to be used to define algorithms.

A system which also separates representation from computation is the ∇-calculus developed by Schürmann and others [24, 25]. This is a theory of a language that manipulates and reasons about logics represented in LF. The operational model is based on logic programming, with elements of functional programming. The semantics is essentially proof search, and may unify existential (instantiable or logic) variables. There is also a non-deterministic pattern match operator. In this system, the notion of higher-order abstract syntax is pursued further. Variables and contexts of the logic represented are internalized as the corresponding concepts of the language itself. However, since open terms have to be reasoned about, a new operator ∇ introduces dynamic assumptions in the course of the computation.

9 Conclusion and Future Work

We have developed a language for certified computation. We showed various important metatheoretic properties of the language, including type safety and canonical forms. We then demonstrated the use of the language by writing a type checker for a simple language, and proving it correct.

We plan to implement the language shortly. This will involve designing an external language, and the study of type reconstruction for easily inferrable types. We are encouraged by the practicality of the type reconstruction algorithm within the Twelf system for LF, which was developed by Pfenning [20].

More serious proposed extensions to the language is to deal with effects. Since we have separated the computation language and the proof representation language, it should be possible to integrate effects into the computation language without interfering with the logical properties of the representation language. We plan to investigate the use of reference types and control mechanisms such as exceptions. These features are useful in writing type checkers for realistic languages, as also for other kinds of programs.

References

[1] Andrew W. Appel. Foundational proof carrying code. In *Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] 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.

[3] 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.

[4] 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.

[5] N G de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, April 1991.

[6] Delphin. `http://cs-www.cs.yale.edu/homes/carsten/delphin/`, September 2004.

[7] Joshua Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University, School of Computer Science, September 2002.

[8] 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.

[9] 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.

[10] The Epigram project. `http://www.e-pig.org`, October 2005.

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[12] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, January 2005. Earlier version in CMU-CS-00-148.

[13] 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.

[14] Zhaohui Luo. *Computation and Reasoning : A Type Theory for Computer Science*. Oxford University Press, 1994.

[15] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[16] 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.

[17] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. Draft, in submission, September 2005.

[18] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.

[19] 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.

[20] 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.

[21] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[22] 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.

[23] Susmit Sarkar. The metatheory of LF extended with dependent pair and unit types. Technical Report CMU-CS-05-179, Carnegie Mellon University, School of Computer Science, 2005.

[24] Carsten Schürmann. Towards functional programming with logical frameworks. Unpublished, available at http://cs-www.cs.yale.edu/homes/carsten/delphin/, July 2003.

[25] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus : Functional programming with higher-order encodings. In *Typed Lambda Calculus and Applications*, 2005.

[26] The Coq Development Team. The Coq proof assistant reference manual. Coq release 8.0, 2004.

[27] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, 1998.

[28] 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.