HybridLF: A System For Reasoning In Higher-Order Abstract Syntax

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

by

Amy Elizabeth Furniss

Department of Computer Science

University of Leicester

March 2015

HybridLF: A System For Reasoning In Higher-Order Abstract Syntax

Amy Elizabeth Furniss

Abstract

In this thesis we describe two new systems for reasoning about deductive systems: HybridLF and Canonical HybridLF.

HybridLF brings together the Hybrid approach (due to Ambler, Crole and Momigliano [15]) to higher-order abstract syntax (HOAS) in Isabelle/HOL with the logical framework LF, a dependently-typed system for proving theorems about logical systems. Hybrid provides a version of HOAS in the form of the lambda calculus, in which Isabelle functions are automatically converted to a nameless de Bruijn represenation. Hybrid allows untyped expressions to be entered as human-readable functions, which are then translated into the machine-friendly de Bruijn form. HybridLF uses and updates these techniques for variable representation in the context of the dependent type theory LF, providing a version of HOAS in the form of LF.

Canonical HybridLF unites the variable representation techniques of Hybrid with Canonical LF, in which all terms are in canonical form and definitional equality is reduced to syntactic equality. We extend the Hybrid approach to HOAS to functions with multiple variables by introducing a family of abstraction functions, and use the Isabelle option type to denote errors instead of including an `ERR` element in the Canonical HybridLF expression type.

In both systems we employ the meta-logic M_2 to prove theorems about deductive systems. M_2 [28] is a first-order logic in which quantifiers range over the objects and types generated by an LF signature (that encodes a deductive system). As part of the implementation of M_2 we explore higher-order unification in LF, adapting existing approaches to work in our setting.

Contents

List of Figures

Typographic conventions

Font	Usage
`Teletype`	Isabelle code and constructors of Isabelle datatypes
Sans-serif	Function and relation names
Small capitals	Inductive definition, lemma and theorem labels and system names

Chapter 1

Introduction

Many of the structures of interest to computer scientists, such as logics and programming languages, are specified as *deductive systems*. Such systems consist of a set of *judgements* that may or may not hold and a set of *axioms* and *inference rules* which are used to create derivations that prove a judgement holds. As well as reasoning within the systems themselves, we often want to prove properties of the system itself. Such properties are referred to as *meta-theorems*.

Computer-verified formal proof dates back to the 1960s with the Automath project [2]. More recent systems such as Isabelle [3], Agda [35] and Coq [12] allow proofs to be created and checked interactively. As well as such general-purpose systems, used to formalise a broad range of mathematics, there are more specialised systems such as Twelf [25] that are designed for the specification of deductive systems and proofs of meta-theorems about them.

Twelf is based on the *logical framework* LF [22], which is designed to provide a system (called a *metalogic*) in which it is possible to specify and prove theorems about a wide range of deductive systems (called *object logics*). One of the key questions when specifying an object logic is how to represent variable binding. The customary approach in LF is to use a technique known as higher-order abstract syntax (HOAS), in which the variable binding of the object logic is carried out through variable binding in the metalogic. This allows the implementation of variable binding in the metalogic to be re-used for a variety of different deductive systems. However, this approach has a drawback, as the types involved in specifying abstractions in higher-order abstract syntax preclude the use of inductive definitions to define the object logic. This is due to positivity constraints: if we had an LF type $\mathtt{tm}$ representing the terms of the simply-typed lambda calculus, the type of the constant representing the lambda-abstraction binding operator would be $(\mathtt{tm} \to \mathtt{tm}) \to \mathtt{tm}$. The size of the function space $\mathtt{tm} \to \mathtt{tm}$ is strictly greater than the (countably infinite) size of `tm`, so this constructor could not be injective, which is not permitted in inductive definitions.

Another approach to higher-order abstract syntax has been the Hybrid project [15]. Hybrid is a package for the Isabelle theorem prover that provides for the user a form of higher-order abstract syntax, and converts HOAS expressions into *de Bruijn notation*, in which bound variables are given a numerical *index* linking them to the corresponding binder. Hybrid automatically converts a subset of Isabelle functions (called *abstractions*) representing higher-order abstract syntax expressions into the corresponding de Bruijn expression. This allows the object logic to be entered in a human-friendly higher order form, and then reasoned about in a machine-friendly de Bruijn form.

In this thesis we detail two new systems, HybridLF and Canonical HybridLF, that implement the logical framework LF and utilise the Hybrid approach to variable binding, combining the two approaches to explore the possibilities that this affords us.

In chapter 2 we detail the LF type theory, and the HybridLF system that implements it. In chapter 3 we discuss the canonical presentation of LF, and the Canonical HybridLF system that implements it. In chapter 4 we survey the mechanisms for proving meta-theorems in the literature, focusing on the Twelf implementation and the M_2 meta-logic, and detail the HybridLF implementation of M_2. In chapter 5 we discuss higher-order unification in LF, which is used in the implementation of M_2, and give an account of its implementation in HybridLF and Canonical HybridLF. In chapter 6 we give some examples of using HybridLF and Canonical HybridLF to create proofs. Finally, in chapter 7 we present our conclusions.

1.1 Logical Frameworks

Logical frameworks are logics designed for the purpose of specifying and reasoning about deductive systems. Reasoning in such frameworks happens at two (or more) levels: reasoning in the logic or deductive system that is to be studied (called the object logic), and reasoning in a metalogic, which is the logic of the logical framework itself.

One of the key goals is to formulate a metalogic within which it is easy to implement and prove properties of object logics. The use of a metalogic also allows tools (such as proof environments) to be written once for the metalogic and then used with a range of different object logics.

There are quite a number of different logical frameworks in the literature; two of the main approaches are type-theoretic (such as LF) and logic programming (such as λ-Prolog).

In the type-theoretic approach, the object logic is encoded as a signature in a higher-order type theory. In the case of LF[22] this is through the judgements-as-types principle, in which judgements of the object logic are represented by the type of their proofs in the LF metalogic. An LF type is created for each syntactic category of the object logic, with constants of appropriate types from which the elements of the syntax may be built. So for example if we are representing the natural numbers, there would be an LF type `nat`, representing the type of natural numbers, and constants `zero` and `succ`, with `zero` representing the number 0 and `succ` representing the successor of another natural number:

Example 1.

$$\begin{array}{rcl} \texttt{nat} & : & \texttt{Type} \\ \texttt{zero} & : & \texttt{nat} \\ \texttt{succ} & : & \texttt{nat} \rightarrow \texttt{nat} \end{array}$$

A family of types is associated with each judgement of the object logic, indexed by the LF type representing the syntactic category that the judgement relates to. So for example we might have a judgement that defines when a term is even like so:

Example 2.

$$\texttt{even} \; : \; \texttt{nat} \rightarrow \texttt{Type}$$

Axioms of the object logic are represented by constants, while inference rules are represented by function constants that take as arguments the parameters together with proofs of the premises of the inference rule:

Example 3.

$$\begin{array}{rcl} \texttt{even\_zero} & : & \texttt{even zero} \\ \texttt{even\_succ} & : & \Pi\texttt{n:nat.even n} \rightarrow \texttt{even (succ (succ n))} \end{array}$$

At the LF level proofs are represented by terms that have the type associated with the appropriate judgement of the object logic. So, for example, proofs of the `even` judgement for a particular natural number `n` will have type `even n`:

Example 4.

```
even_succ (succ (succ zero)) (even_succ zero (even_zero)) :
    even (succ (succ (succ (succ zero))))
```

As such, proof checking is reduced to type checking, which is decidable in LF.

In the logic programming approach, terms are represented by *hereditary Harrop formulas.* In λ-prolog these are higher-order hereditary Harrop formulas. In such an approach [13], types of the object logic are represented by primitive types in the logic programming language, judgements are represented by logic programming language types, while typed constants are introduced to represent the syntax of the object logic:

Example 5.

```
type nat.
type bool.

type even nat → bool.
type zero nat.
type succ nat → nat.
type proof bool → prop.
```

The logic programming language has a distinguished type `prop` of propositions. Axioms of the object logic are constants, while inference rules are represented by clauses in the metalogic:

Example 6.

```
type even_zero (proof (even zero)).
proof (even (succ (succ (N))):- proof (even N).
```

1.2 Representations of syntax with variable binding

There are a number of approaches to encoding syntax with variable binding in the literature.

The PoplMark challenge [1] has provided a benchmark for mechanised reasoning about syntax with variable binding, with submissions covering all of the approaches described in this section. There is no consensus among researchers about a single best way to represent such syntax; it is likely that the requirements of a given proof or situation will dictate the most suitable technique.

The *raw terms* representation (in which binders and variables are labelled with a variable name) is the easiest to define, but the hardest to reason with, as it does not equate α-equivalent terms and requires α-conversion to avoid the capture of free variables. The terms are closer to conventional mathematical presentations of syntax than those of the nameless approaches, but it is common to work with terms up to α-equivalence which is complicated when employing the raw terms approach.

The *locally named* representation avoids the need to rename variables to avoid capture, as the sets of free and bound variables are disjoint. However, implementations do not always equate α-equivalent terms.

The *de Bruijn* approach uses nameless *indices* or *levels*, producing identical representations of α-equivalent terms. It produces terms that are easier to reason about in software or a theorem prover than the raw terms approach or the locally named approach, but does not produce terms that are easily readable by humans.

The *locally nameless* approach combines the advantages of the locally named approach and the de Bruijn approach, using named free variables and nameless indices for bound variables. This method of representation equates terms that are α-equivalent, does not require α-conversion to avoid variable capture and is more human-readable than de Bruijn terms whilst still being suitable for use with mechanised or automated reasoning. Locally nameless terms are perhaps less readable than raw terms or locally named terms, as bound variables are given numerical indices rather than names, but the simplicity of reasoning about them outweighs this for many applications.

Nominal techniques allow reasoning about α-equivalence classes of terms with variable binding, employing notions of name swapping (or *permutation*). A key advantage of the nominal approach is that bound variables are named rather than given numerical indices, and as such terms are human-readable and closer to informal mathematical practice. Nominal techniques require more sophisticated mathematics than the other approaches to variable binding, but the Nominal Isabelle [10] package for the Isabelle theorem prover implements and automates much of this, providing nominal datatype and function constructs that allow the creation of datatypes with variable binding.

1.2.1 Raw terms

The simplest approach to implementing terms with variable binding is to use named variables. For example, the untyped lambda calculus might be represented like so:

Example 7.

$$\texttt{expr} ::= \texttt{App expr expr} \mid \texttt{Abs nat expr} \mid \texttt{Var nat}$$

Here we take variable names to be given by the natural numbers, and label abstractions with the number of the variable that they bind, while variables are represented by the `Var` constructor. While simple, this approach has disadvantages. One such disadvantage is the fact that α-equivalent terms can have different representations, so that `Abs 0 (Var 0)` $\neq$ `Abs 1 (Var 1)` even though both terms describe the (same) identity function. Another disadvantage is that free variable names and bound variable names are drawn from the same set, requiring that substitution perform α-conversion in some instances to ensure that it avoids capturing free variables.

1.2.2 Locally named representation

In the locally named representation the sets of free variable and bound variable names are disjoint: they are given by different syntactic classes. The locally named implementation of the untyped λ-calculus would be as follows:

Example 8.

$$\texttt{expr} ::= \texttt{App expr expr} \mid \texttt{Abs bname expr} \mid \texttt{Bvar bname} \mid \texttt{Fvar fname}$$

In this example, `bname` and `fname` are disjoint types of variable names, instances of `Bvar` represent a bound variable and instances of `Fvar` represent a free variable. In such a representation there is no need to rename bound variables to avoid capturing free variables, as the sets of names do not overlap. However, α-equivalent terms may still have different locally named representations.

Pollack, Sato and Ricciotti [5] describe the locally named representation, and give a *canonical* representation of lambda terms in which α-equivalent terms are identical. They call free variables *global* variables and bound variables *local* variables (hence the term 'locally named', as local variables have names), and define substitution (separately) on both global and local variables in a straightforward manner. They employ a *height* function $F\ x\ m$ that maps a global variable name x and a term m to a local variable name, and define a relation $\mathbb{L}_F$ parameterised by a height function F:

$$\frac{}{\texttt{Fvar x} : \mathbb{L}_F} \qquad \frac{\texttt{M} : \mathbb{L}_F \quad \texttt{N} : \mathbb{L}_F}{(\texttt{App M N}) : \mathbb{L}_F} \qquad \frac{M : \mathbb{L}_F \quad F\ x\ M = y}{(\texttt{Abs y [Bvar y/Fvar x]M}) : \mathbb{L}_F}$$

They show that given a suitable choice of height function F the subset of locally named terms in $\mathbb{L}_F$ adequately represents the set of equivalence classes of lambda terms, and set out a number of properties that the height function must satisfy to ensure that this is true. They give an example formalisation of the multivariate lambda calculus as an example of their representation in use.

1.2.3 De Bruijn representation

The de Bruijn representation [18] is a first-order approach to the representation of syntax with variable binding in which variables are given by a numerical *index* or *level* that associates them with their binder (in the case of bound variables) or with their entry in a context (in the case of free variables). This approach to the representation of variables is therefore *nameless*, and two terms are equivalent up to renaming of bound variables if they have the same de Bruijn representation.

With de Bruijn indices, the number of a bound variable indicates which enclosing binder the variable is bound by. For example, the index 3 would represent the variable bound by the third enclosing binder, and the term $\lambda x.\lambda y.x\ y$ would have the de Bruijn representation $\lambda\lambda 1\ 0$.

To represent free variables requires a naming context, which assigns indices to the free variables contained within the term. For example, we might have the term $\lambda x.(x\ y)\ z$. With the naming context $y \to 0, z \to 1$ the term would have de Bruijn representation $\lambda(0\ 1)\ 2$. The indices of the free variables in the naming context are incremented by 1 as we need to traverse 1 λ binder to reach the root of the term.

Substitution on de Bruijn terms requires an operation called *shifting*. This involves renumbering the free variables in the term that is being substituted in so that the index still indicates the same free variable when situated in the new context. For example, if we have the naming context $y \to 0, z \to 1$ and the term $t = \lambda 0\ 2$, when we naively substitute t for the outermost bound variable in $\lambda\lambda(10)1$ we end up with the term $\lambda((\lambda 0\ 2)\ 0)(\lambda 0\ 2)$. But now the free variable 2 indicates y in the naming context, not z as in the original term because it is now enclosed within an extra binder. We must find a way to increase the index of the free variables to account for the extra binders. This process is known as shifting.

However, we must be careful when shifting to ensure that only the indices of the free variables are increased, so we cannot simply increase every index in the term. We therefore have a *cutoff* value: indices above the cutoff will be increased, and indices below the cutoff will remain the same.

We denote the n-value shift above cutoff c of the term t by $\mathsf{shift}(n, c, t)$, and define it as follows:

Definition 9 (Shift)**.**

$$\begin{aligned}
\mathsf{shift}(n,\ c,\ k) &= k && (\textit{if}\ k < c)\\
\mathsf{shift}(n,\ c,\ k) &= k + n && (\textit{if}\ k \geq c)\\
\mathsf{shift}(n,\ c,\ (\lambda\ t_1)) &= \lambda\ \mathsf{shift}(n,\ c+1,\ t_1)\\
\mathsf{shift}(n,\ c,\ (t_1\ t_2)) &= \mathsf{shift}(n,\ c,\ t_1)\ \mathsf{shift}(n,\ c,\ t_2)
\end{aligned}$$

Note that n, the value to shift by, is added to k in the second equation as the variable index is above the cutoff. The cutoff c is incremented in the third equation as the shifting recurses over a λ binder. In Hybrid, shifting is performed by a function shift that takes as arguments a term, an amount to shift by, and a cutoff value to shift above. We denote substitution of the term t_1 for the variable numbered j in the term t_2 with $[t_1/j]t_2$, and define it as follows:

Definition 10.

$$\begin{aligned}
[t_1/j]k &= t_1 && (\textit{if}\ j = k)\\
[t_1/j]k &= k && (\textit{if}\ j \neq k)\\
[t_1/j](\lambda t) &= (\lambda\ [\mathsf{shift}(1, 0, t_1)/j+1]t)\\
[t_1/j](t_3\ t_4) &= ([t_1/j]t_3\ [t_1/j]t_4)
\end{aligned}$$

De Bruijn levels number variables in the opposite direction to de Bruijn indices, so that the variable bound by the outermost binder has level 0, and variables bound by subsequent binders have increasing indices. For example, the term $\lambda x.\lambda y.\lambda z.z\ (x\ y)$ has de Bruijn index representation $\lambda\lambda\lambda 0\ (2\ 1)$, but has de Bruijn level representation $\lambda\lambda\lambda 2\ (0\ 1)$.

1.2.4 Locally nameless representation

In the locally nameless representation instances of bound variables are denoted by de Bruijn indices, while instances of free variables are named. The key advantages of this methodology are that α-equivalent terms have the same locally nameless representation, there is no need for α-conversion during substitution and there is no need for shifting, as free variables are given names rather than indices that 'point' to entries in a naming context.

The locally nameless representation of the untyped λ-calculus would be as follows:

Example 11.

$$\texttt{expr} ::= \texttt{App expr expr} \mid \texttt{Abs expr} \mid \texttt{Bvar nat} \mid \texttt{Fvar fname}$$

Here the `Abs` binder is not labelled with a variable name, as bound variables are represented namelessly, the `Bvar` constructor has a natural number index and the `Fvar` constructor has a parameter of type `fname` that gives the name of the free variable.

There are two key operations on bound and free variables in the locally nameless representation: *opening* and *closing*.

Opening converts bound variables in the body of an abstraction that are instances of the variable bound by the abstraction into free variables. The operation requires that a name for the new free variables be supplied, and has a natural number parameter that counts the number of binders the function has recursed over (which is therefore 0 initially). Opening is defined like so, where n is the name to give the newly created variables, n' is a previously unused name of type `fname` and i tracks the number of binders that the operation has entered:

Definition 12.

$$\begin{aligned}
\mathsf{open}\ i\ n\ (\texttt{Bvar k}) &= \begin{cases} (\texttt{Fvar n}) & (i = k) \\ (\texttt{Bvar k}) & \text{otherwise} \end{cases} \\
\mathsf{open}\ i\ n\ (\texttt{Fvar j}) &= (\texttt{Fvar j}) \\
\mathsf{open}\ i\ n\ (\texttt{App e e'}) &= (\texttt{App}\ (\mathsf{open}\ i\ n\ \texttt{e})\ (\mathsf{open}\ i\ n\ \texttt{e'})) \\
\mathsf{open}\ i\ n\ (\texttt{Abs e}) &= (\texttt{Abs}\ (\mathsf{open}\ i\ n'\ \texttt{e}))
\end{aligned}$$

The other key operation on locally nameless terms is closing, which builds an abstraction from a term that forms the body of the new abstraction. Closing produces a new term in which the free variables with a given name have been replaced by bound variables with an index that 'points' to the root of the term. Like opening, the closing operation has as parameters a name n and a natural number i that tracks the number of binders recursed over (which again is initially 0).

Closing is defined like so:

Definition 13.

$$
\begin{aligned}
\mathsf{close}\ i\ n\ (\texttt{Bvar k}) &= (\texttt{Bvar k}) \\
\mathsf{close}\ i\ n\ (\texttt{Fvar j}) &= \begin{cases} (\texttt{Bvar i}) & (n = j) \\ (\texttt{Fvar j}) & \text{otherwise} \end{cases} \\
\mathsf{close}\ i\ n\ (\texttt{App e e'}) &= (\texttt{App}\ (\mathsf{close}\ i\ n\ \texttt{e})\ (\mathsf{close}\ i\ n\ \texttt{e'})) \\
\mathsf{close}\ i\ n\ (\texttt{Abs e}) &= (\texttt{Abs}\ (\mathsf{close}\ (i+1)\ n\ \texttt{e}))
\end{aligned}
$$

Note that closing can produce a term with dangling variables; these variables will be bound by the binder to be added at the root of the term.

Charguéraud [4] gives an overview of the locally nameless representation. McBride and McKinna [6] detail their use of the locally nameless representation in the dependently-typed language Epigram [7]. They define a function instantiate that performs opening and a function abstract that performs closing, and introduce a type of names `Name` like so:

```
type Name = Stack (String, Int)
```

This definition allows for *local* generation of fresh names by one or more *agents*. Agents have a `Name` that is used to generate the names of free variables and sub-agents. They always generate names that are longer than their own name. The string element of the datatype is used to give a human-readable prefix to the names created, while the integer element is used to generate a series of similarly named variables $x_1 \ldots x_n$. McBride and McKinna give the example of creating elimination operators for datatype definitions to illustrate their use of the locally nameless representation

1.2.5 Nominal approaches to syntax with variable binding

Nominal approaches to syntax with variable binding are based on notions of *name swapping* (or *permutation*), *support* and *freshness* of names. They allow induction and recursion to be defined on α-equivalence classes of terms with variable binders.

Given a set $\mathbb{A}$ of *atoms* (that can be viewed as names), a *nominal set* X is a set together with a well-defined operation of swapping atoms in elements of the set. Nominal sets must satisfy the *finite support property*: for every $x \in X$ there exists a finite subset $\overline{a} \subset \mathbb{A}$ that *supports* x so that for all $a, a' \in (\mathbb{A} - \overline{a})$ it holds that $(a\ a') \cdot x = x$.

Gabbay and Pitts [9] describe the theoretical basis of nominal models of syntax with variable binding based on Fraenkel and Mostowski set theory (FM-sets). They show how inductively-defined FM-sets can be used to model α-equivalence classes of terms with binders as algebraic datatypes.

Pitts [8] introduces *nominal logic*, a first-order logic with operations for renaming based on atom swapping, and reasoning about freshness of names. He divides sorts in the logic into sorts of atoms and sorts of data, defines a swapping operation for each sort of atoms A with the result x' of swapping the atoms a and a' in x denoted $(a\ a') \cdot x$, and a freshness relation for each sort of atoms A and sort S, written $a\#s$ to indicate that the atom a is fresh for s. A theory in Nominal Logic is given by a signature of sort, function and relation symbols and a set of axioms which may involve freshness, atom swapping and symbols from the signature. Pitts gives as example a theory of λ-terms modulo α-equivalence, producing a structural induction theorem for such equivalence classes.

Urban [10] details Nominal Isabelle, a package for Isabelle/HOL that implements the nominal techniques introduced by Gabbay and Pitts. He bases his work on *permutation types*: sets with a well-defined permutation operation, defines an inductive set of λ-terms that are in bijection with the α-equivalence classes of λ-terms and creates a datatype $\texttt{lam}_\alpha$ from the set, showing that $\texttt{lam}_\alpha$ is a permutation type. He develops a structural induction rule and recursion combinator for $\texttt{lam}_\alpha$, and gives a detailed proof of Barendregt's substitution lemma [11] using the nominal package.

1.3 Higher-order abstract syntax

We might attempt to encode the abstract syntax of a logical system with an inductive datatype, such as in example 7. A key disadvantage with defining the lambda calculus in this way is that we have to manually define operations on variables such as substitution. For example, we might define a function `subst` that substitutes an expression for a specified variable in another expression. From a theorem-proving perspective, the disadvantage with this is that we then have to prove properties of `subst`, which can be quite involved and tedious.

Higher-order abstract syntax (HOAS) is a technique for representing variable binding in an object logic or language. By using the variable binding of the metalogic to represent variable binding in the object logic we can avoid having to reason about object-level substitution, as substitution in the object language is implemented by substitution in the metalanguage. In HOAS object logic functions are represented by metalogic functions, and object logic variables by metalogic variables.

Taking the HOAS approach, we might represent the untyped lambda calculus as follows:

Example 14.

$$\texttt{expr} ::= \texttt{App expr expr} \mid \texttt{Abs} \ (\texttt{expr} \to \texttt{expr}) \mid \texttt{FreeVar nat}$$

Note that abstraction is implemented using function abstraction in the metalogic, and bound variables are implemented using metalogic bound variables (rather than an explicit constructor for bound variables). So a function $(\lambda x.\ z\ x)$ in the object logic would be represented by

$$\texttt{Abs} \ (\lambda \texttt{x}. \ \texttt{App} \ (\texttt{FreeVar 0}) \ \texttt{x})$$

where λ is the binder of the metalogic and x is a metalogic bound variable. This approach in general allows the substitution of the object logic (such as the untyped lambda calculus in this example) to be replaced by substitution in the metalogic (which might in this example be a typed lambda calculus): when we have a function application fx at the object level, we apply the metalogic function representing f to the term representing x, and the substitution of x into the body of f is taken care of by the metalogic implementation of substitution.

Unfortunately it is not possible to create such an inductive datatype. This is because of the relative cardinalities of the sets `expr` $\to$ `expr` and `expr`. If we could create this datatype, the set of expression trees generated by the above grammar would give the set `expr`, and it would be countably infinite with cardinality equal to that of the natural numbers. The set `expr` $\to$ `expr` would have cardinality equal to that of the reals, which is strictly greater than that of the natural numbers. As a consequence, the `Abs` constructor could not be injective. To ensure that this problem does not occur, we use positivity constraints: all instances of `expr` in the datatype must be strictly positive, which is not true of the first (negative) occurrence of `expr` as the source of the function argument in `Abs` (`expr` $\to$ `expr`) $\to$ `expr`. To use HOAS with inductive datatypes it is necessary to devise a solution to this problem.

Another issue with HOAS is the existence of so-called *exotic terms*. These occur when reasoning about an object logic in a metalogic, and are terms that can be constructed in the metalogic but cannot occur in the object logic. For example, Isabelle has an if-then-else construct that allows the creation of functions such as $(\lambda \texttt{x}.\ \texttt{if x} = \texttt{3 then 6 else 4})$ that do not represent any function in the λ-calculus. It is necessary to find a way to exclude these terms.

An implementation of an object logic in a metalogic is referred to as *adequate* if the implementation is both *full* and *faithful* [22]. The first condition ensures that the implementation does not permit the formation of any terms that are not possible in the object logic (such as exotic terms). An implementation is faithful if all of the terms of the object logic become unique terms of the metalogic. In such cases the translation between the object logic and its representation in the metalogic forms a compositional bijection.

Momigliano et al [14] distinguish between *weak* and *full* HOAS, defining an implementation of HOAS to be weak if object logic bound variables are represented as metalogic bound variables and object logic contexts are represented by metalogic contexts. In full HOAS bound variables and contexts are represented as in weak HOAS, but in addition object logic substitution is implemented by metalogic β-conversion.

An example of a weak HOAS approach is that of Despeyroux et al [16], which employs a type of variables *var* to work around the positivity restrictions on inductive datatypes. The above example of the untyped lambda calculus would then be represented like so:

Example 15.

$$\texttt{expr} ::= \texttt{Var var} \mid \texttt{App expr expr} \mid \texttt{Abs}\ (\texttt{var} \to \texttt{expr})$$

Note that the negative occurrence of `expr` has been removed and replaced with `var`; Momigliano et al refer to such a definition as *positivized*. The function $(\lambda x.\ x\ x)$ is then encoded as $(\lambda x : \texttt{var}.\ \texttt{App}\ (\texttt{Var}\ x)\ (\texttt{Var}\ x))$. Since the metalogic function type for abstraction is $\texttt{var} \to \texttt{expr}$ it is not possible to implement substitution in the untyped lambda calculus as metalogic β-reduction. It is therefore necessary to define substitution manually. This task is performed in Coq and Isabelle by an inductive definition.

The other issue with HOAS that must be resolved is the existence of exotic terms. Despeyroux et al construct a hierarchy of types, L_0 to L_n, so that L_0 is the terms of the lambda calculus, L_1 is the terms $\texttt{var} \to L_0$ and so on until $L_n = \texttt{var} \to L_{n-1}$. They define a hierarchy of validity predicates $\texttt{valid}_0$ to $\texttt{valid}_n$ which indicate the set of metalogic terms that represent object logic terms - i.e. those terms that are not exotic. The index on the validity predicate denotes the index of the type that the validity predicate applies to.

Using validity predicates in this way allows the exotic terms to be eliminated.

The other type of HOAS distinguished by Momigliano et al is full HOAS. In full HOAS, the abstract syntax of the object logic is encoded without using an inductive datatype. The untyped lambda calculus would be represented as follows:

Example 16.

$$\mathtt{FreeVar} : \mathtt{nat} \to \mathtt{exp}$$

$$\mathtt{App} : \mathtt{exp} \to \mathtt{exp} \to \mathtt{exp}$$

$$\mathtt{Abs} : (\mathtt{exp} \to \mathtt{exp}) \to \mathtt{exp}$$

These definitions would be part of a *signature* specifying the object constants that define an object logic. Momigliano et al note that this form of HOAS is only possible in an intuitionistic logic.

Chlipala [17] introduces *parametric higher-order abstract syntax* (PHOAS), a variant of weak HOAS. Like weak HOAS, parametric HOAS employs a type parameter as the type of variables, but in the parametric case this parameter is not global as in weak HOAS. Rather, the type parameter can be instantiated with different values at different points during the development of a proof.

In PHOAS, the untyped lambda calculus would be represented by an inductive type `term(V)` as follows:

Example 17.

$$\mathtt{Var} : \mathtt{V} \to \mathtt{term}(\mathtt{V})$$

$$\mathtt{App} : \mathtt{term}(\mathtt{V}) \to \mathtt{term}(\mathtt{V}) \to \mathtt{term}(\mathtt{V})$$

$$\mathtt{Abs} : (\mathtt{V} \to \mathtt{term}(\mathtt{V})) \to \mathtt{term}(\mathtt{V})$$

The type parameter `V` could be instantiated with a variety of different types.

Example 18.

$$
\begin{aligned}
\mathsf{checkTrue} &: \mathtt{term(bool)} \to \mathtt{bool} \\
\mathsf{checkVarsTrue}(\mathtt{Var}\ b) &= b \\
\mathsf{checkVarsTrue}(\mathtt{App}\ e_1\ e_2) &= \mathsf{checkVarsTrue}(e_1) \land \mathsf{checkVarsTrue}(e_2) \\
\mathsf{checkVarsTrue}(\mathtt{Abs}\ f) &= \mathsf{checkVarsTrue}(f\ \mathtt{True}) \\
\mathsf{canEta} &: \mathtt{term(bool)} \to \mathtt{bool} \\
\mathsf{canEta}(\mathtt{Abs}\ f) &= \mathtt{match}\ f\ \mathtt{False}\ \mathtt{with} \\
&\quad \mid \mathtt{App}\ e_1\ (\mathtt{Var\ False}) \Rightarrow \mathsf{checkVarsTrue}(e_1) \\
&\quad \mid \_ \Rightarrow \mathtt{False} \\
\mathsf{canEta}(\_) &= \mathtt{False} \\
\mathsf{canEta} &: \mathtt{Term} \to \mathtt{bool} \\
\mathsf{canEta}(\mathtt{E}) &= \mathsf{canEta}(\mathtt{E\ bool})
\end{aligned}
$$

Chlipala gives the example of a function (see example 18) that computes if a term is η-reducible. The function instantiates the type parameter with boolean values, which are then used to record information about the variables in the term.

If the term is an abstraction, the body of the abstraction is applied to false, and pattern matching is applied to the result. If the result is an application, the checkVarsTrue function tags bound variables by applying the body of abstractions to true, and checks for any occurences of variables that have been tagged false (i.e. the variable that was tagged in canEta).

1.4 Hybrid

The first version of Hybrid was developed by Ambler et al[15]. It provides a form of logical framework, and a key feature is the ability to automatically translate between a HOAS representation and a first-order de Bruijn representation. The system is based around a datatype of de Bruijn terms `expr`, using de Bruijn indices (rather than de Bruijn levels).

Definition 19.

$$'\mathtt{a}\ \mathtt{expr} ::= \mathtt{CON}\ '\mathtt{a} \mid \mathtt{BND\ nat} \mid \mathtt{VAR\ nat} \mid \mathtt{ABS\ expr} \mid \mathtt{expr}\ \$\$\ \mathtt{expr} \mid \mathtt{ERR}$$

The `expr` datatype is parameterised with a datatype consisting of object constant symbols. The `CON` constructor represents an object constant, and its parameter is drawn from the datatype of object constant symbols. The `BND` constructor represents a bound variable, taking as parameter a natural number for the de Bruijn index of the variable. The `ABS` constructor represents an abstraction, and $$ represents application of two expressions. The `VAR` constructor represents free variables, which are indexed by the natural numbers.

Hybrid provides a binder `LAM` which is used to represent abstractions. The term $\texttt{LAM}\ \texttt{v}_1.\ \texttt{LAM}\ \texttt{v}_2.\ (\texttt{VAR}\ 1\ \$\$\ \texttt{v}_2)$ is automatically converted to the de Bruijn term $\texttt{ABS}\ \texttt{ABS}\ (\texttt{VAR}\ 1\ \$\$\ \texttt{BND}\ 1)$.

A key concept in Hybrid is the *level* of a de Bruijn term. This is a different notion from de Bruijn levels in that it is a property of a term rather than something that might appear within a term. A *dangling* variable is a variable whose index is equal to or exceeds the number of enclosing binders and hence has no matching binder. The level of a de Bruijn term is the number of binders that the term would have to be enclosed within to ensure that none of the variables are dangling. For instance, the term $\lambda\ \lambda\ 0\ 3$ is at level 2, because it would have to be enclosed within 2 additional binders to ensure that the variable with index 3 is no longer dangling. In Hybrid, terms at level 0 (i.e. with no dangling variables) are referred to as *proper* terms. These proper terms correspond to valid terms of the λ-calculus. Hybrid has a predicate $\texttt{level} :: \texttt{nat} \Rightarrow \texttt{expr} \Rightarrow \texttt{bool}$ that indicates if an expression is at the specified level, and a predicate $\texttt{proper} :: \texttt{expr} \Rightarrow \texttt{bool}$ that is defined as $\texttt{proper e} \equiv \texttt{level 0 e}$

Another key concept in Hybrid is that of abstractions. An abstraction is a function that is not an exotic term and contains no dangling indices. Hybrid uses two functions to indicate if a function is a valid abstraction: abst and abstr. In the original version of Hybrid [15] abst is defined inductively as in figure 1.1.

The function abstr is then defined as follows:

Definition 20 (abstr)**.**

$$\mathsf{abstr}\ e \equiv \mathsf{abst}\ 0\ e$$

In a later version of Hybrid created by Martin [19] abstr was defined using the Isabelle `function` construct rather than using an inductive relation and then proving that the relation defines a function.

The `abst` and `abstr` functions select a subset of the function space as valid when functions from this subset are used as HOAS terms in Hybrid.

$$\frac{}{\mathsf{abst}\ i\ (\lambda v.\ v)}\ \text{ABST\_V} \qquad \frac{}{\mathsf{abst}\ i\ (\lambda v.\ \mathtt{VAR}\ n)}\ \text{ABST\_FV}$$

$$\frac{j < i}{\mathsf{abst}\ i\ (\lambda v.\ \mathtt{BND}\ j)}\ \text{ABST\_BV} \qquad \frac{\mathsf{abst}\ i\ f \qquad \mathsf{abst}\ i\ g}{\mathsf{abst}\ i\ (\lambda v.\ f\ v\ \$\$\ g\ v)}\ \text{ABST\_APP}$$

$$\frac{\mathsf{abst}\ (\mathrm{Suc}\ i)\ f}{\mathsf{abst}\ i\ (\lambda v.\ \mathtt{ABS}\ (f\ v))}\ \text{ABST\_ABS}$$

Figure 1.1: abst

Once a function has been determined to be an abstraction by the abstr function, the user then needs to be able to convert it to de Bruijn form to reason about it. The function lbind is instrumental in the conversion from higher-order to first-order terms: it takes as argument an Isabelle/HOL function abstraction and returns a level 1 de Bruijn expression with the metavariables of the original abstraction replaced with bound variables. It is defined in terms of another inductively-defined predicate lbnd, which is defined in figure 1.2.

Note the presence of the default case, LBND_NORD, that applies when none of the other rules are applicable. The function lbind makes use of lbnd and the ϵ description operator. It is defined as follows:

Definition 21 (lbind)**.**

$$\mathsf{lbind}\ i\ e \equiv \epsilon\ s.\ \mathsf{lbnd}\ i\ e\ s$$

As the de Bruijn term produced by lbind is at level 1, it potentially has dangling variables. This is resolved by the function lambda, which is defined like so:

Definition 22.

$$\mathsf{lambda}\ e \equiv \mathtt{ABS}\ (\mathsf{lbind}\ 0\ e)$$

The lambda function adds an enclosing abstraction so that the variables are no longer dangling. The `LAM` binder is in fact implemented using `lambda`, for example `LAM v.v` $\equiv$ lambda (λ`v. v`).

The Hybrid encoding of the untyped lambda calculus in Isabelle/HOL would be as follows:

$$\frac{}{\mathsf{lbnd}\ i\ (\lambda v.\ v)\ (\mathtt{BND}\ i)}\ \text{LBND\_VAR} \qquad \frac{}{\mathsf{lbnd}\ i\ (\lambda v.\ \mathtt{VAR}\ n)\ (\mathtt{VAR}\ n)}\ \text{LBND\_FV}$$

$$\frac{}{\mathsf{lbnd}\ i\ (\lambda v.\ \mathtt{BND}\ j)\ (\mathtt{BND}\ j)}\ \text{LBND\_BV}$$

$$\frac{\mathsf{lbnd}\ i\ f\ s \qquad \mathsf{lbnd}\ i\ g\ t}{\mathsf{lbnd}\ i\ (\lambda v.\ (f\ v)\ \$\$\ (g\ v))\ (s\ \$\$\ t)}\ \text{LBND\_APP}$$

$$\frac{\mathsf{lbnd}\ (\mathrm{Suc}\ i)\ f\ s}{\mathsf{lbnd}\ i\ (\lambda v.\ \mathtt{ABS}\ (f\ v))\ (\mathtt{ABS}\ s)}\ \text{LBND\_ABS}$$

$$\frac{\begin{array}{c} e \neq (\lambda v.\ v) \quad e \neq (\lambda v.\ \mathtt{VAR}\ n) \quad e \neq (\lambda v.\ \mathtt{BND}\ j) \\ e \neq (\lambda v.\ (f\ v)\ \$\$\ (g\ v)) \quad e \neq (\lambda v.\ \mathtt{ABS}\ (f\ v)) \end{array}}{\mathsf{lbnd}\ i\ e\ (\mathtt{BND}\ (i+1))}\ \text{LBND\_NORD}$$

Figure 1.2: lbnd

Example 23.

```
datatype cons = c_app | c_abs
```

```
definition app :: "cons expr ⇒ cons expr ⇒ cons expr" where
"app a b  ≡ (CON c_app) $$ a $$ b"
```

```
definition abs :: "(cons expr ⇒ cons expr) ⇒ cons expr" where
"abs f  ≡ (CON c_abs) $$ LAM f"
```

The datatype `cons` defines the set of constant symbols, which are then used with the `CON` constructor.

Crole [20] proves that a formal model of Hybrid adequately represents a λ-calculus with constants that models higher-order abstract syntax.

Chapter 2

HybridLF

2.1 Introduction

HybridLF is a version of Hybrid that implements the metatheory of the logical framework LF. Whereas previous variants of Hybrid provide a version of HOAS in the form of the untyped lambda calculus, HybridLF provides HOAS in the form of a dependently-typed lambda calculus.

LF uses a *signature* which assigns types to object and type level constants. The HOAS functions in HybridLF are used to enter the LF signature to be reasoned about, rather than simply converting terms into de Bruijn format to be reasoned about directly in Isabelle as in the original version of Hybrid.

The system implements the M_2 metalogic formulated by Schürmann and Pfenning [28] for reasoning about LF signatures. M_2 is a sequent calculus in which formulae have the form $\forall x_1 \ldots \forall x_k . \exists x_{k+1} \ldots \exists x_n . \top$ and quantifiers range over closed LF terms from the signature under consideration. We discuss M_2 further in chapter 4.

HybridLF extends the notion of abstraction so that there are now object-level abstractions and type-level abstractions.

Definition 24. A k-ary abstraction is a syntactic function with exactly k bound meta-variables and no dangling indices.

Example 25. So for example $(\lambda x.\ \lambda y.\ \texttt{ABS}\ (\texttt{FCON}\ c)\ (y\ \$\$_{\mathrm{O}}\ (\texttt{BND}\ 0)\ \$\$_{\mathrm{O}}\ x))$ is a binary abstraction because it has 2 bound meta-variables and it is a syntactic function.

In example 25 we have an abstraction with two meta-variables, x and y, containing an ABS node with the type constant FCON c as the type of its parameter, and y applied to the parameter of the ABS node applied to x as its body. For the grammar of HybridLF datatypes, see definition 29.

In HybridLF we expand the definition of abstraction to k-ary abstractions with more than one argument, and through the use of families of lbind-like functions allows abstractions with multiple arguments to be converted into de Bruijn form.

In section 2.2 we set out the LF type theory with three levels of objects, types and kinds, along with the judgements Σ sig indicating that Σ is a valid signature, $\vdash_\Sigma \ \Gamma$ indicating that the context Γ is a valid context, $\Gamma \vdash_\Sigma \ K$ denoting that K is a kind, $\Gamma \vdash_\Sigma \ A : K$ denoting that type A has kind K and $\Gamma \vdash_\Sigma \ M : A$ indicating that term M has type A. We also define the 3 levels of definitional equality: definitional equality on kinds ($\Gamma \vdash_\Sigma K \equiv K'$), definitional equality on types ($\Gamma \vdash_\Sigma A \equiv A'$) and definitional equality on objects ($\Gamma \vdash_\Sigma M \equiv M'$). In section 2.3 we discuss HybridLF itself. We describe its implementation in Isabelle initially by giving the definition of its datatypes, then by defining shifting, substitution and the level predicates as functions, and the LF judgements as relations (implemented using the Isabelle `inductive` construct). In subsection 2.3.1 we describe some properties of typing, kinding and definitional equality in HybridLF that allow us to infer that these judgements hold only for valid (i.e. proper) terms. In subsection 2.3.2 we briefly discuss the possibility of using a dependently-typed language to define o_abstr and f_abstr functions that take a variable number of arguments instead of creating a family of numbered functions, each determining if a function with a particular number of arguments is a valid abstraction. In section 2.4 we develop an induction rule for HybridLF terms and types.

2.2 LF

LF is the Edinburgh Logical Framework, first described by Harper et al [22]. The terms of LF consist of three levels, given by the following:

Definition 26.

$$M ::= \ c \mid x \mid \lambda x{:}A.M \mid MM'$$

$$A ::= \ a \mid \Pi x{:}A.B \mid AM$$

$$K ::= \ \text{Type} \mid \Pi x{:}A.K$$

Here c stands for an object-level constant, a a type-level constant and x a variable. We will use M, N and O to refer to objects, A, B and C to refer to types, and K and L to refer to kinds.

The LF judgements make use of *contexts* (denoted Γ), which assign types to variables, and signatures (denoted Σ) which assign types to object and type level constants.

Definition 27.

$$\Gamma ::= \langle\rangle \mid \Gamma, x{:}A$$

Definition 28.

$$\Sigma ::= \langle\rangle \mid \Sigma, a{:}K \mid \Sigma, c{:}A$$

The LF type theory defines a number of judgements: that a signature is valid (denoted Σ sig), that a context is valid (denoted $\vdash_\Sigma \Gamma$), that K is a kind (denoted $\Gamma \vdash_\Sigma K$), that A has kind K (denoted $\Gamma \vdash_\Sigma A : K$) and that M has type A (denoted $\Gamma \vdash_\Sigma M : A$). In addition, there are 3 levels of definitional equality: definitional equality on kinds (denoted $\Gamma \vdash_\Sigma K \equiv K'$), definitional equality on types (denoted $\Gamma \vdash_\Sigma A \equiv A'$) and definitional equality on objects (denoted $\Gamma \vdash_\Sigma M \equiv M'$). The rules defining the judgements of LF are in figure 2.1.

LF makes use of a notion of *canonical forms*. We follow the definition of canonical forms by Pfenning [21].

The canonical terms are a subset of ordinary terms, and canonical types are a subset of ordinary types. We will use C and C' to refer to a canonical term, and T and T' to refer to a canonical type.

The definition of canonical forms uses the following four judgements: that object M is canonical of type A (denoted $\Gamma \vdash_\Sigma^{\text{CAN}} M : A$), type A is a canonical type (denoted $\Gamma \vdash_\Sigma^{\text{CAN}} A : \text{Type}$), object M is atomic of type A (denoted $\Gamma \vdash_\Sigma^{\text{ATM}} M : A$) and that type A is atomic of kind K (denoted $\Gamma \vdash_\Sigma^{\text{ATM}} A : K$). These judgements are defined by the rules in figure 2.2.

2.3 HybridLF

HybridLF is based around 3 datatypes - `expr`, `type` and `kind` - of which `expr` and `type` are mutually inductively defined. The datatypes correspond to the different levels of the dependent type system. There are two different types of application: object-level application, denoted $\$\$_{\text{O}}$, and type-level application, denoted $\$\$_{\text{F}}$. As well as object constants there are type constants, which are introduced through the `FCON` constructor. The distinguished LF type-kind is denoted by `TYPE`. We add a type parameter to the `ABS` constructor, representing the type of its argument.

$$\frac{}{\langle\rangle \text{ sig}}\ \textsc{sigemp} \qquad \frac{\Sigma \text{ sig} \qquad \vdash_\Sigma\ K}{\Sigma, a{:}K \text{ sig}}\ \textsc{sigfam}$$

$$\frac{\Sigma \text{ sig} \qquad \vdash_\Sigma\ A : \text{Type}}{\Sigma, c{:}A \text{ sig}}\ \textsc{sigobj} \qquad \frac{}{\vdash_\Sigma\ \langle\rangle}\ \textsc{ctxemp}$$

$$\frac{\vdash_\Sigma\ \Gamma \qquad \vdash_\Sigma\ A : \text{Type}}{\vdash_\Sigma\ \Gamma, x{:}A}\ \textsc{ctxobj} \qquad \frac{}{\Gamma \vdash_\Sigma\ \text{Type}}\ \textsc{kndtyp}$$

$$\frac{\Gamma \vdash_\Sigma\ A : \text{Type} \qquad \Gamma, x{:}A \vdash_\Sigma\ K}{\Gamma \vdash_\Sigma\ \Pi x{:}A.K}\ \textsc{kndpi}$$

$$\frac{\Gamma \vdash_\Sigma\ A : K \qquad \Gamma \vdash_\Sigma K \equiv K'}{\Gamma \vdash_\Sigma\ A : K'}\ \textsc{kndcnv} \qquad \frac{a{:}K \in \Sigma}{\Gamma \vdash_\Sigma\ a : K}\ \textsc{famcon}$$

$$\frac{\Gamma \vdash_\Sigma\ A : \text{Type} \qquad \Gamma, x{:}A \vdash_\Sigma\ B : \text{Type}}{\Gamma \vdash_\Sigma\ \Pi x{:}A.B : \text{Type}}\ \textsc{fampi}$$

$$\frac{\Gamma \vdash_\Sigma\ A : \Pi x{:}B.K \qquad \Gamma \vdash_\Sigma\ M : B}{\Gamma \vdash_\Sigma\ AM : K[M/x]}\ \textsc{famapp}$$

$$\frac{c{:}A \in \Sigma}{\Gamma \vdash_\Sigma\ c : A}\ \textsc{objcon} \qquad \frac{x{:}A \in \Gamma}{\Gamma \vdash_\Sigma\ x : A}\ \textsc{objvar}$$

$$\frac{\Gamma \vdash_\Sigma\ A : \text{Type} \qquad \Gamma, x{:}A \vdash_\Sigma\ M : B}{\Gamma \vdash_\Sigma\ \lambda x{:}A.M : \Pi x{:}A.B}\ \textsc{objlam}$$

$$\frac{\Gamma \vdash_\Sigma\ M : \Pi x{:}A.B \qquad \Gamma \vdash_\Sigma\ N : A}{\Gamma \vdash_\Sigma\ MN : B[N/x]}\ \textsc{objapp}$$

$$\frac{\Gamma \vdash_\Sigma\ M : A \qquad \Gamma \vdash_\Sigma A \equiv A'}{\Gamma \vdash_\Sigma\ M : A'}\ \textsc{typcnv}$$

Figure 2.1: LF formation, typing and kinding judgements

$$\frac{\Gamma \vdash^{\text{CAN}}_{\Sigma} T : \text{Type} \qquad \Gamma, x{:}T \vdash^{\text{CAN}}_{\Sigma} C : A}{\Gamma \vdash^{\text{CAN}}_{\Sigma} \lambda x{:}T.C : \Pi x{:}T.A} \text{ CANPI}$$

$$\frac{\Gamma \vdash^{\text{ATM}}_{\Sigma} A : \text{Type} \qquad \Gamma \vdash^{\text{ATM}}_{\Sigma} C : A}{\Gamma \vdash^{\text{CAN}}_{\Sigma} C : A} \text{ CANATM}$$

$$\frac{\Gamma \vdash^{\text{CAN}}_{\Sigma} C : A \qquad \Gamma \vdash_{\Sigma} A \equiv B}{\Gamma \vdash^{\text{CAN}}_{\Sigma} C : B} \text{ CANCNV}$$

$$\frac{c{:}A \in \Sigma}{\Gamma \vdash^{\text{ATM}}_{\Sigma} c : A} \text{ ATMCON} \qquad \frac{x{:}A \in \Gamma}{\Gamma \vdash^{\text{ATM}}_{\Sigma} x : A} \text{ ATMVAR}$$

$$\frac{\Gamma \vdash^{\text{ATM}}_{\Sigma} M : \Pi x{:}A.B \qquad \Gamma \vdash^{\text{CAN}}_{\Sigma} C : A}{\Gamma \vdash^{\text{ATM}}_{\Sigma} MC : B[C/x]} \text{ ATMAPP}$$

$$\frac{\Gamma \vdash^{\text{ATM}}_{\Sigma} M : A \qquad \Gamma \vdash_{\Sigma} A \equiv B}{\Gamma \vdash^{\text{ATM}}_{\Sigma} M : B} \text{ ATMCNV} \qquad \frac{a{:}K \in \Sigma}{\Gamma \vdash^{\text{ATM}}_{\Sigma} a : K} \text{ ATTCON}$$

$$\frac{\Gamma \vdash^{\text{ATM}}_{\Sigma} A : \Pi x{:}B.K \qquad \Gamma \vdash^{\text{CAN}}_{\Sigma} C : B}{\Gamma \vdash^{\text{ATM}}_{\Sigma} AC : K[C/x]} \text{ ATTAPP}$$

$$\frac{\Gamma \vdash^{\text{ATM}}_{\Sigma} A : K \qquad \Gamma \vdash_{\Sigma} K \equiv K'}{\Gamma \vdash^{\text{ATM}}_{\Sigma} A : K'} \text{ ATTCNV} \qquad \frac{\Gamma \vdash^{\text{ATM}}_{\Sigma} T : \text{Type}}{\Gamma \vdash^{\text{CAN}}_{\Sigma} T : \text{Type}} \text{ CNTATM}$$

$$\frac{\Gamma \vdash^{\text{CAN}}_{\Sigma} T : \text{Type} \qquad \Gamma, x{:}T \vdash^{\text{CAN}}_{\Sigma} T' : \text{Type}}{\Gamma \vdash^{\text{CAN}}_{\Sigma} \Pi x{:}T.T' : \text{Type}} \text{ CNTPI}$$

Figure 2.2: LF canonicity judgements

Definition 29 (Datatypes)**.**

$$('a,\ 'b)\ \texttt{expr} ::= \texttt{CON}\ 'a \mid \texttt{BND nat} \mid \texttt{VAR nat} \mid \texttt{ABS}\ ('a,\ 'b)\ \texttt{type}\ ('a,\ 'b)\ \texttt{expr}$$
$$\mid ('a,\ 'b)\ \texttt{expr}\ \$\$_{\text{O}}\ ('a,\ 'b)\ \texttt{expr} \mid \texttt{ERR}$$

$$('a,\ 'b)\ \texttt{type} ::= \texttt{FCON}\ 'b \mid \texttt{FPI}\ ('a,\ 'b)\ \texttt{type}\ ('a,\ 'b)\ \texttt{type}$$
$$\mid ('a,\ 'b)\ \texttt{type}\ \$\$_{\text{F}}\ ('a,\ 'b)\ \texttt{expr} \mid \texttt{FERR}$$

$$('a,\ 'b)\ \texttt{kind} ::= \texttt{TYPE} \mid \texttt{KPI}\ ('a,\ 'b)\ \texttt{type}\ ('a,\ 'b)\ \texttt{kind}$$

The $M\ \$\$_{\text{O}}\ M'$ notation is an abbreviation for `APP` $M\ M'$, while $A\ \$\$_{\text{F}}\ M$ is an abbreviation for `FAPP` $A\ M$.

Example 30. The function $(\lambda x{:}t.\ \lambda y{:}t'.\ z\ (x\ y))$ would then be represented as (`ABS` t (`ABS` t' (`APP` (`VAR` 0) (`APP` (`BND` 1) (`BND` 0))))).

Note that the datatypes have two type parameters which provide the constant symbols, one each for expression and type constants. The symbols are given by two datatypes rather than one as in the original Hybrid to allow a distinction to be made between object-level and type-level constants.

For example, if we have a signature Σ for the untyped lambda calculus with type-level constants `term` and `eval` and object-level constants `abs`, `lam`, `app_eval_1`, `app_eval_2` and `app_eval_3`, we would use two Isabelle/HOL datatypes to represent the constant symbols.

The `term` constant would be used as the type of terms in Σ, while the `eval` constant would be used to label the evaluation judgement. The object constants `abs` and `lam` would be used to represent the syntax of the simply-typed lambda calculus, and the `app_eval_1`, `app_eval_2` and `app_eval_3` constants would be used to label the inference rules for evaluation:

Example 31.

```
datatype ty_cons = term | eval
datatype obj_cons = abs | lam | app_eval_1 | app_eval_2 | app_eval_3
```

In example 31 the `ty_cons` datatype provides the set of type constant symbols, while the object constant symbols are represented by `obj_cons`. The type of Hybrid expressions would therefore be `(obj_cons, ty_cons) expr`.

We adapt the notion of level from the original Hybrid to the dependently-typed setting. In HybridLF there are 3 level predicates: o_level and f_level that act upon objects and types, and k_level that acts upon kinds. The 'f' in f_level and $\$\$_{\mathrm{F}}$ stands for 'family', as this is the level of type families. For objects, the level predicate operates in the same way as level does on expressions in the original Hybrid. At the level of types, although there is no constructor for bound variables in the `type` datatype bound variables may occur within the `expr` component of a type-level application. For kinds, bound variables may appear within the type annotation of a `KPI`.

The type of o_level is `nat` $\rightarrow$ `('a, 'b) expr` $\rightarrow$ `bool`, while the type of f_level is `nat` $\rightarrow$ `('a, 'b) type` $\rightarrow$ `bool` and the type of k_level is `nat` $\rightarrow$ `('a, 'b) kind` $\rightarrow$ `bool`

Definition 32 (o_level)**.**

$$\begin{aligned}
\mathsf{o_level}\ k\ (\mathtt{CON}\ a) &= \mathtt{True}\\
\mathsf{o_level}\ k\ (\mathtt{BND}\ j) &= (j < k)\\
\mathsf{o_level}\ k\ (\mathtt{VAR}\ i) &= \mathtt{True}\\
\mathsf{o_level}\ k\ \mathtt{ERR} &= \mathtt{True}\\
\mathsf{o_level}\ k\ (\mathtt{ABS}\ t\ f) &= (\mathsf{f_level}\ k\ t \wedge \mathsf{o_level}\ (k+1)\ f)\\
\mathsf{o_level}\ k\ (f\ \$\$_{\mathrm{O}}\ g) &= (\mathsf{o_level}\ k\ f \wedge \mathsf{o_level}\ k\ g)
\end{aligned}$$

Definition 33 (f_level)**.**

$$\begin{aligned}
\mathsf{f_level}\ k\ \mathtt{FERR} &= \mathtt{True}\\
\mathsf{f_level}\ k\ (\mathtt{FCON}\ a) &= \mathtt{True}\\
\mathsf{f_level}\ k\ (\mathtt{FPI}\ t\ f) &= (\mathsf{f_level}\ k\ t \wedge \mathsf{f_level}\ (k+1)\ f)\\
\mathsf{f_level}\ k\ (f\ \$\$_{\mathrm{F}}\ g) &= (\mathsf{f_level}\ k\ f \wedge \mathsf{o_level}\ k\ g)
\end{aligned}$$

Definition 34 (k_level)**.**

$$\begin{aligned}
\mathsf{k_level}\ k\ \mathtt{TYPE} &= \mathtt{True}\\
\mathsf{k_level}\ k\ (\mathtt{KPI}\ a\ l) &= (\mathsf{f_level}\ k\ a \wedge \mathsf{k_level}\ (k+1)\ l)
\end{aligned}$$

We define functions o_subst', o_subst, f_subst', f_subst, k_subst' and k_subst that perform substitution of an object for a bound variable on objects, types and kinds. o_subst' performs the work of substitution on objects, while o_subst calls it with a default value of zero for the amount to shift by. The same is true for substitution on types with f_subst' and f_subst, and for substitution on kinds with $\mathsf{k\_subst'}$ and $\mathsf{k\_subst}$.

Because the sets of instances of free variables and bound variables are disjoint (as free variables are denoted by instances of VAR rather than instances of BND) there is no need for shifting to ensure that the indices of free variables are correct in their new context. We indicate the substitution of HybridLF object M for bound variables with index i and with shifting amount n in object M' by $M'[M/i]^n_{\text{O'}}$, in type A by $A[M/i]^n_{\text{F'}}$ and in kind K by $K[M/i]^n_{\text{K'}}$. We will more commonly indicate the substitution of object M for bound variables with index i starting from the default shifting amount of zero in object M' by $M'[M/i]_{\text{O}}$, in type A by $A[M/i]_{\text{F}}$ and in kind K by $K[M/i]_{\text{K}}$. The $M'[M/i]^n_{\text{O'}}$ notation is simply a neater way of writing $\mathsf{o\_subst'}\ i\ n\ M'M$, where i and n are natural numbers and M and M' are HybridLF expressions. Similarly, $A[M/i]^n_{\text{F'}}$ is syntactic sugar for $\mathsf{f\_subst'}\ i\ n\ A\ M$ where i and n are again natural numbers, A is a HybridLF type and M is a HybridLF term, and $K[M/i]^n_{\text{K'}}$ is another way of writing $\mathsf{k\_subst'}\ i\ n\ K\ M$ where i and n are natural numbers as before, K is a HybridLF kind and M is a HybridLF term. In a similar way, $M'[M/i]_{\text{O}}$ is another way of writing $\mathsf{o\_subst}\ i\ M'\ M$, $A[M/i]_{\text{F}}$ is another way of writing $\mathsf{f\_subst}\ i\ A\ M$, and $K[M/i]_{\text{K}}$ is another way of writing $\mathsf{k\_subst}\ i\ K\ M$.

Before we define substitution, we define shifting for objects and types via the $\mathsf{o\_shift}$ and $\mathsf{f\_shift}$ functions.

Definition 35 (o_shift)**.**

$$
\begin{aligned}
\mathsf{o\_shift}\ 0\ k\ n &= n \\
\mathsf{o\_shift}\ i\ k\ (\mathtt{CON}\ c) &= (\mathtt{CON}\ c) \\
\mathsf{o\_shift}\ i\ k\ (\mathtt{VAR}\ n) &= (\mathtt{VAR}\ n) \\
\mathsf{o\_shift}\ i\ k\ \mathtt{ERR} &= \mathtt{ERR} \\
\mathsf{o\_shift}\ i\ k\ (\mathtt{ABS}\ A\ M) &= (\mathtt{ABS}\ (\mathsf{f\_shift}\ i\ k\ a)\ (\mathsf{o\_shift}\ i\ (k+1)\ m)) \\
\mathsf{o\_shift}\ i\ k\ (\mathtt{BND}\ n) &= (\mathtt{if}\ (n \geq k)\ \mathtt{then}\ (\mathtt{BND}\ (n+i))\ \mathtt{else}\ (\mathtt{BND}\ n)) \\
\mathsf{o\_shift}\ i\ k\ (\mathtt{APP}\ M\ N) &= (\mathtt{APP}(\mathsf{o\_shift}\ i\ k\ M)\ (\mathsf{o\_shift}\ i\ k\ N))
\end{aligned}
$$

Definition 36 (f_shift)**.**

$$\begin{aligned}
\mathsf{f_shift}\ 0\ k\ a &= a\\
\mathsf{f_shift}\ i\ k\ \texttt{FERR} &= \texttt{FERR}\\
\mathsf{f_shift}\ i\ k\ (\texttt{FCON}\ c) &= (\texttt{FCON}\ c)\\
\mathsf{f_shift}\ i\ k\ (\texttt{FAPP}\ A\ M) &= (\texttt{FAPP}\ (\mathsf{f_shift}\ i\ k\ A)\ (\mathsf{o_shift}\ i\ k\ M))\\
\mathsf{f_shift}\ i\ k\ (\texttt{FPI}\ A\ B) &= (\texttt{FPI}\ (\mathsf{f_shift}\ i\ k\ A)\ (\mathsf{f_shift}\ i\ (k+1)\ B))
\end{aligned}$$

Shifting on objects is used during substitution to ensure that the indices of the object being substituted in refer to the correct binder. Shifting on types is used during shifting of objects (as f_shift and o_shift are mutually defined).

Definition 37 (Substitution)**.**

$$\begin{aligned}
(\texttt{CON}\ c)[M/i]^{n}_{\mathrm{O'}} &= (\texttt{CON}\ c)\\
(\texttt{VAR}\ k)[M/i]^{n}_{\mathrm{O'}} &= (\texttt{VAR}\ k)\\
(\texttt{BND}\ k)[M/i]^{n}_{\mathrm{O'}} &= (\text{if } k = i \text{ then } \mathsf{o_shift}\ n\ 0\ M \text{ else } (\texttt{BND}\ k))\\
(a\ \$\$_{\mathrm{O}}\ b)[M/i]^{n}_{\mathrm{O'}} &= (a[M/i]^{n}_{\mathrm{O'}})\ \$\$_{\mathrm{O}}\ (b[M/i]^{n}_{\mathrm{O'}})\\
(\texttt{ABS}\ a\ b)[M/i]^{n}_{\mathrm{O'}} &= \texttt{ABS}\ (a[M/i]^{n}_{\mathrm{F'}}\ (b[M/i+1]^{n+1}_{\mathrm{O'}})\\
\texttt{ERR}\ [M/i]^{n}_{\mathrm{O'}} &= \texttt{ERR}
\end{aligned}$$

$$N\ [M/i]_{\mathrm{O}} = N\ [M/i]^{0}_{\mathrm{O'}}$$

$$\begin{aligned}
(\texttt{FPI}\ A\ A')[M/i]^{n}_{\mathrm{F'}} &= \texttt{FPI}\ (A[M/i]^{n}_{\mathrm{F'}})\ (A'[M/i+1]^{n+1}_{\mathrm{F'}})\\
(A\ \$\$_{\mathrm{F}}\ M')[M/i]^{n}_{\mathrm{F'}} &= ((A[M/i]^{n}_{\mathrm{F'}})\ \$\$_{\mathrm{F}}\ (M'[M/i]^{n}_{\mathrm{O'}}))\\
(\texttt{FCON}\ a)[M/i]^{n}_{\mathrm{F'}} &= (\texttt{FCON}\ a)\\
\texttt{FERR}[M/i]^{n}_{\mathrm{F'}} &= \texttt{FERR}
\end{aligned}$$

$$A\ [M/i]_{\mathrm{F}} = A\ [M/i]^{0}_{\mathrm{F'}}$$

$$\begin{aligned}
\texttt{TYPE}[M/i]^{n}_{\mathrm{K'}} &= \texttt{TYPE}\\
(\texttt{KPI}\ A\ K)[M/i]^{n}_{\mathrm{K'}} &= \texttt{KPI}(A[M/i]^{n}_{\mathrm{F'}})(K[M/i+1]^{n+1}_{\mathrm{K'}})
\end{aligned}$$

$$K\ [M/i]_{\mathrm{K}} = A\ [M/i]^{0}_{\mathrm{K'}}$$

The signature consists of into two parts: object constants and type constants (called the object signature and type signature respectively). These parts are implemented as lists of tuples (c, v) where c is a constant symbol drawn from the datatypes which are given as parameters for the `expr`, `type` and `kind` datatypes, and v is the type or kind assigned to that constant. There are functions oconlookup and fconlookup that look up the type or kind for an object constant or a type constant respectively.

We will shortly define the typeof, kindof, obj_def_equal, type_def_equal, kind_def_equal and validkind relations by mutual induction. The levels of objects and types are assigned types and kinds by the typeof and kindof relations. These are mutually-defined, and define 2 judgements: that an object M has a type A (denoted $\Gamma, \psi \vdash_\Sigma M : A$) and that a type A has kind K (denoted $\Gamma, \psi \vdash_\Sigma A : K$). These judgements use a context (denoted Γ) which maps natural variable numbers to types, and a binding environment (denoted ψ) which is an ordered list of types. We use $\psi(n)$ to indicate the nth element of the binding environment, and $A\#\psi$ to indicate the binding environment ψ extended with the type A. We indicate substitution of the object M for the variables bound by the nth binder with $[M/n]_{\mathrm{F}}$, definitional equality of types A and B at kind K with $\Gamma, \psi \vdash_\Sigma A \equiv B : K$ and definitional equality of kinds K and L with $\Gamma, \psi \vdash_\Sigma K \equiv L$. The rules defining the kindof and typeof relations are shown in figure 2.3. In the implementation, typeof consists of a relation between a context, an object signature, a type signature, a binding environment, an expression and a type. kindof consists of a relation between a context, an object signature, a type signature, a binding environment, a type and a kind.

The obj_def_equal, type_def_equal and kind_def_equal relations implement definitional equality of objects, types and kinds respectively. The context and binding environments are the same as for the typeof and kindof relation. We use $\Gamma, \psi \vdash_\Sigma M \equiv N : A$ to indicate that the objects M and N are definitionally equal at type A, $\Gamma, \psi \vdash_\Sigma A \equiv B : K$ to indicate that the types A and B are definitionally equal at kind K and $\Gamma, \psi \vdash_\Sigma K \equiv L$ to indicate that the kinds K and L are definitionally equal. The rules defining the obj_def_equal, type_def_equal and kind_def_equal relations are in figures 2.4 and 2.5. In the Isabelle implementation, the obj_def_equal, type_def_equal and kind_def_equal consist of relations between a context, an object signature, a type signature, a binding environment, two objects and a type (in the case of obj_def_equal) or two types and a kind (in the case of type_def_equal) or two kinds (in the case of kind_def_equal).

$$\frac{\psi(i) = A \qquad \mathsf{f\_level}\ 0\ A}{\Gamma, \psi \vdash_\Sigma \mathtt{BND}\ i : A}\ \textsc{ty\_bnd} \qquad\qquad \frac{\Gamma(j) = A \qquad \mathsf{f\_level}\ 0\ A}{\Gamma, \psi \vdash_\Sigma \mathtt{VAR}\ j : A}\ \textsc{ty\_var}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A : \mathtt{TYPE} \qquad \mathsf{f\_level}\ 0\ A \qquad \Gamma, A\#\psi \vdash_\Sigma M : B}{\Gamma, \psi \vdash_\Sigma \mathtt{ABS}\ A\ M : \mathtt{FPI}\ A\ B}\ \textsc{ty\_abs}$$

$$\frac{\Gamma, \psi \vdash_\Sigma M : \mathtt{FPI}\ A\ B \qquad \Gamma, \psi \vdash_\Sigma N : A \qquad \mathsf{o\_level}\ 0\ N}{\Gamma, \psi \vdash_\Sigma \mathtt{APP}\ M\ N : B[N/0]_{\text{F}}}\ \textsc{ty\_app}$$

$$\frac{\Gamma, \psi \vdash_\Sigma M : A \qquad \Gamma, \psi \vdash_\Sigma A \equiv B : K}{\Gamma, \psi \vdash_\Sigma M : B}\ \textsc{ty\_conv}$$

$$\frac{\Sigma(c) = A \qquad \mathsf{f\_level}\ 0\ A}{\Gamma, \psi \vdash_\Sigma \mathtt{CON}\ c : A}\ \textsc{ty\_con} \qquad\qquad \frac{\Sigma(c) = K \qquad \mathsf{k\_level}\ 0\ K}{\Gamma, \psi \vdash_\Sigma \mathtt{KCON}\ c : K}\ \textsc{k\_con}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A : \mathtt{TYPE} \qquad \Gamma, A\#\psi \vdash_\Sigma B : \mathtt{TYPE}}{\Gamma, \psi \vdash_\Sigma \mathtt{FPI}\ A\ B : \mathtt{TYPE}}\ \textsc{k\_pi}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A : \mathtt{KPI}\ B\ K \qquad \Gamma, \psi \vdash_\Sigma M : B \qquad \mathsf{o\_level}\ 0\ M}{\Gamma, \psi \vdash_\Sigma \mathtt{FAPP}\ A\ M : K[M/0]_{\text{K}}}\ \textsc{k\_app}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A : K \qquad \Gamma, \psi \vdash_\Sigma K \equiv L \qquad \mathsf{k\_level}\ 0\ L}{\Gamma, \psi \vdash_\Sigma A : L}\ \textsc{k\_conv}$$

Figure 2.3: HybridLF typeof and kindof relations

$$\frac{\begin{array}{cc}\Gamma, \psi \vdash_\Sigma A : \texttt{TYPE} & \Gamma, A\#\psi \vdash_\Sigma M : B \\ \textsf{o\_level}\ 0\ N & \textsf{f\_level}\ 0\ A \\ \textsf{o\_level}\ 1\ M & \Gamma, \psi \vdash_\Sigma N : A\end{array}}{\Gamma, \psi \vdash_\Sigma \texttt{APP}\ (\texttt{ABS}\ A\ M)\ N \equiv M[N/0]_{\mathrm{O}} : B[N/0]_{\mathrm{F}}}\ \textsc{obj\_eq\_beta}$$

$$\frac{\begin{array}{c}\Gamma, \psi \vdash_\Sigma A : \texttt{TYPE} \qquad \textsf{f\_level}\ 0\ A \\ \Gamma, A\#\psi \vdash_\Sigma \texttt{APP}\ M\ O \equiv \texttt{APP}\ N\ O : B\end{array}}{\Gamma, \psi \vdash_\Sigma M \equiv N : \texttt{FPI}\ A\ B}\ \textsc{obj\_eq\_ext}$$

$$\frac{\Gamma, \psi \vdash_\Sigma M : \texttt{FPI}\ A\ B \qquad \textsf{o\_level}\ 0\ M}{\Gamma, \psi \vdash_\Sigma \texttt{ABS}\ A\ (\texttt{APP}\ M\ (\texttt{BND}\ 0)) \equiv M : \texttt{FPI}\ A\ B}\ \textsc{obj\_eq\_eta}$$

$$\frac{\begin{array}{c}\Gamma, \psi \vdash_\Sigma M : A \\ \textsf{o\_level}\ 0\ M\end{array}}{\Gamma, \psi \vdash_\Sigma M \equiv M : A}\ \textsc{obj\_eq\_refl} \qquad \frac{\Gamma, \psi \vdash_\Sigma N \equiv M : A}{\Gamma, \psi \vdash_\Sigma M \equiv N : A}\ \textsc{obj\_eq\_sym}$$

$$\frac{\Gamma, \psi \vdash_\Sigma M \equiv M' : A \qquad \Gamma, \psi \vdash_\Sigma M' \equiv N : A}{\Gamma, \psi \vdash_\Sigma M \equiv N : A}\ \textsc{obj\_eq\_trans}$$

$$\frac{\Sigma(c) = A \qquad \textsf{f\_level}\ 0\ A}{\Gamma, \psi \vdash_\Sigma \texttt{CON}\ c \equiv \texttt{CON}\ c : A}\ \textsc{obj\_eq\_cng\_con}$$

$$\frac{\Gamma(i) = A \qquad \textsf{f\_level}\ 0\ A}{\Gamma, \psi \vdash_\Sigma \texttt{VAR}\ i \equiv \texttt{VAR}\ i : A}\ \textsc{obj\_eq\_cng\_var}$$

$$\frac{\Gamma, \psi \vdash_\Sigma M \equiv N : \texttt{FPI}\ A\ B \qquad \Gamma, \psi \vdash_\Sigma M' \equiv N' : A}{\Gamma, \psi \vdash_\Sigma \texttt{APP}\ M\ M' \equiv \texttt{APP}\ N\ N' : B[M'/0]_{\mathrm{F}}}\ \textsc{obj\_eq\_cng\_app}$$

$$\frac{\begin{array}{c}\Gamma, \psi \vdash_\Sigma A \equiv A' : \texttt{TYPE} \qquad \Gamma, \psi \vdash_\Sigma A \equiv A'' : \texttt{TYPE} \\ \Gamma, A\#\psi \vdash_\Sigma M \equiv N : B\end{array}}{\Gamma, \psi \vdash_\Sigma \texttt{ABS}\ A'\ M \equiv \texttt{ABS}\ A''\ N : \texttt{FPI}\ A\ B}\ \textsc{obj\_eq\_cng\_lam}$$

Figure 2.4: HybridLF definitional equality relations

$$\frac{\Gamma, \psi \vdash_\Sigma A : K \qquad \mathsf{f\_level}\ 0\ A}{\Gamma, \psi \vdash_\Sigma A \equiv A : K}\ \text{TY\_EQ\_REFL} \qquad \frac{\Gamma, \psi \vdash_\Sigma A \equiv B : K}{\Gamma, \psi \vdash_\Sigma B \equiv A : K}\ \text{TY\_EQ\_SYM}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A \equiv A' : K \qquad \Gamma, \psi \vdash_\Sigma A' \equiv B : K}{\Gamma, \psi \vdash_\Sigma A \equiv B : K}\ \text{TY\_EQ\_TRANS}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A \equiv B : \mathtt{KPI}\ C\ K \qquad \Gamma, \psi \vdash_\Sigma M \equiv N : C}{\Gamma, \psi \vdash_\Sigma \mathtt{FAPP}\ A\ M \equiv \mathtt{FAPP}\ B\ N : K[M/0]_{\mathrm{K}}}\ \text{TY\_EQ\_CNG\_APP}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A \equiv A' : \mathtt{TYPE} \qquad \Gamma, A\#\psi \vdash_\Sigma B \equiv B' : \mathtt{TYPE}}{\Gamma, \psi \vdash_\Sigma \mathtt{FPI}\ A\ B \equiv \mathtt{FPI}\ A'\ B' : \mathtt{TYPE}}\ \text{TY\_EQ\_CNG\_PI}$$

$$\frac{\Sigma(a) = K \qquad \mathsf{k\_level}\ 0\ K}{\Gamma, \psi \vdash_\Sigma \mathtt{FCON}\ a \equiv \mathtt{FCON}\ a : K}\ \text{TY\_EQ\_CNG\_CON}$$

$$\frac{\Gamma, \psi \vdash_\Sigma K}{\Gamma, \psi \vdash_\Sigma K \equiv K}\ \text{KIND\_EQ\_REFL} \qquad \frac{\Gamma, \psi \vdash_\Sigma K \equiv K'}{\Gamma, \psi \vdash_\Sigma K' \equiv K}\ \text{KIND\_EQ\_SYM}$$

$$\frac{\Gamma, \psi \vdash_\Sigma K \equiv K' \qquad \Gamma, \psi \vdash_\Sigma K' \equiv L}{\Gamma, \psi \vdash_\Sigma K \equiv L}\ \text{KIND\_EQ\_TRANS}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A \equiv A' : \mathtt{TYPE} \qquad \Gamma, A\#\psi \vdash_\Sigma K \equiv K'}{\Gamma, \psi \vdash_\Sigma \mathtt{KPI}\ A\ K \equiv \mathtt{KPI}\ A'\ K'}\ \text{KIND\_EQ\_CNG\_PI}$$

Figure 2.5: HybridLF definitional equality relations (continued)

In Isabelle, the typeof, kindof, validkind and definitional equality relations are defined via mutual induction using the `inductive` command. In figure 2.6 we show some example rules from this definition, and in appendix A we show the full definition.

```
TY_BND : "[lookup bnd i = Some a; f_level 0 a] ⟹
    typeof ctx sig_t sig_k bnd (BND i) a"
| TY_VAR : "[varlookup ctx i = Some a; f_level 0 a] ⟹
    typeof ctx sig_t sig_k bnd (VAR i) a"
| TY_CON : "[oconlookup sig_t c = Some a; f_level 0 a] ⟹
    typeof ctx sig_t sig_k bnd (CON c) a"
| TY_ABS : "[kindof ctx sig_t sig_k bnd ty TYPE; f_level 0 ty;
    typeof ctx sig_t sig_k (ty # bnd) e t1] ⟹
    typeof ctx sig_t sig_k bnd (ABS ty e) (FPI ty t1)"
...
| K_CON : "[fconlookup sig_k a = Some k; k_level 0 k] ⟹
    kindof ctx sig_t sig_k bnd (FCON a) k"
| K_PI : "[kindof ctx sig_t sig_k bnd t1 TYPE;
    kindof ctx sig_t sig_k (t1 # bnd) t2 TYPE] ⟹
    kindof ctx sig_t sig_k bnd (FPI t1 t2) TYPE"
...
| VK_TYPE : "validkind ctx sig_t sig_k bnd TYPE"
| VK_KPI : "[kindof ctx sig_t sig_k bnd ty TYPE; f_level 0 ty;
    validkind ctx sig_t sig_k (ty # bnd) k] ⟹
    validkind ctx sig_t sig_k bnd (KPI ty k)"
...
| OBJ_EQ_BETA : "[kindof ctx sig_t sig_k bnd a TYPE;
    typeof ctx sig_t sig_k (a # bnd) m b; typeof ctx sig_t sig_k bnd n a;
    o_level 0 (APP (ABS a m) n); o_subst 0 m n = m'; f_subst 0 b n = b'] ⟹
    obj_def_equal ctx sig_t sig_k bnd (APP(ABS a m) n) m' b'"
| OBJ_EQ_EXT : "[kindof ctx sig_t sig_k bnd a TYPE; f_level 0 a;
    obj_def_equal ctx sig_t sig_k (a # bnd) (APP m x) (APP n x) b] ⟹
    obj_def_equal ctx sig_t sig_k bnd m n (FPI a b)"
| OBJ_EQ_ETA : "[typeof ctx sig_t sig_k bnd m (FPI a b); o_level 0 m] ⟹
    obj_def_equal ctx sig_t sig_k bnd (ABS a (APP m (BND 0))) m (FPI a b)"
```

Figure 2.6: typeof, kindof, validkind and definitional equality implementation examples

2.3.1 Properties of typing, kinding and definitional equality relations

Note that a number of level assertions appear in typeof, kindof, obj_def_equal, type_def_equal, kind_def_equal and validkind. These are designed to ensure that a number of properties hold for these relations. We first state the properties here in theorem 38, then prove the theorem on page 45.

Theorem 38 (typeof_kindof_def_equal_validkind_level)**.**

$$\Gamma, \psi \vdash_\Sigma M : A \Longrightarrow \mathsf{f\_level}\ 0\ A$$
$$\Gamma, \psi \vdash_\Sigma A : K \Longrightarrow \mathsf{k\_level}\ 0\ K$$
$$\Gamma, \psi \vdash_\Sigma M \equiv N : A \Longrightarrow \mathsf{o\_level}\ 0\ M \wedge \mathsf{o\_level}\ 0\ N \wedge \mathsf{f\_level}\ 0\ A$$
$$\Gamma, \psi \vdash_\Sigma A \equiv B : K \Longrightarrow \mathsf{f\_level}\ 0\ A \wedge \mathsf{f\_level}\ 0\ B \wedge \mathsf{k\_level}\ 0\ K$$
$$\Gamma, \psi \vdash_\Sigma K \equiv L \Longrightarrow \mathsf{k\_level}\ 0\ K \wedge \mathsf{k\_level}\ 0\ L$$
$$\Gamma, \psi \vdash_\Sigma K \Longrightarrow \mathsf{k\_level}\ 0\ K$$

Recall that we refer to a de Bruijn term at level 0 (i.e. with no dangling indices) as a *proper* term. The first two properties, $\Gamma, \psi \vdash_\Sigma M : A \Longrightarrow \mathsf{f\_level}\ 0\ A$ and $\Gamma, \psi \vdash_\Sigma A : K \Longrightarrow \mathsf{k\_level}\ 0\ K$ ensure that when an object has a type or a type has a kind the type or kind is proper. The next three properties ensure that the three forms of definitional equality only apply to proper de Bruijn terms, and that the type or kind that objects or types are definitionally equal at is also a proper term. The final property ensures that valid kinds are proper de Bruijn terms. So for example, if we know that $\Gamma, \psi \vdash_\Sigma M \equiv N : A$, by the third property we know that M and N are proper terms, and that A is a proper type.

Before we can prove these properties we require a number of lemmas. Lemma 39, o_level_o_shift_abs, shows that if an `ABS` node has level p when it is shifted by i, then the body of the `ABS` node has level $p + 1$ when it is shifted by i and the type of the `ABS` node has level p when it is shifted by i.

Lemma 39 (o_level_o_shift_abs)**.**

$$\frac{\mathsf{o\_level}\ p\ (\mathsf{o\_shift}\ i\ k\ (\mathtt{ABS}\ A\ M))}{\mathsf{o\_level}\ (p+1)\ (\mathsf{o\_shift}\ i\ (k+1)\ M)}\ \text{O\_LEVEL\_O\_SHIFT\_ABS\_1}$$

$$\frac{\mathsf{o\_level}\ p\ (\mathsf{o\_shift}\ i\ k\ (\mathtt{ABS}\ A\ M))}{\mathsf{f\_level}\ p\ (\mathsf{f\_shift}\ i\ k\ A)}\ \text{O\_LEVEL\_O\_SHIFT\_ABS\_2}$$

Proof. By induction on i (proved separately in Isabelle). □

Lemma 40, o_level_o_shift_app, shows that if an `APP` node has level p when it is shifted by i, then the two constitutent terms of the `APP` node also have level p when they are shifted by i.

Lemma 40 (o_level_o_shift_app)**.**

$$\frac{\textsf{o\_level}\ p\ (\textsf{o\_shift}\ i\ k\ (\texttt{APP}\ M\ N))}{\textsf{o\_level}\ p\ (\textsf{o\_shift}\ i\ k\ M)}\ \textsc{o\_level\_o\_shift\_}\mathrm{APP\_1}$$

$$\frac{\textsf{o\_level}\ p\ (\textsf{o\_shift}\ i\ k\ (\texttt{APP}\ M\ N))}{\textsf{o\_level}\ p\ (\textsf{o\_shift}\ i\ k\ N)}\ \textsc{o\_level\_o\_shift\_}\mathrm{APP\_2}$$

Proof. By induction on i (proved separately in Isabelle). □

Lemma 41, o_level_o_shift_bnd, shows that if a `BND` node is shifted by i and has level $p + i$ for some p, then the index of the bound variable is lower than p.

Lemma 41 (o_level_o_shift_bnd)**.**

$$\frac{\textsf{o\_level}\ (p+i)\ (\textsf{o\_shift}\ i\ k\ (\texttt{BND}\ b)) \qquad k \le b}{b < p}\ \textsc{o\_level\_o\_shift\_bnd\_}1$$

$$\frac{\textsf{o\_level}\ (p+i)\ (\textsf{o\_shift}\ i\ k\ (\texttt{BND}\ b)) \qquad \neg(k \le b)}{b < p}\ \textsc{o\_level\_o\_shift\_bnd\_}2$$

Proof. By induction on i (proved separately in Isabelle). □

Lemma 42, f_level_f_shift_fpi, is similar to the first, showing that if an `FPI` node has level p when it is shifted by i then the type and body of the abstraction also have level p when they are shifted by i.

Lemma 42 (f_level_f_shift_fpi)**.**

$$\frac{\textsf{f\_level}\ p\ (\textsf{f\_shift}\ i\ k\ (\texttt{FPI}\ A\ B))}{\textsf{f\_level}\ p\ (\textsf{f\_shift}\ i\ k\ A)}\ \textsc{f\_level\_f\_shift\_fpi\_}1$$

$$\frac{\textsf{f\_level}\ p\ (\textsf{f\_shift}\ i\ k\ (\texttt{FPI}\ A\ B))}{\textsf{f\_level}\ p\ (\textsf{f\_shift}\ i\ k\ B)}\ \textsc{f\_level\_f\_shift\_fpi\_}2$$

Proof. By induction on i (proved separately in Isabelle). $\square$

Lemma 43, f_level_o_level_f_shift_o_shift_fapp, is similar to the second, showing that the constituent sub-term and sub-type of an FAPP node both have level p when they are shifted by i if the FAPP expression itself has level p when it is also shifted by i.

Lemma 43 (f_level_o_level_f_shift_o_shift_fapp)**.**

$$\frac{\mathsf{f\_level}\ p\ (\mathsf{f\_shift}\ i\ k\ (\mathtt{FAPP}\ A\ M))}{\mathsf{o\_level}\ p\ (\mathsf{o\_shift}\ i\ k\ M)}\ \textsc{f\_level\_o\_level\_f\_shift\_o\_shift\_fapp\_1}$$

$$\frac{\mathsf{f\_level}\ p\ (\mathsf{f\_shift}\ i\ k\ (\mathtt{FAPP}\ A\ M))}{\mathsf{f\_level}\ p\ (\mathsf{f\_shift}\ i\ k\ A)}\ \textsc{f\_level\_o\_level\_f\_shift\_o\_shift\_fapp\_2}$$

Proof. By induction on i (proved separately in Isabelle). $\square$

The next lemma that we need shows that if a term or type has level $p+q$ when it is shifted by q, then the term or type has level $p+q+1$ when shifted by level $q+1$.

Lemma 44 (o_level_f_level_succ)**.**

$$\frac{\mathsf{o\_level}\ (p+q)\ (\mathsf{o\_shift}\ q\ r\ M)}{\mathsf{o\_level}\ (p+q+1)\ (\mathsf{o\_shift}\ (q+1)\ r\ M)}\ \textsc{o\_level\_f\_level\_succ\_1}$$

$$\frac{\mathsf{f\_level}\ (p+q)\ (\mathsf{f\_shift}\ q\ r\ A)}{\mathsf{f\_level}\ (p+q+1)\ (\mathsf{f\_shift}\ (q+1)\ r\ A)}\ \textsc{o\_level\_f\_level\_succ\_2}$$

Proof. We proceed by mutual induction on M and A. The cases where $M = (\mathtt{CON}\ c), M = (\mathtt{VAR}\ v), M = \mathtt{ERR}, A = (\mathtt{FCON}\ c)$ and $A = \mathtt{FERR}$ are all trivial. For the case where $M = (\mathtt{ABS}\ A\ N)$ we have

$$\mathsf{o\_level}\ p+i\ (\mathsf{o\_shift}\ i\ k\ (\mathtt{ABS}\ A\ N))$$

and need to show

$$\mathsf{o\_level}\ (p+i+2)\ (\mathsf{o\_shift}\ (i+1)\ (k+1)\ N)$$

and

$$\mathsf{o\_level}\ (p+i+1)\ (\mathsf{o\_shift}\ (i+1)\ k\ M)$$

which we can do using lemma 39. For the case where $M = (\texttt{APP}\ M\ N)$ we have

$$\mathsf{o\_level}\ (p+i)\ (\mathsf{o\_shift}\ i\ k\ (\texttt{APP}\ M\ N))$$

and need to show

$$\mathsf{o\_level}\ (p+i+1)\ (\mathsf{o\_shift}\ (i+1)\ k\ M)$$

and

$$\mathsf{o\_level}\ (p+i+1)\ (\mathsf{o\_shift}\ (i+1)\ k\ N)$$

which we can do using lemma 40. Where $M = (\texttt{BND}\ b)$ we have two cases: $k \leq b$ and $\neg(k \leq b)$. These cases can be solved using lemma 41. For the case where $A = (\texttt{FPI}\ B\ C)$ we have

$$\mathsf{f\_level}\ (p+i)\ (\mathsf{f\_shift}\ i\ k\ \texttt{FPI}\ B\ C)$$

and we need to show

$$\mathsf{f\_level}\ (p+i+1)\ (\mathsf{f\_shift}\ (i+1)\ k\ B)$$

and

$$\mathsf{f\_level}\ (p+i+2)\ (\mathsf{f\_shift}\ (i+1)\ (k+1)\ C)$$

which we can do using lemma 42. Finally, for the case where $A = (\texttt{FPI}\ A\ M)$ we have

$$\mathsf{f\_level}\ (p+i)\ (\mathsf{f\_shift}\ i\ k\ (\texttt{FAPP}\ A\ M))$$

and need to show

$$\mathsf{f\_level}\ (p+i+1)\ (\mathsf{f\_shift}\ (i+1)\ k\ A)$$

and

$$\mathsf{o\_level}\ (p+i+1)\ (\mathsf{o\_shift}\ (i+1)\ k\ M)$$

which we can do using lemma 43. □

The next lemma shows that if a term or type has level p and is shifted by i then the resulting term or type has level $p+i$.

Lemma 45 (o_level_f_level_o_shift_f_shift)**.**

$$\frac{\mathsf{o\_level}\ p\ M}{\mathsf{o\_level}\ (p+i)\ (\mathsf{o\_shift}\ i\ k\ M)}\ \textsc{o\_level\_f\_level\_o\_shift\_f\_shift\_1}$$

$$\frac{\textsf{f\_level}\ p\ A}{\textsf{f\_level}\ (p+i)\ (\textsf{f\_shift}\ i\ k\ A)}\ \text{O\_LEVEL\_F\_LEVEL\_O\_SHIFT\_F\_SHIFT\_2}$$

Proof. By induction on i and lemma 44. □

Now we need a brief lemma showing that if a term or type is at level k and $k \leq p$ then the term or type is also at level p:

Lemma 46 (o_level_f_level_gt)**.**

$$\frac{\textsf{o\_level}\ k\ M \qquad k \leq p}{\textsf{o\_level}\ p\ M}\ \text{O\_LEVEL\_F\_LEVEL\_GT\_1}$$

$$\frac{\textsf{f\_level}\ k\ A \qquad k \leq p}{\textsf{f\_level}\ p\ A}\ \text{O\_LEVEL\_F\_LEVEL\_GT\_2}$$

Proof. By mutual induction on M and A. □

Lemma 47 (o_level_f_level_o_subst_f_subst_succ)**.**

$$\frac{\textsf{f\_level}\ (i+k+1)\ A \qquad \textsf{o\_level}\ l\ N}{\textsf{f\_level}\ i+k+l\ (A[N/i+k]^{k}_{\text{F'}})}\ \text{O\_LEVEL\_F\_LEVEL\_O\_SUBST\_F\_SUBST\_SUCC\_1}$$

$$\frac{\textsf{o\_level}\ (i+k+1)\ M \qquad \textsf{o\_level}\ l\ N}{\textsf{o\_level}\ i+k+l\ (M[N/i+k]^{k}_{\text{O'}})}\ \text{O\_LEVEL\_F\_LEVEL\_O\_SUBST\_F\_SUBST\_SUCC\_2}$$

Proof. By mutual induction on A and M, lemma 45 and lemma 46. □

We require two lemmas about substitution on kinds. The first concerns the function k_subst', and shows that if a term of level 0 is substituted for variable $p + p'$ in a term of level $p + p' + 1$, the result has level $p + p'$.

Lemma 48 (k_level_k_subst'_succ)**.**

$$\frac{\textsf{k\_level}\ (p+p'+1)\ K \qquad \textsf{o\_level}\ 0\ N}{\textsf{k\_level}\ (p+p')\ (K[N/(p+p')]^{p}_{\text{K'}})}\ \text{K\_LEVEL\_K\_SUBST'\_SUCC}$$

Proof. By induction on K and lemma 47. □

The next lemma is very similar to the previous lemma, but for `k_subst` rather than `k_subst'`.

Lemma 49 (k_level_k_subst_succ)**.**

$$\frac{\textsf{k\_level}\ (i+1)\ K \qquad \textsf{o\_level}\ 0\ M}{\textsf{k\_level}\ (i)\ (K[M/i]_{\text{K}})}\ \text{K\_LEVEL\_K\_SUBST\_SUCC}$$

Proof. By induction on K and lemmas 47 and 48. □

Finally we are in a position to prove theorem 38:

Theorem 38 (continuing from p. 40)**.**

$$\frac{\Gamma, \psi \vdash_\Sigma M : A}{\textsf{f\_level}\ 0\ A}\ \text{TYPEOF\_KINDOF\_DEF\_EQUAL\_VALIDKIND\_LEVEL\_1}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A : K}{\textsf{k\_level}\ 0\ K}\ \text{TYPEOF\_KINDOF\_DEF\_EQUAL\_VALIDKIND\_LEVEL\_2}$$

$$\frac{\Gamma, \psi \vdash_\Sigma M \equiv N : A}{\textsf{o\_level}\ 0\ M \wedge \textsf{o\_level}\ 0\ N \wedge \textsf{f\_level}\ 0\ A}\ \text{TYPEOF\_KINDOF\_DEF\_EQUAL\_VALIDKIND\_LEVEL\_3}$$

$$\frac{\Gamma, \psi \vdash_\Sigma A \equiv B : K}{\textsf{f\_level}\ 0\ A \wedge \textsf{f\_level}\ 0\ B \wedge \textsf{k\_level}\ 0\ K}\ \text{TYPEOF\_KINDOF\_DEF\_EQUAL\_VALIDKIND\_LEVEL\_4}$$

$$\frac{\Gamma, \psi \vdash_\Sigma K \equiv L}{\textsf{k\_level}\ 0\ K \wedge \textsf{k\_level}\ 0\ L}\ \text{TYPEOF\_KINDOF\_DEF\_EQUAL\_VALIDKIND\_LEVEL\_5}$$

$$\frac{\Gamma, \psi \vdash_\Sigma K}{\textsf{k\_level}\ 0\ K}\ \text{TYPEOF\_KINDOF\_DEF\_EQUAL\_VALIDKIND\_LEVEL\_6}$$

Proof. We proceed by mutual induction across the six relations. Of the 32 cases, 15 follow trivially from the hypotheses. The remaining cases can be solved simply, using lemmas 47, 49, 75 and 77. □

There are also canonical, atomic_of_type, atomic_of_kind and canonical_type relations. Contexts and binding environments are the same as in the typeof and kindof relations.

We use the notation defined in table 2.1.

Notation	Meaning
$\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} C : A$	The object C is canonical of type A
$\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} M : A$	The object M is atomic of type A
$\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} A : K$	The type A is atomic of kind K
$\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} T : \texttt{TYPE}$	The type T is a canonical type

Table 2.1: Canonicity and atomicity notation

Example 50. For example, if we have a signature Σ such that $\Sigma(a) = \texttt{TYPE}, \Sigma(b) = \texttt{TYPE}, \Sigma(c) = \texttt{FPI}\ (\texttt{FCON}\ a)\ (\texttt{FCON}\ b)$ and $\Sigma(d) = \texttt{FCON}\ a$, then given context Γ and binding environment ψ, we can derive a proof that $\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma}$ $\texttt{APP}\ (\texttt{CON}\ c)\ (\texttt{CON}\ d) : \texttt{FCON}\ b[(\texttt{CON}\ d)/0]_{\text{F}}$ as shown in figure 2.7.

The rules defining $\mathsf{canonical}$, $\mathsf{atomic\_of\_type}$, $\mathsf{atomic\_of\_kind}$ and $\mathsf{canonical\_type}$ are shown in figures 2.8 and 2.9.

In HybridLF there are object-level abstractions and type-level abstractions. Recall that a k-ary abstraction is a proper syntactic function with k bound meta-variables. We define functions $\mathsf{o\_abstr}$ and $\mathsf{f\_abstr}$ that determine if an object or type respectively is a valid unary abstraction. In the Isabelle implementation, $\mathsf{o\_abstr}$ and $\mathsf{f\_abstr}$ are directly defined as functions. This differs from the abst predicate in the original Hybrid [15], which was defined as an inductive relation that was then shown to define a function. By defining $\mathsf{o\_abstr}$ and $\mathsf{f\_abstr}$ using Isabelle's `function` construct it is possible to reduce the size of the definitions and proof necessary for their implementation. In this respect HybridLF follows the version of Hybrid produced by Martin [19].

Definition 51 (o_abstr)**.**

$$\begin{aligned}
\mathsf{o_abstr}\ i\ (\lambda x.\ x) &= \texttt{True} \\
\mathsf{o_abstr}\ i\ (\lambda x.\ \texttt{CON}\ a) &= \texttt{True} \\
\mathsf{o_abstr}\ i\ (\lambda x.\ \texttt{BND}\ n) &= (n < i) \\
\mathsf{o_abstr}\ i\ (\lambda x.\ \texttt{ERR}) &= \texttt{True} \\
\mathsf{o_abstr}\ i\ (\lambda x.\ \texttt{VAR}\ n) &= \texttt{True} \\
\mathsf{o_abstr}\ i\ (\lambda x.\ (f\ x)\ \$\$_{\text{O}}\ (g\ x)) &= (\mathsf{o_abstr}\ i\ f \wedge \mathsf{o_abstr}\ i\ g) \\
\mathsf{o_abstr}\ i\ (\lambda x.\ \texttt{ABS}\ (t\ x)(f\ x)) &= (\mathsf{f_abstr}\ i\ t \wedge \mathsf{o_abstr}\ (i+1)\ f) \\
\neg \mathsf{obj_ordinary}\ f \implies \mathsf{o_abstr}\ i\ f &= \texttt{False}
\end{aligned}$$

$$T = \frac{\Sigma(c) = \texttt{FPI}\ (\texttt{FCON}\ a)\ (\texttt{FCON}\ b) \qquad \textsf{f\_level}\ 0\ \texttt{FPI}\ (\texttt{FCON}\ a)\ (\texttt{FCON}\ b)}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} (\texttt{CON}\ c) : \texttt{FPI}\ (\texttt{FCON}\ a)\ (\texttt{FCON}\ b)}$$

$$T' = \frac{\dfrac{\Sigma(a) = \texttt{TYPE} \qquad \textsf{k\_level}\ 0\ \texttt{TYPE}}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} \texttt{FCON}\ a : \texttt{TYPE}} \qquad \dfrac{\Sigma(d) = \texttt{FCON}\ a \qquad \textsf{f\_level}\ 0\ \texttt{FCON}\ a}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} \texttt{CON}\ d : \texttt{FCON}\ a}}{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} \texttt{CON}\ d : \texttt{FCON}\ a}$$

$$T'' = \Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} (\texttt{FCON}\ b)[\texttt{CON}\ d/0]_{\text{F}} : \texttt{TYPE}$$

$$\frac{\dfrac{T \qquad T'}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} \texttt{APP}\ (\texttt{CON}\ c)\ (\texttt{CON}\ d) : (\texttt{FCON}\ b)[\texttt{CON}\ d/0]_{\text{F}}} \qquad T''}{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} \texttt{APP}\ (\texttt{CON}\ c)\ (\texttt{CON}\ d) : (\texttt{FCON}\ b)[\texttt{CON}\ d/0]_{\text{F}}}$$

Figure 2.7: Example canonicity judgement

$$\frac{\Sigma(c) = A \qquad \mathsf{f\_level}\ 0\ A}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} \texttt{CON}\ c : A}\ \text{ATM\_OBJ\_CON}$$

$$\frac{\Gamma(i) = A \qquad \mathsf{f\_level}\ 0\ A}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} \texttt{VAR}\ i : A}\ \text{ATM\_OBJ\_VAR}$$

$$\frac{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} M : \texttt{FPI}\ A\ B \qquad \Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} N : A}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} \texttt{APP}\ M\ N : B[N/0]_{\text{F}}}\ \text{ATM\_OBJ\_APP}$$

$$\frac{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} M : A \qquad \Gamma, \psi \vdash_{\Sigma} A \equiv B : \texttt{TYPE}}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} M : B}\ \text{ATM\_OBJ\_CNV}$$

$$\frac{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} A : \texttt{TYPE} \qquad \Gamma, A\#\psi \vdash^{\text{CAN}}_{\Sigma} M : B}{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} \texttt{ABS}\ A\ M : \texttt{FPI}\ A\ B}\ \text{CAN\_OBJ\_PI}$$

$$\frac{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} A : \texttt{TYPE} \qquad \Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} M : A}{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} M : A}\ \text{CAN\_OBJ\_ATM}$$

$$\frac{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} M : A \qquad \Gamma, \psi \vdash_{\Sigma} A \equiv B : \texttt{TYPE}}{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} M : B}\ \text{CAN\_OBJ\_CNV}$$

$$\frac{\Sigma(a) = K \qquad \mathsf{k\_level}\ 0\ K}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} \texttt{FCON}\ a : K}\ \text{ATM\_TY\_CON}$$

$$\frac{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} A : \texttt{KPI}\ B\ K \qquad \Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} M : B}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} \texttt{FAPP}\ A\ M : K[M/0]_{\text{K}}}\ \text{ATM\_TY\_APP}$$

$$\frac{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} A : K \qquad \Gamma, \psi \vdash_{\Sigma} K \equiv K'}{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} A : K'}\ \text{ATM\_TY\_CNV}$$

Figure 2.8: canonical, atomic_of_type, atomic_of_kind and canonical_type relations

$$\frac{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} A : \texttt{TYPE} \qquad \Gamma, A\#\psi \vdash^{\text{CAN}}_{\Sigma} B : \texttt{TYPE}}{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} \texttt{FPI}\ A\ B : \texttt{TYPE}}\ \text{CAN\_TY\_PI}$$

$$\frac{\Gamma, \psi \vdash^{\text{ATM}}_{\Sigma} a : \texttt{TYPE}}{\Gamma, \psi \vdash^{\text{CAN}}_{\Sigma} A : \texttt{TYPE}}\ \text{CAN\_TY\_ATM}$$

Figure 2.9: canonical, atomic_of_type, atomic_of_kind and canonical_type relations (cont.)

Definition 52 (f_abstr)**.**

$$\begin{aligned}
\mathsf{f_abstr}\ i\ (\lambda x.\ \texttt{FCON}\ a) &= \texttt{True}\\
\mathsf{f_abstr}\ i\ (\lambda x.\ \texttt{FERR}) &= \texttt{True}\\
\mathsf{f_abstr}\ i\ (\lambda x.\ \texttt{FPI}\ (t\ x)\ (f\ x)) &= (\mathsf{f_abstr}\ i\ t \wedge \mathsf{f_abstr}\ (i+1)\ f)\\
\mathsf{f_abstr}\ i\ (\lambda x.\ (f\ x)\ \$\$_{\text{F}}\ (g\ x)) &= (\mathsf{f_abstr}\ i\ f \wedge \mathsf{o_abstr}\ i\ g)\\
\neg\mathsf{fam_ordinary}\ f \Longrightarrow \mathsf{f_abstr}\ i\ f &= \texttt{False}
\end{aligned}$$

The natural number argument i is used to check for dangling variables in the case for BND. Note the use of the guards ¬obj_ordinary and ¬fam_ordinary; these allow us to exclude functions that do not match one of the other cases and are not a syntactic function. The definitions of obj_ordinary and fam_ordinary are as follows:

Definition 53 (obj_ordinary)**.**

$$\begin{aligned}
\mathsf{obj_ordinary}\ e \equiv (e &= (\lambda x.\ x)\\
&\vee\ e = (\lambda x.\ \texttt{ERR})\\
&\vee\ \exists n.\ e = (\lambda x.\ \texttt{CON}\ n)\\
&\vee\ \exists n.\ e = (\lambda x.\ \texttt{VAR}\ n)\\
&\vee\ \exists n.\ e = (\lambda x.\ \texttt{BND}\ n)\\
&\vee\ \exists f\ g.\ e = (\lambda x.\ (f\ x)\ \$\$_{\text{O}}\ (g\ x))\\
&\vee\ \exists f\ t.\ e = (\lambda x.\ \texttt{ABS}(t\ x)\ (f\ x)))
\end{aligned}$$

Definition 54 (fam_ordinary).

$$\begin{aligned}\mathsf{fam\_ordinary}\ e \equiv (&\exists n.\ e = (\lambda x.\mathtt{FCON}\ n)\\ &\vee e = (\lambda x.\ \mathtt{FERR})\\ &\vee \exists f\ g.\ e = (\lambda x.\ (f\ x)\ \$\$_{\mathrm{F}}\ (g\ x))\\ &\vee \exists f\ t.\ e = (\lambda x.\ \mathtt{FPI}\ (t\ x)\ (f\ x)))\end{aligned}$$

We further define instances of o_abstr, f_abstr, obj_ordinary and fam_ordinary for k-ary functions for values of k up to 8. Recall from definition 24 that a k-ary abstraction is a function with exactly k bound meta-variables and no dangling indices. We append the value of k to the names of these functions and definitions (so we have o_abstr . . . o_abstr8).

For example, the definitions of o_abstr3 and f_abstr3 are as follows:

Definition 55 (o_abstr3).

$$\begin{aligned}\mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ x) &= \mathtt{True}\\ \mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ y) &= \mathtt{True}\\ \mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ z) &= \mathtt{True}\\ \mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ \mathtt{CON}\ a) &= \mathtt{True}\\ \mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ \mathtt{BND}\ n) &= (n < i)\\ \mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ \mathtt{ERR}) &= \mathtt{True}\\ \mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ \mathtt{VAR}\ n) &= \mathtt{True}\\ \mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ (f\ x\ y\ z)\ \$\$_{\mathrm{O}}\ (g\ x\ y\ z)) &= (\mathsf{o\_abstr3}\ i\ f \wedge \mathsf{o\_abstr3}\ i\ g)\\ \mathsf{o\_abstr3}\ i\ (\lambda x\ y\ z.\ \mathtt{ABS}\ (t\ x\ y\ z)\ (f\ x\ y\ z)) &= (\mathsf{f\_abstr3}\ i\ t \wedge \mathsf{o\_abstr3}\ (i+1)\ f)\\ \neg\mathsf{obj\_ordinary3}\ f \Longrightarrow \mathsf{o\_abstr3}\ i\ f &= \mathtt{False}\end{aligned}$$

Definition 56 (f_abstr3).

$$\begin{aligned}\mathsf{f\_abstr3}\ i\ (\lambda x\ y\ z.\ \mathtt{FCON}\ a) &= \mathtt{True}\\ \mathsf{f\_abstr3}\ i\ (\lambda x\ y\ z.\ \mathtt{FERR}) &= \mathtt{True}\\ \mathsf{f\_abstr3}\ i\ (\lambda x\ y\ z.\ \mathtt{FPI}\ (t\ x\ y\ z)\ (f\ x\ y\ z)) &= (\mathsf{f\_abstr3}\ i\ t \wedge \mathsf{f\_abstr3}\ (i+1)\ f)\\ \mathsf{f\_abstr3}\ i\ (\lambda x\ y\ z.\ (f\ x\ y\ z)\ \$\$_{\mathrm{F}}\ (g\ x\ y\ z)) &= (\mathsf{f\_abstr3}\ i\ f \wedge \mathsf{o\_abstr3}\ i\ g)\\ \neg\mathsf{fam\_ordinary3}\ f \Longrightarrow \mathsf{f\_abstr3}\ i\ f &= \mathtt{False}\end{aligned}$$

We define functions obind' :: $\mathtt{nat} \to (\mathtt{expr} \to \mathtt{expr}) \to \mathtt{expr}$ and

$\mathsf{fbind'} :: \mathtt{nat} \to (\mathtt{expr} \to \mathtt{type}) \to \mathtt{type}$ that perform the conversion from Isabelle/HOL unary function abstraction to de Bruijn term. obind' and fbind' are defined as follows:

Definition 57 (obind')**.**

$$\begin{aligned}
\mathsf{obind'}\ i\ (\lambda x.\ x) &= (\mathtt{BND}\ i)\\
\mathsf{obind'}\ i\ (\lambda x.\ \mathtt{CON}\ a) &= (\mathtt{CON}\ a)\\
\mathsf{obind'}\ i\ (\lambda x.\ \mathtt{ABS}\ (t\ x)\ (f\ x)) &= \mathtt{ABS}\ (\mathsf{fbind'}\ i\ t)\ (\mathsf{obind'}\ (i+1)\ f)\\
\mathsf{obind'}\ i\ (\lambda x.\ \mathtt{APP}\ (f\ x)\ (g\ x)) &= \mathtt{APP}\ (\mathsf{obind'}\ i\ f)\ (\mathsf{obind'}\ i\ g)\\
\mathsf{obind'}\ i\ (\lambda x.\ \mathtt{BND}\ k) &= (\mathtt{BND}\ k)\\
\mathsf{obind'}\ i\ (\lambda x.\ \mathtt{VAR}\ n) &= (\mathtt{VAR}\ n)\\
\mathsf{obind'}\ i\ (\lambda x.\ \mathtt{ERR}) &= \mathtt{ERR}\\
\neg\mathsf{obj_ordinary}\ e \Longrightarrow \mathsf{obind'}\ i\ e &= \mathtt{ERR}
\end{aligned}$$

Definition 58 (fbind')**.**

$$\begin{aligned}
\mathsf{fbind'}\ i\ (\lambda x.\ \mathtt{FERR}) &= \mathtt{FERR}\\
\mathsf{fbind'}\ i\ (\lambda x.\ \mathtt{FCON}\ a) &= (\mathtt{FCON} a)\\
\mathsf{fbind'}\ i\ (\lambda x.\ \mathtt{FPI}(t\ x)\ (f\ x)) &= \mathtt{FPI}\ (\mathsf{fbind'}\ i\ t)(\mathsf{fbind'}\ (i+1)\ f)\\
\mathsf{fbind'}\ i\ (\lambda x.\ (f\ x)\ \$\$_{\mathrm{F}}\ (g\ x) &= (\mathsf{fbind'}\ i\ f)\ \$\$_{\mathrm{F}}\ (\mathsf{obind'}\ i\ g)\\
\neg\mathsf{fam_ordinary}\ e \Longrightarrow \mathsf{fbind'}\ i\ e &= \mathtt{FERR}
\end{aligned}$$

Again we define instances of obind' and fbind' for functions with up to 8 bound variables, appending the number of variables to the function name. We then create families of definitions obind and fbind, numbered in the same way as obind' and fbind'. By 'families' of definitions, all we mean is repeated definitions that perform the same task for different arguments, all given the same name except for a distinguishing natural number that indicates the argument that the definition applies to.

Definition 59 (obind)**.**

$$\mathsf{obind}\ t\ e \equiv \mathtt{if}\ \mathsf{o\_abstr}\ 0\ e\ \mathtt{then}\ \mathtt{ABS}\ t\ (\mathsf{obind'}\ 0\ e)\ \mathtt{else}\ \mathtt{ERR}$$

Definition 60 (fbind)**.**

$$\mathsf{fbind}\ t\ e \equiv \mathtt{if}\ \mathsf{f\_abstr}\ 0\ e\ \mathtt{then}\ \mathtt{FPI}\ t\ (\mathsf{fbind'}\ 0\ e)\ \mathtt{else}\ \mathtt{FERR}$$

obind checks if the function e is a valid term abstraction, returning ERR if it is not, and then creates an `ABS` node with t and the result of calling obind' on e. fbind works similarly, except that it calls f_abstr and fbind' on e and returns an `FPI` node if it is a type abstraction, or `FERR` if it is not.

The type argument t to fbind is not a function because there are no variables free in it: since it is the type of the first parameter of the `FPI` node, no `FPI`-bound variables that may appear. For fbind2 and above, the type of the second `FPI` node is a function with one variable, as the variable bound in the preceeding instance of `FPI` may appear in the type of the second instance. The type of the third `FPI` node in fbind3 is a function with two variables, and so on.

We give obind5 and fbind5 as an example:

Definition 61 (obind5).

obind5 $t1$ $t2$ $t3$ $t4$ $t5$ $e \equiv$ `if` o_abstr5 0 e $\wedge$ f_abstr 0 $t2$ $\wedge$ f_abstr2 0 $t3$ $\wedge$ f_abstr3 0 $t4$ $\wedge$ f_abstr4 0 $t5$ `then ABS` $t1$ (`ABS`(fbind' 0 $t2$) (`ABS` (fbind'2 0 $t3$) (`ABS` (fbind'3 0 $t4$) (`ABS` (fbind'4 0 $t5$) (obind'5 0 e))))) `else ERR`

Definition 62 (fbind5).

fbind5 $t1$ $t2$ $t3$ $t4$ $t5$ $e \equiv$ `if` f_abstr5 0 e $\wedge$ f_abstr 0 $t2$ $\wedge$ f_abstr2 0 $t3$ $\wedge$ f_abstr3 0 $t4$ $\wedge$ f_abstr4 0 $t5$ `then FPI` $t1$ (`FPI`(fbind' 0 $t2$) (`FPI` (fbind'2 0 $t3$) (`FPI` (fbind'3 0 $t4$) (`FPI` (fbind'4 0 $t5$) (fbind'5 0 e))))) `else FERR`

2.3.2 Families of functions?

Whether the approach of forming families of numbered functions is the best way of handling k-ary functions is a topic for further research. One possible such avenue of investigation would be to define o_abstr and f_abstr in a dependently-typed language such as Agda or Gallina (the specification language of the Coq theorem prover). In such a language, it is possible to define a function that returns a type of variable length and use it in the type of the o_abstr function, like in the following partial (Gallina) example:

Example 63.

```
Inductive expr : Set :=
| CON : expr
| BND : expr
| VAR : expr
| APP : expr -> expr -> expr
| ERR : expr
| ABS : expr -> expr.

Fixpoint build_ty (n : nat) :=
match n with
| 0 => expr
| S n' => expr -> build_ty n'
end.

Fixpoint o_abstr (n : nat) (i : nat) (x : build_ty (n - 1)) :=
match x with
| ?
end.
```

However, we cannot pattern-match on the argument `x` in o_abstr, as it does not have a fixed number of arguments. We might try to write a function that applies the argument to o_abstr to n elements of the `expr` datatype defined in example 63, where n is the arity.

Example 64.

```
Fixpoint o_apply (n : nat) (x : build_ty (n - 1)) :=
match n with
| 0 => x
| S m => (o_apply m (x CON))
end.
```

However, the function definition in example 64 is rejected by the type-checker, as statically the type of `x` is build_ty `(n - 1)`, not `expr` $\rightarrow \ldots \rightarrow$ `expr`. It is unclear whether it is possible to find a way around these problems to create dependently-typed $\mathsf{o\_abstr}$ and obind-like functions without creating an instance for each number of parameters in the function argument.

2.4 Induction rule

In the original version of Hybrid [15], Ambler et al create an induction rule for untyped terms of type `expr`.

The rule is as follows:

$$\frac{\begin{array}{c}\mathsf{proper}\ u \qquad \forall a.\ P\ (\texttt{CON}\ a) \qquad \forall n.\ P\ (\texttt{VAR}\ n) \\ \forall s. \forall t.\ \mathsf{proper}\ s \wedge P\ s \wedge \mathsf{proper}\ t\ \wedge P\ t \Longrightarrow P\ (s\ \$\$\ t) \\ \forall e.\ \mathsf{abstr}\ e \wedge \exists n.\ P\ (e\ (\texttt{VAR}\ n)) \Longrightarrow P\ (\texttt{LAM}\ x.\ e\ x)\end{array}}{P\ u}\ \textsc{proper\_VAR\_induct}$$

In Ambler et al's version of Hybrid proper is true if a term is at level 0, and abstr is true if a term is a unary abstraction. The `LAM` binder converts an abstraction into de Bruijn form.

As a comparison to this, we define a mutual-induction rule for HybridLF terms and types, adapting the method used by Ambler et al. Before we present the induction rule itself together with its proof, we need some auxillary lemmas.

Since we need to mutually induct over terms and types, we adapt Isabelle's built-in measure induction rule to perform mutual induction of two properties (P and Q):

Lemma 65 (mut_measure_induct)**.**

$$\frac{\begin{array}{c}\forall x.\ \forall y.\ f\ y < f\ x \longrightarrow P\ y \longrightarrow P\ x \\ \forall x.\ \forall y.\ g\ y < g\ x \longrightarrow Q\ y \longrightarrow Q\ x\end{array}}{P\ a \wedge Q\ b}\ \textsc{mut\_measure\_induct}$$

Proof. By measure induction. □

Isabelle automatically generates a size function on the `expr` and `type` datatypes for us.

Definition 66.

$$
\begin{aligned}
\mathsf{size}\ (\texttt{CON}\ a) &= 0\\
\mathsf{size}\ (\texttt{ABS}\ t\ e) &= \mathsf{size}\ t + \mathsf{size}\ e + 1\\
\mathsf{size}\ (\texttt{VAR}\ n) &= 0\\
\mathsf{size}\ (e_1\ \$\$_{\mathrm{O}}\ e_2) &= \mathsf{size}\ e_1 + \mathsf{size}\ e_2 + 1\\
\mathsf{size}\ (\texttt{BND}\ n) &= 0\\
\mathsf{size}\ \texttt{ERR} &= 0\\
\mathsf{size}\ (\texttt{FPI}\ t_1\ t_2) &= \mathsf{size}\ t_1 + \mathsf{size}\ t_2 + 1\\
\mathsf{size}\ (\texttt{FCON}\ b) &= 0\\
\mathsf{size}(t\ \$\$_{\mathrm{F}}\ e) &= \mathsf{size}\ t + \mathsf{size}\ e + 1\\
\mathsf{size}\ \texttt{FERR} &= 0
\end{aligned}
$$

Next we define functions o_inst and f_inst that instantiate a bound variable k with a term u.

Definition 67 (o_inst)**.**

$$
\begin{aligned}
\mathsf{o\_inst}\ k\ u\ (\texttt{CON}\ a) &= \texttt{CON}\ a\\
\mathsf{o\_inst}\ k\ u\ (\texttt{VAR}\ n) &= \texttt{VAR}\ n\\
\mathsf{o\_inst}\ k\ u\ (\texttt{BND}\ i) &= (\text{if } i = k \text{ then } u \text{ else } \texttt{BND}\ i)\\
\mathsf{o\_inst}\ k\ u\ (s\ \$\$_{\mathrm{O}}\ t) &= ((\mathsf{o\_inst}\ k\ u\ s)\ \$\$_{\mathrm{O}}\ (\mathsf{o\_inst}\ k\ u\ t))\\
\mathsf{o\_inst}\ k\ u\ (\texttt{ABS}\ t\ s) &= \texttt{ABS}\ (\mathsf{f\_inst}\ k\ u\ t)\ (\mathsf{o\_inst}\ (\mathrm{Suc}\ k)\ u\ s)\\
\mathsf{o\_inst}\ k\ u\ \texttt{ERR} &= \texttt{ERR}
\end{aligned}
$$

Definition 68 (f_inst)**.**

$$
\begin{aligned}
\mathsf{f\_inst}\ k\ u\ (\texttt{FCON}\ a) &= \texttt{FCON}\ a\\
\mathsf{f\_inst}\ k\ u\ \texttt{FERR} &= \texttt{FERR}\\
\mathsf{f\_inst}\ k\ u\ (\texttt{FPI}\ t\ s) &= \texttt{FPI}\ (\mathsf{f\_inst}\ k\ u\ t)\ (\mathsf{f\_inst}(\mathrm{Suc}\ k)\ u\ s)\\
\mathsf{f\_inst}\ k\ u\ (s\ \$\$_{\mathrm{F}}\ t) &= ((\mathsf{f\_inst}\ k\ u\ s)\ \$\$_{\mathrm{F}}\ (\mathsf{o\_inst}\ k\ u\ t))
\end{aligned}
$$

Next we need a lemma about the interaction between obind', o_inst, fbind' and f_inst, which is used in the proof of lemma 70:

Lemma 69 (obind_fbind_inst)**.**

$$\forall n.\ \mathsf{obind'}\ n\ (\lambda x.\mathsf{o\_inst}\ n\ x\ s) = s$$
$$\forall n.\ \mathsf{fbind'}\ n\ (\lambda y.\mathsf{f\_inst}\ n\ y\ t) = t$$

Proof. By induction and case analysis. □

Lemma abs_inst demonstrates a property of the interaction between o_inst, f_inst and obind, which is used in the proof of the induction rule in theorem 80.

Lemma 70 (abs_inst)**.**

$$\frac{\mathsf{o\_abstr}\ 0\ (\lambda x.\ \mathsf{o\_inst}\ 0\ x\ s) \quad \mathsf{f\_abstr}\ 0\ (\lambda y.\ \mathsf{f\_inst}\ 0\ y\ t)}{\text{obind}\ (\lambda x.\ \mathsf{f\_inst}\ 0\ x\ t)(\lambda y.\ \mathsf{o\_inst}\ 0\ y\ s) = \mathtt{ABS}\ t\ s}\ \text{ABS\_INST}$$

Proof. By lemma 69 and definition 59. □

The next lemma proves that when a term or type has level n, the function formed from using o_inst or f_inst to replace the bound variable with index $n-1$ with a lambda-abstracted parameter is an abstraction of level $n-1$:

Lemma 71 (o_level_f_level_abstr)**.**

$$\frac{\mathsf{o\_level}\ (\text{Suc}\ n)\ s}{\mathsf{o\_abstr}\ n\ (\lambda x.\ \mathsf{o\_inst}\ n\ x\ s)}\ \text{O\_LEVEL\_ABSTR}$$

$$\frac{\mathsf{f\_level}\ (\text{Suc}\ k)\ t}{\mathsf{f\_abstr}\ k\ (\lambda y.\ \mathsf{f\_inst}\ k\ y\ t)}\ \text{F\_LEVEL\_ABSTR}$$

Proof. By induction and case analysis. □

The following lemma shows the connection between terms and types of level 0 and the abstractions formed using o_inst and f_inst to substitute the bound variable with index 0:

Lemma 72 (o_level_f_level_0_abstr)**.**

$$\frac{\mathsf{o\_level}\ 0\ e}{\mathsf{o\_abstr}\ 0\ (\lambda x.\ \mathsf{o\_inst}\ 0\ x\ e)}\ \text{O\_LEVEL\_0\_ABSTR}$$

$$\frac{\mathsf{f\_level}\ 0\ t}{\mathsf{f\_abstr}\ 0\ (\lambda y.\ \mathsf{f\_inst}\ 0\ y\ t)}\ \text{F\_LEVEL\_0\_ABSTR}$$

Proof. By induction. □

The next lemma concerns the size of terms when a `VAR` constructor is substituted in using o_inst or f_inst:

Lemma 73 (size_o_inst_f_inst_VAR)**.**

$$\forall i.\ \text{size } (\text{o\_inst } i\ (\texttt{VAR}\ n)\ s = \mathsf{size}\ s$$
$$\forall j.\ \text{size } (\text{f\_inst } j\ (\texttt{VAR}\ n)\ t = \mathsf{size}\ t$$

Proof. By induction. ☐

o_level_f_level_succ demonstrates that terms or types at level n are also at level $n+1$:

Lemma 74 (o_level_f_level_succ)**.**

$$\frac{\mathsf{o\_level}\ n\ e}{\mathsf{o\_level}\ (n+1)\ e}\ \text{O\_LEVEL\_SUCC}$$

$$\frac{\mathsf{f\_level}\ k\ t}{\mathsf{f\_level}\ (k+1)\ t}\ \text{F\_LEVEL\_SUCC}$$

Proof. By induction. ☐

The next lemma shows that when a term or type is at level 0, it is also at any arbitrary level n:

Lemma 75 (o_level_f_level_0)**.**

$$\frac{\mathsf{o\_level}\ 0\ e}{\mathsf{o\_level}\ n\ e}\ \text{O\_LEVEL\_0}$$

$$\frac{\mathsf{f\_level}\ 0\ t}{\mathsf{f\_level}\ k\ t}\ \text{F\_LEVEL\_0}$$

Proof. By induction and lemma 74. ☐

The next lemma is equivalent to lemma 74, but for kinds instead of terms and types:

Lemma 76 (k_level_succ)**.**

$$\frac{\mathsf{k\_level}\ n\ k}{\mathsf{k\_level}\ (n+1)\ k}\ \text{K\_LEVEL\_SUCC}$$

Proof. By induction on k and lemma 74. ☐

k_level_0 is again equivalent to lemma 75, but for kinds:

Lemma 77 (k_level_0)**.**

$$\frac{\mathsf{k\_level}\ 0\ K}{\mathsf{k\_level}\ n\ K}\ \text{K\_LEVEL\_0}$$

Proof. By induction on n and lemma 76. □

The next lemma proves a property of the interaction between o_level and f_level, o_inst and f_inst:

Lemma 78 (level_inst)**.**

$$\frac{\mathsf{o\_level}\ (\mathrm{Suc}\ j)\ s \quad \mathsf{o\_level}\ 0\ q}{\mathsf{o\_level}\ j\ (\mathsf{o\_inst}\ j\ q\ s)}\ \text{O\_LEVEL\_INST}$$

$$\frac{\mathsf{f\_level}\ (\mathrm{Suc}\ j)\ t \quad \mathsf{o\_level}\ 0\ r}{\mathsf{f\_level}\ j\ (\mathsf{f\_inst}\ j\ r\ t)}\ \text{F\_LEVEL\_INST}$$

Proof. By induction and lemma 75. □

The following lemma concerns the level of a term when a free variable is substituted in:

Lemma 79 (o_level_f_level_VAR)**.**

$$\frac{\text{o\_level}\ (\mathrm{Suc}\ k)\ s}{\mathsf{o\_level}\ k\ (\mathsf{o\_inst}\ k\ (\mathtt{VAR}\ a)\ s)}\ \text{O\_LEVEL\_VAR}$$

$$\frac{\mathsf{f\_level}\ (\mathrm{Suc}\ n)\ t}{\mathsf{f\_level}\ n\ (\mathsf{f\_inst}\ n\ (\mathtt{VAR}\ b)\ t)}\ \text{F\_LEVEL\_VAR}$$

Proof. By lemma 78. □

We are now in a position to state and prove the induction rule itself.

Theorem 80 (var_induct)**.**

$$\frac{\begin{array}{c} \mathsf{o\_level}\ 0\ e \qquad \mathsf{f\_level}\ 0\ t \qquad \forall a.\ P\ (\mathtt{CON}\ a) \\ \forall n.\ P\ (\mathtt{VAR}\ n) \qquad \forall s\ t.\ P\ s \wedge P\ t \longrightarrow P\ (s\ \$\$_{\mathrm{O}}\ t) \\ \forall s\ t.\ \mathsf{o\_abstr}\ 0\ s \wedge \forall n.P\ (s\ (\mathtt{VAR}\ n)) \wedge \mathsf{f\_abstr}\ 0\ t \longrightarrow P\ (\mathrm{obind}\ t\ s) \\ P\ \mathtt{ERR} \qquad \forall a.\ Q\ (\mathtt{FCON}\ a) \\ \forall a\ b.\ Q\ a \longrightarrow Q\ (a\ \$\$_{\mathrm{F}}\ b) \qquad Q\ \mathtt{FERR} \\ \forall s\ t.\ (\mathsf{f\_abstr}\ 0\ s \wedge \forall n.\ Q\ (s\ (\mathtt{VAR}\ n)) \wedge \mathsf{f\_abstr}\ 0\ t \wedge \\ \forall n.\ Q\ (t\ (\mathtt{VAR}\ n))) \longrightarrow Q\ (\mathsf{fbind}\ t\ s) \end{array}}{P\ e \wedge Q\ t}\ \text{VAR\_INDUCT}$$

Proof. We proceed by measure induction on the size of terms, using lemma 65 and definition 66. To prove Pe we perform case analysis over e. P ($\texttt{CON}\ c$), P ($\texttt{VAR}\ a$) and P $\texttt{ERR}$ are trivially true from the hypotheses. For $e = (s\ \$\$_{\mathrm{o}}\ t)$ as $\mathsf{size}\ s < \mathsf{size}\ (s\ \$\$\ t)$ and $\mathsf{size}\ t < \mathsf{size}\ (s\ \$\$_{\mathrm{o}}\ t)$ we have $\mathsf{o\_level}\ 0\ s \longrightarrow P\ s$ and $\mathsf{o\_level}\ 0\ t \longrightarrow P\ t$. Since we know from the hypotheses $\mathsf{o\_level}\ 0\ (s\ \$\$_{\mathrm{o}}\ t)$ from definition 32 we have $\mathsf{o\_level}\ 0\ s$ and $\mathsf{o\_level}\ 0\ t$, and hence $P\ s$ and $P\ t$. Since we have that $\forall s\ t.\ P\ s \wedge P\ t \longrightarrow P\ (s\ \$\$_{\mathrm{o}}\ t)$, we have $P\ (s\ \$\$_{\mathrm{o}}\ t)$ as required. If $e = (\texttt{BND}\ n)$ we have from the hypotheses $\mathsf{o\_level}\ 0\ (\texttt{BND}\ n)$, which by definition 32 can only be true if $n < 0$ and hence cannot be true, allowing us to conclude $P\ (\texttt{BND}\ n)$.

The most interesting case is when $e = (\texttt{ABS}\ u\ f)$. We have from the hypotheses that

$$\forall s\ t.\ \mathsf{o\_abstr}\ 0\ s \wedge \forall n.P\ (s\ (\texttt{VAR}\ n)) \wedge \mathsf{f\_abstr}\ 0\ t \longrightarrow P\ (\text{obind}\ t\ s)$$

Since in the hypotheses for this case we have $\mathsf{o\_abstr}\ 0\ e$ and $\mathsf{f\_abstr}\ 0\ t$, definition 59 becomes simply $\texttt{ABS}$ ($\mathsf{fbind'}\ 0\ t$) ($\mathsf{obind'}\ 0\ e$) so we have

$$\forall s\, t.\ \mathsf{o\_abstr}\, 0\, s \wedge \forall n.P\, (s\, (\texttt{VAR}\, n)) \wedge \mathsf{f\_abstr}\, 0\, t \longrightarrow P\ \texttt{ABS}\ (\mathsf{fbind'}\ 0\ t)\ (\mathsf{obind'}\ 0\ e)$$

.

Since by lemma 69

$$\mathsf{obind'}\ n\ (\lambda x.\ \mathsf{o\_inst}\ n\ x\ s) = s$$

and

$$\mathsf{fbind'}\ n\ (\lambda y.\ \mathsf{f\_inst}\ n\ y\ t) = t$$

we only need to show

$$P\ (\texttt{ABS}\ (\mathsf{fbind'}\ 0\ (\lambda x.\ \mathsf{f\_inst}\ 0\ x\ u))\ (\mathsf{obind'}\ 0\ (\lambda x.\ \mathsf{o\_inst}\ 0\ x\ f)))$$

Using lemma 71 and lemma 72 we can conclude $\mathsf{o\_abstr}\ 0\ (\lambda x.\ \mathsf{o\_inst}\ 0\ x\ f)$ and $\mathsf{f\_abstr}\ 0\ (\lambda y.\ \mathsf{f\_inst}\ 0\ y\ u)$. Now we need to show that P holds for $((\lambda x.\ \mathsf{o\_inst}\ 0\ x\ f)(\texttt{VAR}\ a))$. To do this we use the induction hypothesis. Since by lemma 73 $\mathsf{size}\ (\mathsf{o\_inst}\ 0\ (\texttt{VAR}\ a)\ f) = \mathsf{size}\ f$ and since by definition 66 $\mathsf{size} f < (\mathsf{size} f + \mathsf{size}\ u)$ we have that

$$\mathsf{o\_level}\ 0\ (\mathsf{o\_inst}\ 0\ (\texttt{VAR}\ a)\ f) \longrightarrow P\ (\mathsf{o\_inst}\ 0\ (\texttt{VAR}\ a)\ f)$$

Now we can use lemma 79 to conclude $\mathsf{o\_level}\ 0\ (\mathsf{o\_inst}\ 0\ (\texttt{VAR}\ a)\ f)$ and therefore P ($\mathsf{o\_inst}\ 0\ (\mathtt{VAR}\ a)\ f$).

We therefore have

$$P\ (\mathtt{ABS}\ (\mathsf{fbind'}\ 0\ (\lambda x.\ \mathsf{f\_inst}\ 0\ x\ u))\ (\mathsf{obind'}\ 0\ (\lambda x.\ \mathsf{o\_inst}\ 0\ x\ f)))$$

and thus $P\ (\mathtt{ABS}\ u\ f)$.

The proof of $Q\ t$ is very similar to that of $P\ e$, again by case analysis and using the same lemmas and definitions.

□

2.5 Chapter summary

In this chapter we have described HybridLF, a version of Hybrid that implements the metatheory of LF.

Although HybridLF is an effective implementation of LF, allowing easy creation of LF signatures through a HOAS interface, there are obstacles to its practical use in proving meta-theorems about these signatures. The first such obstacle is the presence of terms that are not in canonical form. Such terms do not correspond to any term of the object logic. Given the signature for the natural numbers in example 1, an example of such a term is (λ`x. succ x`) `zero`, which does not correspond to any natural number (but is definitionally equivalent to `succ zero`). As we will see in chapter 5, terms that are not in canonical form cause problems when it comes to the process of unification that is necessary during proof search. In the next chapter we will discuss another system, Canonical HybridLF, which is based upon the canonical presentation of LF, in which such terms do not occur.

The implementation of many HybridLF judgements as relations is also difficult when using the system to create proofs. When we need to compute the type of a term, for example, it is in many cases necessary to apply a number of rules step-by-step to come to the result. The main alternative to implementing judgements as relations would be to use the Isabelle `fun` or `function` constructs, which would allow the result to be calculated automatically by the simplifier. However, this approach may not be possible for HybridLF, as Isabelle requires all functions to be total and provably terminating.

Chapter 3

Canonical HybridLF

3.1 Introduction

As we mentioned in section 1.3, one of the main considerations when representing an object logic in LF is adequacy. An encoding of an object logic is adequate if the translation function between the object logic and the canonical forms of the LF encoding is a compositional bijection. The presence of objects and types that are not in canonical form in the LF type theory requires the establishment of definitional equality relations: every term or type that is not in canonical form is definitionally equivalent to a term or type in canonical form. Since the canonical forms are β-reduced, η-long, the definitional equality relations are based around β-reduction and η-conversion. As we discussed in section 2.5, having to reason about definitional equality and terms that are not in canonical form can be a burden when reasoning about object logics in LF.

Canonical HybridLF is a version of HybridLF that is based on the canonical presentation of LF [38]. The key property of Canonical LF is that it is only possible to construct canonical terms due to restrictions on the grammar. As a result, definitional equality is reduced to syntactic equality - there is no need for the definitional equality relations used in LF. The main disadvantage of Canonical LF is that substitution is complicated with the requirement that the result of substitution is a canonical term - a notion of *hereditary* substitution is introduced to resolve this problem.

Hereditary substition essentially ensures that the result of substituting a canonical term into another canonical term is itself a canonical term. The key rule of hereditary substitution concerns substitution of a term for free variables in an application $R\ M$. In most traditional definitions of substitution when the result of substition on the first subterm R of the application is a lambda abstraction the result would not be in canonical form as it would be a β-redex. In hereditary substitution we go a step further, and the term is β-reduced during substition with the result of substitution on the second subterm M being substituted into the lambda abstraction itself, resulting in a term that is in canonical form.

In section 3.2 we define the grammar of Canonical LF and give its signature, context, typing and kinding judgements. In section 3.3 we discuss hereditary substitution, and in section 3.4 we review the implementation of CANONICAL HYBRIDLF.

3.2 The language of Canonical LF

In addition to the standard definition of kinds, the grammar of canonical LF introduces syntactic categories of atomic and canonical type families (denoted by P and A respectively) and atomic and canonical terms (denoted by R and M respectively):

Definition 81.

$$\begin{aligned}
K &::= \mathrm{Type} \mid \Pi x{:}A.K \\
A &::= P \mid \Pi x{:}A.A \\
P &::= a \mid P\ M \\
M &::= R \mid \lambda x.M \\
R &::= x \mid c \mid R\ M
\end{aligned}$$

We use a to denote a type-level constant, c to denote a term-level constant and x to denote a variable. Note that the definition of term-level application precludes the formation of β-redices: since the first operand of an application must be an atomic term, we can never construct a term such as $(\lambda x.x)\ c$.

Signatures and contexts are defined like so:

Definition 82.

$$\begin{aligned}
\Sigma &::= \langle\rangle \mid \Sigma, a{:}K \mid \Sigma, c{:}A \\
\Gamma &::= \langle\rangle \mid \Gamma, x{:}A
\end{aligned}$$

The judgements K kind and A type are defined in figure 3.1.

The judgements Γ ctx and Σ sig indicate valid contexts and signatures. They are defined in figure 3.2.

The typing rules for Canonical LF are shown in figure 3.3.

$$\frac{}{\Gamma \vdash_\Sigma \ \text{Type kind}}\ \text{TYPE\_KIND} \qquad \frac{\Gamma \vdash_\Sigma \ A : \text{Type}}{A \text{ type}}\ \text{PI\_TYPE}$$

$$\frac{\Gamma \vdash_\Sigma \ A \text{ type} \qquad \Gamma, x{:}A \vdash_\Sigma \ K \text{ kind}}{\Gamma \vdash_\Sigma \ \Pi x{:}A.K \text{ kind}}\ \text{PI\_KIND}$$

$$\frac{\Gamma \vdash_\Sigma \ A' \text{ type} \qquad \Gamma, x{:}A' \vdash_\Sigma \ A \text{ type}}{\Gamma \vdash_\Sigma \ \Pi x{:}A'.A \text{ type}}\ \text{PI\_TYPE}$$

Figure 3.1: Canonical LF kind and type judgements

$$\frac{\vdash_\Sigma \ A \text{ type} \qquad \Gamma \text{ ctx} \qquad x \notin \Gamma}{\vdash_\Sigma \ \Gamma, x{:}A \text{ ctx}}\ \text{CTX\_VAR}$$

$$\frac{}{\vdash_\Sigma \ \langle\rangle \text{ ctx}}\ \text{CTX\_EMPTY} \qquad \frac{}{\vdash_\Sigma \ \langle\rangle \text{ sig}}\ \text{SIG\_EMPTY}$$

$$\frac{\vdash_\Sigma \ A \text{ type} \qquad \Sigma \text{ sig} \qquad c \notin \Sigma}{\vdash_\Sigma \ \Sigma, c{:}A \text{ sig}}\ \text{SIG\_OBJ\_CON}$$

$$\frac{\vdash_\Sigma \ K \text{ kind} \qquad \Sigma \text{ sig} \qquad a \notin \Sigma}{\vdash_\Sigma \ \Sigma, a{:}K \text{ sig}}\ \text{SIG\_TY\_CON}$$

Figure 3.2: Canonical LF ctx and sig judgements

$$\frac{a{:}K \in \Sigma}{\Gamma \vdash_\Sigma \; a : K}\ \text{CON\_AT\_KIND} \qquad \frac{\Gamma, x{:}A \vdash_\Sigma \; M : A'}{\Gamma \vdash_\Sigma \; \lambda x.M : \Pi x{:}A.A'}\ \text{ABS\_CAN\_TY}$$

$$\frac{\Gamma \vdash_\Sigma \; P : \Pi x{:}A.K \qquad \Gamma \vdash_\Sigma \; M : A \qquad [M/x]^k_A K = K'}{\Gamma \vdash_\Sigma \; P\ M : K'}\ \text{APP\_AT\_KIND}$$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash_\Sigma \; x : A}\ \text{VAR\_AT\_TY} \qquad \frac{c{:}A \in \Sigma}{\Gamma \vdash_\Sigma \; c : A}\ \text{CON\_AT\_TY}$$

$$\frac{\Gamma \vdash_\Sigma \; R : \Pi x{:}A.A' \qquad \Gamma \vdash_\Sigma \; M : A \qquad [M/x]^a_A A' = A''}{\Gamma \vdash_\Sigma \; R\ M : A''}\ \text{APP\_AT\_TY}$$

Figure 3.3: Canonical LF typing judgements

3.3 Hereditary substitution

The principal complication of Canonical LF is that since only canonical terms can exist (as a result of the grammar in definition 81), we must ensure that the result of substituting one canonical term into another canonical term is also canonical. Problems may arise when the result of substituting the first operand of an application is an abstraction, which would normally result in a term that is not in canonical form. The solution to this problem is to further substitute the result of substitution on the second operand of the application into the body of the abstraction, so that a canonical term results. In this way we can avoid the creation of terms that are not in canonical form.

Example 83. So for example if we had $[(\lambda x.\ x)/y]^r_\alpha(yM)$, and $[(\lambda x.\ x)/y]^m_\alpha M = M'$ the result of simply substituting $(\lambda x.\ x)$ for y would be $(\lambda x.\ x)\ M'$, which is not a valid term formed by the grammar in definition 81. As a result, we must substitute M' for x.

This requires a particular definition of substitution, known as *hereditary substitution*. We adapt the presentation of hereditary substitution set out by Harper and Licata [24].

We use the notation $[M/x]^m_\alpha M' = M''$ to indicate the hereditary substitution of the object M for free occurrences of the variable x with simple type α in the object M', resulting in the object M''. The superscript m is used simply to indicate which category of the grammar we are substituting over.

Simple types are defined as follows:

Definition 84.

$$\alpha ::= a \mid \alpha \to \alpha$$

Hereditary substitution is defined in figures 3.4 and 3.5.

3.4 Canonical HybridLF

3.4.1 Datatypes

Canonical HybridLF is based around 5 datatypes that define the syntactic categories of kinds (`kind`), canonical types (`ctype`), atomic types (`atype`), canonical terms (`cterm`) and atomic terms (`aterm`).

Definition 85.

```
datatype ('a, 'b) kind = TYPE
                       | KPI"('a, 'b) ctype" "('a, 'b) kind"
    and ('a, 'b) ctype = PI "('a, 'b) ctype" "('a, 'b) ctype"
                       | ATYPE "('a, 'b) atype"
    and ('a, 'b) atype = FCON 'b
                       | FAPP "('a, 'b) atype" "('a, 'b) cterm"
                           (infixl "$$T" 50)
    and ('a, 'b) cterm = ABS "('a, 'b) ctype" "('a, 'b) cterm"
                       | ATERM "('a, 'b) aterm"
    and ('a, 'b) aterm = VAR nat
                       | BND nat
                       | CON 'a
                       | APP "('a, 'b) aterm" "('a, 'b) cterm"
                           (infixl "$$O" 50)
```

Note the `ATERM` and `ATYPE` constructors that bring the atomic terms into the syntactic category of canonical terms and atomic types into the syntactic category of canonical types.

$$\frac{[M/x]^a_\alpha A = A' \qquad [M/x]^k_\alpha K = K'}{[M/x]^k_\alpha \Pi x{:}A.K = \Pi x{:}A'.K'}\ \textsc{kpi\_k\_subst}$$

$$\frac{}{[M/x]^k_\alpha \text{Type} = \text{Type}}\ \textsc{type\_k\_subst} \qquad \frac{[M/x]^p_\alpha P = P'}{[M/x]^a_\alpha P = P'}\ \textsc{atom\_cty\_subst}$$

$$\frac{[M/x]^a_\alpha A = A'' \qquad [M/x]^a_\alpha A' = A'''}{[M/x]^a_\alpha \Pi x{:}A.A' = \Pi x{:}A''.A'''}\ \textsc{pi\_cty\_subst}$$

$$\frac{}{[M/x]^p_\alpha a = a}\ \textsc{con\_aty\_subst}$$

$$\frac{[M/x]^p_\alpha P = P' \qquad [M/x]^m_\alpha M = M'}{[M/x]^p_\alpha P\ M = P'\ M'}\ \textsc{app\_aty\_subst}$$

$$\frac{[M/x]^r_\alpha R = M' : \alpha}{[M/x]^m_\alpha R = M'}\ \textsc{atom\_can\_res\_ctm\_subst}$$

$$\frac{[M/x]^r_\alpha R = R'}{[M/x]^m_\alpha R = R'}\ \textsc{atom\_ctm\_subst} \qquad \frac{[M/x]^m_\alpha M = M'}{[M/x]^m_\alpha \lambda x.M = \lambda x.M'}\ \textsc{abs\_ctm\_subst}$$

$$\frac{}{[M/x]^r_\alpha x = M : \alpha}\ \textsc{var\_can\_res\_atm\_subst}$$

$$\frac{\begin{array}{c}[M/x]^r_\alpha R = \lambda x.M' : \alpha' \to \alpha'' \\ [M/x]^m_\alpha M'' = M''' \qquad [M'''/x]^m_{\alpha'} M' = M''''\end{array}}{[M/x]^r_\alpha R\ M'' = M'''' : \alpha''}\ \textsc{app\_can\_res\_atm\_subst}$$

Figure 3.4: Canonical LF hereditary substitution judgement

$$\frac{x \neq x'}{[M/x]^r_\alpha x' = x'}\ \text{VAR\_ATM\_SUBST} \qquad \frac{}{[M/x]^r_\alpha c = c}\ \text{CON\_ATM\_SUBST}$$

$$\frac{[M/x]^r_\alpha R = R' \qquad [M/x]^m_\alpha M' = M''}{[M/x]^r_\alpha R\ M' = R'\ M''}\ \text{APP\_ATM\_SUBST}$$

Figure 3.5: Canonical LF hereditary substitution judgement (cont.)

3.4.2 Contexts, signatures and binding environments

Contexts are defined as a list of pairs of natural numbers and canonical types.

```
type_synonym ('a, 'b) ctx = "(nat × ('a, 'b) ctype) list"
```

As in HybridLF, the signature is split into two: `sig_t` that defines object constants and `sig_k` that defines type constants. Also similar to HybridLF is the manner in which the datatypes making up the syntax of Canonical HybridLF are parameterised with two datatypes that provide the object and type level constant symbols. The signature therefore consists of two lists of pairs of object constants and the corresponding type or kind:

```
type_synonym ('a, 'b) sig_t = "('a × ('a, 'b) ctype) list"
type_synonym ('a, 'b) sig_k = "('b × ('a, 'b) kind) list"
```

When considering a term in de Bruijn form, binding environments provide the ability to determine the type of binders that enclose the term and hence allow the type of the term to be calculated. They are implemented as a list of canonical types, with the nth enclosing binder at position n in the list.

```
type_synonym ('a, 'b) bndenv = "(('a, 'b) ctype) list"
```

3.4.3 Levels and shifting

We define cterm_level, ctype_level, aterm_level, atype_level and kind_level functions that determine if a given canonical term, canonical type, atomic term or

Name	Purpose
cterm_level	Determine if a canonical term is at a given level
aterm_level	Determine if an atomic term is at a given level
ctype_level	Determine if a canonical type is at a given level
atype_level	Determine if an atomic type is at a given level
kind_level	Determine if a kind is at a given level
cterm_shift	Shift the bound variables in a canonical term
aterm_shift	Shift the bound variables in an atomic term
ctype_shift	Shift the bound variables in a canonical type
atype_shift	Shift the bound variables in an atomic type
validkind	Determine if a kind is valid
validtype	Determine if a type is valid
canon_typeof	Find the type of a canonical term
atom_typeof	Find the type of an atomic term
atom_kindof	Find the type of an atomic term
cterm_subst_bv	Substitute a canonical term for a bound variable in a canonical term
aterm_subst_bv	Substitute a canonical term for a bound variable in an atomic term resulting in an atomic term
ctype_subst_bv	Substitute a canonical term for a bound variable in a canonical type
aterm_subst_bv	Substitute a canonical term for a bound variable in an atomic type
aterm_can_subst_bv	Substitute a canonical term for a bound variable in an atomic term resulting in a canonical term
kind_subst_bv	Substitute a canonical term for a bound variable in a kind
ctx_subst_bv	Substitute a canonical term for a bound variable in a context
cterm_subst_fv	Substitute a canonical term for a free variable in a canonical term
aterm_subst_fv	Substitute a canonical term for a free variable in an atomic term
ctype_subst_fv	Substitute a canonical term for a free variable in a canonical type
aterm_subst_fv	Substitute a canonical term for a free variable in an atomic type resulting in an atomic term
aterm_can_subst_fv	Substitute a canonical term for a free variable in an atomic term resulting in a canonical term
kind_subst_fv	Substitute a canonical term for a free variable in a kind
ctx_subst_fv	Substitute a canonical term for a free variable in a context

Table 3.1: Functions defined in Canonical HybridLF

Name	**Purpose**
cterm_ordinary to cterm_ordinary12	A family of functions, determining if a function returning a canonical term is syntactic
aterm_ordinary to aterm_ordinary12	A family of functions, determining if a function returning an atomic term is syntactic
ctype_ordinary to ctype_ordinary12	A family of functions, determining if a function returning a canonical type is syntactic
atype_ordinary to atype_ordinary12	A family of functions, determining if a function returning an atomic type is syntactic
cterm_bind' to cterm_bind'12	A family of functions, used during the conversion of a HOAS function to a canonical term
aterm_bind' to aterm_bind'12	A family of functions, used during the conversion of a HOAS function to an atomic term
ctype_bind' to ctype_bind'12	A family of functions, used during the conversion of a HOAS function to a canonical type
atype_bind' to atype_bind'12	A family of functions, used during the conversion of a HOAS function to an atomic type
cterm_abstr to cterm_abstr12	A family of functions, determining if a given function that returns a canonical term represents a valid abstraction
aterm_abstr to aterm_abstr12	A family of functions, determining if a given function that returns an atomic term represents a valid abstraction
ctype_abstr to ctype_abstr12	A family of functions, determining if a given function that returns a canonical type represents a valid abstraction
atype_abstr to atype_abstr12	A family of functions, determining if a given function that returns an atomic type represents a valid abstraction
cterm_bind to cterm_bind12	A family of functions, converting a HOAS function to a canonical term
ctype_bind to ctype_bind12	A family of functions, converting a HOAS function to a canonical type

Table 3.2: Functions defined in CANONICAL HYBRIDLF (cont.)

atomic type are at a given level. The level indicates the number of ABS or PI binders that would have to be added to the start of the term to ensure that there are no dangling indices. Terms at level 0 have no dangling indices.

Definition 86 (Level functions).

$$\begin{aligned}
\mathsf{cterm_level}\ k\ (\mathtt{ABS}\ t\ f) &= (\mathsf{ctype_level}\ k\ t \wedge \mathsf{cterm_level}\ (k+1)\ f) \\
\mathsf{cterm_level}\ k\ (\mathtt{ATERM}\ a) &= \mathsf{aterm_level}\ k\ a
\end{aligned}$$

$$\begin{aligned}
\mathsf{aterm_level}\ k\ (\mathtt{CON}\ a) &= \mathtt{True} \\
\mathsf{aterm_level}\ k\ (\mathtt{BND}\ j) &= (j < k) \\
\mathsf{aterm_level}\ k\ (\mathtt{VAR}\ i) &= \mathtt{True} \\
\mathsf{aterm_level}\ k\ (f\ \$\$_{\mathrm{O}}\ g) &= (\mathsf{aterm_level}\ k\ f \wedge \mathsf{cterm_level}\ k\ g)
\end{aligned}$$

$$\begin{aligned}
\mathsf{ctype_level}\ k\ (\mathtt{PI}\ t\ f) &= (\mathsf{ctype_level}\ k\ t \wedge \mathsf{ctype_level}\ (k+1)\ f) \\
\mathsf{ctype_level}\ k\ (\mathtt{ATYPE}\ t) &= \mathsf{atype_level}\ k\ t
\end{aligned}$$

$$\begin{aligned}
\mathsf{atype_level}\ k\ (\mathtt{FCON}\ a) &= \mathtt{True} \\
\mathsf{atype_level}\ k\ (f\ \$\$_{\mathrm{T}}\ g) &= (\mathsf{atype_level}\ k\ f \wedge (\mathsf{cterm_level}\ k\ g)
\end{aligned}$$

$$\begin{aligned}
\mathsf{kind_level}\ k\ (\mathtt{TYPE}) &= \mathtt{True} \\
\mathsf{kind_level}\ k\ (\mathtt{KPI}\ t\ k) &= (\mathsf{ctype_level}\ k\ t \wedge \mathsf{kind_level}\ (k+1)\ k)
\end{aligned}$$

We also define cterm_shift, ctype_shift, aterm_shift and atype_shift functions that perform shifting on the bound variables of a given canonical term, canonical type, atomic term or atomic type:

Definition 87 (Shift functions)**.**

$$
\begin{aligned}
\mathsf{cterm\_shift}\ 0\ k\ n &= n \\
\mathsf{cterm\_shift}\ i\ k\ (\texttt{ABS}\ a\ m) &= (\texttt{ABS}\ (\mathsf{ctype\_shift}\ i\ k\ a)\ (\mathsf{cterm\_shift}\ i\ (k+1)\ m)) \\
\mathsf{cterm\_shift}\ i\ k\ (\texttt{ATERM}\ r) &= (\texttt{ATERM}\ (\mathsf{aterm\_shift}\ i\ k\ r))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{aterm\_shift}\ i\ k\ (\texttt{CON}\ a) &= (\texttt{CON}\ a) \\
\mathsf{aterm\_shift}\ i\ k\ (\texttt{VAR}\ n) &= (\texttt{VAR}\ n) \\
\mathsf{aterm\_shift}\ i\ k\ (\texttt{BND}\ n) &= \begin{cases} (\texttt{BND}\ (n+i)) & (n \geq k) \\ (\texttt{BND}\ n) & (n < k) \end{cases} \\
\mathsf{aterm\_shift}\ i\ k\ (\texttt{APP}\ m\ n) &= \texttt{APP}\ (\mathsf{aterm\_shift}\ i\ k\ m)\ (\mathsf{cterm\_shift}\ i\ k\ n)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{ctype\_shift}\ 0\ k\ n &= n \\
\mathsf{ctype\_shift}\ i\ k\ (\texttt{ATYPE}\ p) &= (\texttt{ATYPE}\ (\mathsf{atype\_shift}\ i\ k\ p)) \\
\mathsf{ctype\_shift}\ i\ k\ (\texttt{PI}\ a\ b) &= (\texttt{PI}\ (\mathsf{ctype\_shift}\ i\ k\ a)\ (\mathsf{ctype\_shift}\ i\ (k+1)\ b))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{atype\_shift}\ i\ k\ (\texttt{FCON}\ a) &= (\texttt{FCON}\ a) \\
\mathsf{atype\_shift}\ i\ k\ (\texttt{FAPP}\ a\ m) &= (\texttt{FAPP}\ (\mathsf{atype\_shift}\ i\ k\ a)\ (\mathsf{cterm\_shift}\ i\ k\ m))
\end{aligned}
$$

These functions perform shifting on a given term or type. This is intended to be used when substituting a term or type containing instances of `BND` to represent bound variables into a second term or type that may potentially include binders. When performing such a substitution, the indices of the bound variables in the first term that 'point' to binders outside the term should be increased to ensure that they are still linked to the correct binder. As such, we shift indices above a given *cutoff* (which is the second parameter to this function, the first being the amount to shift by). When shifting during substitution the cutoff is initially 0, ensuring that all variables are shifted. When shifting recurses over an instance of `ABS` the cutoff is incremented so that variables that refer to the binder that we recursed over are not shifted. This takes place in the equation for `ABS` above.

We also define ctx_lookup, sig_t_lookup, sig_k_lookup and bndenv_lookup functions that look up a variable in a context, a given object constant in a signature, a type constant in a signature or the type of a given binder in a binding environment. We omit their simple definitions.

3.4.4 Substitution

Substitution is carried out by two sets of mutually inductively defined functions. Those ending in '`bv`' perform substitution for a bound variable, while those ending in '`fv`' perform substitution for a free variable. See tables 3.1 and 3.2 for a list of the functions in CANONICAL HYBRIDLF.

Although the functions are mutually defined, in the listing below we separate them for clarity.

The first parameter of all of the functions is a natural number, used to ensure that substitution terminates. Note that all of the functions have a case for when this argument is zero (such as `zero_kind`, `zero_type`, etc) which simply returns `None` to indicate failure. The cases for when this parameter is non-zero all pattern-match on `Suc q` for some `q`, distinguishing them from the zero case, and recursive calls within the body of the functions all give `q` as the first parameter, ensuring that this decreases with each recursive call. The second parameter is a context, while the third and fourth are the signature, split into object constants and type constants. The fifth parameter is the binding environment. In `validkind` and `validtype` the sixth argument is the kind or type to check for validity. In `canon_typeof` and `atom_typeof`, the sixth parameter is the canonical term or atomic term to determine the type of. The sixth parameter in the substitution functions is a canonical term. This is the term that we are substituting for bound variables. The seventh and eighth parameters in the substitution functions are natural numbers. The seventh is the number of the variable to substitute for, while the eighth tracks the number of binders that substitution recurses over (and hence this parameter should be 0 when the substitution function is initially called).

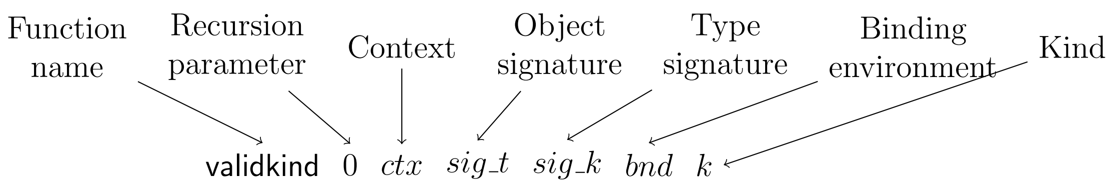

Figure 3.6: Parameters to validkind function

The validkind function indicates if a given kind is a valid kind:

Definition 88 (validkind).

validkind 0 ctx sig_t sig_k bnd k = None
validkind $(q+1)$ ctx sig_t sig_k bnd TYPE = Some True
validkind $(q+1)$ ctx sig_t sig_k bnd (KPI a k) = (case validtype q ctx sig_t sig_k
bnd a of Some $t1$ ⇒ (case validkind q ctx sig_t sig_k $(a \# bnd)$ k of
Some $t2$ ⇒ Some $(t1 \wedge t2)$ | None ⇒ None) | None ⇒ None)

The validtype function indicates if a given type is a valid canonical LF type:

Definition 89 (validtype).

validtype 0 ctx sig_t sig_k bnd t = None
validtype $(q+1)$ ctx sig_t sig_k bnd (ATYPE p) = Some (atom_kindof q ctx sig_t
sig_k bnd p = Some TYPE)
validtype $(q+1)$ ctx sig_t sig_k bnd (PI a' a) = (case validtype q ctx sig_t
sig_k bnd a' of Some $t1$ ⇒ (case validtype q ctx sig_t sig_k $(a' \# bnd)$
a of Some $t2$ ⇒ Some $(t1 \wedge t2)$ | None ⇒ None) | None ⇒ None)

The atom_kindof function computes the kind of an atomic type:

Definition 90 (atom_kindof).

atom_kindof 0 ctx sig_t sig_k bnd a = None
atom_kindof $(q+1)$ ctx sig_t sig_k bnd (FCON a) = (case sig_k_lookup
sig_k a of Some k ⇒ (if kind_level 0 k then Some k else None)
| None ⇒ None)
atom_kindof $(q+1)$ ctx sig_t sig_k bnd (FAPP p m) = (case atom_kindof
q ctx sig_t sig_k bnd p of Some (KPI a k) ⇒ (case canon_typeof q ctx sig_t
sig_k bnd m of Some a ⇒ kind_subst_bv q ctx sig_t sig_k bnd m 0 0 k
| None ⇒ None) | None ⇒ None)

The canon_typeof function computes the type of a canonical term:

Definition 91 (canon_typeof).

$$\begin{aligned}
&\mathsf{canon_typeof}\ 0\ ctx\ sig_t\ sig_k\ bnd\ m = \texttt{None}\\
&\mathsf{canon_typeof}\ (q+1)\ ctx\ sig_t\ sig_k\ bnd\ (\texttt{ATERM}\ r) =\\
&\quad \mathsf{atom_typeof}\ q\ ctx\ sig_t\ sig_k\ bnd\ r\\
&\mathsf{canon_typeof}\ (q+1)\ ctx\ sig_t\ sig_k\ bnd\ (\texttt{ABS}\ a'\ m) =\\
&\quad (\texttt{case}\ \mathsf{canon_typeof}\ q\ ctx\ sig_t\ sig_k\ (a'\ \#\ bnd)\ m\ \texttt{of Some}\ a \Rightarrow\\
&\quad (\texttt{if}\ \mathsf{ctype_level}\ 0\ a'\ \texttt{then Some}\ (\texttt{PI}\ a'\ a)\ \texttt{else None}) \mid \texttt{None} \Rightarrow \texttt{None})
\end{aligned}$$

The `atom_typeof` function computes the type of an atomic term:

Definition 92 (atom_typeof).

$$\begin{aligned}
&\mathsf{atom_typeof}\ 0\ ctx\ sig_t\ sig_k\ bnd\ r\ = \texttt{None}\\
&\mathsf{atom_typeof}\ (q+1)\ ctx\ sig_t\ sig_k\ bnd\ (\texttt{VAR}\ v) = (\texttt{case}\ \mathsf{ctx_lookup}\ ctx\ v\\
&\quad \texttt{of Some}\ t \Rightarrow (\texttt{if}\ \mathsf{ctype_level}\ 0\ t\ \texttt{then Some}\ t\ \texttt{else None}) \mid \texttt{None} \Rightarrow \texttt{None})\\
&\mathsf{atom_typeof}\ (q+1)\ ctx\ sig_t\ sig_k\ bnd\ (\texttt{BND}\ b) = (\texttt{case}\ \mathsf{bndenv_lookup}\ bnd\ b\\
&\quad \texttt{of Some}\ t \Rightarrow (\texttt{if}\ \mathsf{ctype_level}\ 0\ t\ \texttt{then Some}\ t\ \texttt{else None}) \mid \texttt{None} \Rightarrow \texttt{None})\\
&\mathsf{atom_typeof}\ (q+1)\ ctx\ sig_t\ sig_k\ bnd\ (\texttt{CON}\ c) = (\texttt{case}\ \mathsf{sig_t_lookup}\ sig_t\ c\\
&\quad \texttt{of Some}\ t \Rightarrow (\texttt{if}\ \mathsf{ctype_level}\ 0\ t\ \texttt{then Some}\ t\ \texttt{else None}) \mid \texttt{None} \Rightarrow \texttt{None})\\
&\mathsf{atom_typeof}(q+1)\ ctx\ sig_t\ sig_k\ bnd\ (\texttt{APP}\ r\ m) =\\
&\quad (\texttt{case}\ \mathsf{atom_typeof}\ q\ ctx\ sig_t\ sig_k\ bnd\ r\ \texttt{of Some}\ (\texttt{PI}\ a'\ a) \Rightarrow\\
&\quad (\texttt{case}\ \mathsf{canon_typeof}\ q\ ctx\ sig_t\ sig_k\ bnd\ m\ \texttt{of Some}\ a' \Rightarrow\\
&\quad \mathsf{ctype_subst_bv}\ q\ ctx\ sig_t\ sig_k\ bnd\ m\ 0\ 0\ a \mid \texttt{None} \Rightarrow \texttt{None})\\
&\quad \mid \texttt{None} \Rightarrow \texttt{None})
\end{aligned}$$

Substitution is performed by a number of functions, all of which are defined in a similar fashion and are listed in table 3.3.

We show kind_subst_bv here as an example, and include the rest in appendix B. The kind_subst_bv function performs substitution of a canonical term for a bound variable in a given kind:

Notation	Function
$[M/x]^k_\alpha$	kind_subst_bv kind_subst_fv
$[M/x]^a_\alpha$	ctype_subst_bv ctype_subst_fv
$[M/x]^p_\alpha$	atype_subst_bv atype_subst_fv
$[M/x]^m_\alpha$	cterm_subst_bv cterm_subst_fv
$[M/x]^r_\alpha$	aterm_subst_bv aterm_subst_fv aterm_can_subst_bv aterm_can_subst_fv

Table 3.3: Canonical HybridLF substitution functions

Definition 93 (kind_subst_bv)**.**

kind_subst_bv 0 $ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ p\ a$ $=$ `None`

kind_subst_bv $(q+1)$ $ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ p$ `TYPE` $=$

(`if` cterm_level 0 m `then Some TYPE else None`)

kind_subst_bv $(q+1)$ $ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ p$ (`KPI` $a\ k$) $=$

(`case` ctype_subst_bv $q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ p\ a$ `of Some` $a' \Rightarrow$

(`case` kind_subst_bv $q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ (p+1)\ k$ `of Some` $k' \Rightarrow$

`Some` (`KPI` $a'\ k'$) $|$ `None` $\Rightarrow$ `None`) $|$ `None` $\Rightarrow$ `None`)

3.4.5 Syntactic terms

We define families of functions cterm_ordinary to cterm_ordinary12, aterm_ordinary to aterm_ordinary12, ctype_ordinary to ctype_ordinary12 and atype_ordinary to atype_ordinary12. These are similar to the obj_ordinary and fam_ordinary functions of HybridLF in their definition and intent: they determine if a function is a syntactic term.

cterm_ordinary is defined as follows:

Definition 94 (cterm_ordinary)**.**

$$
\begin{aligned}
&\mathsf{cterm\_ordinary} \equiv \lambda e.(e = (\lambda x.\ \mathtt{ATERM}\ x)) \\
&\lor\ (\exists n.\ e = (\lambda x.\ \mathtt{ATERM}\ (\mathtt{CON}\ n))) \\
&\lor\ (\exists n.\ e = (\lambda x.\ \mathtt{ATERM}\ (\mathtt{VAR}\ n))) \\
&\lor\ (\exists n.\ e = (\lambda x.\ \mathtt{ATERM}\ (\mathtt{BND}\ n))) \\
&\lor\ (\exists f\ g.\ e = (\lambda x.\ \mathtt{ATERM}\ (\mathtt{APP}\ (f\ x)\ (g\ x)))) \\
&\lor\ (\exists f\ ty.\ e = (\lambda x.\ \mathtt{ABS}\ (ty\ x)\ (f\ x)))
\end{aligned}
$$

The definition of cterm_ordinary12 is as follows:

Definition 95 (cterm_ordinary12)**.**

$$
\begin{aligned}
&\mathsf{cterm\_ordinary12} \equiv \lambda e.\ (e = (\lambda x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ x)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ y)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ z)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ a)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ b)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ c)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ d)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ e)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ f)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ g)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ h)) \\
&\lor (e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ i)) \\
&\lor (\exists n.\ e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ (\mathtt{CON}\ n))) \\
&\lor (\exists n.\ e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ (\mathtt{VAR}\ n))) \\
&\lor (\exists n.\ e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ (\mathtt{BND}\ n))) \\
&\lor (\exists fa\ ga.\ e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ATERM}\ (\mathtt{APP} \\
&\qquad (fa\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i)\ (ga\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i)))) \\
&\lor (\exists fa\ ty.\ e = (\lambda\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i.\ \mathtt{ABS} \\
&\qquad (ty\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i)\ (fa\ x\ y\ z\ a\ b\ c\ d\ e\ f\ g\ h\ i)))
\end{aligned}
$$

The number appended to the end of the function name indicates the number of lambda-bound variables that the abstraction has (so, for example, cterm_ordinary7 is a function with 7 parameters).

The definition of aterm_ordinary is as follows:

Definition 96 (aterm_ordinary)**.**

$$\begin{aligned}
&\mathsf{aterm_ordinary} \ \equiv \lambda e.\ (\exists n.\ e = (\lambda x.\ (\texttt{CON}\ n)))\\
&\vee (e = (\lambda x.\ x))\\
&\vee (\exists n.\ e = (\lambda x.\ (\texttt{VAR}\ n)))\\
&\vee (\exists n.\ e = (\lambda x.\ (\texttt{BND}\ n)))\\
&\vee (\exists f\ g.\ e = (\lambda x.\ (\texttt{APP}\ (f\ x)\ (g\ x))))\text{''}
\end{aligned}$$

ctype_ordinary is defined like so:

Definition 97 (ctype_ordinary)**.**

$$\begin{aligned}
&\mathsf{ctype_ordinary} \equiv \lambda e.\ (\exists n.\ e = (\lambda x.\ (\texttt{ATYPE}\ (\texttt{FCON}\ n))))\\
&\vee (\exists f\ g.\ e = (\lambda x.\ \texttt{ATYPE}\ (\texttt{FAPP}\ (f\ x)\ (g\ x))))\\
&\vee (\exists f\ ty.\ e = (\lambda x.\ \texttt{PI}\ (ty\ x)\ (f\ x)))\text{''}
\end{aligned}$$

atype_ordinary is defined as follows:

Definition 98 (atype_ordinary)**.**

$$\begin{aligned}
&\mathsf{atype_ordinary} \equiv \lambda e.\ (\exists n.\ e = (\lambda x.\ (\texttt{FCON}\ n)))\\
&\vee (\exists f\ g.\ e = (\lambda x.\ (\texttt{FAPP}\ (f\ x)\ (g\ x))))
\end{aligned}$$

The aterm_ordinary, ctype_ordinary and atype_ordinary families of functions all have numbered variants for different numbers of variables up to 12 in the same way as cterm_ordinary; we do not show the definitions here.

3.4.6 Conversion from HOAS functions

We define mutually-recursive functions cterm_bind', aterm_bind', ctype_bind' and atype_bind'. These perform conversion from a HOAS function to a Canonical HybridLF canonical term, atomic term, canonical type or atomic type. Note the equation in each function that handles the case where the input function does not represent a syntactic term; these are the equations that are guarded by a use of an 'ordinary' function. In the case of cterm_bind' this equation is $\neg\mathsf{cterm\_ordinary}\ expr \Longrightarrow \mathsf{cterm\_bind'}\ i\ expr = \texttt{None}$.

Although the functions are mutually recursively defined, we show them separately here for clarity.

cterm_bind' is defined like so:

Definition 99 (cterm_bind').

$$\begin{aligned}
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ \mathtt{ATERM}\ x) = \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{BND}\ i))\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{CON}\ a)) = \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{CON}\ a))\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{APP}\ (F\ x)\ (G\ x))) = (\mathtt{case}\\
&\qquad (\mathsf{aterm_bind'}\ i\ F)\ \mathtt{of}\ \mathtt{Some}\ atm\ \Rightarrow (\mathtt{case}\ (\mathsf{cterm_bind'}\ i\ G)\\
&\qquad \mathtt{of}\ \mathtt{Some}\ ctm \Rightarrow \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{APP}\ atm\ ctm))\mid \mathtt{None} \Rightarrow \mathtt{None})\\
&\qquad \mid \mathtt{None} \Rightarrow \mathtt{None})\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ \mathtt{ABS}\ (ty\ x)\ (F\ x)) = (\mathtt{case}\\
&\qquad (\mathsf{ctype_bind'}\ i\ ty)\ \mathtt{of}\ \mathtt{Some}\ t \Rightarrow (\mathtt{case}\ (\mathsf{cterm_bind'}\ (i+1)\ F)\\
&\qquad \mathtt{of}\ \mathtt{Some}\ m \Rightarrow \mathtt{Some}\ (\mathtt{ABS}\ t\ m)\mid \mathtt{None} \Rightarrow \mathtt{None})\\
&\qquad \mid \mathtt{None} \Rightarrow \mathtt{None})\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ (\mathtt{ATERM}\ (\mathtt{BND}\ k))) = \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{BND}\ k))\\
&\mathsf{cterm_bind'}\ i\ (\lambda x.\ (\mathtt{ATERM}\ (\mathtt{VAR}\ n))) = \mathtt{Some}\ (\mathtt{ATERM}\ (\mathtt{VAR}\ n))\\
&\neg \mathsf{cterm_ordinary}\ expr \Longrightarrow \mathsf{cterm_bind'}\ i\ expr = \mathtt{None}
\end{aligned}$$

Note that in the APP case the operands of the APP node are pattern-matched as functions F and G. When we recursively call aterm_bind' i F and cterm_bind' i G, the abstraction over the variable x is 'pushed inwards' over the APP node. The same mechanism is used in ABS nodes for the type and body of the abstraction. Note the use of cterm_ordinary in the guard of the final equation to rule out functions that are not syntactic.

The first parameter, the natural number i, is used to keep track of the number of ABS nodes that the function has recursed over. Note that this is increased in the cterm_bind' $(i+1)$ F call in the ABS equation.

aterm_bind' is defined as follows:

Definition 100 (aterm_bind').

$$
\begin{aligned}
&\textsf{aterm\_bind'}\ i\ (\lambda x.\ (\texttt{APP}\ (F\ x)\ (G\ x))) = (\texttt{case}\\
&\quad (\textsf{aterm\_bind'}\ i\ F)\ \texttt{of Some}\ atm \Rightarrow (\texttt{case}\ (\textsf{cterm\_bind'}\ i\ G)\ \texttt{of}\\
&\quad \texttt{Some}\ ctm \Rightarrow \texttt{Some}\ (\texttt{APP}\ atm\ ctm) \mid \texttt{None} \Rightarrow \texttt{None}) \mid \texttt{None} \Rightarrow \texttt{None})\\
&\textsf{aterm\_bind'}\ i\ (\lambda x.\ x) = \texttt{Some}\ (\texttt{BND}\ i)\\
&\textsf{aterm\_bind'}\ i\ (\lambda x.\ (\texttt{BND}\ k)) = \texttt{Some}\ (\texttt{BND}\ k)\\
&\textsf{aterm\_bind'}\ i\ (\lambda x.\ (\texttt{VAR}\ n)) = \texttt{Some}\ (\texttt{VAR}\ n)\\
&\textsf{aterm\_bind'}\ i\ (\lambda x.\ (\texttt{CON}\ c)) = \texttt{Some}\ (\texttt{CON}\ c)\\
&\neg\textsf{aterm\_ordinary}\ expr \Longrightarrow \textsf{aterm\_bind'}\ i\ expr = \texttt{None}
\end{aligned}
$$

The definition of ctype_bind' is as follows:

Definition 101 (ctype_bind').

$$
\begin{aligned}
&\textsf{ctype\_bind'}\ i\ (\lambda x.\ \texttt{ATYPE}\ (\texttt{FCON}\ a)) = \texttt{Some}\ (\texttt{ATYPE}\ (\texttt{FCON}\ a))\\
&\textsf{ctype\_bind'}\ i\ (\lambda x.\ \texttt{PI}\ (ty\ x)\ (F\ x)) = (\texttt{case}\\
&\quad (\textsf{ctype\_bind'}\ i\ ty)\ \texttt{of Some}\ t \Rightarrow (\texttt{case}\ (\textsf{ctype\_bind'}\ (i+1)\ F)\\
&\quad \texttt{of Some}\ t' \Rightarrow \texttt{Some}\ (\texttt{PI}\ t\ t') \mid \texttt{None} \Rightarrow \texttt{None}) \mid \texttt{None} \Rightarrow \texttt{None})\\
&\textsf{ctype\_bind'}\ i\ (\lambda x.\ \texttt{ATYPE}\ (\texttt{FAPP}\ (F\ x)\ (G\ x))) = (\texttt{case}\\
&\quad (\textsf{atype\_bind'}\ i\ F)\ \texttt{of Some}\ t \Rightarrow (\texttt{case}\ (\textsf{cterm\_bind'}\ i\ G)\\
&\quad \texttt{of Some}\ m \Rightarrow \texttt{Some}\ (\texttt{ATYPE}\ (\texttt{FAPP}\ t\ m)) \mid \texttt{None} \Rightarrow \texttt{None})\\
&\quad \mid \texttt{None} \Rightarrow \texttt{None})\\
&\neg\textsf{ctype\_ordinary}\ ty \Longrightarrow \textsf{ctype\_bind'}\ i\ ty = \texttt{None}
\end{aligned}
$$

atype_bind' is defined like so:

Definition 102 (atype_bind').

$$
\begin{aligned}
&\textsf{atype\_bind'}\ i\ (\lambda x.\ \texttt{FCON}\ a) = \texttt{Some}\ (\texttt{FCON}\ a)\\
&\textsf{atype\_bind'}\ i\ (\lambda x.\ \texttt{FAPP}\ (F\ x)\ (G\ x)) = (\texttt{case}\ (\textsf{atype\_bind'}\ i\ F)\\
&\quad \texttt{of Some}\ t \Rightarrow (\texttt{case}\ (\textsf{cterm\_bind'}\ i\ G)\ \texttt{of Some}\ m \Rightarrow \texttt{Some}\ (\texttt{FAPP}\ t\ m)\\
&\quad \mid \texttt{None} \Rightarrow \texttt{None}) \mid \texttt{None} \Rightarrow \texttt{None})\\
&\neg\textsf{atype\_ordinary}\ ty \Longrightarrow \textsf{atype\_bind'}\ i\ ty = \texttt{None}
\end{aligned}
$$

Like the 'ordinary' functions, cterm_bind', aterm_bind', ctype_bind' and atype_bind' are families of n-ary functions, with variants numbered up to 12 for n up to 12.

Here we show cterm_bind'5, aterm_bind'5, ctype_bind'5 and atype_bind'5.

cterm_bind'5 is defined like so:

Definition 103 (cterm_bind'5).

$$
\begin{aligned}
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \texttt{ATERM}\ x) = \\
&\qquad \texttt{Some}\ (\texttt{ATERM}\ (\texttt{BND}\ (i+4))) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \texttt{ATERM}\ y) = \\
&\qquad \texttt{Some}\ (\texttt{ATERM}\ (\texttt{BND}\ (i+3))) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \texttt{ATERM}\ z) = \\
&\qquad \texttt{Some}\ (\texttt{ATERM}\ (\texttt{BND}\ (i+2))) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \texttt{ATERM}\ a) = \\
&\qquad \texttt{Some}\ (\texttt{ATERM}\ (\texttt{BND}\ (i+1))) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \texttt{ATERM}\ b) = \\
&\qquad \texttt{Some}\ (\texttt{ATERM}\ (\texttt{BND}\ i)) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \texttt{ATERM}\ (\texttt{CON}\ c)) = \\
&\qquad \texttt{Some}\ (\texttt{ATERM}\ (\texttt{CON}\ c)) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ (\texttt{ATERM}\ (\texttt{BND}\ k))) = \\
&\qquad \texttt{Some}\ (\texttt{ATERM}\ (\texttt{BND}\ k)) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ (\texttt{ATERM}\ (\texttt{VAR}\ n))) = \\
&\qquad \texttt{Some}\ (\texttt{ATERM}\ (\texttt{VAR}\ n)) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \texttt{ATERM} \\
&\qquad (\texttt{APP}\ (F\ x\ y\ z\ a\ b)\ (G\ x\ y\ z\ a\ b))) = (\textsf{case}\ (\textsf{aterm\_bind'5}\ i\ F) \\
&\qquad \texttt{of Some}\ atm \Rightarrow (\textsf{case}\ (\textsf{cterm\_bind'5}\ i\ G)\ \texttt{of Some}\ ctm \\
&\qquad\ \Rightarrow \texttt{Some}\ (\texttt{ATERM}\ (\texttt{APP}\ atm\ ctm))\ |\ \texttt{None} \Rightarrow \texttt{None})\ |\ \texttt{None} \Rightarrow \texttt{None}) \\
&\textsf{cterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \texttt{ABS}\ (ty\ x\ y\ z\ a\ b)\ (F\ x\ y\ z\ a\ b)) = \\
&\qquad \textsf{case}\ ((\textsf{ctype\_bind'5}\ i\ ty)\ \texttt{of Some}\ t \Rightarrow (\textsf{case}\ (\textsf{cterm\_bind'5} \\
&\qquad (i+1)\ F)\ \texttt{of Some}\ m \Rightarrow \texttt{Some}\ (\texttt{ABS}\ t\ m)\ |\ \texttt{None} \Rightarrow \texttt{None}) \\
&\qquad |\ \texttt{None} \Rightarrow \texttt{None}) \\
&\neg\textsf{cterm\_ordinary5}\ expr \Longrightarrow \textsf{cterm\_bind'5}\ i\ expr = \texttt{None}
\end{aligned}
$$

Note the additional equations for variables y, z, a and b compared to definition 99. The variable b translates to BND i because we have recursed over i

ABS nodes to reach the variable, so we need to give the variable an index of i to cause it to 'point' to the binder that we will eventually add to the start of the term. The other equations for variables have indexes of $i + n$ for some n because the variables in these equations refer to the 5 ABS binders that will be added at the start of the term by cterm_bind5. As we have recursed over i ABS nodes in the body of the term, we need to give the variables an index of $i + 1$, $i + 2$, etc, to make them refer to the correct binder.

aterm_bind'5 is defined as follows:

Definition 104 (aterm_bind'5)**.**

$$
\begin{aligned}
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ (\mathtt{APP}\ (Fx\ y\ z\ a\ b)\ (Gx\ y\ z\ a\ b)))\\
&\qquad = (\mathtt{case}\ (\mathsf{aterm\_bind'5}\ i\ F)\ \mathtt{of}\ \mathtt{Some}\ atm \Rightarrow (\mathtt{case}\\
&\qquad (\mathsf{cterm\_bind'5}\ i\ G)\ \mathtt{of}\ \mathtt{Some}\ ctm \Rightarrow \mathtt{Some}\ (\mathtt{APP}\ atm\ ctm)\\
&\qquad \mid \mathtt{None}\ \Rightarrow \mathtt{None})\mid \mathtt{None}\ \Rightarrow \mathtt{None})\\
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ x) = \mathtt{Some}\ (\mathtt{BND}\ (i+4))\\
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ y) = \mathtt{Some}\ (\mathtt{BND}\ (i+3))\\
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ z) = \mathtt{Some}\ (\mathtt{BND}\ (i+2))\\
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ a) = \mathtt{Some}\ (\mathtt{BND}\ (i+1))\\
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ b) = \mathtt{Some}\ (\mathtt{BND}\ i)\\
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ (\mathtt{BND}\ k)) = \mathtt{Some}\ (\mathtt{BND}\ k)\\
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ (\mathtt{VAR}\ n)) = \mathtt{Some}\ (\mathtt{VAR}\ n)\\
&\mathsf{aterm\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ (\mathtt{CON}\ c)) = \mathtt{Some}\ (\mathtt{CON}\ c)\\
&\neg\mathsf{aterm\_ordinary5}\ expr\ \Longrightarrow \mathsf{aterm\_bind'5}\ i\ expr = \mathtt{None}
\end{aligned}
$$

ctype_bind'5 is defined like so:

Definition 105 (ctype_bind'5)**.**

$\mathsf{ctype\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \mathtt{ATYPE}\ (\mathtt{FCON}\ c)) =$
$\quad \mathtt{Some}\ (\mathtt{ATYPE}\ (\mathtt{FCON}\ c))$
$\mathsf{ctype\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \mathtt{PI}\ (ty\ x\ y\ z\ a\ b)\ (F\ x\ y\ z\ a\ b)) =$
$\quad (\mathtt{case}\ (\mathsf{ctype\_bind'5}\ i\ ty)\ \mathtt{of}\ \mathtt{Some}\ t \Rightarrow (\mathtt{case}\ (\mathsf{ctype\_bind'5}$
$\quad (i+1)\ F)\ \mathtt{of}\ \mathtt{Some}\ t' \Rightarrow \mathtt{Some}\ (\mathtt{PI}\ t\ t') \mid \mathtt{None}\ \Rightarrow \mathtt{None})$
$\quad \mid \mathtt{None}\ \Rightarrow \mathtt{None})$
$\mathsf{ctype\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \mathtt{ATYPE}$
$\quad (\mathtt{FAPP}\ (Fx\ y\ z\ a\ b)\ (Gx\ y\ z\ a\ b))) = (\mathtt{case}\ (\mathsf{atype\_bind'5}\ i\ F)$
$\quad \mathtt{of}\ \mathtt{Some}\ t \Rightarrow (\mathtt{case}\ (\mathsf{cterm\_bind'5}\ i\ G)\ \mathtt{of}\ \mathtt{Some}\ m \Rightarrow \mathtt{Some}$
$\quad (\mathtt{ATYPE}\ (\mathtt{FAPP}\ t\ m)) \mid \mathtt{None}\ \Rightarrow \mathtt{None}) \mid \mathtt{None}\ \Rightarrow \mathtt{None})$
$\neg\mathsf{ctype\_ordinary5}\ ty \Longrightarrow \mathsf{ctype\_bind'5}\ i\ ty = \mathtt{None}$

$\mathsf{atype\_bind'5}$ is defined as follows:

Definition 106 (atype_bind'5)**.**

$\mathsf{atype\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \mathtt{FCON}\ c) = \mathtt{Some}\ (\mathtt{FCON}\ c)$
$\mathsf{atype\_bind'5}\ i\ (\lambda x\ y\ z\ a\ b.\ \mathtt{FAPP}\ (F\ x\ y\ z\ a\ b)\ (G\ x\ y\ z\ a\ b)) =$
$\quad (\mathtt{case}\ (\mathsf{atype\_bind'5}\ i\ F)\ \mathtt{of}\ \mathtt{Some}\ t \Rightarrow (\mathtt{case}\ (\mathsf{cterm\_bind'5}\ i\ G)\ \mathtt{of}$
$\quad \mathtt{Some}\ m \Rightarrow \mathtt{Some}\ (\mathtt{FAPP}\ t\ m) \mid \mathtt{None}\ \Rightarrow \mathtt{None}) \mid \mathtt{None}\ \Rightarrow \mathtt{None})$
$\neg\mathsf{atype\_ordinary5}\ ty \Longrightarrow \mathsf{atype\_bind'5}\ i\ ty = \mathtt{None}$

The bind' functions are later used in the definition of families of bind functions. First we need to define the families of abstr functions that determine if a given function represents a valid abstraction.

Although the functions are mutually recursively defined, we again separate them for clarity.

The definition of $\mathsf{cterm\_abstr}$ is as follows:

Definition 107 (cterm_abstr).

$$\begin{aligned}
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ x) = \mathtt{True}\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{CON}\ a)) = \mathtt{True}\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{BND}\ n)) = (n < i)\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ATERM}\ (\mathtt{VAR}\ n)) = \mathtt{True}\\
&\mathsf{cterm_abstr}\ i(\lambda x.\ \mathtt{ATERM}\ ((f\ x)\ \$\$_{\mathrm{o}}\ (g\ x))) =\\
&\qquad (\mathsf{aterm_abstr}\ i\ f \wedge \mathsf{cterm_abstr}\ i\ g)\\
&\mathsf{cterm_abstr}\ i\ (\lambda x.\ \mathtt{ABS}\ (t\ x)\ (f\ x)) = (\mathsf{ctype_abstr}\ i\ t\\
&\qquad \wedge \mathsf{cterm_abstr}(i+1)\ f)\\
&\neg\mathsf{cterm_ordinary}\ f \Longrightarrow \mathsf{cterm_abstr}\ i\ f = \mathtt{False}
\end{aligned}$$

The `nat` first parameter is used to track the number of abstractions that the function has recursed over to check that bound variables are not dangling (this is performed in the equation for `BND`). It is intended to be initially called with 0, and is then increased in the cterm_abstr $(i + 1)\ f$ call in the equation for functions consisting of an `ABS` node above. Note the use of cterm_ordinary in the guard for the last equation to exclude functions that are not syntactic.

The definition of aterm_abstr is as follows:

Definition 108 (aterm_abstr).

$$\begin{aligned}
&\mathsf{aterm_abstr}\ i\ (\lambda x.\ (\mathtt{CON}\ a)) = \mathtt{True}\\
&\mathsf{aterm_abstr}\ i\ (\lambda x.\ (\mathtt{BND}\ n)) = (n < i)\\
&\mathsf{aterm_abstr}\ i\ (\lambda x.\ (\mathtt{VAR}\ n)) = \mathtt{True}\\
&\mathsf{aterm_abstr}\ i\ (\lambda x.\ (f\ x)\ \$\$_{\mathrm{o}}\ (g\ x)) = (\mathsf{aterm_abstr}\ i\ f \wedge \mathsf{cterm_abstr}\ i\ g)\\
&\mathsf{aterm_abstr}\ i\ (\lambda x.\ x) = \mathtt{True}\\
&\neg\mathsf{aterm_ordinary}\ f \Longrightarrow \mathsf{aterm_abstr}\ i\ f = \mathtt{False}
\end{aligned}$$

ctype_abstr is defined like so:

Definition 109 (ctype_abstr).

$$\textsf{ctype\_abstr}\ i\ (\lambda x.\ \texttt{ATYPE}\ (\texttt{FCON}\ a)) = \texttt{True}$$
$$\textsf{ctype\_abstr}\ i\ (\lambda x.\ \texttt{ATYPE}\ ((f\ x)\ \$\$_{\text{T}}\ (g\ x))) = (\textsf{atype\_abstr}\ i\ f \wedge \textsf{cterm\_abstr}\ i\ g)$$
$$\textsf{ctype\_abstr}\ i\ (\lambda x.\ \texttt{PI}\ (t\ x)\ (f\ x)) = (\textsf{ctype\_abstr}\ i\ t \wedge \textsf{ctype\_abstr}\ (i+1)\ f)$$
$$\neg\textsf{ctype\_ordinary}\ f \Longrightarrow \textsf{ctype\_abstr}\ i\ f = \texttt{False}$$

Finally, atype_abstr is defined like this:

Definition 110 (atype_abstr).

$$\textsf{atype\_abstr}\ i\ (\lambda x.\ \texttt{FCON}\ a) = \texttt{True}$$
$$\textsf{atype\_abstr}\ i\ (\lambda x.\ (f\ x)\ \$\$_{\text{T}}\ (g\ x)) = (\textsf{atype\_abstr}\ i\ f \wedge \textsf{cterm\_abstr}\ i\ g)$$
$$\neg\textsf{atype\_ordinary}\ f \Longrightarrow \textsf{atype\_abstr}\ i\ f = \texttt{False}$$

The cterm_bind and ctype_bind functions that perform the conversion from a function `aterm` $\rightarrow$ `cterm` or `aterm` $\rightarrow$ `ctype` are defined using the abstr and bind' functions. cterm_bind creates an `ABS` term, with the first `ctype` parameter of the function being the type of the variable bound by the abstraction. ctype_bind creates a `PI` term, with the first `ctype` parameter of the function again as the type of the bound variable. The functions return a `cterm option` or `ctype option`. This is in contrast to the original Hybrid and to HybridLF, in which an `ERR` (and in the case of HybridLF `FERR`) element is included in the `expr` datatype to indicate failure. Using Isabelle's `option` type allows us to separate the success or failure of the function from the datatypes that the function is defined over, giving a cleaner design overall.

Definition 111 (cterm_bind).

$$\textsf{cterm\_bind}\ t\ e \equiv \texttt{if}\ \textsf{cterm\_abstr}\ 0\ e\ \texttt{then}\ (\textsf{case}\ (\textsf{cterm\_bind'}\ 0\ e)$$
$$\texttt{of Some}\ e' \Rightarrow \texttt{Some}\ (\texttt{ABS}\ t\ e')\ |\ \texttt{None}\ \Rightarrow \texttt{None})\ \texttt{else None}$$

Definition 112 (ctype_bind).

$$\textsf{ctype\_bind}\ t\ e \equiv \texttt{if}\ \textsf{ctype\_abstr}\ 0\ e\ \texttt{then}\ (\textsf{case}\ (\textsf{ctype\_bind'}\ 0\ e)$$
$$\texttt{of Some}\ e' \Rightarrow \texttt{Some}\ (\texttt{PI}\ t\ e')\ |\ \texttt{None}\ \Rightarrow \texttt{None})\ \texttt{else None}$$

cterm_bind and ctype_bind are again families of functions, with numbered variants up to 12.

As an example, here is the definition of cterm_bind3:

Definition 113 (cterm_bind3)**.**

$$\mathsf{cterm\_bind3}\ t1\ t2\ t3\ e \equiv \mathtt{if}\ \mathsf{cterm\_abstr3}\ 0\ e \wedge \mathsf{ctype\_abstr}\ 0\ t2$$
$$\wedge\ \mathsf{ctype\_abstr2}\ 0\ t3\ \mathtt{then}\ (\mathtt{case}\ (\mathsf{ctype\_bind'}\ 0\ t2)\ \mathtt{of}\ \mathtt{Some}\ t2'$$
$$\Rightarrow (\mathtt{case}\ (\mathsf{ctype\_bind'2}\ 0\ t3)\ \mathtt{of}\ \mathtt{Some}\ t3' \Rightarrow (\mathtt{case}$$
$$(\mathsf{cterm\_bind'3}\ 0\ e)\ \mathtt{of}\ \mathtt{Some}\ e' \Rightarrow \mathtt{Some}\ (\mathtt{ABS}\ t1\ (\mathtt{ABS}\ t2'$$
$$(\mathtt{ABS}\ t3'\ e')))\ |\ \mathtt{None}\ \Rightarrow \mathtt{None})\ |\ \mathtt{None}\ \Rightarrow \mathtt{None})\ |\ \mathtt{None}\ \Rightarrow \mathtt{None})$$
$$\mathtt{else\ None}$$

Note that the first parameter is a `ctype`, while the second and third parameters are functions `aterm` ⇒ `ctype` and `aterm` ⇒ `aterm` ⇒ `ctype`. The cterm_bind3 function creates 3 `ABS` nodes, and the first three parameters are the types of these nodes. The second parameter is a function because the type of the variable bound by the second `ABS` node depends upon the variable bound by the first `ABS` node - this variable may appear in the type of the variable bound by the second `ABS` node. Likewise, the type of the third `ABS` node depends upon the types of the variables bound by both the first and second `ABS` nodes, etc. The first parameter is not a function because the type of the variable bound by the first `ABS` node does not depend upon any other variable. The type of the fourth parameter is a function `aterm` ⇒ `aterm` ⇒ `aterm` ⇒ `cterm`, which is the body of the third abstraction node. This is a function with three parameters for the three variables bound by the abstractions. As discussed above, the return type of the `cterm_bind3` function is `cterm option`.

The definition of ctype_bind3 is as follows:

Definition 114 (ctype_bind3)**.**

$$\mathsf{ctype\_bind3}\ t1\ t2\ t3\ e \equiv \mathtt{if}\ \mathsf{ctype\_abstr3}\ 0\ e \wedge \mathsf{ctype\_abstr}\ 0\ t2$$
$$\wedge\ \mathsf{ctype\_abstr2}\ 0\ t3\ \mathtt{then}\ (\mathtt{case}\ (\mathsf{ctype\_bind'}\ 0\ t2)\ \mathtt{of}\ \mathtt{Some}\ t2'$$
$$\Rightarrow (\mathtt{case}\ (\mathsf{ctype\_bind'2}\ 0\ t3)\ \mathtt{of}\ \mathtt{Some}\ t3' \Rightarrow (\mathtt{case}$$
$$(\mathsf{ctype\_bind'3}\ 0\ e)\ \mathtt{of}\ \mathtt{Some}\ e' \Rightarrow \mathtt{Some}\ (\mathtt{PI}\ t1\ (\mathtt{PI}\ t2'$$
$$(\mathtt{PI}\ t3'\ e')))\ |\ \mathtt{None} \Rightarrow \mathtt{None})\ |\ \mathtt{None}\ \Rightarrow \mathtt{None})\ |\ \mathtt{None}\ \Rightarrow \mathtt{None})$$
$$\mathtt{else\ None}$$

The parameters of ctype_bind are similar to those of cterm_bind.

3.5 Chapter summary

In this chapter we discussed the canonical presentation of LF, and the Canonical HybridLF system based upon it.

In many ways Canonical HybridLF is an improvement upon HybridLF. The key advantage is the lack of terms that are not in canonical form, and the resulting absence of definitional equality relations. This is of further benefit when we consider unification (in chapter 5), as there is no need to reduce terms into canonical form during unification itself. The main cost of modifying the grammar so that only terms in canonical form can occur is the need for the `ATERM` and `ATYPE` constructors to include the atomic terms and atomic types into the canonical terms and canonical types. This can add visual clutter to proofs, but it is a price worth paying.

The implementation of typing, kinding and substitution as functions is convenient when using the system to create proofs, as the type or kind or result of substitution is calculated automatically by Isabelle. One drawback with the functions described here is the need for a natural number parameter to bound the potential recursion depth to ensure termination. This is the first parameter of functions such as kind_subst_bv and atom_kindof. Having to give such a parameter is inconvenient, as it can make it hard to tell exactly why substitution or typing failed. However, as the result of these functions is automatically determined it is easy to experiment with different values for the recursion depth parameter, and most instances of substitution require a relatively small number of recursive calls to the substitution functions.

The use of the Isabelle `option` type in the cterm_bind and ctype_bind families of functions allows the removal of the `ERR` and `FERR` elements of the core datatypes that are present in HybridLF. This enables a better separation between the constructors of the datatypes and the indicators that an error has occurred. The use of the `option` type requires a little extra work when using the system, as the type of signatures is a list of pairs of constants and types or kinds, not pairs of constants and type or kind `option`s. As a result, the user must create a function to remove the `option` element of each entry in the signature, a fairly trivial task.

Chapter 4

Proving meta-theorems

4.1 Introduction

There are two main approaches to proving meta-theorems in the literature. *Schema-checking*, implemented in the Twelf system [25], employs logic programming to search for a derivation of a proof. Variables of meta-theorems are *moded* to indicate whether the variable represents an input or an output, and Twelf checks the mode declaration specified by the user to check that it is consistent with the logic programming interpretation of the meta-theorem. Twelf then performs *coverage checking*, in which it checks that the meta-theorem represents a total function from input variables to output variables.

The other approach to proving meta-theorems in LF is through meta-logics such as M_2 [28]. M_2 allows proofs of formulae of the form $\forall x_1{:}A_1 \ldots \forall x_k{:}A_k . \exists x_{k+1}{:}A_{k+1} \ldots \exists x_m{:}A_m . \top$ to be derived, where quantifiers range over LF objects from the signature. M_2 is defined as a sequent calculus with proof terms, and the proof terms of a complete derivation form a total function from the $\forall$-quantified inputs of the formula to the $\exists$-quantified outputs.

In section 4.2 we examine schema-checking as implemented in Twelf, in section 4.3 we discuss the M_2 metalogic and in section 4.4 we discuss the implementation of M_2 in HybridLF.

4.2 Meta-theorems in Twelf

Schuřmann and Pfenning [27] describe the schema-checking approach used in Twelf, in which the meta-theorem is implemented as a relation that is interpreted as a logic program. Type families in the signature are interpreted as *predicates*, while constant declarations are interpreted as *clauses*. For a constant $c : \Pi x_1{:}A_1 \ldots \Pi x_m{:}A_m.B_1 \rightarrow \ldots \rightarrow B_n \rightarrow H$ the *head* of the clause is H and the *body* of the clause or *subgoals* are $B_1 \ldots B_n$. Execution of the relation is carried out via *backchaining*. Given a goal G and a set of clauses $C_1 \ldots C_n$ where each clause is of the form $H \vdash S_1 \ldots S_k$, a backchaining inference system attempts to unify the heads $H_1 \ldots H_n$ of clauses in the signature with the goal. If the head of a clause unifies with G producing a substitution σ, the substitution is applied to the subgoals in the body of the clause $S_1\sigma \ldots S_k\sigma$ which become the new goals, and the process is repeated. If the goal is not reached, the system backtracks to try other clauses in the signature. As a result, backchaining implements a depth-first search strategy.

Example 115. For example, if we have the following signature Σ:

$$\begin{array}{rcl}
\texttt{nat} & : & \texttt{type} \\
\texttt{zero} & : & \texttt{nat} \\
\texttt{succ} & : & \texttt{nat} \rightarrow \texttt{nat} \\
\texttt{odd} & : & \texttt{nat} \rightarrow \texttt{type} \\
\texttt{odd_one} & : & \texttt{odd}\ (\texttt{succ}\ \texttt{zero}) \\
\texttt{odd_succ} & : & \Pi\texttt{a:nat.}\ \texttt{odd a} \rightarrow \texttt{odd}\ (\texttt{succ}\ (\texttt{succ a}))
\end{array}$$

and the goal

$$\vdash \texttt{odd succ}\ (\texttt{succ}\ (\texttt{succ zero}))$$

the system would first try to unify the goal with the heads of all of the constants in the signature, failing until it reached `odd_succ` with which unification would succeed, producing the substitution [`succ zero`/`a`] and new goal `odd succ zero`. The system would again try to unify `odd succ zero` with the heads of constants in the signature, failing until it reached `odd_one`, at which point it would succeed and the query would succeed.

In Twelf, some elements of the relation representing the meta-theorem (called parameters) are designated by the user as inputs (with *positive polarity*), while others are designated as outputs (with *negative polarity*). This is known as moding. Rohwedder and Pfenning [26] describe the Twelf moding process. A mode declaration for a type family is well-defined if all of the parameters designated as inputs only contain other input parameters; in particular, no input parameter should depend upon an output parameter.

Definition 116 (Groundness)**.** A term M is ground with respect to a context Γ if all of the free variables $x_1 \ldots x_n$ contained within M are bound within Γ.

A term is ground if it is ground with respect to the empty context.

Twelf checks that the logic-programming interpretation of the meta-theorem corresponds to the mode declaration - that if ground terms are assigned to the input parameters, ground terms will be produced for the output parameters. This is performed using *abstract interpretation* and *abstract substitutions*, which record whether a variable is known to be ground.

Once Twelf has determined that the meta-theorem is well-moded, it performs *termination checking* on the meta-theorem. This involves checking that the execution of the meta-theorem using backchaining terminates in a finite number of execution steps using a *termination ordering*. By default, Twelf uses a subterm ordering. The basic premise is that the execution terminates if arguments given to recursive subgoals are proper subterms of the arguments to the original goal.

The system then performs coverage checking - it checks that the relation implements a total function from inputs to outputs. As a result, meta-theorems in Twelf are restricted to ∀∃-theorems (i.e. theorems of the form $\forall \overrightarrow{x}. \exists \overrightarrow{y}.$ which relate the inputs $\overrightarrow{x}$ to outputs $\overrightarrow{y}$).

Schürmann and Pfenning [27] describe the Twelf coverage checking algorithm. A *coverage goal* is a term or type with free variables. A *coverage problem* consists of a coverage goal and a set of patterns (which are terms with free variables). During coverage checking, the system checks whether all ground instances of the coverage goal are instances of the set of patterns. We let U stand for a term or type, and V stand for a type or kind, and introduce an additional context Δ that contains the free variables in the coverage goal, writing such variables u and v. We say that a coverage goal $\Delta \vdash U : V$ is *immediately covered* by a collection of patterns $\Delta_i \vdash U_i : V_i$ iff there exists an i and a substitution σ such that $\Delta \vdash U \equiv \sigma U_i : V$. A coverage goal $\Delta \vdash U : V$ is *covered* by a collection of patterns $\Delta_i \vdash U_i : V_i$ iff every ground instance $\langle\rangle \vdash \sigma U : \sigma V$ is immediately covered by $\Delta_i \vdash U_i : V_i$.

Twelf converts general coverage problems (involving terms as well as types) into a type-level form, where a goal $\Delta \vdash A : \text{Type}$ is covered by a collection of patterns $\Delta_i \vdash A_i : \text{Type}$ if every ground instance $\langle\rangle \vdash \sigma A : \text{Type}$ is immediately covered by $\Delta_i \vdash A_i : \text{Type}$.

In general, coverage checking is undecidable. To circumvent this problem, Schürmann and Pfenning [27] require that patterns are *strict*. A variable u has a *strict occurrence* in a term or type U if the judgement $\Delta; \Gamma \vdash_u U$ holds as defined by the rules in figure 4.1.

A pattern $\Delta_i \vdash A_i : \text{Type}$ is *strict* if all variables in Δ_i have a strict occurrence.

$$\frac{\Delta; \Gamma \vdash_u A}{\Delta; \Gamma \vdash_u \lambda x{:}A.M}\ \text{LS\_LD} \qquad \frac{\Delta; \Gamma, x{:}A \vdash_u M}{\Delta; \Gamma \vdash_u \lambda x{:}A.M}\ \text{LS\_LB}$$

$$\frac{\Delta; \Gamma \vdash_u A_i}{\Delta; \Gamma \vdash_u \Pi x{:}A_1.A_2}\ \text{LS\_PD} \qquad \frac{\Delta; \Gamma, x{:}A_1 \vdash_u A_2}{\Delta; \Gamma \vdash_u \Pi x{:}A_1.A_2}\ \text{LS\_PB}$$

$$\frac{\Delta; \Gamma \vdash_u M_i}{\Delta; \Gamma \vdash_u c\ M_1 \ldots M_n}\ \text{LS\_C}\ (1 \leq i \leq n)$$

$$\frac{\Delta; \Gamma \vdash_u M_i}{\Delta; \Gamma \vdash_u a\ M_1 \ldots M_n}\ \text{LS\_A}\ (1 \leq i \leq n)$$

$$\frac{x{:}A \in \Gamma \quad \Delta; \Gamma \vdash_u M_i}{\Delta; \Gamma \vdash_u x\ M_1 \ldots M_n}\ \text{LS\_VAR}\ (1 \leq i \leq n) \qquad \frac{\Gamma \vdash u\ x_1 \ldots x_n \text{ pat}}{\Delta; \Gamma \vdash_u u\ x_1 \ldots x_n}\ \text{LS\_PAT}$$

Figure 4.1: Strictness rules

Schürmann [29] demonstrates that it is decidable if, given a coverage goal $\Delta \vdash A : \text{Type}$ and a strict pattern $\Delta_i \vdash A_i : \text{Type}$, there exists a uniquely determined substitution σ such that $A \equiv \sigma A_i$.

The coverage goal $\Delta \vdash A : \text{Type}$ may be immediately covered by the set of patterns $\Delta_i \vdash A_i : \text{Type}$. If it is not, the set of patterns may still cover the ground instances of the coverage goal. To determine if this is the case, Schürmann and Pfenning [27] gradually instantiate the coverage goal via *splitting*.

The splitting operation results in a set of coverage goals, all of which must be covered for coverage to hold. During splitting, the system selects an uncovered coverage goal $\Delta \vdash A : \text{Type}$ and a free variable u from Δ which is known as the *splitting variable*.

The definition of splitting requires another operation, *raising*. If $\Gamma = x_1, \ldots x_n$ is a context and $c : A$ a constant, raising A by $x_1 \ldots x_n$ produces a raised type A' of form $\Pi x_1 \ldots \Pi x_n.B$ and a context of raised existential variables Δ. We write this as $\langle \Delta \vdash \Pi x_1 \ldots \Pi x_n.A' \rangle$.

Raising is defined like so:

Definition 117.

$\texttt{raise}(\Gamma \vdash A) = \langle \langle \rangle \ \vdash \Pi x_1 \ldots \Pi x_n.A \rangle$ if A is atomic and $\Gamma = x_1 \ldots x_n$.

$\texttt{raise}(\Gamma \vdash A) = \langle u : \Pi x_1 \ldots \Pi x_n.A_1, \Delta \vdash A' \rangle$ if $\Gamma = x_1 \ldots x_n$ and $A = \Pi u : A_1.A_2$ and $\langle \Delta \vdash A' \rangle = \texttt{raise}(\Gamma \vdash A_2[u\ x_1 \ldots x_n/u])$

The splitting operation produces a set of substitutions using higher-order pattern unification. Splitting is defined like so:

If $\Gamma = x_1 : A_1 \ldots x_n : A_n$ and u in $\Delta = \Delta_1, u : \Pi x_1 \ldots \Pi x_n.B_u, \Delta_2$ is a splitting variable and $\Delta \vdash C$: Type is a coverage goal, splitting produces a set of substitutions by looking at each constant in the signature Σ and each local parameter in Γ:

Definition 118.

Constants: Let $c : \Pi y_1 \ldots \Pi y_i.B_c \in \Sigma$ and
$\texttt{raise}(\Gamma \vdash \Pi y_1 \ldots \Pi y_i.B_c) = \langle \Delta'_c \vdash \Pi x_1 \ldots \Pi x_n.B'_c \rangle$ where
$\Delta'_c = z_1{:}C_1 \ldots z_k{:}C_k$
Let $\Delta' \vdash \sigma_c : \Delta, \Delta'_c$ be the most general unifier of
$\exists \Delta . \exists \Delta'_c.(\Pi x_1 \ldots \Pi x_n.B_u \approx \Pi x_1 \ldots \Pi x_n.B'_c) \wedge$
$(u \approx \lambda x_1 \ldots \lambda x_n.c\ (z_1\ x_1 \ldots x_n) \ldots (z_k\ x_1 \ldots x_n))$ if such a most general unifier exists

Bound variables: Let $\Delta_y = q_1{:}E_1 \ldots q_m{:}E_m,\ y : \Pi q_1 \ldots \Pi q_m.B_y \in \Gamma$ and
$\texttt{raise}(\Gamma \vdash \Pi q_1 \ldots \Pi q_m.B_y) = \langle \Delta'_y \vdash \Pi x_1 \ldots \Pi x_n.B'_y \rangle$ where
$\Delta'_y = r_1{:}F_1 \ldots r_p{:}F_p$
Let $\Delta' \vdash \sigma_y : \Delta, \Delta'_y$ be the most general unifier of
$\exists \Delta . \exists \Delta'_c.(\Pi x_1 \ldots \Pi x_n.B_u \approx \Pi x_1 \ldots \Pi x_n.B'_y) \wedge$
$(u \approx \lambda x_1 \ldots \lambda x_n.y\ (r_1\ x_1 \ldots x_n) \ldots (r_p\ x_1 \ldots x_n))$ if such a most general unifier exists

Example 119. So for example, given the following signature Σ:

$$
\begin{aligned}
\texttt{nat} &: \texttt{type} \\
\texttt{zero} &: \texttt{nat} \\
\texttt{succ} &: \texttt{nat} \to \texttt{nat} \\
\texttt{odd} &: \texttt{nat} \to \texttt{type} \\
\texttt{odd\_one} &: \texttt{odd}\ (\texttt{succ}\ \texttt{zero}) \\
\texttt{odd\_succ} &: \Pi\texttt{a:nat.}\ \texttt{odd a} \to \texttt{odd}\ (\texttt{succ}\ (\texttt{succ a}))
\end{aligned}
$$

and the coverage goal $a : \texttt{nat} \vdash \texttt{odd a} : \text{Type}$ if we split on a, the following substitutions σ_1 and σ_2 will be produced:

$$
\begin{aligned}
\sigma_1 &= [\texttt{zero}\ \ /\ a] \\
\sigma_2 &= [\texttt{succ}\ \ b\ /\ a]
\end{aligned}
$$

resulting in the following coverage goals:

$$
\begin{aligned}
&\vdash \texttt{odd zero} : \text{Type} \\
b : \texttt{nat} &\vdash \texttt{odd}\ (\texttt{succ}\ b) : \text{Type}
\end{aligned}
$$

Given a coverage goal and a pattern, the Twelf implementation uses a form of *rigid matching* to produce a set of equations for which immediate coverage fails. A set of candidate variables for splitting is determined from the form of these equations. Twelf attempts to split the variables, starting with right-most variable in the context. If no candidates for splitting remain, the coverage goal is added to a set of *counterexamples*, and the system picks another coverage goal to work on.

4.3 The metalogic M_2

The other approach to proving meta-theorems in LF is via the meta-logic M_2, first formulated by Schürmann and Pfenning [28]. M_2 is defined as a sequent calculus with proof terms. In M_2, formulae take the $\forall\exists$ form, and are given by $\forall x_1{:}A_1 \ldots \forall x_k{:}A_k.\exists x_{k+1}{:}A_{k+1} \ldots \exists x_m{:}A_m.\top$, in which all $A_1 \ldots A_m$ are valid types, $x_1 \ldots x_k$ and $x_{k+1} \ldots x_m$ are valid contexts, the quantifiers range over the objects in the LF signature that the theorems are defined over and the $\top$ symbol stands for truth. We use F to refer to an arbitrary formula. Each sequent has a context Γ and a set of assumptions Δ, which have the form $x \in F$ for some proof term variable x and formula F. We abbreviate the formula $\forall x_1{:}A_1 \ldots \forall x_k{:}A_k.\exists x_{k+1}{:}A_{k+1} \ldots \exists x_m{:}A_m.\top$ to $\forall\Gamma_1.\exists\Gamma_2.\top$, using Γ_1 to refer to the input variables and Γ_2 to refer to the output variables respectively.

The proof terms of M_2 are defined like so:

Definition 120.

$$
\begin{aligned}
P ::= {} & \texttt{Let}\ y = x\ \sigma\ \texttt{ in }\ P \\
& \mid \lambda\Gamma.P \\
& \mid \texttt{Split}\ x\ \texttt{ as }\ \langle\Gamma\rangle\ \texttt{ in }\ P \\
& \mid \langle\sigma\rangle \\
& \mid \texttt{Case}\ x\ \texttt{ of }\ \Theta
\end{aligned}
$$

Given a complete M_2 derivation for a formula $\forall\Gamma_1\exists\Gamma_2.\top$, the proof terms form a function from the universally quantified inputs Γ_1 of the formula to the existentially quantified outputs Γ_2.

Cases are defined as follows:

Definition 121.

$$\Theta ::= \ \cdot \mid R \rightarrow P$$

Patterns are defined like so:

Definition 122.

$$R ::= \Gamma; \Gamma' \rhd M$$

Substitutions σ take the form of a function from variables in a context Γ to objects in a second context Γ', denoted by the judgement $\Gamma' \vdash \sigma : \Gamma$ which is defined by the following rules:

$$\frac{}{\Gamma \vdash [] : \langle\rangle}\ \text{subst\_empty}$$

$$\frac{\Gamma' \vdash M{:}A[\sigma] \qquad \Gamma' \vdash \sigma : \Gamma}{\Gamma' \vdash (\sigma, M/x) : (\Gamma, x : A)}\ \text{subst\_nonempty}$$

4.3.1 Proof rules

The proof rules of M_2 are shown in figure 4.2. The case rule makes use of another judgement $\longrightarrow_\Sigma$, which is shown in figure 4.3.

$$\frac{\Gamma \vdash \sigma : \Gamma_1 \qquad \Gamma; \Delta_1, x \in \forall \Gamma_1.F_1, \Delta_2, y \in F_1[\sigma] \longrightarrow P \in F_2}{\Gamma; \Delta_1, x \in \forall \Gamma_1.F_1, \Delta_2 \longrightarrow \texttt{Let } y = x\ \sigma \texttt{ in } P \in F_2} \forall\text{L}$$

$$\frac{\Gamma, \Gamma_1; \Delta_1, x \in \exists \Gamma_1.\top, \Delta_2 \longrightarrow P \in F}{\Gamma; \Delta_1, x \in \exists \Gamma_1.\top, \Delta_2 \longrightarrow \texttt{Split } x \texttt{ as } \Gamma_1 \texttt{ in } P \in F} \exists\text{L}$$

$$\frac{\Gamma, \Gamma_1; \Delta \longrightarrow P \in F}{\Gamma; \Delta \longrightarrow \lambda \Gamma_1.P \in \forall \Gamma_1.F} \forall\text{R} \qquad \frac{\Gamma \vdash \sigma : \Gamma_1}{\Gamma; \Delta \longrightarrow \langle \sigma \rangle \in \exists \Gamma_1.\top} \exists\text{R}$$

$$\frac{\Gamma; \Delta, x \in F \longrightarrow P \in F}{\Gamma; \Delta \longrightarrow \mu x \in F.P \in F} \text{FIX} \quad [\mu x \in F.P \text{ terminates in } x]$$

$$\frac{\Gamma_1, x{:}A_x, \Gamma_2; \Delta \longrightarrow_\Sigma \Theta \in F}{\Gamma; \Delta \longrightarrow \texttt{Case } x \texttt{ of } \Theta \in F} \text{CASE}$$

Figure 4.2: M_2 proof rules

$$\frac{}{\Gamma_1, x{:}A_x, \Gamma_2; \Delta \longrightarrow_{\langle\rangle} \cdot \in F} \text{SIG\_EMPTY}$$

$$\frac{\Gamma_1, x{:}A_x \Gamma_2; \Delta \longrightarrow_\Sigma \Theta \in F}{\Gamma_1, x{:}A_x.\Gamma_2; \Delta \longrightarrow_{\Sigma, c:\Pi\Gamma_c.A_c} \Theta \in F} \text{SIG\_NON\_UNI } [A_x \text{ and } A_c \text{ do not unify}]$$

$$\frac{\Gamma', \Gamma_2[\sigma]; \Delta[\sigma] \longrightarrow P \in F[\sigma] \qquad \Gamma_1, x{:}A_x, \Gamma_2; \Delta \longrightarrow_\Sigma \Theta \in F}{\Gamma_1, x{:}A_x, \Gamma_2; \Delta \longrightarrow_{\Sigma, c:\Pi\Gamma_c.A_c} \Theta, (\Gamma'; \Gamma_2[\sigma] \rhd (c\ \Gamma_c)[\sigma] \to P) \in F} \text{SIG\_UNI } [\dagger]$$

where † stands for $\sigma = \text{mgu}(A_x = A_c, x = c\ \Gamma_c), \Gamma' \vdash \sigma : (\Gamma_1, x{:}A_x, \Gamma_c)$

Figure 4.3: $M_2 \to_\Sigma$ rules

The $\forall L$ rule instantiates the universally-quantified inputs of an assumption using a given substitution, creating a new assumption. The ∃L rule introduces the existentially quantified outputs of an assumption into the context. ∀R introduces input variables of a goal formula into the context, and ∃R concludes branch in the proof by providing a substitution that instantiates the output variables of a goal formula. FIX implements recursion, and introduces a goal formula as an inductive assumption. CASE performs case analysis on a variable from the context. The $\longrightarrow_\Sigma$ judgement essentially works through all possible instantiations of a context variable to a ground term given by elements of the signature, attempting to unify A_x, the type of $x{:}A_x$ from the context with the base type A_c of each element of the signature $c : \Gamma_c.A_c$. If the types do not unify, the $\longrightarrow_\Sigma$ judgement moves on to the next element of the signature. However, if the types do unify, producing an MGU σ, we proceed to create a derivation showing that the proof term given in the pattern corresponding to c is a valid proof term for $F[\sigma]$, the result of substitution on the goal formula using the MGU σ. The case analysis in M_2 corresponds to the splitting operation in Twelf. Wang and Nadathur [30] formalise the connection between case analysis in M_2 and splitting in Twelf, proving that input coverage checking in Twelf can be translated into a series of case analysis steps in an M_2 proof.

4.4 Implementation of M_2 in HybridLF

4.4.1 Types and proof terms

We first define type synonyms for substitutions, contexts, formulae and patterns:

Definition 123.

```
type_synonym ('a, 'b) subst = (nat × ('a, 'b) expr) list
type_synonym ('a, 'b) con = (nat × ('a, 'b) type) list
type_synonym ('a, 'b) form = (('a, 'b) con × ('a, 'b) con)
type_synonym ('a, 'b) patt = ('a, 'b) expr
```

A substitution consists of pairs relating a variable number to be substituted for to the expression that is to be substituted for it. Contexts consist of pairs of a free variable number and the type of the free variable. Formulae are given by a pair of contexts - the first containing universally quantified input variables, and the second containing existentially quantified output variables. Patterns are simply terms.

We then define the datatype representing M_2 proof terms:

Definition 124.

```
datatype ('a, 'b) pterm = Let nat nat "('a, 'b) subst"
                               "('a, 'b) pterm"
                          | Lam "('a, 'b) con" "('a, 'b) pterm"
                          | Split nat "('a, 'b) con"
                               "('a, 'b) pterm"
                          | Subst "('a, 'b) subst"
                          | Var nat
                          | Fix "('a, 'b) pterm" "('a, 'b) form"
                               "('a, 'b) pterm"
                          | Case nat "('a, 'b) cases"
                      and ('a, 'b) cases = EmptyCase
                          | PattCase "('a, 'b) pterm"
                               "('a, 'b) cases"
```

and hence can now define the type of assumptions:

Definition 125.

```
type_synonym ('a, 'b) assms = (('a, 'b) pterm × ('a, 'b) form) list
```

4.4.2 Bound and free variables

We implement functions for converting the variables in a given term or type from bound variables represented by `BND` nodes to free variables represented by instances of `VAR`.

Example 126. So for example if the conversion function was called on the term (`ABS` (`FCON` a) (`ABS` (`FAPP` (`FCON` b) (`BND` 0)) (`APP` (`BND` 1) (`BND` 0)))), with free variables numbered from 0, the term would be converted to
(`ABS` (`FCON` a) (`ABS` (`FAPP` (`FCON` b) (`VAR` 1)) (`APP` (`VAR` 1) (`VAR` 2)))).

This change of notation is necessary during reasoning in M_2 because the types of the constants representing formulae to be shown to be total will be converted to contexts with a free variable for each `PI` clause. In such a situation, the `BND` representation of variables within these contexts is meaningless, as we do not have the full constant with the `PI` binders to determine which variable is being referenced. As a result, we must convert bound variables to free variables. The unification algorithm also requires all variables to be converted to metavariables before unification; this requires variables to be numbered and implies conversion of variables from their BND form to VAR notation before unification takes place.

We have four functions: bv_to_fv_expr' and bv_to_fv_type' that perform the actual work, and bv_to_fv_expr and bv_to_fv_type that call the previous two functions with a default value.

Definition 127.

$$
\begin{aligned}
\mathsf{bv\_to\_fv\_expr'}\ (\mathtt{ABS}\ a\ b)\ n\ n' &= \mathtt{ABS}\ (\mathsf{bv\_to\_fv\_type'}\ a\ n\ n')\\
&\qquad (\mathsf{bv\_to\_fv\_expr'}\ b\ n\ (n'+1))\\
\mathsf{bv\_to\_fv\_expr'}\ (\mathtt{BND}\ b)\ n\ n' &= \mathtt{VAR}\ (n + (n' - b))\\
\mathsf{bv\_to\_fv\_expr'}\ (\mathtt{CON}\ c)\ n\ n' &= \mathtt{CON}\ c\\
\mathsf{bv\_to\_fv\_expr'}\ \mathtt{ERR}\ n\ n' &= \mathtt{ERR}\\
\mathsf{bv\_to\_fv\_expr'}\ (\mathtt{APP}\ a\ b)\ n\ n' &= \mathtt{APP}\ (\mathsf{bv\_to\_fv\_expr'}\ a\ n\ n')(\mathsf{bv\_to\_fv\_expr'}\ b\ n\ n')\\
\mathsf{bv\_to\_fv\_expr'}\ (\mathtt{VAR}\ v)\ n\ n' &= \mathtt{VAR}\ v
\end{aligned}
$$

Note that the equation for BND returns an instance of VAR. The value n is used as a 'base' value above which all of the new free variable numbers will be created. The value n' tracks the number of ABS nodes that the function has recursed over: note that it is incremented in the second recursive call in the ABS equation. We give each instance of ABS a free variable number starting from n and incremented with each abstraction node that we recurse over. Bound variables that have been converted from an instance of BND b are given free variable number $n + n' - b$ because the bound variable index b refers to the binder b ABS nodes away. Since we have traversed n' ABS nodes, variables bound to this binder will be represented by the free variable number $n' - b$. As we want the variables created to start from n, we add n, making the final result $n + n' - b$.

bv_to_fv_type' performs the same task as bv_to_fv_expr', except that it acts on types instead of terms:

Definition 128.

$$\begin{aligned}
\mathsf{bv_to_fv_type'}\ \mathtt{FERR}\ n\ n' &= \mathtt{FERR}\\
\mathsf{bv_to_fv_type'}\ (\mathtt{FPI}\ a\ b)\ n\ n' &= \mathtt{FPI}\ (\mathsf{bv_to_fv_type'}\ a\ n\ n')\\
&\qquad (\mathsf{bv_to_fv_type'}\ b\ n\ (n'+1))\\
\mathsf{bv_to_fv_type'}\ (\mathtt{FAPP}\ a\ b)\ n\ n' &= (\mathtt{FAPP}\ (\mathsf{bv_to_fv_type'}\ a\ n\ n')(\mathsf{bv_to_fv_expr'}\ b\ n\ n'))\\
\mathsf{bv_to_fv_type'}\ (\mathtt{FCON}\ c)\ n\ n' &= \mathtt{FCON}\ c
\end{aligned}$$

bv_to_fv_expr and bv_to_fv_type simply call bv_to_fv_expr' and bv_to_fv_type' with 0 as the default value of n'.

Definition 129.

$$\mathsf{bv\_to\_fv\_expr}\ c\ n = \mathsf{bv\_to\_fv\_expr'}\ c\ n\ 0$$

Definition 130.

$$\mathsf{bv\_to\_fv\_type}\ c\ n = \mathsf{bv\_to\_fv\_type'}\ c\ n\ 0$$

We define bv_to_fv_con_expr' and bv_to_fv_con_type'. These functions take as arguments a term or type, two natural numbers, a binding environment and a context, and return a context containing entries for all of the occurrences of free variables that arise when converting the free variables of the term or type to bound variables. n and n' are as in bv_to_fv_expr' and bv_to_fv_type', while the function recurses over `ABS` nodes (note that the type a is added to the end of the binding environment in the equation for `ABS`). The context argument contains a partial result for the function - many equations simply return this context. In the equation for `BND`, if there already exists an entry for the variable in the context c then c is simply returned. If no such entry exists then one is created (if there exists an entry for the bound variable in the binding environment) and the updated context is returned. If there is no entry for the bound variable in the binding environment then `None` is returned.

Definition 131.

$$
\begin{aligned}
\mathsf{bv\_to\_fv\_con\_expr'}\ (\mathtt{ABS}\ a\ b)\ n\ n'\ bnd\ c &= (\mathtt{case}\ (\mathsf{bv\_to\_fv\_con\_type'}\ a\ n\ n'\ bnd\ c)\\
&\quad \mathtt{of\ Some}\ c' \Rightarrow (\mathsf{bv\_to\_fv\_con\_expr'}\ b\ n\\
&\quad (n'+1)\ (a \mathbin{\#} bnd)\ c') \mid \mathtt{None} \Rightarrow \mathtt{None})\\
\mathsf{bv\_to\_fv\_con\_expr'}\ (\mathtt{BND}\ b)\ n\ n'\ bnd\ c &= (\mathtt{case}\ (\mathsf{con\_lookup}\ c\ (n + (n' - b)))\ \mathtt{of}\\
&\quad \mathtt{Some}\ t \Rightarrow \mathtt{Some}\ c \mid \mathtt{None} \Rightarrow (\mathtt{case}\\
&\quad (\mathsf{lookup}\ bnd\ b)\ \mathtt{of\ Some}\ t' \Rightarrow\\
&\quad \mathtt{Some}\ (((n + n' - b),\ t') \mathbin{\#} c)\\
&\quad \mid \mathtt{None} \Rightarrow\ \mathtt{None}))\\
\mathsf{bv\_to\_fv\_con\_expr'}\ (\mathtt{CON}\ c')\ n\ n'\ bnd\ c &= \mathtt{Some}\ c\\
\mathsf{bv\_to\_fv\_con\_expr'}\ \mathtt{ERR}\ n\ n'\ bnd\ c &= \mathtt{Some}\ c\\
\mathsf{bv\_to\_fv\_con\_expr'}\ (\mathtt{APP}\ a\ b)\ n\ n'\ bnd\ c &= (\mathtt{case}\ (\mathsf{bv\_to\_fv\_con\_expr'}\ a\ n\ n'\ bnd\ c)\\
&\quad \mathtt{of\ Some}\ c' \Rightarrow (\mathsf{bv\_to\_fv\_con\_expr'}\ b\ n\ n'\\
&\quad bnd\ c') \mid \mathtt{None} \Rightarrow \mathtt{None})\\
\mathsf{bv\_to\_fv\_con\_expr'}\ (\mathtt{VAR}\ v)\ n\ n'\ bnd\ \ &= \mathtt{Some}\ c
\end{aligned}
$$

Definition 132.

$$
\begin{aligned}
\mathsf{bv\_to\_fv\_con\_type'}\ (\mathtt{FPI}\ a\ b)\ n\ n'\ bnd\ c &= (\mathtt{case}\ (\mathsf{bv\_to\_fv\_con\_type'}\ a\ n\ n'\ bnd\ c)\\
&\quad \mathtt{of\ Some}\ c' \Rightarrow (\mathsf{bv\_to\_fv\_con\_type'}\ b\ n\\
&\quad (n'+1)\ (a \mathbin{\#} bnd)\ c') \mid \mathtt{None} \Rightarrow \mathtt{None})\\
\mathsf{bv\_to\_fv\_con\_type'}\ (\mathtt{FAPP}\ a\ b)\ n\ n'\ bnd\ c &= (\mathtt{case}\ (\mathsf{bv\_to\_fv\_con\_type'}\ a\ n\ n'\ bnd\ c)\\
&\quad \mathtt{of\ Some}\ c' \Rightarrow (\mathsf{bv\_to\_fv\_con\_expr'}\ b\ n\ n'\\
&\quad bnd\ c') \mid \mathtt{None} \Rightarrow \mathtt{None})\\
\mathsf{bv\_to\_fv\_con\_type'}\ (\mathtt{FCON}\ c')\ n\ n'\ bnd\ c &= \mathtt{Some}\ c\\
\mathsf{bv\_to\_fv\_con\_type'}\ \mathtt{FERR}\ n\ n'\ bnd\ c &= \mathtt{Some}\ c
\end{aligned}
$$

The bv_to_fv_con_expr and bv_to_fv_con_type functions simply call the previous two functions, providing default values.

Definition 133.

$$\mathsf{bv\_to\_fv\_con\_expr}\ c\ n = \mathsf{bv\_to\_fv\_con\_expr'}\ c\ n\ 0\ [\,][\,]$$

Definition 134.

$$\mathsf{bv\_to\_fv\_con\_type}\ c\ n = \mathsf{bv\_to\_fv\_con\_type'}\ c\ n\ 0\ [\,]\,[\,]$$

4.4.3 Translation from unification representation

The implementation of M_2 requires the use of unification, and as we will see in section 5.3.1, the implementation of unification requires the use of different datatypes to represent terms and types. It is therefore necessary to have functions that translate between the standard `expr` and `type` datatypes and the `uexpr` and `utype` datatypes defined in section 5.3.1. `uexpr` and `utype` are much the same as `expr` and `type`, except that they are expanded with entries for metavariables (`UMVAR`) and placeholders (`UPH`) and lack the `ERR` and `FERR` error indicators.

We define functions translate_to_unify_expr, translate_to_unify_type, translate_to_unify_kind, translate_to_unify_ctx, translate_to_unify_sig_t, translate_to_unify_sig_k and translate_to_unify_bnd that translate a term, type, kind, context, type signature, kind signature or binding environment into the corresponding unification representation.

Example 135. These functions are relatively straight-forward; for example, translate_to_unify_expr (`ABS` (`FCON` a) (`ABS` (`FCON` b) (`APP` (`BND` 1) (`BND` 0)))) = `Some` (`UABS` (`UFCON` a) (`UABS` (`UFCON` b) (`UAPP` (`UBND` 1) (`UBND` 0)))).

The only complication is that there are no elements in `uexpr` or `utype` corresponding to `ERR` and `FERR`, since these error elements of `expr` and `type` are not applicable during unification. In these cases, the translation functions return `None`.

As well as the functions for translating *into* unification representation, there are corresponding functions for translating *out of* unification representation. These are translate_from_unify_uexpr and translate_from_unify_utype, which perform the inverse of translate_to_unify_expr and translate_to_unify_type. The only difference is that `UPH` is translated to `None`, while `UMVAR` $v\ s$ becomes `Some` (`VAR` v).

We also have functions create_solution_subst and create_solution_subst' that create a solution substitution from a unification state or unification equation list respectively.

Definition 136.

$$\mathsf{create\_solution\_subst}\ \mathtt{FAIL} = \mathtt{None}$$
$$\mathsf{create\_solution\_subst}\ (\mathtt{EQNS}\ l) = \mathsf{create\_solution\_subst'}\ l$$

Definition 137.

$$\begin{aligned}
\mathsf{create_solution_subst'}\ [] &= \mathtt{Some}\ []\\
\mathsf{create_solution_subst'}\ ((\mathtt{Solved}\ (x,\ y))\ \#\ xs) &= (\mathtt{case}\ (\mathsf{create_solution_subst'}\ xs)\\
&\quad\ \mathtt{of}\ \mathtt{Some}\ l \Rightarrow \mathtt{Some}\ ((x,\ y)\ \#\ l)\\
&\quad\ |\ \mathtt{None} \Rightarrow \mathtt{None})\\
\mathsf{create_solution_subst'}\ ((\mathtt{TermEqn}\ (x,\ y))\ \#\ xs) &= \mathtt{None}\\
\mathsf{create_solution_subst'}\ ((\mathtt{TyEqn}\ (x,\ y))\ \#\ xs) &= \mathtt{None}
\end{aligned}$$

Note that create_solution_subst' returns Some l if the list of equations consists completely of solved-form equations, and None otherwise.

4.4.4 Substitution

A number of different substitution functions are used in the implementation of M_2.

The function subst_all_expr_bv takes as arguments a substitution and a term, and applies the substitution to the bound variables of the term, producing another term. subst_all_expr_bv performs much the same task as the substitution function o_subst, except that it applies an entire substitution to a given term rather than substituting a single term for a single variable.

Definition 138.

$$\begin{aligned}
\mathsf{subst_all_expr_bv}\ []\ m &= m\\
\mathsf{subst_all_expr_bv}\ ((x,\ y)\ \#\ xs)\ m &= \mathsf{subst_all_expr_bv}\ xs\ (\mathsf{o_subst}\ x\ y\ m)
\end{aligned}$$

Example 139. For example, given the term $M = \mathtt{ABS}\ (\mathtt{FCON}\ a)\ (\mathtt{ABS}\ (\mathtt{FCON}\ b)\ (\mathtt{BND}\ 1)\ \$\$_{\mathrm{o}}\ (\mathtt{BND}\ 3))$ and the substitution $\sigma = [(0,\ (\mathtt{CON}\ c)),\ (1,\ (\mathtt{CON}\ d)),\ (3, (\mathtt{ABS}\ (\mathtt{FCON}\ e)\ (\mathtt{BND}\ 0)))]$, subst_all_expr_bv $\sigma\ M = \mathtt{ABS}\ (\mathtt{FCON}\ a)\ (\mathtt{ABS}\ (\mathtt{FCON}\ b)\ (\mathtt{BND}\ 1)\ \$\$_{\mathrm{o}}\ (\mathtt{CON}\ d))$.

The corresponding function subst_all_expr_fv performs the same task, but substituting for free variables:

Definition 140.

$$\begin{aligned}
\mathsf{subst_all_expr_fv}\ []\ m &= m\\
\mathsf{subst_all_expr_fv}\ ((x,\ y)\ \#\ xs)\ m &= \mathsf{subst_all_expr_fv}\ xs\ (\mathsf{o_subst_fv}\ x\ y\ m)
\end{aligned}$$

There are also subst_all_type_bv and subst_all_type_fv functions; these are very similar to the previous two functions, but operate on types instead of terms.

Next we define substitution on contexts, formulae and assumptions. These functions are necessary for the implementation of the M_2 $\forall L$ and sig_uni rules, where they are responsible for applying the given substitution (in the case of $\forall L$) and propagating the results of unification (in the case of sig_uni).

To substitute for bound variables in contexts we have con_subst_bv:

Definition 141.

$$\begin{aligned}\mathsf{con\_subst\_bv}\ []\ s &= []\\ \mathsf{con\_subst\_bv}\ ((w,\ x)\ \#\ xs)\ s &= ((w,\ (\mathsf{subst\_all\_type\_bv}\ s\ x))\ \#\\ &\qquad \mathsf{con\_subst\_bv}\ xs\ s)\end{aligned}$$

Note the use of subst_all_type_bv to apply the entire substitution s to the type x; this function works through the entire context, applying s to each element in turn. For free variables we have con_subst_fv, which is defined similarly. To substitute for bound variables in formulae we have form_subst_bv:

Definition 142.

$$\begin{aligned}\mathsf{form\_subst\_bv}\ ([],\ [])\ s &= ([],\ [])\\ \mathsf{form\_subst\_bv}\ (x1,\ x2)\ s &= (\mathsf{con\_subst\_bv}\ x1\ s,\ \mathsf{con\_subst\_bv}\ x2\ s)\end{aligned}$$

Since formulae are simply a pair of contexts, we use con_subst_bv on both to obtain our result. We also define form_subst_fv, which subsitutes for free variables, and is defined in the same way. To substitute for bound variables in assumptions we define assms_subst_bv:

Definition 143.

$$\begin{aligned}\mathsf{assms\_subst\_bv}\ []\ s &= []\\ \mathsf{assms\_subst\_bv}\ ((x,\ y)\ \#\ xs)\ s &= ((x,\ \mathsf{form\_subst\_bv}\ y\ s)\ \#\\ &\qquad (\mathsf{assms\_subst\_bv}\ xs\ s))\end{aligned}$$

We again define the corresponding function for free variables, assms_subst_fv, which is defined similarly.

4.4.5 Operations on contexts and substitutions

To implement the proof rules of M_2 we require a number of functions that act on contexts. We have a simple recursive function con_lookup that looks up the type of a variable in a context, returning `None` if the variable was not found, and `Some` A if the variable has type A.

We define another function split_con_after that returns the portion of a context that occurs after the context entry for a given variable. split_con_after is defined like so:

Definition 144.

$$\begin{aligned}\mathsf{split\_con\_after}\ []\ n &= [] \\ \mathsf{split\_con\_after}\ ((x,\ y)\ \#\ xs)\ n &= (\mathtt{if}\ n = x\ \mathtt{then}\ xs\ \mathtt{else}\ (\mathsf{split\_con\_after}\ xs\ n))\end{aligned}$$

Another function on contexts that is used in the implementation of M_2 is con_append, which appends two contexts together. Note that for all of the variables of the entries in the first context, con_append checks that there does not already exist an entry in the second context, returning `None` if such an entry exists.

Definition 145.

$$\begin{aligned}&\mathsf{con\_append}\ []\ d = \mathtt{Some}\ d \\ &\mathsf{con\_append}\ c\ [] = \mathtt{Some}\ c \\ &\mathsf{con\_append}\ ((x,\ y)\ \#\ xs)\ d = (\mathtt{case}\ (\mathsf{con\_lookup}\ d\ x)\ \mathtt{of}\ \mathtt{Some}\ y' \Rightarrow \mathtt{None} \\ &\qquad \mid \mathtt{None} \Rightarrow (\mathtt{case}\ \mathsf{con\_append}\ xs\ d\ \mathtt{of}\ \mathtt{Some}\ d' \Rightarrow \mathtt{Some}\ ((x,\ y)\ \#\ d') \\ &\qquad \mid \mathtt{None} \Rightarrow \mathtt{None}))\end{aligned}$$

We define a relation subst_ctx_fv, which holds if a substitution (given as the sixth argument) maps variables in a context c (the fifth argument) to objects in a context c' (given as the seventh argument).

subst_ctx_fv is defined in figure 4.4

We define a function create_con that takes a list l of free variables and a context c, and looks up the free variables in c to create a new context c'.

$$\frac{}{\mathsf{subst\_ctx\_fv}\ ctx\ sig\_t\ sig\_k\ bnd\ c\ [\,]\,[\,]}\ \text{s\_c\_fv\_empty}$$

$$\frac{\begin{array}{c}\mathsf{con\_lookup}\ c\ x = \mathtt{Some}\ a\\ \mathsf{subst\_all\_type\_fv}\ xs\ a = a'\\ \mathsf{typeof}\ ctx\ sig\_t\ sig\_k\ bnd\ m\ a'\\ \mathsf{subst\_ctx\_fv}\ ctx\ sig\_t\ sig\_k\ bnd\ c\ xs\ xs'\end{array}}{\mathsf{subst\_ctx\_fv}\ ctx\ sig\_t\ sig\_k\ bnd\ c\ ((x,\ m)\ \#\ xs)\ ((x,\ a')\ \#\ xs')}\ \text{s\_c\_fv\_nonempty}$$

Figure 4.4: Rules for subst_ctx_fv

Definition 146.

$$\begin{aligned}
\mathsf{create_con}\ [\,]\ c &= \mathtt{Some}\ [\,]\\
\mathsf{create_con}\ (x\ \#\ xs)\ c &= (\mathtt{case}\ (\mathsf{con_lookup}\ c\ x)\ \mathtt{of}\ \mathtt{Some}\ y \Rightarrow (\mathtt{case}\\
&\qquad (\mathsf{create_con}\ xs\ c)\ \mathtt{of}\ \mathtt{Some}\ xs' \Rightarrow \mathtt{Some}\ ((x,\ y)\ \#\ xs')\\
&\qquad \mid \mathtt{None}\ \Rightarrow \mathtt{None}) \mid \mathtt{None} \Rightarrow \mathtt{None})
\end{aligned}$$

There is a function unique_list that takes as argument a list l, and returns another list l' containing only a single instance of each element in the list (so unique_list$[1, 2, 3, 2, 1, 4]$ would return $[1, 2, 3, 4]$). We omit the definition of this function here.

We define functions get_fvs and get_fvs_for_subst that return the free variables of an expression and a substitution respectively. They are defined like so:

Definition 147.

$$\begin{aligned}
\mathsf{get_fvs}\ (\mathtt{VAR}\ a) &= [a]\\
\mathsf{get_fvs}\ (\mathtt{BND}\ b) &= [\,]\\
\mathsf{get_fvs}\ (\mathtt{APP}\ a\ b) &= (\mathsf{unique_list}\ ((\mathsf{get_fvs}\ a)\ @\ (\mathsf{get_fvs}\ b)))\\
\mathsf{get_fvs}\ (\mathtt{CON}\ c) &= [\,]\\
\mathsf{get_fvs}\ (\mathtt{ABS}\ t\ e) &= \mathsf{get_fvs}\ e\\
\mathsf{get_fvs}\ \mathtt{ERR} &= [\,]
\end{aligned}$$

Definition 148.

$$\begin{aligned}\mathsf{get\_fvs\_for\_subst}\ [\,] &= [\,]\\ \mathsf{get\_fvs\_for\_subst}\ ((x,\ y)\ \#\ xs) &= (\mathsf{unique\_list}\ ((\mathsf{get\_fvs}\ y)\ @\ (\mathsf{get\_fvs\_for\_subst}\ xs)))\end{aligned}$$

We can then define a function get_con_for_subst as:

Definition 149.

$$\mathsf{get\_con\_for\_subst}\ xs\ c = (\mathsf{create\_con}\ (\mathsf{get\_fvs\_for\_subst}\ xs)\ c)$$

We have further functions get_max_var_con and get_max_var_ctx that find the maximum free variable in a context. These are simply defined like so:

Definition 150.

$$\mathsf{get\_max\_var\_con}\ c = \text{fold max (map fst } c\text{) } 0$$

get_max_var_ctx is defined in the same way.

We define a function params_to_con that takes as arguments a type a and a natural number n, and returns a context c. If $a = (\texttt{FPI}\ b\ b')$, params_to_con will include b in the context c, assigning it variable number n, and recursively call params_to_con on b' with $n + 1$ as the variable number. The end result is a context containing entries for variables sequentially numbered from n with types given by the FPI type abstractions.

Definition 151.

$$\begin{aligned}\mathsf{params\_to\_con}\ (\texttt{FPI}\ a\ b)\ n &= ((\mathsf{params\_to\_con}\ b\ (n+1))\ @\ [(n,\ a)])\\ \mathsf{params\_to\_con}\ (\texttt{FCON}\ a)\ n &= [\,]\\ \mathsf{params\_to\_con}\ (\texttt{FAPP}\ a\ b)\ n &= [\,]\end{aligned}$$

The function make_args takes as arguments a term m, a context c with k entries and a natural number n and creates a term

$$m\ \$\$_{\text{o}}\ (\texttt{VAR}\ n) \ldots \$\$_{\text{o}}\ (\texttt{VAR}\ n + k)$$

that has the term m followed by k VAR nodes, each with ascending variable numbers starting from n.

Definition 152.

$$\mathsf{make\_args}\ m\ []\ n = m$$
$$\mathsf{make\_args}\ m\ (x \mathbin{\#} xs)\ n = \mathsf{make\_args}\ (m\ \$\$_{\mathrm{O}}\ (\mathtt{VAR}\ n))\ xs\ (n+1)$$

4.4.6 Proof rules

We can now define relations derivation and sig_derivation that implement the proof rules of M_2. The derivation relation corresponds to the $\longrightarrow$ judgement defined in section 4.3, while the sig_derivation relation corresponds to the $\longrightarrow_\Sigma$ judgement.

derivation is defined in figure 4.5:

sig_derivation is defined in figures 4.6 and 4.7

4.5 Chapter summary

In this chapter we have discussed ways in the literature for proving meta-theorems in LF, along with their implementation in HybridLF. In section 4.2 we discussed the method of schema-checking and its execution in the Twelf system. In section 4.3 we discussed the M_2 metalogic, and in section 4.4 the implementation of M_2 in HybridLF.

While M_2 provides a way to prove meta-theorems in HybridLF, it is not as powerful as the implementation of schema-checking in Twelf. This is because Twelf allows a relation representing a meta-theorem to be proved total in a non-empty LF context (a set of which are known as a *regular world*), whereas M_2 does not. Schürmann [29] describes the logic M_2^+ that allows such contexts to be specified and theorems that depend upon these contexts to be proved. However, M_2^+ is more complicated than M_2, and it is not clear if M_2^+ would be usable for the purpose of actually writing proofs were it to be implemented in HybridLF or Canonical HybridLF.

The implementation of M_2 in HybridLF is relatively faithful to the definition given in section 4.3. However, there are two deviations from the abstract definition of M_2: the side condition of the FIX rule and the side condition of the SIG_NON_UNI rule of the $\longrightarrow_\Sigma$ relation. In the FIX rule, the side condition is that the recursion terminates, but this is not actually checked in the implementation of M_2 in HybridLF. In the SIG_NON_UNI rule, the side condition is that the type of the variable and the base type of the constant from the signature do not unify. This is not actually checked in the implementation, so it is possible to create a 'proof' that appears valid in HybridLF or

$$\frac{\begin{array}{c}\mathsf{con\_append}\ con\ con' = \mathtt{Some}\ con''' \\ \mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ con'''\ assms\ p\ ([],\ con'')\end{array}}{\mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ con\ assms\ (\mathtt{Lam}\ con'\ p)\ (con',\ con'')}\ \text{FORALL\_R}$$

$$\frac{\begin{array}{c}\mathsf{subst\_ctx\_fv}\ ctx\ sig\_t\ sig\_k\ bnd\ c\ s\ c' \\ (\mathtt{Var}\ x,\ (c',\ c'')) \in \text{set}\ assms \\ assms' = ((\mathtt{Var}\ y,\ ([],\ c''))\ \#\ assms) \\ \mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ c\ assms'\ p\ f'\end{array}}{\mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ c\ assms\ (\mathtt{Let}\ y\ x\ s\ p)\ f}\ \text{FORALL\_L}$$

$$\frac{\mathsf{subst\_ctx\_fv}\ ctx\ sig\_t\ sig\_k\ bnd\ c\ s\ c'}{\mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ c\ a\ (\mathtt{Subst}\ s)\ ([],\ c')}\ \text{EXISTS\_R}$$

$$\frac{\begin{array}{c}\mathsf{con\_append}\ con\ con' = \mathtt{Some}\ con'' \\ (\mathtt{Var}\ x,\ ([],\ con')) \in \text{set}\ assms \\ \mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ con''\ assms\ p\ f\end{array}}{\mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ con\ assms\ (\mathtt{Split}\ x\ con'\ p)\ f}\ \text{EXISTS\_L}$$

$$\frac{\mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ con\ ((\mathtt{Var}\ x,\ f)\ \#\ assms)\ p\ f}{\mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ con\ assms\ (\mathtt{Fix}\ (\mathtt{Var}\ x)\ f\ p)\ f}\ \text{RECUR}$$

$$\frac{\begin{array}{c}\mathsf{con\_lookup}\ con\ x = \mathtt{Some}\ t \\ \mathsf{sig\_derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ x\ con\ assms\ sig\_t\ patt\ form\end{array}}{\mathsf{derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ con\ assms\ (\mathtt{Case}\ x\ patt)\ form}\ \text{CASES}$$

Figure 4.5: HybridLF M_2 proof rules - derivation relation

$$\frac{}{\textsf{sig\_derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ x\ con\ assms\ []\ s\ f}\ \textsc{sig\_empty}$$

$$\frac{\textsf{con\_lookup}\ con\ x\ = \texttt{Some}\ y \qquad t' = (t\#ts) \quad \textsf{sig\_derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ x\ con\ assms\ ts\ p\ f}{\textsf{sig\_derivation}\ ctx\ sig\_t\ sig\_k\ bnd\ x\ con\ assms\ t'\ p\ f}\ \textsc{sig\_non\_uni}$$

Figure 4.6: HybridLF M_2 proof rules - sig_derivation relation

Canonical HybridLF but actually applies the sig_non_uni rule in places where it cannot actually be used.

$$\frac{\begin{array}{c}
n = ((\max\ (\text{length}\ con)\ (\max\ (\mathsf{get_max_var_ctx}\ ctx) \\
(\mathsf{get_max_var_con}\ con)) + 1)) \\
\mathsf{bv_to_fv_type}\ t\ n = t' \\
\mathsf{translate_to_unify_ctx}\ ctx' = \mathsf{Some}\ ctxa \\
\mathsf{translate_to_unify_sig_t}\ sig_t = \mathsf{Some}\ sig_ta \\
\mathsf{translate_to_unify_sig_k}\ sig_k = \mathsf{Some}\ sig_ka \\
\mathsf{translate_to_unify_bnd}\ bnd = \mathsf{Some}\ bnda \\
\mathsf{translate_to_unify_type}\ y = \mathsf{Some}\ ya \\
\mathsf{translate_to_unify_expr}\ tm = \mathsf{Some}\ tma \\
\mathsf{var_to_umvar_type}\ y = \mathsf{Some}\ y' \\
\mathsf{var_to_umvar_type}\ (\mathsf{base_type}\ t') = \mathsf{Some}\ bt' \\
\mathsf{utransitions}\ [(x,\ [],\ ya)]\ ctxa\ sig_ta\ sig_ka\ bnda\ (\mathtt{EQNS} \\
[(\mathtt{TyEqn}\ (y',\ bt'), (\mathtt{TermEqn}\ ((\mathtt{UMVAR}\ x\ []),\ tma))])\ e \\
\mathsf{create_solution_subst}\ e = \mathsf{Some}\ s' \\
\mathsf{translate_solution_subst}\ s' = \mathsf{Some}\ s \\
\mathsf{con_lookup}\ con\ x = \mathsf{Some}\ y \\
\mathsf{split_con_after}\ con\ x = con2 \\
\mathsf{bv_to_fv_con_type}\ t\ n = \mathsf{Some}\ ctx' \\
\mathsf{con_append}\ ctx\ ctx' = \mathsf{Some}\ ctx'' \\
\mathsf{con_subst_fv}\ con2\ s = con2' \\
\mathsf{form_subst_fv}\ f\ s = f' \qquad \mathsf{assms_subst_fv}\ a\ s = a' \\
\mathsf{con_append}\ ctx''\ tm_con = \mathsf{Some}\ ctx''' \\
\mathsf{get_con_for_subst}\ s\ ctx''' = \mathsf{Some}\ con' \\
\mathsf{con_append}\ con'\ con2' = \mathsf{Some}\ con'' \\
\mathsf{derivation}\ ctx''\ sig_t\ sig_k\ bnd\ con'''\ a'\ p\ f' \\
\mathsf{con_subst_fv}\ con''\ s = con''' \\
\mathsf{sig_derivation}\ ctx''\ sig_t\ sig_k\ bnd\ x\ con\ a\ sig\ patt\ f \\
sig' = ((c,\ t)\ \#\ sig) \qquad max_var = (\text{foldl max 0 (map fst}\ ctx'')) \\
tm = ((\mathsf{make_args}\ (\mathtt{CON}\ c)\ (\mathsf{params_to_con}\ t'\ (max_var + 1))\ (max_var + 1))) \\
tm_con = (\mathsf{params_to_con}\ t'\ (max_var + 1)) \\
\mathsf{subst_all_expr_fv}\ s\ tm = tm'
\end{array}}{\mathsf{sig_derivation}\ ctx\ sig_t\ sig_k\ bnd\ x\ con\ a\ sig'\ (\mathtt{PattCase}\ p\ patt)\ f}\ \textsc{sig_uni}$$

Figure 4.7: HybridLF M_2 proof rules - sig_derivation relation (cont.)

Chapter 5

Higher-order unification in LF

5.1 Introduction

One of the key operations when implementing meta-theorem proving in LF is unification. Since variables can be of functional type, this unification is higher-order in nature. It is known that higher-order unification in general is not decidable [32], even for a second-order language with a single function constant.

Elliott [33] gives a pre-unification algorithm for terms in LF based on Huet's algorithm [34] for the simply-typed λ-calculus. As a pre-unification algorithm, it produces a set of *solved form* equations to be unified that contain only a particular kind of pair (flexible-flexible) for which unification always succeeds. It does not, however, produce a most general unifier. A key notion in Elliot's algorithm is that of *approximate well-typedness*, in which he defines functions that map terms M to simply-typed terms $\overline{M}$ and types A to simple types $\overline{A}$. Terms that are approximately well-typed can be put into *long $\beta\eta$ head-normal form*, in which terms have the form $\lambda x_1 : \sigma_1 \dots \lambda x_n : \sigma_n.\ a\ M_1 \dots M_p$ for some $n \geq 0$ and $p \geq 0$ and where a is an occurrence of a constant or variable. We refer to $\lambda x_1 \dots \lambda x_n.a$ as the *heading* of such a long $\beta\eta$ head-normal form term, and a as the *head*. Elliott shows that every approximately well-typed term has a long head-normal form (LHNF), and that every LHNF of an approximately well-typed term has the same heading. He defines three transformations that each replace a unification problem with a further set of unification problems, maintaining a typing invariant known as *acceptability* that ensures that the resulting sets of unification problems have disjoint sets of unifiers, and that the sets of unifiers are complete (i.e. that all possible unifiers are included).

Miller [36] introduces a fragment of the higher-order unification problem known as *pattern unification*, which is decidable and for which most general unifiers exist. In pattern unification, metavariables of function type must appear with a distinct series of bound variables as their arguments. Miller describes pattern unification for the simply-typed λ-calculus as part of a logic programming language L_λ.

Reed [37] describes an algorithm for higher-order pattern unification in LF, where unification for pairs of equations that fall into the pattern fragment proceeds, and unification of equations that do not fall into the pattern fragment is postponed in the hope that solutions may be found for some metavariables that will bring the equations into the pattern fragment. Reed assumes that all terms are approximately well-typed, and uses a *canonical* version of LF [38] in which only canonical (β-normal η-long) forms are representable. This is enforced through the grammar of the language, and by the use of a substitution technique known as *hereditary substitution*, in which all β-redices created during substitution are reduced in a single operation so that the result of substitution is again in canonical form. The other main device used in [37] is *contextual modal type theory*, in which metavariables have a *local context* of variables, and their arguments are represented as a substitution for these variables. He writes $u[\sigma]$ where u is the metavariable in question, and σ is the substitution representing the arguments of the metavariable.

5.2 Unification in HybridLF

We base our unification algorithm upon that of Reed. For HybridLF, the first technical consideration is that we are using a formulation of LF in which non-canonical terms can arise, in contrast to the canonical presentation of LF that Reed employs and is implemented in Canonical HybridLF. As a result, we must introduce normalisation steps into the algorithm to ensure that the equations we are unifying are in normal form. Since equations are only approximately well-typed (that is, well-typed in the simply-typed lambda calculus when all dependencies are erased) we cannot rely upon their well-typedness in LF and therefore do not know if they have canonical forms in LF.

The second main difference between our algorithm and that of Reed is that we consider unification of types, introducing *type equations* alongside the *term equations* of Reed's algorithm. This is necessary to implement the M_2 SIG_UNI rule.

Definition 153. The *approximation* $\overline{A}$ of a type A is defined as follows:

$$\begin{aligned} \overline{a} &= a \\ \overline{A\ M} &= \overline{A} \\ \overline{\Pi x : A.B} &= \overline{A} \to \overline{B} \end{aligned}$$

The approximation of a term M, written $\overline{M}$, is as follows:

$$\begin{aligned} \overline{c} &= c \\ \overline{x} &= x \\ \overline{\lambda x{:}A.M} &= \lambda x{:}\overline{A}.\overline{M} \\ \overline{M\ N} &= \overline{M}\ \overline{N} \end{aligned}$$

The approximation of a kind K, written $\overline{K}$, is as follows:

$$\begin{aligned} \overline{\mathrm{Type}} &= \mathrm{Type} \\ \overline{\Pi x{:}A.K} &= \overline{A} \to \overline{K} \end{aligned}$$

The approximation of a context Γ, written $\overline{\Gamma}$, is as follows:

$$\begin{aligned} \overline{\langle\rangle} &= \langle\rangle \\ \overline{\Gamma, x : A} &= \overline{\Gamma}, x : \overline{A} \end{aligned}$$

The approximation of a signature Σ, written $\overline{\Sigma}$ is as follows:

$$\begin{aligned} \overline{\langle\rangle} &= \langle\rangle \\ \overline{\Sigma, c : A} &= \overline{\Sigma}, c : \overline{A} \\ \overline{\Sigma, a : K} &= \overline{\Sigma}, a : \overline{K} \end{aligned}$$

We define the typing rules for the simply-typed lambda calculus $\lambda_{\to}$ in figure 5.1

Definition 154. An LF term M has approximate (simple) type A if $\overline{\Gamma} \vdash_{\lambda\to} \overline{M} : A$. M is then *approximately well-typed*.

Definition 155. A term M is in long $\beta\eta$ head normal form if it has the form $\lambda x_1{:}A_1 \ldots \lambda x_n{:}A_n.x\ M_1 \ldots M_p$ or $\lambda x_1{:}A_1 \ldots \lambda x_n{:}A_n.c\ M_1 \ldots M_p$ for some $n \geq 0$ and $p \geq 0$ and the term is fully η-expanded (that is, $x\ M_1 \ldots M_p$ or $c\ M_1 \ldots M_p$ is not of function type).

$$\frac{\Sigma(c) = A}{\Gamma \vdash_{\lambda\rightarrow} c : A}\text{ST\_CON} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash_{\lambda\rightarrow} x : A}\text{ST\_VAR}$$

$$\frac{\Gamma, x : A \vdash_{\lambda\rightarrow} M : B}{\Gamma \vdash_{\lambda\rightarrow} \lambda x : A.M : A \rightarrow B}\text{ST\_FUN}$$

$$\frac{\Gamma \vdash_{\lambda\rightarrow} M : A \rightarrow B \quad \Gamma \vdash_{\lambda\rightarrow} N : A}{\Gamma \vdash_{\lambda\rightarrow} MN : B}\text{ST\_APP}$$

Figure 5.1: Typing rules for $\lambda_\rightarrow$, the simply-typed lambda calculus

$$\frac{}{(\lambda x.M)M' \rightarrow_\beta [M'/x]M}\text{BETA\_RED} \qquad \frac{M \rightarrow_\beta M'}{MM'' \rightarrow_\beta M'M''}\text{BETA\_FST}$$

$$\frac{M' \rightarrow_\beta M''}{MM' \rightarrow_\beta MM''}\text{BETA\_SND} \qquad \frac{\Gamma \vdash_{\lambda\rightarrow} M : A \rightarrow B}{M \rightarrow_\eta \lambda x{:}A.Mx}\text{ETA} \ \ [x \notin FVM]$$

Figure 5.2: β-reduction and η-expansion for the simply-typed lambda calculus

Definition 156. We define the relations $M \rightarrow_\beta M'$ and $M \rightarrow_\eta M'$ in the conventional way in figure 5.2

The eta rule ensures that terms can be η-expanded to reach $\beta\eta$-long head normal form.

Lemma 157. Given a term M and its approximation $\overline{M}$ there is a correspondence between M and $\overline{M}$ such that if a β-redex exists in M then it exists in $\overline{M}$ and no additional β-redices are introduced in $\overline{M}$.

Proof. This follows from induction on the structure of M and definition 153. □

Lemma 158. If we reduce a β-redex in M to obtain the term M' and reduce the corresponding β-redex in $\overline{M}$ to obtain M'', we have that $M'' \equiv \overline{M'}$.

Proof. The result follows from a simple rule induction across the 3 β rules of definition 156. ☐

Lemma 159. Long head normal forms exist for all approximately well-typed terms.

Proof. Given a term M and its (simply-typed) approximation $\overline{M}$ by lemma 157 $\overline{M}$ has the same β-redices as M. We can reduce a redex of M and the corresponding redex in $\overline{M}$ to obtain M' and $\overline{M'}$ by lemma 158. Since the simply-typed λ-calculus is strongly normalising [39], if we continue to reduce β-redices in $\overline{M}$ and the corresponding redices in M we will eventually reach some $\overline{M'}$ which is in β-normal form. M' is then also in normal form. ☐

Lemma 160. All long head normal forms of an approximately well-typed term have the same heading

Proof. Given a term M and its approximation $\overline{M}$ by lemma 157 $\overline{M}$ has the same β-redices as M. Similarly to lemma 159, we can reduce a redex of M and the corresponding redex in $\overline{M}$ to obtain M' and $\overline{M'}$ by lemma 158. Since the simply-typed lambda calculus has the Church-Rosser property, it does not matter in which order we reduce the β-redices in $\overline{M}$ as eventually the sequences of reductions will reach a common approximate term $\overline{M''}$ and its equivalent M''. From $\overline{M''}$ and M'' we apply further β-reductions and η-expansions until we reach long head normal form. ☐

We follow Reed [37] in making use of contextual modal type theory, extending the language of LF terms with metavariables, written $u[\sigma]$ where u is the metavariable in question and σ is a substitution for a *local context* ψ of variables. We extend the typing judgement with a *modal context*, written Δ, which contains typing information for metavariables. The typing rules for LF are extended with an additional rule for metavariables as follows:

$$\frac{\Delta(u) = u : (\psi \vdash a\ M_1 \ldots M_n) \quad \Delta, \Gamma \vdash_\Sigma \sigma : \Gamma'}{\Delta, \Gamma \vdash_\Sigma u[\sigma] : a\ \sigma M_1 \ldots \sigma M_n}\ (n \geq 0) \quad \textsc{ty\_mvar}$$

The primary advantage of using metavariables $u[\sigma]$ with a suspended substitution σ is that metavariables of function type are *lowered* to their base type rather than explicitly having function types. Function application then becomes substitution for the local context of variables contained within the modal context.

The other addition to the LF term syntax introduced by Reed [37] is the placeholder term _. This is used to replace terms that appear as an argument to a metavariable, but cannot appear in the solution: all solutions must disregard these terms.

Definition 161. We use σ to denote an arbitrary substitution, ρ to denote a *pattern* substitution which consists of only unique-variable-for-variable and placeholder-for-variable substitutions, and ϕ to denote a *strong pattern substitution* which are similar to pattern substitutions but have no placeholders. We denote the *inversion* of a strong pattern substitution with ϕ^{-1}. If $(x/y) \in \phi$ then $(y/x) \in \phi^{-1}$, else $(\_/x) \in \phi^{-1}$.

Pattern substitutions are used in the transition rules for the arguments of a function-type metavariable to enforce that the equation falls into the pattern fragment.

Definition 162. The *intersection* of a strong pattern substitution with the identity substitution, written $\phi \cap \mathrm{id}$, substitutes the placeholder $\_$ for individual substitutions in ϕ that are not part of the identity substitution. Formally:

$$
\begin{aligned}
[] \cap \mathrm{id} &= [] \\
(\phi, (x/y)) \cap \mathrm{id} &= (\phi \cap \mathrm{id}), (\_/y) \\
(\phi, (x/x)) \cap \mathrm{id} &= (\phi \cap \mathrm{id}), (x/x)
\end{aligned}
$$

Definition 163. A *modal substitution* θ consists of a substitution of constants $c\ M_1 \ldots M_n$, bound variables $x\ M_1 \ldots M_n$ or metavariables $u[\sigma]$ for the modal variables contained within Δ. We call a modal substitution *ground* if it contains no metavariables.

Definition 164. An equation is either a *term equation*: a pair of terms to be unified, written $(M \doteq M')$, a *type equation*, written $(A \doteq A')$, or a solution, written $u \leftarrow M$, indicating that the solution for the metavariable u is M. We write ε to indicate an arbitrary equation.

Definition 165. An *equation set* Θ consists of an empty equation set $\emptyset$ or a finite conjunction of equations $\Theta' \wedge \varepsilon$.

Definition 166. Given a modal context and an equation set, a *unification problem* $\Delta \vdash \Theta$ is the problem of whether we can find a modal substitution for the metavariables in the modal context that unifies all of the equations in Θ.

Definition 167. A *solution* to $\Delta \vdash \Theta$ is a ground modal substitution θ for every metavariable in Δ so that all of the terms in θ do not contain the placeholder $\_$, for every equation $M \doteq M'$ we have $\theta M \equiv \theta M'$ and for every solution $u \leftarrow M$ we have that $(\theta M/u) \in \theta$.

We use ς to indicate a term in $\beta\eta$ long head normal form. We write $M\{M'\}$ to indicate a term M containing another term M' (and similarly $\varepsilon\{M\}$ to indicate an equation ε containing a term M). We use $M_{rig}\{M'\}$ to indicate a term M containing another term M' that is in a *rigid* position - not in the argument substitution of a metavariable. We write $M_{srig}\{M'\}$ to denote a term M containing a term M' in a *strongly rigid* position - not in the argument substitution of a metavariable, or as one of the arguments of a bound variable. For example, in the term `ABS` t (`BND` 0 $\$\$_{\mathrm{o}}$ `VAR` 0) the sub-term `VAR` 0 is in a rigid position, as it is not in the argument substitution of a metavariable, but not in a strongly rigid position, as it is an argument of a bound variable.

The result of a transition of the unification algorithm is either $\perp$ to indicate failure, or a unification problem $\Delta \vdash \Theta$

The transition rules of the unification algorithm are shown in figures 5.3 and 5.4.

In addition to the transition rules of the unification algorithm, we have β-reduction and η-expansion rules for the terms contained within the equations. These allow an arbitrary term M to be reduced to its $\beta\eta$ long head normal form. We use $M[M'/n]$ to indicate the term resulting from substituting M' for the bound variable n in the term M. These rules are shown in figure 5.5.

Although the transition rules are very similar, our algorithm contains more rules than Reed's; this is because we consider simple unification of types, and because of the restricted nature of the grammar of the canonical version of LF that Reed works with. Since we do not consider terms in spine form (as Reed does) we need to introduce multiple rules for inversion and occurs check, along with rules for type equations.

The algorithm consists of choosing applicable rules in any order except that following a use of the pruning rule, the algorithm must make use of an instantiation rule to instantiate the appropriate variable.

Decomposition

$$\Delta \vdash (\texttt{FCON}\ c \doteq \texttt{FCON}\ c') \wedge \Theta \longrightarrow \begin{cases} \Delta \vdash \Theta & (c = c') \\ \bot & (c \neq c') \end{cases}$$

$$\Delta \vdash (\texttt{FAPP}\ A\ \varsigma \doteq \texttt{FAPP}\ A'\ \varsigma') \wedge \Theta \longrightarrow (A \doteq A') \wedge (\varsigma \doteq \varsigma') \wedge \Theta$$

$$\Delta \vdash (\texttt{FPI}\ A\ B \doteq \texttt{FPI}\ A'\ B') \wedge \Theta \longrightarrow (A \doteq A') \wedge (B \doteq B') \wedge \Theta$$

$$\Delta \vdash (\texttt{ABS}\ A\ \varsigma \doteq \texttt{ABS}\ A'\ \varsigma') \wedge \Theta \longrightarrow \Delta \vdash (\varsigma \doteq \varsigma') \wedge (A \doteq A') \wedge \Theta$$

$$\Delta \vdash (\texttt{APP}\ \varsigma\ \varsigma' \doteq \texttt{APP}\ \varsigma''\ \varsigma''') \wedge \Theta \longrightarrow \Delta \vdash (\varsigma \doteq \varsigma'') \wedge (\varsigma' \doteq \varsigma''') \wedge \Theta$$

$$\Delta \vdash (\texttt{CON}\ c \doteq \texttt{CON}\ c') \wedge \Theta \longrightarrow \begin{cases} \Delta \vdash \Theta & (c = c') \\ \bot & (c \neq c') \end{cases}$$

$$\Delta \vdash (\texttt{BND}\ b \doteq \texttt{BND}\ b') \wedge \Theta \longrightarrow \begin{cases} \Delta \vdash \Theta & (b = b') \\ \bot & (b \neq b') \end{cases}$$

$$\Delta \vdash (\texttt{VAR}\ v \doteq \texttt{VAR}\ v') \wedge \Theta \longrightarrow \begin{cases} \Delta \vdash \Theta & (v = v') \\ \bot & (v \neq v') \end{cases}$$

$$\Delta \vdash \varepsilon_{rig}\{\_\} \wedge \Theta \longrightarrow \bot$$

Inversion

$$\Delta \vdash (u[\phi] \doteq \texttt{CON}\ c\ \varsigma_1 \ldots \varsigma_n) \wedge \Theta \longrightarrow \Delta \vdash (u \doteq [\phi^{-1}](\texttt{CON}\ c\ \varsigma_1 \ldots \varsigma_n)) \wedge \Theta$$

$$\Delta \vdash (u[\phi] \doteq \texttt{VAR}\ v\ \varsigma_1 \ldots \varsigma_n) \wedge \Theta \longrightarrow \Delta \vdash (u \doteq [\phi^{-1}](\texttt{BND}\ b\ \varsigma_1 \ldots \varsigma_n)) \wedge \Theta$$

$$\Delta \vdash (u[\phi] \doteq \texttt{BND}\ b\ \varsigma_1 \ldots \varsigma_n) \wedge \Theta \longrightarrow \Delta \vdash (u \doteq [\phi^{-1}](\texttt{VAR}\ v\ \varsigma_1 \ldots \varsigma_n)) \wedge \Theta$$

$$\Delta \vdash (u[\phi] \doteq v[\sigma]) \wedge \Theta \longrightarrow \Delta \vdash (u \doteq [\phi^{-1}]v[\sigma]) \wedge \Theta$$

Occurs check

$$\begin{array}{ll} \Delta \vdash (u \doteq \texttt{CON}\ c\ \varsigma_1 \ldots \varsigma_i\{u[\phi]\} \ldots \varsigma_n) \wedge \Theta \longrightarrow & \\ \qquad \Delta \vdash (u \doteq \texttt{CON}\ c\ \varsigma_1 \ldots \varsigma_i\{\_\} \ldots \varsigma_n) \wedge \Theta & (1 \leq i \leq n) \\ \Delta \vdash (u \doteq \texttt{VAR}\ v\ \varsigma_1 \ldots \varsigma_i\{u[\phi]\} \ldots \varsigma_n) \wedge \Theta \longrightarrow & \\ \qquad \Delta \vdash (u \doteq \texttt{VAR}\ v\ \varsigma_1 \ldots \varsigma_i\{\_\} \ldots \varsigma_n) \wedge \Theta & (1 \leq i \leq n) \\ \Delta \vdash (u \doteq \texttt{BND}\ b\ \varsigma_1 \ldots \varsigma_i\{u[\phi]\} \ldots \varsigma_n) \wedge \Theta \longrightarrow & \\ \qquad \Delta \vdash (u \doteq \texttt{BND}\ b\ \varsigma_1 \ldots \varsigma_i\{\_\} \ldots \varsigma_n) \wedge \Theta & (1 \leq i \leq n) \\ \Delta \vdash (u \doteq \texttt{CON}\ c\ \varsigma_1 \ldots \varsigma_{i\ srig}\{u[\sigma]\} \ldots \varsigma_n) \longrightarrow \bot & (1 \leq i \leq n) \end{array}$$

Figure 5.3: HybridLF unification algorithm transition rules

Intersection

$$\Delta \vdash (u \doteq u[\phi]) \wedge \Theta \longrightarrow \begin{cases} \Theta & \text{if } \phi \cap \mathrm{id} = \phi \\ \Delta \vdash (u \doteq u[\phi \cap \mathrm{id}] \wedge \Theta & \text{otherwise} \end{cases}$$

Pruning

$$\begin{aligned}(\Delta, u :: (\Gamma \vdash A)) \vdash \varepsilon_{rig}\{u[\rho]\} \wedge \Theta \longrightarrow\ & (\Delta, u :: (\Gamma \vdash A), v :: ((\Gamma \vdash A) \setminus x)) \vdash \\ & u \doteq v[\rho \setminus x] \wedge \varepsilon_{rig}\{u[\rho]\} \wedge \Theta\end{aligned}$$

Instantiation

$$\begin{aligned}
\Delta \vdash (u \doteq \mathtt{CON}\ c\ \varsigma_1 \ldots \varsigma_n) \wedge \Theta &\longrightarrow [\mathtt{CON}\ c\ \varsigma_1 \ldots \varsigma_n/u]\Delta \vdash \\
&(u \leftarrow \mathtt{CON}\ c\ \varsigma_1 \ldots \varsigma_n) \wedge [\mathtt{CON}\ c\ \varsigma_1 \ldots \varsigma_n/u]\Theta \\
&\qquad (u \notin \mathrm{FV}\ \mathtt{CON}\ c\ \varsigma_1 \ldots \varsigma_n, n \geq 0) \\
\Delta \vdash (u \doteq \mathtt{VAR}\ v\ \varsigma_1 \ldots \varsigma_n) \wedge \Theta &\longrightarrow [\mathtt{VAR}\ v\ \varsigma_1 \ldots \varsigma_n/u]\Delta \vdash \\
&(u \leftarrow \mathtt{VAR}\ v\ \varsigma_1 \ldots \varsigma_n) \wedge [\mathtt{VAR}\ v\ \varsigma_1 \ldots \varsigma_n/u]\Theta \\
&\qquad (u \notin \mathrm{FV}\ \mathtt{VAR}\ v\ \varsigma_1 \ldots \varsigma_n, n \geq 0) \\
\Delta \vdash (u \doteq v[\sigma]) \wedge \Theta &\longrightarrow ([v[\sigma]/u]\Delta \vdash (u \leftarrow v[\sigma])[v[\sigma]/u]\Theta
\end{aligned}$$

Figure 5.4: HybridLF unification algorithm transition rules (cont.)

$$
\begin{array}{rll}
\texttt{APP}\ (\texttt{ABS}\ A\ M)\ M' & \longrightarrow_\beta M[M'/0] & \\
M & \longrightarrow_\eta (\texttt{ABS}\ A\ (\texttt{APP}\ M\ (\texttt{BND}\ 0))) & (\Gamma \vdash_\Sigma\ M : A \to B) \\
\texttt{APP}\ M\ N & \longrightarrow \texttt{APP}\ M'\ N & (M \longrightarrow M') \\
\texttt{APP}\ M\ N & \longrightarrow \texttt{APP}\ M\ N' & (N \longrightarrow N')
\end{array}
$$

Figure 5.5: HybridLF unification β-reduction and η-expansion rules

5.3 Implementation of unification for HybridLF

5.3.1 Datatypes, levels, shifting, substitution, typing, kinding and equations

We first define datatypes `uexpr`, `utype` and `ukind` that are almost the same as the `expr`, `type` and `kind` datatypes of HybridLF except that they are extended with the `UMVAR` constructor to represent meta-variables, and `UPH` to represent the place-holder _. The `uexpr` and `utype` datatypes also lack `ERR` and `FERR`, which are used to signal an error during the conversion from Isabelle HOAS functions to de Bruijn representation, and are therefore not valid terms or types to be unified.

Definition 168 (Unification representation datatypes).

```
datatype ('a, 'b) uexpr = UCON 'a
                        | UABS "('a, 'b) utype" "('a, 'b) uexpr"
                        | UVAR nat
                        | UAPP "('a, 'b) uexpr" "('a, 'b) uexpr"
                          (infixl "$$_UO" 50)
                        | UBND nat
                        | UMVAR nat "(nat × ('a, 'b) uexpr) list"
                        | UPH

    and ('a, 'b) utype = UFPI "('a, 'b) utype" "('a, 'b) utype"
                        | UFCON 'b
                        | UFAPP "('a, 'b) utype" "('a, 'b) uexpr"
                          (infixl "$$_UF" 50)

datatype ('a, 'b) ukind = UTYPE
                        | UKPI "('a, 'b) utype" "('a, 'b) ukind"
```

We then define `ueqn`, the type of unification equations. These can be type or term equations, or a solution for a meta-variable.

Definition 169 (ueqn).

```
datatype ('a, 'b) ueqn = TermEqn "(('a, 'b) uexpr × ('a, 'b) uexpr)"
                       | TyEqn "(('a, 'b) utype × ('a, 'b) utype)"
                       | Solved "(nat × ('a, 'b) uexpr)"
```

Note that the meta-variable in the `Solved` constructor is represented by a natural number, and that the `TermEqn` and `TyEqn` constructors take as arguments pair of terms or types to be unified.

We define the datatype `ustate` that represents a particular state of the unification algorithm. This is either a list of equations, or `FAIL` to represent failure.

Definition 170 (ustate).

```
datatype ('a, 'b) ustate = EQNS "('a, 'b) ueqn list"
                         | FAIL
```

We define functions uo_level, uf_level and uk_level that determine if a `uexpr`, `utype` or `ukind` respectively is at a given level. The implementation of these functions is very similar to definitions 32, 33 and 34.

We define functions o_shift, f_shift and subst_shift that perform shifting on a `uexpr`, `utype` or substitution respectively. Their definitions are similar to definitions 35 and 36 except that they are defined over the `uexpr` and `utype` datatypes; we do not show the definitions here.

We further define functions subst, f_subst, s_subst, ustate_subst and ueqn_subst that perform substitution on unification terms, types, substitutions, states and equations respectively. The definitions of subst and f_subst are almost identical to definition 37. The definition of s_subst simply performs substitution (using subst) on all of the terms in the substitution. The definition of ueqn_subst performs substitution on the two terms or types in the equation, unless the equation is a solution, in which case it performs substitution only on the term that is the solution for the metavariable. The ustate_subst function performs substitution on all of the equations in the unification state (using ueqn_subst), or simply returns `FAIL` if it is called on `FAIL`.

We define functions umvar_subst_uexpr, umvar_subst_utype, umvar_subst_ueqn and umvar_subst_ustate that substitute a term for a metavariable in an expression, type, equation and unification state respectively. The implementation of these functions is similar to those of subst, f_subst, etc.

We create a function usubst that takes as arguments a substitution and an expression, and returns the result of applying the substitution to the expression (using the subst function). It is defined as follows:

Definition 171 (usubst).

$$\begin{aligned} \mathsf{usubst}\ []\ e &= e \\ \mathsf{usubst}\ ((n,\ x)\ \#\ xs)\ e &= \mathsf{usubst}\ xs\ (\mathsf{subst}\ n\ x\ e) \end{aligned}$$

Following the definition of substitution functions, we define typing, kinding and definitional equality relations utypeof, ukindof, uobj_def_equal, utype_def_equal, ukind_def_equal and uvalidkind for the `uexpr` and `utype` datatypes. These are almost identical to the rules defined in section 2.3, except that they are defined over `uexpr` and `utype` rather than `expr` and `type`, they take an additional argument $mctx$ for the modal context, and they have rules for meta-variables.

The additional typing rule for meta-variables is like so:

$$\frac{\mathsf{umvarlookup}\ mctx\ v = \texttt{Some}\ t \qquad \mathsf{uf\_level}\ 0\ t \qquad \mathsf{umvar\_apply\_args}\ s\ t = \texttt{Some}\ t'}{\mathsf{utypeof}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{UMVAR}\ v\ s)\ t'}\ \textsc{ty\_MVAR}$$

The umvar_apply_args function applies the arguments from the argument substitution of the meta-variable to the base type, calculating the result type of the meta-variable.

We define functions find_next_mvar_uexpr, find_next_mvar_utype, find_next_mvar_ustate and find_next_mvar_ueqn that return an integer one higher than the maximum metavariable number in an expression, type, unification state or equation respectively.

The functions subst_remove takes as arguments a substitution and a substitution pair, and returns the substitution with the pair removed. The subst_contains function takes as arguments a substitution and a term, and returns true if the substitution contains a substitution pair that substitutes the term for a variable, or false otherwise.

The remove_eqn function removes an equation from a list of equations, while the ustate_remove_eqn function removes an equation from a unification state.

5.3.2 Occurrences of terms in terms and types

We define functions occurs_rigid_uexpr and occurs_rigid_utype that determine if a term occurs in a rigid position in a given term or type.

Definition 172.

$$\begin{aligned}
\mathsf{occurs_rigid_uexpr}\ e\ (\texttt{UVAR}\ v) &= (e = (\texttt{UVAR}\ v)) \\
\mathsf{occurs_rigid_uexpr}\ e\ (\texttt{UCON}\ c) &= (e = (\texttt{UCON}\ c)) \\
\mathsf{occurs_rigid_uexpr}\ e\ (\texttt{UABS}\ t\ e') &= (e = (\texttt{UABS}\ t\ e') \lor \mathsf{occurs_rigid_uexpr}\ e\ e') \\
\mathsf{occurs_rigid_uexpr}\ e\ (\texttt{UAPP}\ a\ b) &= (e = (\texttt{UAPP}\ a\ b) \lor \mathsf{occurs_rigid_uexpr}\ e\ a\ \lor \\
&\qquad \mathsf{occurs_rigid_uexpr}\ e\ b) \\
\mathsf{occurs_rigid_uexpr}\ e\ (\texttt{UBND}\ b) &= (e = (\texttt{UBND}\ b)) \\
\mathsf{occurs_rigid_uexpr}\ e\ (\texttt{UPH}) &= (e = \texttt{UPH}) \\
\mathsf{occurs_rigid_uexpr}\ e\ (\texttt{UMVAR}\ n\ s) &= (e = (\texttt{UMVAR}\ n\ s))
\end{aligned}$$

Definition 173.

$$\begin{aligned}
\mathsf{occurs_rigid_utype}\ e\ (\texttt{UFCON}\ c) &= \texttt{False} \\
\mathsf{occurs_rigid_utype}\ e\ (\texttt{UFAPP}\ a\ b) &= (\mathsf{occurs_rigid_utype}\ e\ a \lor \mathsf{occurs_rigid_uexpr}\ e\ b) \\
\mathsf{occurs_rigid_utype}\ e\ (\texttt{UFPI}\ a\ b) &= (\mathsf{occurs_rigid_utype}\ e\ a \lor \mathsf{occurs_rigid_utype}\ e\ b)
\end{aligned}$$

Recall that a term in a rigid position is not in the argument substitution of a metavariable. occurs_rigid_uexpr and occurs_rigid_utype operate in the expected way: when determining if a term M exists in a rigid position in another term M', occurs_rigid_uexpr checks to see if $M = M'$ and then if this is not the case checks to see if M occurs rigidly in any subterms or types of M'. When determining if M exists in a rigid position in a type A occurs_rigid_utype simply checks to see if M occurs rigidly in subterms and types of A. Note that occurs_rigid_uexpr does not check to see if the terms occurs within the substitution s of a meta-variable represented by an instance of `UMVAR`: this ensures that (if found) the term is in a rigid position.

Using occurs_rigid_uexpr and occurs_rigid_utype, we can then implement the function occurs_rigid that determines if a term occurs in a rigid position within an equation.

Definition 174.

$$\begin{aligned}
\mathsf{occurs_rigid}\ e\ (\texttt{TyEqn}\ (a,\ b)) &= (\mathsf{occurs_rigid_utype}\ e\ a\ \ \lor \mathsf{occurs_rigid_utype}\ e\ b) \\
\mathsf{occurs_rigid}\ e\ (\texttt{TermEqn}\ (a,\ b)) &= (\mathsf{occurs_rigid_uexpr}\ e\ a\ \ \lor \mathsf{occurs_rigid_uexpr}\ e\ b) \\
\mathsf{occurs_rigid}\ e\ (\texttt{Solved}\ (n,\ x)) &= (e = (\texttt{UMVAR}\ n\ []) \lor \mathsf{occurs_rigid_uexpr}\ e\ x)
\end{aligned}$$

occurs_rigid simply checks in both terms or types to be unified if the term occurs in a rigid position. In solution equations, the function determines if the term is equal to the metavariable that the solution is for, and checks if the term appears in a rigid position in the term that the metavariable stands for.

We also define the occurs_bnd_head function, which determines if a term has an instance of a bound variable in its head position, so that it is of the form (`BND` b) $M_1 \ldots M_n$ for some $n \geq 0$ and b.

Definition 175.

$$\begin{aligned}
\mathsf{occurs_bnd_head}\ (\mathtt{UVAR}\ v) &= \mathtt{False}\\
\mathsf{occurs_bnd_head}\ (\mathtt{UCON}\ c) &= \mathtt{False}\\
\mathsf{occurs_bnd_head}\ (\mathtt{UBND}\ b) &= \mathtt{True}\\
\mathsf{occurs_bnd_head}\ (\mathtt{UPH}) &= \mathtt{False}\\
\mathsf{occurs_bnd_head}\ (\mathtt{UMVAR}\ v\ s) &= \mathtt{False}\\
\mathsf{occurs_bnd_head}\ (\mathtt{UABS}\ t\ e) &= \mathtt{False}\\
\mathsf{occurs_bnd_head}\ (\mathtt{UAPP}\ a\ b) &= \mathsf{occurs_bnd_head}\ a
\end{aligned}$$

We define a function occurs_top that determines if a term (except an application) occurs at the top-level (i.e. not within an abstraction).

Definition 176.

$$\begin{aligned}
\mathsf{occurs_top}\ e\ (\mathtt{UVAR}\ v) &= (e = \mathtt{UVAR}\ v)\\
\mathsf{occurs_top}\ e\ (\mathtt{UCON}\ c) &= (e = \mathtt{UCON}\ c)\\
\mathsf{occurs_top}\ e\ (\mathtt{UBND}\ b) &= (e = \mathtt{UBND}\ b)\\
\mathsf{occurs_top}\ e\ \mathtt{UPH} &= (e = \mathtt{UPH})\\
\mathsf{occurs_top}\ e\ (\mathtt{UMVAR}\ v\ s) &= (e = \mathtt{UMVAR}\ v\ s)\\
\mathsf{occurs_top}\ e\ (\mathtt{UABS}\ t\ e') &= (e = \mathtt{UABS}\ t\ e')\\
\mathsf{occurs_top}\ e\ (\mathtt{UAPP}\ a\ b) &= (\mathsf{occurs_top}\ e\ a \vee \mathsf{occurs_top}\ e\ b)
\end{aligned}$$

We can then define a function is_argument_to_bound, which determines if a term (except an application) occurs as an argument to a bound variable within another term.

Definition 177.

$$\begin{aligned}
\mathsf{is_argument_to_bound}\ e\ (\texttt{UVAR}\ v) &= \texttt{False}\\
\mathsf{is_argument_to_bound}\ e\ (\texttt{UBND}\ b) &= \texttt{False}\\
\mathsf{is_argument_to_bound}\ e\ (\texttt{UCON}\ c) &= \texttt{False}\\
\mathsf{is_argument_to_bound}\ e\ \texttt{UPH} &= \texttt{False}\\
\mathsf{is_argument_to_bound}\ e\ (\texttt{UMVAR}\ v\ s) &= \texttt{False}\\
\mathsf{is_argument_to_bound}\ e\ (\texttt{UABS}\ t\ e') &= \mathsf{is_argument_to_bound}\ e\ e'\\
\mathsf{is_argument_to_bound}\ e\ (\texttt{UAPP}\ a\ b) &= ((\mathsf{occurs_bnd_head}\ a \land (\mathsf{occurs_top}\ e\ a\\
&\quad \lor \mathsf{occurs_top}\ e\ b))\\
&\quad \lor \mathsf{is_argument_to_bound}\ e\ a\\
&\quad \lor \mathsf{is_argument_to_bound}\ e\ b)
\end{aligned}$$

Using the previous few functions we can define functions occurs_strongly_rigid_uexpr and occurs_strongly_rigid_utype that determine if a term occurs in a strongly rigid position within a term or type. Recall that a term exists in a strongly rigid position if it is not an argument to a metavariable or an argument to a bound variable.

Definition 178.

$$\begin{aligned}
\mathsf{occurs_strongly_rigid_uexpr}\ e\ (\texttt{UVAR}\ v) &= (e = \texttt{UVAR}\ v)\\
\mathsf{occurs_strongly_rigid_uexpr}\ e\ (\texttt{UCON}\ c) &= (e = \texttt{UCON}\ c)\\
\mathsf{occurs_strongly_rigid_uexpr}\ e\ (\texttt{UABS}\ t\ e') &= (e = (\texttt{UABS}\ t\ e')\\
&\quad \lor \mathsf{occurs_strongly_rigid_uexpr}\ e\ e'\\
&\quad \lor \mathsf{occurs_strongly_rigid_utype}\ e\ t)\\
\mathsf{occurs_strongly_rigid_uexpr}\ e\ (\texttt{UAPP}\ a\ b) &= (e = (\texttt{UAPP}\ a\ b) \lor\\
&\quad ((\mathsf{occurs_strongly_rigid_uexpr}\ e\ a\\
&\quad \lor \mathsf{occurs_strongly_rigid_uexpr}\ e\ b)\\
&\quad \land \neg(\mathsf{occurs_bnd_head}\ a\\
&\quad \land (\mathsf{occurs_top}\ e\ a\\
&\quad \lor \mathsf{occurs_top}\ e\ b))))\\
\mathsf{occurs_strongly_rigid_uexpr}\ e\ (\texttt{UBND}\ b) &= (e = \texttt{UBND}\ b)\\
\mathsf{occurs_strongly_rigid_uexpr}\ e\ \texttt{UPH} &= (e = \texttt{UPH})\\
\mathsf{occurs_strongly_rigid_uexpr}\ e\ (\texttt{UMVAR}\ n\ s) &= (e = \texttt{UMVAR}\ n\ s)
\end{aligned}$$

Definition 179.

$$
\begin{aligned}
\mathsf{occurs\_strongly\_rigid\_utype}\ e\ (\texttt{UFCON}\ c) &= \texttt{False} \\
\mathsf{occurs\_strongly\_rigid\_utype}\ e\ (\texttt{UFAPP}\ a\ b) &= (\mathsf{occurs\_strongly\_rigid\_utype}\ e\ a \\
&\qquad \vee \mathsf{occurs\_strongly\_rigid\_uexpr}\ e\ b) \\
\mathsf{occurs\_strongly\_rigid\_utype}\ e\ (\texttt{UFPI}\ a\ b) &= (\mathsf{occurs\_strongly\_rigid\_utype}\ e\ a \\
&\qquad \vee \mathsf{occurs\_strongly\_rigid\_utype}\ e\ b)
\end{aligned}
$$

So for example if we have the term

$$M = \texttt{ABS}\ (\texttt{UFCON}\ t\ \$\$_{\text{UF}}\ (\texttt{UCON}\ c))\ (\texttt{UBND}\ 0\ \$\$_{\text{UO}}\ \texttt{UCON}\ c')$$

then $\mathsf{occurs\_strongly\_rigid\_uexpr}$ ($\texttt{UCON}$ c) M will evaluate to `True`, as the occurrence of `UCON` c is not an argument to a metavariable or a bound variable. On the other hand, $\mathsf{occurs\_strongly\_rigid\_uexpr}$ ($\texttt{UCON}$ c') M will evaluate to `False`, as the only occurrence of `UCON` c' is an argument to a bound variable.

We can now define the function $\mathsf{occurs\_strongly\_rigid}$, which checks to see if a term exists in a strongly rigid position within an equation. In the `TermEqn` and `TyEqn` cases, this function simply checks the two terms or types within the equation. In the `Solved` case, the function checks if the term that is to occur in the strongly rigid position matches the metavariable in the solution, and then checks to see if an occurrence exists within the term that the solution is assigning to the metavariable.

Definition 180.

$$
\begin{aligned}
\mathsf{occurs\_strongly\_rigid}\ e\ (\texttt{TyEqn}\ (a,\ b)) &= (\mathsf{occurs\_strongly\_rigid\_utype}\ e\ a \\
&\qquad \vee \mathsf{occurs\_strongly\_rigid\_utype}\ e\ b) \\
\mathsf{occurs\_strongly\_rigid}\ e\ (\texttt{TermEqn}\ (a,\ b)) &= (\mathsf{occurs\_strongly\_rigid\_uexpr}\ e\ a \\
&\qquad \vee \mathsf{occurs\_strongly\_rigid\_uexpr}\ e\ b) \\
\mathsf{occurs\_strongly\_rigid}\ e\ (\texttt{Solved}\ (n,\ x)) &= (e = (\texttt{UMVAR}\ n\ []) \\
&\qquad \vee \mathsf{occurs\_strongly\_rigid\_uexpr}\ e\ x)
\end{aligned}
$$

We create a function $\mathsf{occurs\_strongly\_rigid\_mvar}$ that determines if a metavariable occurs in a strongly rigid position within an equation. It is very similar in definition to $\mathsf{occurs\_strongly\_rigid}$, except that the parameter e is a natural number for the number of the metavariable rather than a term. The definition of $\mathsf{occurs\_strongly\_rigid\_mvar}$ uses two functions, $\mathsf{occurs\_strongly\_rigid\_mvar\_uexpr}$ and $\mathsf{occurs\_strongly\_rigid\_mvar\_utype}$. We will not show the definitions of these functions here.

We define a function replace_mvar_with_ph that replaces a metavariable in a term with an instance of the placeholder `UPH` if the argument substitution to the metavariable is not empty. The definition of this function is straightforward.

5.3.3 Pattern substitutions

Recall that a pattern substitution is one that consists entirely of distinct bound variables and placeholders. We define a function is_pattern_subst' that determines if a substitution is a pattern substitution, taking a substitution and a set as parameters. If the substitution is empty, the result of this function is trivially true. If the substitution is non-empty we consider the type of the term in the substitution pair at the head of the substitution. If this is a bound variable, and the bound variable does not appear in the set that is given as the second parameter to the function, we add the variable to the set and recursively call is_pattern_subst' on the rest of the list making up the substitution. If the bound variable does already appear in the set, we know that the bound variables appearing in the substitution are not distinct, so the substitution is not a pattern substitution. If the term in the pair at the head of the substitution is the placeholder `UPH`, we again call the is_pattern_subst' function recursively on the rest of the substitution. If the term is anything else, we know that the substitution cannot be pattern.

Definition 181 (is_pattern_subst')**.**

$$\begin{aligned}
\textsf{is_pattern_subst'}\ []\ _ &= \texttt{True}\\
\textsf{is_pattern_subst'}\ ((x,\ y)\ \#\ xs)\ s &= (\texttt{case}\ y\ \texttt{of}\ (\texttt{UBND}\ b) \Rightarrow (\texttt{if}\ (b \notin s)\ \texttt{then}\\
&\qquad (\textsf{is_pattern_subst'}\ xs\ (b \cup s))\ \texttt{else False})\\
&\qquad \mid \texttt{UPH} \Rightarrow (\textsf{is_pattern_subst'}\ xs\ s) \mid _ \Rightarrow \texttt{False})
\end{aligned}$$

The is_pattern_subst' function is intended to be initially called with an empty set as the second parameter. We define a function is_pattern_subst that does exactly this:

Definition 182 (is_pattern_subst)**.**

$$\textsf{is\_pattern\_subst}\ x = \textsf{is\_pattern\_subst'}\ x\ \emptyset$$

We define a function is_strong_pattern_subst' that determines if a function is a strong pattern substitution. This function works in the same way as is_pattern_subst', except that instances of the placeholder UPH in the substitution pair produce a negative result. This is in line with the definition of strong pattern substitutions, which requires that the substitution be a pattern substitution with no placeholders

Definition 183 (is_strong_pattern_subst').

$$\begin{aligned}\textsf{is\_strong\_pattern\_subst'}\ []\ \_ &= \texttt{True}\\ \textsf{is\_strong\_pattern\_subst'}\ ((x,\ y)\ \#\ xs)\ s &= (\texttt{case}\ y\ \texttt{of}\ (\texttt{UBND}\ b) \Rightarrow (\texttt{if}\ (b \notin s)\ \texttt{then}\\ &\quad (\textsf{is\_strong\_pattern\_subst'}\ xs\ (b \cup s))\\ &\quad \texttt{else False})\ |\ \_ \Rightarrow \texttt{False})\end{aligned}$$

Similarly to is_pattern_subst, we define a function is_strong_pattern_subst that simply calls is_strong_pattern_subst' with an empty set:

Definition 184 (is_strong_pattern_subst).

$$\textsf{is\_strong\_pattern\_subst}\ x = \textsf{is\_strong\_pattern\_subst'}\ x\ \emptyset$$

To implement unification, we need two operations on pattern substitutions: inversion and intersection with identity.

The function ureplace takes a substitution s, a natural number n and a term e as arguments and returns a substitution in which the term to substitute for the variable for the first pair whose variable to be substituted for matches n is replaced by e.

Definition 185 (ureplace).

$$\begin{aligned}\textsf{ureplace}\ []\ \_\ e &= \texttt{None}\\ \textsf{ureplace}\ ((x,\ y)\ \#\ xs)\ n\ e &= (\texttt{if}\ x = n\ \texttt{then}\ (\texttt{Some}\ ((x,\ e)\ \#\ xs))\ \texttt{else}\ (\texttt{case}\\ &\quad (\textsf{ureplace}\ xs\ n\ e)\ \texttt{of}\ (\texttt{Some}\ l) \Rightarrow \texttt{Some}\ ((x,\ y)\ \#\ l)\\ &\quad |\ \texttt{None} \Rightarrow \texttt{None}))\end{aligned}$$

We define a function invert' that is used during inversion of a strong pattern substitution. Recall that inversion returns (y/x) if (x/y) is defined in the substitution, and $(\_/x)$ otherwise. invert' takes as arguments two substitutions and returns a substitution option.

Definition 186 (invert').

$$\begin{aligned}
\mathsf{invert'}\ []\ l &= \mathtt{Some}\ l \\
\mathsf{invert'}\ ((x,\ (\mathtt{UBND}\ b))\ \#\ xs)\ s &= (\mathtt{case}\ (\mathsf{ureplace}\ s\ b\ (\mathtt{UBND}\ x))\ \mathtt{of}\ \mathtt{Some}\ l \Rightarrow \\
&\qquad \mathsf{invert'}\ xs\ l \mid \mathtt{None} \Rightarrow \mathtt{None}) \\
\mathsf{invert'}(_\ \#\ xs)\ s &= \mathtt{None}
\end{aligned}$$

Note that invert' returns None if the substitution is not a strong pattern substitution, due to the last equation.

The function invert uses the function invert' to carry out inversion on a strong pattern substitution. invert takes as its argument the substitution to perform inversion on. It uses a function create_ph_list, which simply creates a substitution containing pairs substituting the placeholder UPH for all variables from zero to the specified number. In invert, since we fold the max function across the variable numbers to substitute for of all of the pairs in the substitution, this will produce a list of pairs from zero to the maximum variable number in the substitution, substituting the placeholder for each variable. When we call invert' with this substitution as its second argument, invert' works through the substitution given as its first argument. It uses the ureplace function to replace the pair in its second argument corresponding to the number of the bound variable from the term of the current pair from its first argument with the inverted current pair. The result of inversion is a substitution containing placeholders for all terms except those that substitute a bound variable for another variable in the original substitution.

Definition 187 (invert).

$$\begin{aligned}
\mathsf{invert}\ [] &= \mathtt{Some}[] \\
\mathsf{invert}\ xs &= \mathsf{invert'}\ xs\ (\mathsf{create_ph_list}\ (\mathrm{foldl}\ \mathrm{max}\ 0\ (\mathrm{map}\ \mathrm{fst}\ xs)))
\end{aligned}$$

The other operation on strong pattern substitutions that needs to be implemented for unification is intersection with identity.

We define a function intersection_id that takes a substitution as argument, and returns a substitution in which the identity pairs remain the same, and the term in every other pair is replaced by the placeholder UPH.

$$\frac{\mathsf{utypeof}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b\ t \qquad c = \mathsf{subst}\ 0\ a\ b}{\mathsf{red\_step}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ ((\mathtt{UABS}\ t\ a)\ \$\$_{\text{UO}}\ b)\ c}\ \text{ABS\_RED}$$

$$\frac{\mathsf{red\_step}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ t1\ t1'}{\mathsf{red\_step}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (t1\ \$\$_{\text{UO}}\ t2)\ (t1'\ \$\$_{\text{UO}}\ t2)}\ \text{APP\_RED1}$$

$$\frac{\mathsf{red\_step}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ t2\ t2'}{\mathsf{red\_step}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (t1\ \$\$_{\text{UO}}\ t2)\ (t1\ \$\$_{\text{UO}}\ t2')}\ \text{APP\_RED2}$$

$$\frac{\mathsf{utypeof}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ m\ (\mathtt{UFPI}\ a\ b)}{\mathsf{red\_step}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ m\ (\mathtt{UABS}\ a\ m\ \$\$_{\text{UO}}\ (\mathtt{UBND}\ 0))}\ \text{ETA\_EXP}$$

Figure 5.6: HybridLF unification red_step relation

Definition 188 (intersection_id)**.**

$$\begin{aligned}\mathsf{intersection\_id}\ [] &= []\\ \mathsf{intersection\_id}\ ((x,\ y)\ \#\ xs) &= (\mathtt{if}\ y = (\mathtt{UBND}\ x)\ \mathtt{then}\ (x,\ (\mathtt{UBND}\ x))\ \mathtt{else}\\ &\qquad (x,\ \mathtt{UPH}))\ \#\ (\mathsf{intersection\_id}\ xs)\end{aligned}$$

5.3.4 Normal forms and reductions

We have a relation red_step that defines single-step β-reductions and η-expansions. It is shown in figure 5.6.

We then define a relation reduce that produces the reflexive transitive closure of reduction steps. This relation is shown in figure 5.7.

The head function finds the head of a term in long head-normal form:

$$\frac{}{\mathsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ m\ m}\ \text{RED\_NONE}$$

$$\frac{\mathsf{red\_step}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ m\ m'}{\mathsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ m\ m'}\ \text{RED\_SINGLE}$$

$$\frac{\begin{array}{c}\mathsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ m\ m' \\ \mathsf{red\_step}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ m'\ m''\end{array}}{\mathsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ m\ m''}\ \text{RED\_TRANS}$$

Figure 5.7: HybridLF unification reduce relation

Definition 189 (head).

$$\begin{aligned}
\mathsf{head}\ (\mathtt{UCON}\ c) &= \mathtt{Some}\ (\mathtt{UCON}\ c)\\
\mathsf{head}\ (\mathtt{UVAR}\ v) &= \mathtt{Some}\ (\mathtt{UVAR}\ v)\\
\mathsf{head}\ (\mathtt{UMVAR}\ v\ s) &= \mathtt{Some}\ (\mathtt{UMVAR}\ v\ s)\\
\mathsf{head}\ (\mathtt{UABS}\ t\ e) &= \mathsf{head}\ e\\
\mathsf{head}\ (\mathtt{UBND}\ b) &= \mathtt{Some}\ (\mathtt{UBND}\ b)\\
\mathsf{head}\ \mathtt{UPH} &= \mathtt{None}\\
\mathsf{head}\ (\mathtt{UAPP}\ a\ b) &= \mathsf{head}\ a
\end{aligned}$$

The head is either a constant or a bound variable.

We define a find_body function that finds the body of a term:

Definition 190 (find_body).

$$\begin{aligned}
\mathsf{find_body}\ (\mathtt{UCON}\ c) &= (\mathtt{UCON}\ c)\\
\mathsf{find_body}\ \mathtt{UPH} &= \mathtt{UPH}\\
\mathsf{find_body}\ (\mathtt{UMVAR}\ v\ s) &= (\mathtt{UMVAR}\ v\ s)\\
\mathsf{find_body}\ (\mathtt{UABS}\ t\ e) &= (\mathsf{find_body}\ e)\\
\mathsf{find_body}\ (\mathtt{UBND}\ b) &= (\mathtt{UBND}\ b)\\
\mathsf{find_body}\ (\mathtt{UVAR}\ v) &= (\mathtt{UVAR}\ v)\\
\mathsf{find_body}\ (\mathtt{UAPP}\ a\ b) &= (\mathtt{UAPP}\ a\ b)
\end{aligned}$$

$$\frac{\begin{array}{c}\mathsf{head}\ e = \mathtt{Some}\ (\mathtt{UBND}\ n)\ \lor\ \mathsf{head}\ e = \mathtt{Some}\ (\mathtt{UCON}\ c) \lor \\ \mathsf{head}\ e = \mathtt{Some}\ (\mathtt{UMVAR}\ v\ s)\ \lor\ \mathsf{head}\ e = \mathtt{Some}\ (\mathtt{UVAR}\ v') \\ \mathsf{utypeof}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathsf{find\_body}\ e)\ t \\ \forall a.\ \forall b.\ t \neq (\mathtt{FPI}\ a\ b)\end{array}}{\mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e}\ \text{LHNF}$$

$$\frac{}{\mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{UVAR}\ v)}\ \text{LHNF\_VAR}$$

$$\frac{}{\mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{UBND}\ b)}\ \text{LHNF\_BND}$$

$$\frac{}{\mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{UMVAR}\ v\ s)}\ \text{LHNF\_MVAR}$$

Figure 5.8: Rules for lhnf

Using the head and find_body functions, we can now define a relation lhnf that holds if a term is in long head-normal form. It is shown in figure 5.8.

We define a further relation all_lhnf that determines if all of the terms in an application are in long head-normal form. It is shown in figure 5.9

5.3.5 Transition rules

We define a relation utransition that implements the transition rules of the unification algorithm. The rules are given in figures 5.10, 5.11, 5.12 and 5.13.

We define a further relation utransitions, which implements the transitive closure of the transition relation, in figure 5.14.

5.4 Unification for Canonical HybridLF

Unification for Canonical HybridLF is simpler than that for HybridLF, because since only canonical forms can exist we do not need reduction rules, nor do we need to explicitly state that terms are in long head-normal form.

$$\frac{\begin{array}{c}\mathsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ a\\ \mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b\end{array}}{\mathsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{UAPP}\ a\ b)}\ \textsc{all\_lhnf\_app}$$

$$\frac{}{\mathsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{UBND}\ b)}\ \textsc{all\_lhnf\_bnd}$$

$$\frac{}{\mathsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\mathtt{UCON}\ c)}\ \textsc{all\_lhnf\_con}$$

Figure 5.9: HybridLF unification all_lhnf relation

5.4.1 Transition rules

The transition rules for unification in Canonical HybridLF are very similar to those for HybridLF. They are shown in figure 5.15

5.4.2 Implementation of unification in Canonical HybridLF

The implementation of unification for Canonical HybridLF is very similar to the implementation of unification for HybridLF.

The main difference is in the datatypes used for the representation of terms and types (extended with the metavariables and placeholder).

Instead of the `uexpr` and `utype` datatypes defined in the HybridLF implementation of unification, we have 5 (mutually defined) datatypes: `ukind`, `uctype`, `uatype`, `ucterm` and `uaterm`, corresponding to kinds, canonical types, atomic types, canonical terms and atomic terms respectively.

Their definition is like so:

$$\frac{e \in \text{set } s \qquad \mathsf{occurs\_rigid}\ \texttt{UPH}\ e}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ \texttt{FAIL}}\ \text{RIGID\_PH}$$

$$\frac{\begin{array}{c}(\texttt{TyEqn}((\texttt{UFCON}\ c),(\texttt{UFCON}\ c))) \in \text{set } s\\ s' = (\mathsf{remove\_eqn}\ s\ (\texttt{TyEqn}\ (\texttt{UFCON}\ c, \texttt{UFCON}\ c)))\end{array}}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UFCON\_SAME}$$

$$\frac{(\texttt{TyEqn}((\texttt{UFCON}\ c),(\texttt{UFCON}\ c'))) \in \text{set } s \qquad c \neq c'}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ \texttt{FAIL}}\ \text{UFCON\_DIFF}$$

$$\frac{\begin{array}{c}\texttt{TyEqn}\ (a\ \$\$_{\text{UF}}\ b,\ a'\ \$\$_{\text{UF}}\ b')) \in \text{set } s\\ \mathsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b\ b''\\ \mathsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b'\ b'''\\ \mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b''\\ \mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b'''\\ s' = ([(\texttt{TyEqn}\ (a,\ a')),\ (\texttt{TermEqn}\ (b,\ b'))]@\\ (\mathsf{remove\_eqn}\ s\ (\texttt{TyEqn}\ (a\ \$\$_{\text{UF}}\ b, a'\ \$\$_{\text{UF}}\ b'))))\end{array}}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UFAPP}$$

$$\frac{\begin{array}{c}(\texttt{TyEqn}\ (\texttt{UFPI}\ a\ b, \texttt{UFPI}\ a'\ b')) \in \text{set } s\\ s' = ([(\texttt{TyEqn}\ (a,\ a')), (\texttt{TyEqn}\ (b,\ b'))]@\\ (\mathsf{remove\_eqn}\ s\ (\texttt{TyEqn}\ (\texttt{UFPI}\ a\ b, \texttt{UFPI}\ a'\ b'))))\end{array}}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UFPI}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}(\texttt{UABS}\ t\ e, \texttt{UABS}\ t'\ e')) \in \text{set } s\\ \mathsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ (t\ \#\ bnd)\ e\ e''\\ \mathsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ (t'\ \#\ bnd)\ e'\ e'''\\ \mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ (t\ \#\ bnd)\ e''\\ \mathsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ (t'\ \#\ bnd)\ e'''\\ s' = ((\texttt{TyEqn}\ (t,\ t'))\#(\texttt{TermEqn}\ (e,\ e'))\#\\ (\mathsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UABS}\ t\ e, \texttt{UABS}\ t'\ e'))))\end{array}}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UABS}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UCON}\ c,\ \texttt{UCON}\ c)) \in \text{set } s\\ s' = (\mathsf{remove\_eqn}\ s\ ((\texttt{TermEqn}\ (\texttt{UCON}\ c, \texttt{UCON}\ c))))\end{array}}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UCON\_SAME}$$

$$\frac{(\texttt{TermEqn}\ (\texttt{UCON}\ c,\ \texttt{UCON}\ c')) \in \text{set } s \qquad c \neq c'}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ \texttt{FAIL}}\ \text{UCON\_DIFF}$$

Figure 5.10: Implementation of HybridLF unification transition rules

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UBND}\ n,\ \texttt{UBND}\ n)) \in \text{set}\ s \\ s' = (\textsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UBND}\ n, \texttt{UBND}\ n)))\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UBND\_SAME}$$

$$\frac{(\texttt{TermEqn}\ (\texttt{UBND}\ n,\ \texttt{UBND}\ n')) \in \text{set}\ s \qquad n \neq n'}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ \texttt{FAIL}}\ \text{UBND\_DIFF}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (a\ \$\$_{\text{UO}}\ b,\ a'\ \$\$_{\text{UO}}\ b')) \in \text{set}\ s \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ a\ a'' \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ a'\ a''' \\ \textsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ a'' \\ \textsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ a''' \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b\ b'' \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b'\ b''' \\ \textsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b'' \\ \textsf{lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ b''' \\ s' = ([(\texttt{TermEqn}\ (a,\ a')), (\texttt{TermEqn}(b,\ b'))] @ \\ (\textsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (a\ \$\$_{\text{UO}}\ b,\ a'\ \$\$_{\text{UO}}\ b'))))\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ S)\ (\texttt{EQNS}\ s')}\ \text{UAPP}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UVAR}\ v, \texttt{UVAR}\ v)) \in \text{set}\ s \\ s' = (\textsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UVAR}\ v, \texttt{UVAR}\ v)))\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UVAR\_SAME}$$

$$\frac{(\texttt{TermEqn}\ (\texttt{UVAR}\ v, \texttt{UVAR}\ v')) \in \text{set}\ s \qquad v \neq v'}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ \texttt{FAIL}}\ \text{UVAR\_DIFF}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ ((\texttt{UMVAR}\ v\ s),\ e)) \in \text{set}\ s'' \\ \textsf{head}\ e = \textsf{Some}\ (\texttt{UCON}\ c) \vee \textsf{head}\ e = \textsf{Some}\ (\texttt{UBND}\ b)\ \vee \\ \textsf{head}\ e = \textsf{Some}\ (\texttt{UVAR}\ v) \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e\ e' \\ \textsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e' \\ \textsf{is\_strong\_pattern\_subst}\ s \qquad \textsf{invert}\ s = \textsf{Some}\ s' \\ s''' = ((\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [],\ \textsf{usubst}\ s'\ e'))\# \\ (\textsf{remove\_eqn}((\texttt{UMVAR}\ v\ s),\ e)))\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s'')(\texttt{EQNS} s''')}\ \text{INVERT}$$

Figure 5.11: Implementation of HybridLF unification transition rules (cont. 1)

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [],\ (\texttt{UCON}\ c)\ \$\$_{\text{UO}}\ e)) \in \text{set}\ s \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e\ e' \\ \textsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e' \\ \textsf{replace\_mvar\_with\_ph}((\texttt{UCON}\ c)\ \$\$_{\text{UO}}\ e')\ v = e'' \\ s' = ((\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], e''))\ \# \\ (\textsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], (\texttt{UCON}\ c)\ \$\$_{\text{UO}}\ e))))\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{OCCURS\_UCON}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [],\ (\texttt{UBND}\ b)\ \$\$_{\text{UO}}\ e)) \in \text{set}\ s \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e\ e' \\ \textsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e' \\ \textsf{replace\_mvar\_with\_ph}((\texttt{UBND}\ b)\ \$\$_{\text{UO}}\ e')\ v = e'' \\ s' = ((\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], e''))\ \# \\ (\textsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], (\texttt{UBND}\ b)\ \$\$_{\text{UO}}\ e))))\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{OCCURS\_UBND}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [],\ (\texttt{UVAR}\ v')\ \$\$_{\text{UO}}\ e)) \in \text{set}\ s \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e\ e' \\ \textsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e' \\ \textsf{replace\_mvar\_with\_ph}((\texttt{UVAR}\ v')\ \$\$_{\text{UO}}\ e')\ v = e'' \\ s' = ((\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], e''))\ \# \\ (\textsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], (\texttt{UVAR}\ v')\ \$\$_{\text{UO}}\ e))))\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{OCCURS\_UVAR}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [],\ ((\texttt{UCON}\ c)\ \$\$_{\text{UO}}\ e))) \in \text{set}\ s \\ \textsf{reduce}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e\ e' \\ \textsf{all\_lhnf}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ e' \\ \textsf{occurs\_strongly\_rigid\_mvar\_uexpr}\ v\ e'\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ \texttt{FAIL}}\ \text{OCCURS\_RIGID\_FAIL}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [],\ \texttt{UMVAR}\ v\ s)) \in \text{set}\ s' \qquad \textsf{intersection\_id}\ s = s \\ s'' = (\textsf{remove\_eqn}\ s'\ (\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], \texttt{UMVAR}\ v\ s)))\end{array}}{\textsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s')\ (\texttt{EQNS}\ s'')\text{''}}\ \text{INTERSECTION\_SAME}$$

Figure 5.12: Implementation of HybridLF unification transition rules (cont. 2)

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], \texttt{UMVAR}\ v\ s))\ \in \text{set}\ t\\ \mathsf{intersection\_id}\ s = s' \qquad s' \neq s\\ t' = (((\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], \texttt{UMVAR}\ v\ s')))\#\\ (\mathsf{remove\_eqn}\ t\ (\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [], \texttt{UMVAR}\ v\ s))))\end{array}}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ t)\ (\texttt{EQNS}\ t')}\ \text{INTERSECTION\_DIFF}$$

$$\frac{\begin{array}{c}eqn \in \text{set}t \qquad \mathsf{find\_next\_mvar\_ustate}(\texttt{EQNS}\ t) = v\\ \mathsf{occurs\_rigid}\ (\texttt{UMVAR}\ v'\ s)\ eqn \qquad \mathsf{is\_pattern\_subst}\ s; \setminus\ x\ s = \texttt{Some}\ s'\\ t' = (((\texttt{TermEqn}(\texttt{UMVAR}\ v'\ [],\ \texttt{UMVAR}\ v\ s'))\ \#\ t))\end{array}}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ t)\ (\texttt{EQNS}\ t')}\ \text{PRUNING}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [],\ e)) \in \text{set}\ s\\ s' = (\texttt{Solved}\ (v,\ e))\#(\text{map}\ = (\lambda x.\ \mathsf{umvar\_subst\_ueqn}\ x\ v\ e)\\ (\mathsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UMVAR}\ v\ [],\ e))))\end{array}}{\mathsf{utransition}\ mctx\ ctx\ sig\_t\ sig\_k\ bnd\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')\text{''}}\ \text{NSTANTIATE}$$

Figure 5.13: Implementation of HybridLF unification transition rules (cont. 3)

$$\frac{\mathsf{utransition}\ umctx\ ctx\ sig\_t\ sig\_k\ bnd\ s\ s}{\mathsf{utransitions}\ umctx\ ctx\ sig\_t\ sig\_k\ bnd\ s\ s'}\ \text{UTRANS\_SINGLE}$$

$$\frac{\begin{array}{c}\mathsf{utransition}\ umctx\ ctx\ sig\_t\ sig\_k\ bnd\ s\ s'\\ \mathsf{utransitions}\ umctx\ ctx\ sig\_t\ sig\_k\ bnd\ s'\ s''\end{array}}{\mathsf{utransitions}\ umctx\ ctx\ sig\_t\ sig\_k\ bnd\ s\ s''}\ \text{UTRANS\_TRANS}$$

Figure 5.14: utransitions in HybridLF

Decomposition

$$\Delta \vdash (\texttt{FCON}\ c \doteq \texttt{FCON}\ c') \wedge \Theta \longrightarrow \begin{cases} \Delta \vdash \Theta & (c = c') \\ \bot & (c \neq c') \end{cases}$$

$$\Delta \vdash (\texttt{FAPP}\ A\ M \doteq \texttt{FAPP}\ A'\ M') \wedge \Theta \longrightarrow (A \doteq A') \wedge (M \doteq M') \wedge \Theta$$

$$\Delta \vdash (\texttt{PI}\ A\ B \doteq \texttt{PI}\ A'\ B') \wedge \Theta \longrightarrow (A \doteq A') \wedge (B \doteq B') \wedge \Theta$$

$$\Delta \vdash (\texttt{ABS}\ A\ M \doteq \texttt{ABS}\ A'\ M') \wedge \Theta \longrightarrow \Delta \vdash (M \doteq M') \wedge (A \doteq A') \wedge \Theta$$

$$\Delta \vdash (\texttt{APP}\ M\ N \doteq \texttt{APP}\ M'\ N') \wedge \Theta \longrightarrow \Delta \vdash (M \doteq M') \wedge (N \doteq N') \wedge \Theta$$

$$\Delta \vdash (\texttt{CON}\ c \doteq \texttt{CON}\ c') \wedge \Theta \longrightarrow \begin{cases} \Delta \vdash \Theta & (c = c') \\ \bot & (c \neq c') \end{cases}$$

$$\Delta \vdash (\texttt{BND}\ b \doteq \texttt{BND}\ b') \wedge \Theta \longrightarrow \begin{cases} \Delta \vdash \Theta & (b = b') \\ \bot & (b \neq b') \end{cases}$$

$$\Delta \vdash (\texttt{VAR}\ v \doteq \texttt{VAR}\ v') \wedge \Theta \longrightarrow \begin{cases} \Delta \vdash \Theta & (v = v') \\ \bot & (v \neq v') \end{cases}$$

$$\Delta \vdash \varepsilon_{rig}\{\_\} \wedge \Theta \longrightarrow \bot$$

Inversion

$$\Delta \vdash (u[\phi] \doteq \texttt{CON}\ c\ M_1 \ldots M_n) \wedge \Theta \longrightarrow \Delta \vdash (u \doteq [\phi^{-1}](\texttt{CON}\ c\ M_1 \ldots M_n)) \wedge \Theta$$

$$\Delta \vdash (u[\phi] \doteq \texttt{VAR}\ v\ M_1 \ldots M_n) \wedge \Theta \longrightarrow \Delta \vdash (u \doteq [\phi^{-1}](\texttt{BND}\ b\ M_1 \ldots M_n)) \wedge \Theta$$

$$\Delta \vdash (u[\phi] \doteq \texttt{BND}\ b\ M_1 \ldots M_n) \wedge \Theta \longrightarrow \Delta \vdash (u \doteq [\phi^{-1}](\texttt{VAR}\ v\ M_1 \ldots M_n)) \wedge \Theta$$

$$\Delta \vdash (u[\phi] \doteq v[\sigma]) \wedge \Theta \longrightarrow \Delta \vdash (u \doteq [\phi^{-1}]v[\sigma]) \wedge \Theta$$

Occurs check

$$\begin{aligned} &\Delta \vdash (u \doteq \texttt{CON}\ c\ M_1 \ldots M_i\{u[\phi]\} \ldots M_n) \wedge \Theta \longrightarrow \\ &\qquad \Delta \vdash (u \doteq \texttt{CON}\ c\ M_1 \ldots M_i\{\_\} \ldots M_n) \wedge \Theta \qquad (1 \leq i \leq n) \\ &\Delta \vdash (u \doteq \texttt{VAR}\ v\ M_1 \ldots M_i\{u[\phi]\} \ldots M_n) \wedge \Theta \longrightarrow \\ &\qquad \Delta \vdash (u \doteq \texttt{VAR}\ v\ M_1 \ldots M_i\{\_\} \ldots M_n) \wedge \Theta \qquad (1 \leq i \leq n) \\ &\Delta \vdash (u \doteq \texttt{BND}\ b\ M_1 \ldots M_i\{u[\phi]\} \ldots M_n) \wedge \Theta \longrightarrow \\ &\qquad \Delta \vdash (u \doteq \texttt{BND}\ b\ M_1 \ldots M_i\{\_\} \ldots M_n) \wedge \Theta \qquad (1 \leq i \leq n) \\ &\Delta \vdash (u \doteq \texttt{CON}\ c\ M_1 \ldots M_{i\ srig}\{u[\sigma]\} \ldots M_n) \longrightarrow \bot \quad (1 \leq i \leq n) \end{aligned}$$

Figure 5.15: Canonical HybridLF unification transition rules

Intersection

$$\Delta \vdash (u \doteq u[\phi]) \wedge \Theta \longrightarrow \begin{cases} \Theta & \text{if } \phi \cap \text{id} = \phi \\ \Delta \vdash (u \doteq u[\phi \cap \text{id}] \wedge \Theta & \text{otherwise} \end{cases}$$

Pruning

$$\begin{aligned}(\Delta, u :: (\Gamma \vdash A)) \vdash \varepsilon_{rig}\{u[\rho]\} \wedge \Theta \longrightarrow {} & (\Delta, u :: (\Gamma \vdash A), v :: ((\Gamma \vdash A) \setminus x)) \vdash \\ & u \doteq v[\rho \setminus x] \wedge \varepsilon_{rig}\{u[\rho]\} \wedge \Theta\end{aligned}$$

Instantiation

$$\begin{aligned}
\Delta \vdash (u \doteq \texttt{CON}\ c\ M_1 \dots M_n) \wedge \Theta \longrightarrow {} & [\texttt{CON}\ c\ M_1 \dots M_n/u]\Delta \vdash \\
& (u \leftarrow \texttt{CON}\ c\ M_1 \dots M_n) \wedge [\texttt{CON}\ c\ M_1 \dots M_n/u]\Theta \\
& (u \notin \text{FV}\ \texttt{CON}\ c\ M_1 \dots M_n, n \geq 0) \\
\Delta \vdash (u \doteq \texttt{VAR}\ v\ M_1 \dots M_n) \wedge \Theta \longrightarrow {} & [\texttt{VAR}\ v\ M_1 \dots M_n/u]\Delta \vdash \\
& (u \leftarrow \texttt{VAR}\ v\ M_1 \dots M_n) \wedge [\texttt{VAR}\ v\ M_1 \dots M_n/u]\Theta \\
& (u \notin \text{FV}\ \texttt{VAR}\ v\ M_1 \dots M_n, n \geq 0) \\
\Delta \vdash (u \doteq v[\sigma]) \wedge \Theta \longrightarrow {} & ([v[\sigma]/u]\Delta \vdash (u \leftarrow v[\sigma])[v[\sigma]/u]\Theta
\end{aligned}$$

Figure 5.16: Canonical HybridLF unification transition rules (cont.)

```
datatype ('a, 'b) ukind = UTYPE
    | UKPI "('a, 'b) uctype" "('a, 'b) ukind"
and ('a, 'b) uctype = UPI "('a, 'b) uctype" "('a, 'b) uctype"
    | UATYPE "('a, 'b) uatype"
and ('a, 'b) uatype = UFCON 'b
    | UFAPP "('a, 'b) uatype" "('a, 'b) ucterm" (infixl "$$_UT" 50)
and ('a, 'b) ucterm = UABS "('a, 'b) uctype" "('a, 'b) ucterm"
    | UATERM "('a, 'b) uaterm"
and ('a, 'b) uaterm = UVAR nat
    | UBND nat
    | UCON 'a
    | UAPP "('a, 'b) uaterm" "('a, 'b) ucterm" (infixl "$$_UO" 50)
    | UMVAR nat "(nat × ('a, 'b) ucterm) list"
    | UPH
```

We define the following type synonyms for contexts, the type declaration part of signatures, the kind declaration part of signatures, binding environments, metvariable contexts and substitutions:

```
type_synonym ('a, 'b) uctx = "(nat × ('a, 'b) uctype) list"
type_synonym ('a, 'b) usig_t = "('a × ('a, 'b) uctype) list"
type_synonym ('a, 'b) usig_k = "('b × ('a, 'b) ukind) list"
type_synonym ('a, 'b) ubndenv = "(('a, 'b) uctype) list"
type_synonym ('a, 'b) umvarctx = "(nat × (('a, 'b) uctype list
    × ('a, 'b) uctype)) list"
type_synonym ('a, 'b) usubst = "(nat × ('a, 'b) ucterm) list"
```

We define functions ucterm_shift, uctype_shift, uaterm_shift, uatype_shift and subst_shift that perform shifting on canonical terms, canonical types, atomic terms, atomic types and substitutions. These are similar to the functions in definition 87 with the exception of subst_shift, which is defined like so:

Definition 191 (subst_shift).

$$\mathsf{subst\_shift}\ i\ k\ l = \ \mathrm{map}\ (\lambda(x,\ y).\ (x,\ \mathsf{ucterm\_shift}\ i\ k\ y))\ l$$

We also define functions ucterm_level, uctype_level, uaterm_level, uatype_level, ukind_level and subst_level which are very similar to the functions in definition 86, with the exception of subst_level, which is defined as follows:

Definition 192 (subst_level).

$$\begin{aligned}\mathsf{subst\_level}\ k\ [] &= \mathtt{True}\\ \mathsf{subst\_level}\ k\ ((x,\ y)\ \#\ xs) &= (\mathsf{ucterm\_level}\ k\ y \wedge \mathsf{subst\_level}\ k\ xs)\end{aligned}$$

We define valid types and kinds, typing and kinding, and substitution for bound variables through the functions uvalidkind, uvalidtype, uatom_kindof, ukind_subst_bv, ucterm_subst_bv, uctype_subst_bv, uatype_subst_bv, uaterm_can_subst_bv, uaterm_subst_bv, uctx_subst_bv, ucanon_typeof, uatom_typeof, and usub_subst_bv. These functions are similar to those in definitions 88, 89, 90, 93, 200, 198, 199, 201, 202, 203, 91 and 92. usub_subst_bv is defined like so:

Definition 193 (usub_subst_bv).

$\mathsf{usub\_subst\_bv}\ 0\ \mathit{ctx}\ \mathit{sig\_t}\ \mathit{sig\_k}\ \mathit{bnd}\ \mathit{umctx}\ m\ n\ n'\ c = \mathtt{None}$

$\mathsf{usub\_subst\_bv}\ (\mathtt{Suc}\ q)\ \mathit{ctx}\ \mathit{sig\_t}\ \mathit{sig\_k}\ \mathit{bnd}\ \mathit{umctx}\ m\ n\ n'\ [] = (\mathtt{if}\ \mathsf{ucterm\_level}$
$\quad 0\ m\ \mathtt{then}\ \mathtt{Some}\ []\ \mathtt{else}\ \mathtt{None})$

$\mathsf{usub\_subst\_bv}\ (\mathtt{Suc}\ q)\ \mathit{ctx}\ \mathit{sig\_t}\ \mathit{sig\_k}\ \mathit{bnd}\ \mathit{umctx}\ m\ n\ n'\ ((x,\ y)\ \#\ xs) = (\mathtt{case}$
$\quad \mathsf{usub\_subst\_bv}\ q\ \mathit{ctx}\ \mathit{sig\_t}\ \mathit{sig\_k}\ \mathit{bnd}\ \mathit{umctx}\ m\ n\ n'\ xs\ \mathtt{of}\ \mathtt{Some}\ xs' \Rightarrow (\mathtt{case}$
$\quad \mathsf{ucterm\_subst\_bv}\ q\ \mathit{ctx}\ \mathit{sig\_t}\ \mathit{sig\_k}\ \mathit{bnd}\ \mathit{umctx}\ m\ n\ n'\ y\ \mathtt{of}\ \mathtt{Some}\ y' \Rightarrow \mathtt{Some}$
$\quad ((x,\ y')\ \#\ xs')\ |\ \mathtt{None}\ \Rightarrow \mathtt{None})\ |\ \mathtt{None} \Rightarrow \mathtt{None})$

We again define a function usubst that applies a substitution to a canonical term; this is very similar to definition 171 but returns a `cterm option` value and takes an additional numerical argument to ensure termination (like all of the substitution functions defined for Canonical HybridLF).

We create mutually-defined functions occurs_rigid_ucterm, occurs_rigid_uaterm, occurs_rigid_uctype and occurs_rigid_uatype; these are similar to definitions 172 and 173 and perform the same task of determining if a term exists in a rigid position over the `ucterm`, `uaterm`, `uctype` and `uatype` datatypes respectively.

We define a function occurs_rigid that determines if a term is in a rigid position within an equation; this is almost identical to definition 174.

We define functions occurs_bnd_head_ucterm, occurs_bnd_head_uaterm, occurs_top_ucterm, occurs_top_uaterm, occurs_top_mvar_ucterm, occurs_top_mvar_uaterm, is_argument_to_bound_ucterm and is_argument_to_bound_uaterm. These are the 'helper' functions that we will use to determine if a term is in a strongly rigid position, and are very similar to definitions 175, 176 and 177.

It is then possible to define the functions occurs_strongly_rigid_ucterm, occurs_strongly_rigid_uaterm, occurs_strongly_rigid_uctype and occurs_strongly_rigid_uatype, which are the Canonical HybridLF equivalents of the HybridLF definitions 178 and 179. The function occurs_strongly_rigid is then defined similarly to definition 180.

We define functions occurs_strongly_rigid_mvar_ucterm, occurs_strongly_rigid_mvar_uaterm, occurs_strongly_rigid_mvar_uctype and occurs_strongly_rigid_mvar_uatype which determine if a metavariable occurs in a strongly rigid position within a canonical term, atomic term, canonical type or atomic type respectively. The function occurs_strongly_rigid_mvar then uses these functions to determine if a metavariable exists in a strongly rigid position within an equation.

We define functions is_pattern_subst' and is_pattern_subst which determine if a substitution is a strong pattern substitution; they are very similar to definitions 181 and 182. We further define functions is_strong_pattern_subst' and is_strong_pattern_subst that determine if a substitution is a strong pattern substitution, and are the Canonical HybridLF equivalents of definitions 183 and 184.

We further define functions create_ph_list, ureplace, invert', invert and intersection_id that are defined in the same way as the HybridLF functions in definitions 185, 186, 187 and 188. We also define functions find_next_mvar_ucterm, find_next_mvar_uaterm, find_next_mvar_uctype, find_next_mvar_uatype, find_next_mvar_ueqn, find_next_mvar_ustate and find_next_mvar_usubst that return a value one higher than the highest numbered metavariable in a canonical term, atomic term, canonical type, atomic type, equation, unification state or substitution respectively.

The next set of functions define substitution for metavariables. We create functions ukind_subst_mv, ucterm_subst_mv, uctype_subst_mv, uatype_subst_mv, uaterm_subst_mv, uctx_subst_mv and usub_subst_mv which substitute a canonical term for a metavariable in a kind, canonical term, canonical type, atomic type, atomic term, context or substitution respectively. We also define the function uaterm_can_subst_mv; this performs substitution of a canonical term for a metavariable in an atomic term, resulting in a canonical term. These functions are very similar to definitions 93, 200, 198, 199, 202, 203 and 201, except that instead of substituting for instances of `BND` they substitute for instances of `UMVAR`.

Using the previously defined functions for substitution of metavariables, we define ueqn_subst_mv, which substitutes for a metavariable in an equation, ueqns_subst_mv, which substitutes for a metavariable in a list of equations, and ustate_subst_mv which substitutes for a metavariable in a unification state.

The make_id, subst_contains, subst_remove, diff, remove_eqn and ustate_remove_eqn functions are all defined in the same way as for HybridLF.

The transition rules of the unification algorithm are given inductively in figures 5.17, 5.18, 5.19 and 5.20.

Similarly to unification in HybridLF, we create a utransitions relation that defines the transitive closure of the single-step utransition relation. The rules defining utransitions are in figure 5.21.

5.5 Chapter summary

In this chapter we discussed unification in LF, a necessary part of implementing the M_2 metalogic that we described in section 4.3. In section 5.2 we defined unification for HybridLF, basing our algorithm upon that of Reed but adding additional transition and reduction rules. In section 5.3 we examined the implementation of unification for HybridLF. In section 5.4 we discussed the definition of unification for Canonical HybridLF, and in subsection 5.4.2 its implementation.

The unification algorithms for HybridLF and Canonical HybridLF are very similar, except for the presence of β-reduction and η-expansion rules in HybridLF, and the constraint that many of the terms on the left of transition rules are in long head-normal form. The reduction and expansion rules and LHNF constraints are not necessary in Canonical HybridLF, as all terms are by definition in canonical form.

The Isabelle execution of the unification algorithms functions relatively well. The implementation of the transition rules as relations (using Isabelle's `inductive` command) rather than as functions is unavoidable, as the transition rules may be applied in any order, and we may reach states in which no further transition rules can be applied but the result is not `FAIL`. The main potential issue with the algorithms as they are implemented is that many of the transition rules, such as ufcon_same and ufapp, have the conclusion utransition ... (`EQNS` s) (`EQNS` s') and constrain the s and s' in the hypotheses. As a result, it is possible to apply these rules even when the s and s' given

$$\frac{e \in \text{set} s \qquad \mathsf{occurs\_rigid}\ (\mathtt{UATERM\ UPH})\ e}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ \mathtt{FAIL}}\ \text{RIGID\_PH}$$

$$\frac{\begin{array}{c}(\mathtt{TyEqn}\ (\mathtt{UATYPE}\ (\mathtt{UFCON}\ c),\ \mathtt{UATYPE}\ (\mathtt{UFCON}\ c))) \in \text{set}\ s \\ s' = (\mathsf{remove\_eqn}\ s\ (\mathtt{TyEqn}\ (\mathtt{UATYPE}\ (\mathtt{UFCON}\ c), \\ (\mathtt{UATYPE}\ (\mathtt{UFCON}\ c)))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ (\mathtt{EQNS}\ s')}\ \text{UFCON\_SAME}$$

$$\frac{\begin{array}{c}(\mathtt{TyEqn}\ (\mathtt{UATYPE}\ (\mathtt{UFCON}\ c),\ (\mathtt{UATYPE}\ (\mathtt{UFCON}\ c')))) \in \text{set}\ s \\ c \neq c'\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ \mathtt{FAIL}}\ \text{UCON\_DIFF}$$

$$\frac{\begin{array}{c}(\mathtt{TyEqn}\ (\mathtt{UATYPE}\ (a\ \$\$_{\text{UT}}\ b),\ (\mathtt{UATYPE}\ (a'\ \$\$_{\text{UT}}\ b')))) \in \text{set}\ s \\ s' = ([(\mathtt{TyEqn}\ ((\mathtt{UATYPE}\ a),\ (\mathtt{UATYPE}\ a'))),\ (\mathtt{TermEqn}\ (b,\ b'))]\ @ \\ (\mathsf{remove\_eqn}\ s\ (\mathtt{TyEqn}\ (\mathtt{UATYPE}\ (a\ \$\$_{\text{UT}}\ b),\ (\mathtt{UATYPE}\ (a'\ \$\$_{\text{UT}}\ b'))))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ (\mathtt{EQNS}\ s')}\ \text{UFAPP}$$

$$\frac{\begin{array}{c}(\mathtt{TyEqn}\ (\mathtt{UPI}\ a\ b,\ \mathtt{UPI}\ a'\ b')) \in \text{set}\ s \\ s' = ([(\mathtt{TyEqn}\ (a,\ a')),\ (\mathtt{TyEqn}\ (b,\ b'))]\ @ \\ (\mathsf{remove\_eqn}\ s\ (\mathtt{TyEqn}(\mathtt{UPI}\ a\ b, \mathtt{UPI}\ a'\ b'))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ (\mathtt{EQNS}\ s')}\ \text{UPI}$$

$$\frac{\begin{array}{c}\mathtt{TermEqn}\ (\mathtt{UABS}\ t\ e,\ \mathtt{UABS}\ t'\ e')) \in \text{set}\ s \\ s' = ((\mathtt{TermEqn}\ (e,\ e'))\ \#\ (\mathtt{TyEqn}\ (t,\ t'))\ \# \\ (\mathsf{remove\_eqn}\ s\ (\mathtt{TermEqn}\ (\mathtt{UABS}\ t\ e,\ \mathtt{UABS}\ t'\ e'))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ (\mathtt{EQNS}\ s')}\ \text{UABS}$$

$$\frac{\begin{array}{c}(\mathtt{TermEqn}\ (\mathtt{UATERM}\ (\mathtt{UCON}\ c),\ (\mathtt{UATERM}\ (\mathtt{UCON}\ c)))) \in \text{set}\ s \\ s' = (\mathsf{remove\_eqn}\ s\ ((\mathtt{TermEqn}\ (\mathtt{UATERM}\ (\mathtt{UCON}\ c), \\ (\mathtt{UATERM}\ (\mathtt{UCON}\ c))))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ (\mathtt{EQNS}\ s')}\ \text{UCON\_SAME}$$

$$\frac{\begin{array}{c}(\mathtt{TermEqn}\ (\mathtt{UATERM}\ (\mathtt{UCON}\ c),\ (\mathtt{UATERM}(\mathtt{UCON}\ c')))) \in \text{set} s \\ c \neq c'\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ \mathtt{FAIL}}\ \text{UCON\_DIFF}$$

Figure 5.17: Transition rules for unification in CANONICAL HYBRIDLF

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UBND}\ n),\ (\texttt{UATERM}\ (\texttt{UBND}\ n)))) \in \text{set}\ s \\ s' = (\mathsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UBND}\ n), \\ (\texttt{UATERM}\ (\texttt{UBND}\ n)))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UBND\_SAME}$$

$$\frac{\begin{array}{c}(termeqn\ (\texttt{UATERM}\ (\texttt{UBND}\ n),\ (\texttt{UATERM}\ (\texttt{UBND}\ n')))) \in \text{set}\ s \\ n \neq n'\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\ \texttt{FAIL}}\ \text{UBND\_DIFF}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (a\ \$\$_{\text{UO}}\ b),\ (\texttt{UATERM}\ (a'\ \$\$_{\text{UO}}\ b')))) \in \text{set}\ s \\ s' = ([(\texttt{TermEqn}\ ((\texttt{UATERM}\ a),\ (\texttt{UATERM}\ a'))), (\texttt{TermEqn}\ (b,\ b'))]\ @ \\ (\mathsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ (\texttt{UATERM}\ (a\ \$\$_{\text{UO}}\ b), \\ (\texttt{UATERM}\ (a'\ \$\$_{\text{UO}}\ b'))))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UAPP}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ ((\texttt{UATERM}\ (\texttt{UVAR}\ v),\ \texttt{UATERM}\ (\texttt{UVAR}\ v)))) \in \text{set}\ s \\ s' = (\mathsf{remove\_eqn}\ s\ (\texttt{TermEqn}\ ((\texttt{UATERM}\ (\texttt{UVAR}\ v), \\ (\texttt{UATERM}\ (\texttt{UVAR}\ v))))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{UVAR\_SAME}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UVAR}\ v),\ (\texttt{UATERM}\ (\texttt{UVAR}\ v')))) \in \text{set}\ s \\ v \neq v'\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\texttt{FAIL}}\ \text{UVAR\_DIFF}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ s),\ e)) \in \text{set}\ s'' \\ \mathsf{invert}\ s = \mathsf{Some}\ s' \\ \mathsf{usubst}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ s'\ e = \mathsf{Some}\ e' \\ s''' = ((\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ e'))\ \# \\ (\mathsf{remove\_eqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ s),\ e)))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s'')\ (eqns\ s''')}\ \text{INVERT}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []), \texttt{UATERM}((\texttt{UCON}\ c)\ \$\$_{\text{UO}}\ e))) \in \text{set}\ s \\ \mathsf{replace\_mvar\_with\_ph}\ (\texttt{UATERM}\ ((\texttt{UCON}\ c)\ \$\$_{\text{UO}}\ e))\ v = e' \\ s' = ((\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ e'))\ \#\ (\mathsf{remove\_eqn}\ s \\ (\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ \texttt{UATERM}\ ((\texttt{UCON}\ c)\ \$\$_{\text{UO}}\ e)))))\end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{OCCURS\_CON}$$

Figure 5.18: Transition rules for unification in CANONICAL HYBRIDLF (cont.) (1)

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ \texttt{UATERM}\ ((\texttt{UBND}\ b)\ \$\$_{\text{UO}}\ e))) \in \text{set}\ s\\ \textsf{replace\_mvar\_with\_ph}\ (\texttt{UATERM}((\texttt{UBND}\ b)\ \$\$_{\text{UO}}\ e))\ v = e'\\ s' = ((\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ e'))\ \#\ (\textsf{remove\_eqn}\ s\\ (\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ \texttt{UATERM}\ ((\texttt{UBND}\ b)\$\$_{\text{UO}}\ e)))))\end{array}}{\textsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{OCCURS\_BND}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ \texttt{UATERM}\ ((\texttt{UVAR}\ v)\ \$\$_{\text{UO}}\ e))) \in \text{set}\ s\\ \textsf{replace\_mvar\_with\_ph}\ (\texttt{UATERM}\ ((\texttt{UVAR}\ v)\$\$_{\text{UO}}\ e))\ v = e'\\ s' = ((\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ e'))\ \#\ (\textsf{remove\_eqn}\ s\\ (\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ \texttt{UATERM}\ ((\texttt{UVAR}\ v)\ \$\$_{\text{UO}}\ e)))))\end{array}}{\textsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\ (\texttt{EQNS}\ s')}\ \text{OCCURS\_VAR}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ (\texttt{UATERM}\\ ((\texttt{UCON}\ c)\ \$\$_{\text{UO}}\ e)))) \in \text{set}\ s\\ \textsf{occurs\_strongly\_rigid\_mvar\_ucterm}\ v\ e\end{array}}{\textsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s)\ \texttt{FAIL}}\ \text{OCCURS\_RIGID\_FAIL}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\\ \texttt{UATERM}\ (\texttt{UMVAR}\ v\ s))) \in \text{set}\ s'\\ \textsf{intersection\_id}\ s = s\\ s'' = (\textsf{remove\_eqn}\ s'\ (\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\\ \texttt{UATERM}\ (\texttt{UMVAR}\ v\ s))))\end{array}}{\textsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ s')\ (\texttt{EQNS}\ s'')}\ \text{INTERSECTION\_SAME}$$

$$\frac{\begin{array}{c}(\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ \texttt{UATERM}\ (\texttt{UMVAR}\ v\ s))) \in \text{set}\ t\\ \textsf{intersection\_id}\ s = s'\\ s' \neq s\\ t' = (((\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\ \texttt{UATERM}\ (\texttt{UMVAR}\ v\ s'))))\\ \#\ (\textsf{remove\_eqn}\ t\ (\texttt{TermEqn}\ (\texttt{UATERM}\ (\texttt{UMVAR}\ v\ []),\\ \texttt{UATERM}\ (\texttt{UMVAR}\ v\ s)))))\end{array}}{\textsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\texttt{EQNS}\ t)\ (\texttt{EQNS}\ t')}\ \text{INTERSECTION\_DIFF}$$

Figure 5.19: Transition rules for unification in CANONICAL HYBRIDLF (cont.) (2)

$$\frac{\begin{array}{c} eqn \in \mathrm{set}\ t \\ \mathsf{find\_next\_mvar\_ustate}\ (\mathtt{EQNS}\ t) = v \\ \mathsf{occurs\_rigid}\ (\mathtt{UATERM}\ (\mathtt{UMVAR}\ v'\ s))\ eqn \\ \mathsf{is\_pattern\_subst}\ s \\ \mathsf{diff}\ x\ s = \mathtt{Some}\ s' \\ (\mathsf{ueqns\_subst\_mv}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{UATERM}\ (\mathtt{UMVAR}\ v\ s'))\ v' \\ (\mathsf{remove\_eqn}\ t\ (\mathtt{TermEqn}\ (\mathtt{UATERM}\ (\mathtt{UMVAR}\ v\ []), \\ \mathtt{UATERM}\ (\mathtt{UMVAR}\ v\ s'))))) = \mathtt{Some}\ t'' \\ t' = (\mathtt{Solved}\ (v',\ \mathtt{UATERM}\ (\mathtt{UMVAR}\ v\ s')))\ \#\ t'' \end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ t)\ (\mathtt{EQNS}\ t')}\ \textsc{pruning}$$

$$\frac{\begin{array}{c} (\mathtt{TermEqn}\ (\mathtt{UATERM}\ (\mathtt{UMVAR}\ v\ []),\ e)) \in \mathrm{set}\ s \\ (\mathsf{ueqns\_subst\_mv}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ e\ v\ (\mathsf{remove\_eqn} \\ s\ (\mathtt{TermEqn}\ (\mathtt{UATERM}\ (\mathtt{UMVAR}\ v\ []),\ e)))) = \mathtt{Some}\ s'' \\ s' = (\mathtt{Solved}\ (v,\ e))\ \#\ s'' \end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ (\mathtt{EQNS}\ s')}\ \textsc{instantiate}$$

$$\frac{\begin{array}{c} (\mathtt{TermEqn}\ (e,\ \mathtt{UATERM}\ (\mathtt{UMVAR}\ v\ []))) \in \mathrm{set} s \\ (\mathsf{ueqns\_subst\_mv}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ e\ v\ (\mathsf{remove\_eqn} \\ s\ (\mathtt{TermEqn}\ (e,\ \mathtt{UATERM}\ (\mathtt{UMVAR}\ v\ []))))) = \mathtt{Some}\ s'' \\ s' = (\mathtt{Solved}\ (v,\ e))\ \#\ s'' \end{array}}{\mathsf{utransition}\ d\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ (\mathtt{EQNS}\ s)\ (\mathtt{EQNS}\ s')}\ \textsc{instantiate\_r}$$

Figure 5.20: Transition rules for unification in Canonical HybridLF (cont.) (3)

$$\frac{\mathsf{utransition}\ n\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ s\ s'}{\mathsf{utransitions}\ n\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ s\ s'}\ \textsc{utrans\_single}$$

$$\frac{\begin{array}{c} \mathsf{utransition}\ n\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ s\ s' \\ \mathsf{utransitions}\ n\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ s'\ s'' \end{array}}{\mathsf{utransitions}\ n\ ctx\ sig\_t\ sig\_k\ bnd\ umvct\ s\ s''}\ \textsc{utrans\_trans}$$

Figure 5.21: utransitions in Canonical HybridLF

in the hypotheses do not match the current sub-goal. Once the rule has been applied, the hypotheses concerning s and s' will evaluate to `False`, so the proof will not succeed, but the fact that it is possible to apply these rules in places where they should not be applied complicates the implementation of unification.

Chapter 6

Using HybridLF and Canonical HybridLF

6.1 Introduction

Now that we have created HybridLF and Canonical HybridLF, we want to actually use them to prove meta-theorems of deductive systems. The principal approach is to define a signature by creating types to represent the constants of the signature then using the fbind family of functions with HOAS functions to give the types corresponding to each constant. We then create a pair of contexts giving the input and output parameters of the meta-theorem, define the proof-terms that specify the proof of totality, then prove that these describe a complete M_2 derivation using the derivation rules.

6.2 Creating proofs

When creating a proof in HybridLF or Canonical HybridLF we first create a pair of types to represent the constants of the signature - one for object constants and one for type constants. These are given as type parameters to the datatypes implementing the expressions, types and kinds of LF (whether these be `expr`, `type` and `kind` in HybridLF, or `cterm`, `aterm`, `ctype`, `atype` and `kind` in Canonical HybridLF).

Example 194. For example, we might extend the LF signature Σ from examples 115 and 119 on pages 88 and 91 representing the natural numbers, with a judgement `even` that holds when a number is even and a judgement that represents the metatheorem that the successor to an odd number is even. The signature is shown in figure 6.1.

$\Sigma' =$

```
           nat : type
          zero : nat
          succ : nat → nat
           odd : nat → type
          even : nat → type
 odd_succ_even : Πa:nat. odd a → even (succ a) → type
       odd_one : odd (succ zero)
      odd_succ : Πa:nat. odd a → odd (succ (succ a))
     even_zero : even zero
     even_succ : Πa:nat. even a → even (succ (succ a))
odd_succ_even_one : odd_succ_even (succ zero) odd_one (even_succ zero
                      even_zero)
odd_succ_even_succ : Πa:nat. Πb:odd a. Πc:even (succ a).
                      odd_succ_even a b c →
                      odd_succ_even (succ (succ a)) (odd_succ a b)
                      (even_succ (succ a) c)
```

Figure 6.1: LF signature for natural numbers with odd/even judgements and metatheorem

The datatypes representing the constants of the signature would be like so:

```
datatype o_cons = zero | succ | odd_one | odd_succ | even_zero |
                        even_succ
datatype t_cons = nat | odd | even
```

The signature itself would be split into two parts, one for object constants and one for type constants, represented by a list of pairs with the first element being the constant symbol and the second the type or kind of the constant. The signature is shown in figure 6.2.

The first branch of the proof that `odd_succ_even` is total is shown in figure 6.3. We define constants `con_i` and `con_o` that contain the input and output variables of the goal formula respectively.

```
definition sig_obj :: "(o_cons × (o_cons, t_cons) type) list"
where "sig_obj ≡ [(zero, FCON nat),
(succ, fbind (FCON nat) (λx. FCON nat $$_F x)),
(odd_one, FCON odd $$_F (CON succ $$_O CON zero)),
(odd_succ, fbind2 (FCON nat) (λx. FCON odd $$_F x)
    (λx. λy. FCON odd $$_F (CON succ $$_O (CON succ $$_O x))))
(even_zero, FCON even $$_F CON zero),
(even_succ, fbind2 (FCON nat) (λx. FCON even $$_F x)
    (λx. λy. FCON even $$_F (CON succ $$_O (CON succ $$_O x)))),
(odd_succ_even_one, FCON odd_succ_even $$_F (CON succ $$_O CON zero) $$_F
    (CON odd_one) $$_F (CON even_succ $$_O CON zero $$_O CON even_zero)),
(odd_succ_even_succ, (fbind4 (FCON nat) (λx. FCON odd $$_F x)
    (λx. λy. FCON even $$_F (CON succ $$_O x))
    (λx. λy. λz. FCON odd_succ_even $$_F x $$_F y $$_F z)
    (λx. λy. λz. λa. FCON odd_succ_even $$_F (CON succ $$_O (CON succ
    $$_O x)) $$_F(CON odd_succ $$_O x$$_O y) $$_F
    (CON even_succ $$_O (CON succ $$_O x)$$_O z)))))]"

definition sig_ty :: "(t_cons × (o_cons, t_cons) kind) list"
where "sig_ty ≡ [(nat, TYPE),
(odd, (KPI (FCON nat) TYPE)),
(even, (KPI (FCON nat) TYPE)),
(odd_succ_even, (KPI (FCON nat) (KPI (FCON odd $$_F (BND 0))
    (KPI (FCON even $$_F (CON succ $$_O (BND 1))) TYPE)))]"
```

Figure 6.2: HybridLF signature for natural numbers with odd/even judgements and metatheorem

```
definition con_i :: "(o_cons, t_cons) con"
where "con_i ≡ [(1, FCON odd $$_F (VAR 0))]"

definition con_o :: "(o_cons, t_cons) con"
where "con_o ≡ [(2, FCON odd_succ_even $$_F (VAR 0) $$_F (VAR 1) $$_F (VAR 3))]"



lemma totality : "derivation ((0, FCON nat) # con_i @ [(3, FCON even
    $$_F (CON succ $$_O VAR 0))] @ con_o) sig_obj sig_ty [] ((0, FCON nat) #
    con_i @ [(3, FCON even $$_F (CON succ $$_O VAR 0))] @ con_o) []
    (Fix (Var 0) (con_i, con_o)
    (Case 1 (PattCase
        (Case 3 (PattCase
            (Case 6 (PattCase
                (Case 7 (PattCase
                    (Lam [(1, FCON odd $$_F (CON succ $$_O CON zero))]
                    (Subst [(2, CON odd_succ_even_one)]))
                EmptyCase))
            (PattCase
                (Case 7 (EmptyCase))
            EmptyCase)))
        EmptyCase))
    ...
```

Figure 6.3: First branch of proof of totality for odd_succ_even

As discussed in Wang and Nadathur [30], the proof starts with use of the `Fix` recursion construct to bring the goal formula into the assumptions. There then follows a series of case analysis steps, which are analogous to Twelf splitting on a variable. These case analysis steps are implemented by the `sig_uni` relation, which works through the signature, attempting to unify the variable that case analysis is being performed upon with the type of each constant in the signature. In the proof, this manifests itself as a series of applications of the `sig_non_uni` rule followed by the `sig_uni` rule as follows (for the initial splitting step see figure 6.3).

Example 195.

```
apply(rule cases) (* Case 1 *)
apply(auto)
apply(rule sig_non_uni) (* zero *)
apply(auto)
apply(rule sig_non_uni) (* succ *)
apply(auto)
apply(rule sig_uni) (* odd_one *)
apply(auto)
```

Once the progress through the signature has reached a constant whose type should unify with the type of the variable that case analysis is being performed upon, the process of unification starts:

Example 196.

```
apply(rule utrans_trans)
apply(rule UFAPP)
apply(auto)
apply(rule red_none)
apply(rule red_none)
apply(rule lhnf.intros)
apply(auto)
apply(rule ty_MVAR)
apply(auto)
apply(rule lhnf.intros)
apply(auto)
apply(rule ty_APP)
apply(rule ty_CON)
apply(auto)
apply(rule ty_CON)
apply(auto)
apply(rule utrans_trans)
apply(rule UFCON_same)
apply(auto)
apply(rule utrans_trans)
apply(rule instantiate)
apply(auto)
apply(rule utrans_single)
apply(rule instantiate)
apply(auto)
```

Note that since the proof is in HybridLF, as part of unification we must show that the terms can be put into long head-normal form using the reduction rules (in this case `red_none`, as the terms do not need reducing), the `lhnf.intros` rule and the various typing rules with the `ty` prefix.

For the proof in figure 6.3, once the case analysis steps have been completed the `Lam` and `Subst` constructs are used to provide a constant with type matching the output variable in the goal formula using the ∀R and ∃R rules.

The proof proceeds as follows:

Example 197.

```
apply(rule forall_r)
apply(auto)
apply(rule exists_r)
apply(rule s_c_fv_nonempty)
apply(auto)
apply(rule ty_CON)
apply(auto)
apply(rule s_c_fv_empty)
```

As an example of a Canonical HybridLF signature, in appendix C we give a signature defining a proof of type preservation for the simply-typed lambda calculus, based on an example from the Twelf documentation [31] (to allow comparison between the Twelf signature and the Canonical HybridLF signature).

6.3 Chapter summary

In this chapter we have discussed the methodology of using HybridLF and Canonical HybridLF to create proofs of meta-theorems about deductive systems, based around an example of part of a proof in HybridLF. The proofs are long and verbose, as operations such as unification and reduction to long head-normal form must be performed manually by applying rules step-by-step. This is a drawback compared to systems such as Twelf, in which totality checking and unification are automatic. This could be improved in HybridLF and Canonical HybridLF with more work on automation, such as customised tactics created at the ML level of Isabelle.

Chapter 7

Conclusions

In this thesis, we have described the theory and implementation of two systems: HybridLF and Canonical HybridLF, based on LF [22] and Canonical LF [38] respectively. There are positive and negative aspects to both.

For HybridLF, the chief advantages are the simplicity of substitution, which does not require a numerical argument to limit the number of recursive calls and ensure termination like the substitution of Canonical HybridLF, and the properties that we have proved about the relations of HybridLF such as that all terms with a type or types with a kind are proper terms (i.e. have level 0). The main disadvantage of HybridLF is the complication of definitional equality of terms, types and kinds.

In Canonical LF, on the other hand, all definitionally equal terms, types and kinds are also syntactically equal, so definitional equality is reduced to syntactic equality. In Canonical HybridLF, the price of this is the introduction of another pair of datatypes to represent atomic terms and types, and the use of `ATERM` and `ATYPE` annotations, which can introduce visual clutter. The definition of substitution in Canonical HybridLF is considerably more complicated than that of HybridLF; this is because of the hereditary nature of substitution.

Since all Isabelle functions are required to terminate, to define substitution for Canonical HybridLF in Isabelle as a function we needed to introduce an additional natural number argument that is decremented upon each recursive call to the substitution functions. Once the number reaches 0, the result of substitution is `None`. Using this approach, all functions and relations that make use of substitution are required to provide a numerical argument, so we have the choice of either introducing a numerical argument to these relations and functions as well, or hard-coding a number (that may not be sufficient to reach a result in all cases) into the function or relation itself. We have chosen the former option. In practice, using substitution in Canonical HybridLF may entail some experimentation with different values for this termination measure, as if substitution fails it may not be immediately apparent whether there is no result from substitution itself, or if the level of recursion exceeded the numerical argument provided for termination.

The alternative to defining substitution as a function would be to define it as a relation. We experimented with this approach, and found that while it was much easier to define hereditary substitution as a relation, and no numerical recursion depth argument was required, actually using the implementation of substitution was considerably more cumbersome in practice. With the definition of substitution as a function the result of substitution is automatically calculated by Isabelle, whereas if substitution is defined as a relation the user is required to apply rules in order to perform substitution step by step.

Another advantage that Canonical HybridLF has over HybridLF is in the definition of unification. Since all terms are in canonical form by design, we do not need to perform any normalisation steps between transitions of the unification algorithm. In Reed's work [37] on higher-order unification he shows termination of the algorithm by demonstrating that the resulting equations of applying a transition of the algorithm are smaller each time. Although we do not formally show that this is the case for unification in Canonical HybridLF the algorithms are similar enough that this is most likely true. In HybridLF, on the other hand, we require normalisation of terms between applications of the unification rules. This normalisation may increase the size of terms within the equations, and therefore we cannot show that the equations always decrease in size. As a result, this approach to proof of termination of the unification algorithm is not applicable to HybridLF.

The extensions to the Hybrid approach to higher-order abstract syntax that we made in HybridLF and Canonical HybridLF all function correctly. These include the use of a family of abstraction functions to allow functions with more than one argument to be converted to de Bruijn form; in the original version of Hybrid [15], only functions with one variable could be converted to de Bruijn form. The family of functions approach could be considered a brute-force solution to the problem, but since Isabelle does not allow a single function to have a variable number of arguments, we do not lose anything by defining one function for each number of arguments since we would have to define one anyway.

Another technique introduced in Canonical HybridLF is the use of Isabelle's `option` type to replace the `ERR` and `FERR` elements of the core `expr` and `type` datatypes. This slightly complicates the use of the signature generated by the abstraction functions, as we must remove the `option` element of the type, but this is relatively easy to accomplish. The unification algorithm does not allow the unification of `ERR` and `FERR` (as these constructors are used to indicate an error during the conversion from HOAS to de Bruijn terms), so using the `option` type and removing `ERR` and `FERR` brings the core datatypes into line with the unification representation. This approach also removes error reporting from the terms themselves; if the result of calling an abstraction function is `Some M` for some term M, we know that no errors have occurred, and that the term M is a valid term with no error indicators buried within it.

One of the biggest differences between the original version of Hybrid and the systems described in this thesis is the introduction of types, as the system created by Ambler et al [15] was untyped. The typing (and kinding) mechanisms of HybridLF and Canonical HybridLF work well, with the introduction of binding environments to provide a way to record information about the types of enclosing binders when considering the types of abstraction bodies. The two systems differ in their implementation of typing: in HybridLF typing is implemented as a relation, while in Canonical HybridLF it is implemented as a function. The former entails more effort and overhead for the user of the system, as typing must be performed by applying rules step-by-step, whereas the latter is calculated automatically by Isabelle itself. From a practical perspective the implementation of typing as a function is preferable, and were the systems to be re-implemented we would investigate the possibility of performing typing for HybridLF via this mechanism.

For practical use in proving properties of deductive systems, the systems described in this thesis are somewhat limited compared to a system such as Twelf. Creating an LF signature is relatively straightforward in both HybridLF and Canonical HybridLF. The introduction of higher-order abstract syntax allows the variables bound by `FPI` binders to be used in the types of binders that follow easily through use of Isabelle bound variables. However, the signature definitions of HybridLF and Canonical HybridLF are not as clear as Twelf definitions. This is partly because in Twelf variables can be bound implicitly using variable names that start with an upper-case letter, simplifying the definitions. The Twelf signatures are also visually clearer, as application is denoted by simple textual juxtaposition, rather than by using an infix operator such as $\$\$_{\mathrm{O}}$ or $\$\$_{\mathrm{F}}$.

There is also the question of automation. For a system to be practically useful in creating proofs, automation of proof steps is desirable (as long as the system itself can be proven correct). In Twelf, the splitting operations when searching for a proof of the totality of a type family representing a metatheorem are performed automatically: we need only provide a mode declaration to indicate which parameters are to be interpreted as inputs and which as outputs. In HybridLF and Canonical HybridLF, the case analysis steps are performed manually by application of various rules. The same is true of unification, as unification in Twelf is automatic while unification in HybridLF and Canonical HybridLF must be performed manually. The fact that case analysis and unification is an explicit part of HybridLF and Canonical HybridLF proofs is a disadvantage. We found it hard to produce a complete proof of the totality of even the simplest metatheorem due to the sheer number of individual steps required; as a system for the practical production of proofs this is clearly undesirable. We might construct a system that automatically performs splitting and unification, and produces a HybridLF or Canonical HybridLF proof as a result. The systems described in this thesis would then be used to 'certify' the automatically generated proof.

The M_2 logic employed in HybridLF and Canonical HybridLF is also less powerful than the logic implemented in Twelf, as M_2 does not allow reasoning in non-empty contexts. Twelf requires that the user give a *worlds* declaration that specifies *regular worlds* - the contexts within which a totality assertion is valid. The logic M_2^+, described by Schürmann [29], allows reasoning in such contexts, so we could replace the M_2 implementation in HybridLF and Canonical HybridLF with an implementation of M_2^+, but reasoning in this logic is more complicated than reasoning in M_2. However, as the systems currently exist, there are proofs that are expressible in Twelf that cannot be created in HybridLF or Canonical HybridLF.

While HybridLF and Canonical HybridLF implement the M_2 logic relatively faithfully, there are two omissions that must be taken into consideration when creating a proof, to ensure that the proof is in fact valid. The first is that the system does not check that a recursive call using the `Fix` construct terminates. In Twelf, this is performed using a sub-term ordering, ensuring that the recursively called term is a subterm of the original term. The second omission is in the `sig_derivation` relation, where we use `sig_uni` and `sig_non_uni` rules to work through the signature. `sig_uni` is used to indicate that the type of the variable for which case analysis is being performed unifies with the base type of the constant in the signature, while `sig_non_uni` indicates that it does not unify. While unification is performed when `sig_uni` is applied, the systems do not check that unification fails when `sig_non_uni` is used. We could create a 'proof' that simply consists of a case analysis step and the application of `sig_non_uni` to all of the constants in the signature (regardless of whether they actually unify or not, as the system does not check) followed by application of the `sig_empty` rule to finish the proof. Such a 'proof' would be invalid, but HybridLF and Canonical HybridLF would accept it as a valid proof.

At the outset of the creation of HybridLF and Canonical HybridLF we aimed to answer one main question: what happens when you combine the Hybrid approach to higher-order abstract syntax with a type theory such as LF? This thesis and the systems described within it provide an answer to that question. The resulting systems as they stand are perhaps not usable for the purpose of practical proof-creation, but with additional automation could provide an alternative to Twelf. HybridLF and Canonical HybridLF allow the creation of explicit proofs of totality that are not possible in Twelf. They are strictly less capable than Twelf in terms of proving theorems in contexts, but this could be remedied by implementing the M_2^+ meta-logic instead of M_2. The Hybrid approach to variable binding, as used and extended in the systems, works well and provides a user-friendly interface for specifying signatures.

Appendix A

HybridLF **typeof**, **kindof** and definitional equality relations

```
inductive typeof :: "(nat × ('a, 'b) type) list ⇒ ('a × ('a, 'b) type)
    list ⇒ ('b × ('a, 'b) kind) list ⇒ ('a, 'b) type list ⇒
    ('a, 'b) expr ⇒ ('a, 'b) type ⇒ bool"
and kindof :: "(nat × ('a, 'b) type) list ⇒ ('a × ('a, 'b) type) list ⇒
    ('b × ('a, 'b) kind) list ⇒ ('a, 'b) type list ⇒ ('a, 'b) type ⇒
    ('a, 'b) kind ⇒ bool"
and obj_def_equal :: "(nat × ('a, 'b) type) list ⇒ ('a × ('a, 'b) type)
    list ⇒ ('b × ('a, 'b) kind) list ⇒ ('a, 'b) type list ⇒
    ('a, 'b) expr ⇒ ('a, 'b) expr ⇒ ('a, 'b) type ⇒ bool"
and type_def_equal :: "(nat × ('a, 'b) type) list ⇒ ('a × ('a, 'b) type)
    list ⇒ ('b × ('a, 'b) kind) list ⇒ ('a, 'b) type list ⇒
    ('a, 'b) type ⇒ ('a, 'b) type ⇒ ('a, 'b) kind ⇒ bool"
and kind_def_equal :: "(nat × ('a, 'b) type) list ⇒ ('a × ('a, 'b) type)
    list ⇒ ('b × ('a, 'b) kind) list ⇒ ('a, 'b) type list ⇒
    ('a, 'b) kind ⇒ ('a, 'b) kind ⇒ bool"
and validkind :: "(nat × ('a, 'b) type) list ⇒ ('a × ('a, 'b) type) list
    ⇒ ('b × ('a, 'b) kind) list ⇒ ('a, 'b) type list ⇒ ('a, 'b) kind ⇒
    bool"
where
```

TY_BND : "[lookup bnd i = Some a; f_level 0 a] ⟹
    typeof ctx sig_t sig_k bnd (BND i) a"
| TY_VAR : "[varlookup ctx i = Some a; f_level 0 a] ⟹
    typeof ctx sig_t sig_k bnd (VAR i) a"
| TY_CON : "[oconlookup sig_t c = Some a; f_level 0 a] ⟹
    typeof ctx sig_t sig_k bnd (CON c) a"
| TY_ABS : "[kindof ctx sig_t sig_k bnd ty TYPE; f_level 0 ty;
    typeof ctx sig_t sig_k (ty # bnd) e t1] ⟹
    typeof ctx sig_t sig_k bnd (ABS ty e) (FPI ty t1)"
| TY_APP : "[typeof ctx sig_t sig_k bnd a (FPI t1 t2); o_level 0 b;
    typeof ctx sig_t sig_k bnd b t1; f_subst 0 t2 b = t3] ⟹
    typeof ctx sig_t sig_k bnd (APP a b) t3"
| TY_CONV : "[typeof ctx sig_t sig_k bnd m t1;
    type_def_equal ctx sig_t sig_k bnd t1 t2 k] ⟹
    typeof ctx sig_t sig_k bnd m t2"
| K_CON : "[fconlookup sig_k a = Some k; k_level 0 k] ⟹
    kindof ctx sig_t sig_k bnd (FCON a) k"
| K_PI : "[kindof ctx sig_t sig_k bnd t1 TYPE;
    kindof ctx sig_t sig_k (t1 # bnd) t2 TYPE] ⟹
    kindof ctx sig_t sig_k bnd (FPI t1 t2) TYPE"
| K_APP : "[kindof ctx sig_t sig_k bnd a (KPI t1 k);
    typeof ctx sig_t sig_k bnd m t1; o_level 0 m; k_subst 0 k m = b] ⟹
    kindof ctx sig_t sig_k bnd(FAPP a m) b"
| K_CONV : "[kindof ctx sig_t sig_k bnd a k1;
    kind_def_equal ctx sig_t sig_k bnd k1 k2; k_level 0 k2] ⟹
    kindof ctx sig_t sig_k bnd a k2"
| VK_TYPE : "validkind ctx sig_t sig_k bnd TYPE"
| VK_KPI : "[kindof ctx sig_t sig_k bnd ty TYPE; f_level 0 ty;
    validkind ctx sig_t sig_k (ty # bnd) k] ⟹
    validkind ctx sig_t sig_k bnd (KPI ty k)"

| OBJ_EQ_BETA : "[kindof ctx sig_t sig_k bnd a TYPE;
typeof ctx sig_t sig_k (a # bnd) m b; typeof ctx sig_t sig_k bnd n a;
o_level 0 (APP (ABS a m) n); o_subst 0 m n = m$'$; f_subst 0 b n = b$'$] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd (APP(ABS a m) n) m$'$ b$'$"

| OBJ_EQ_EXT : "[kindof ctx sig_t sig_k bnd a TYPE; f_level 0 a;
obj_def_equal ctx sig_t sig_k (a # bnd) (APP m x) (APP n x) b] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd m n (FPI a b)"

| OBJ_EQ_ETA : "[typeof ctx sig_t sig_k bnd m (FPI a b); o_level 0 m] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd (ABS a (APP m (BND 0))) m (FPI a b)"

| OBJ_EQ_REFL : "[typeof ctx sig_t sig_k bnd m a; o_level 0 m] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd m m a"

| OBJ_EQ_SYM : "obj_def_equal ctx sig_t sig_k bnd n m a $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd m n a"

| OBJ_EQ_TRANS : "[obj_def_equal ctx sig_t sig_k bnd m m$'$ a;
obj_def_equal ctx sig_t sig_k bnd m$'$ n a] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd m n a"

| OBJ_EQ_CNG_CON : "[oconlookup sig_t c = Some a; f_level 0 a] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd (CON c) (CON c) a"

| OBJ_EQ_CNG_VAR : "[varlookup ctx x = Some a; f_level 0 a] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd (VAR x) (VAR x) a"

| OBJ_EQ_CNG_APP : "[obj_def_equal ctx sig_t sig_k bnd m n (FPI a b);
obj_def_equal ctx sig_t sig_k bnd m$'$ n$'$ a; f_subst 0 b m$'$ = b$'$] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd (APP m m$'$) (APP n n$'$) b$'$"

| OBJ_EQ_CNG_LAM : "[type_def_equal ctx sig_t sig_k bnd a a$'$ TYPE;
type_def_equal ctx sig_t sig_k bnd a a$''$ TYPE;
obj_def_equal ctx sig_t sig_k (a # bnd) m n b] $\Longrightarrow$
obj_def_equal ctx sig_t sig_k bnd (ABS a$'$ m)(ABS a$''$ n) (FPI a b)"

| TY_EQ_REFL : "[kindof ctx sig_t sig_k bnd a k; f_level 0 a] $\Longrightarrow$
type_def_equal ctx sig_t sig_k bnd a a k"

| TY_EQ_SYM : "type_def_equal ctx sig_t sig_k bnd a b k $\Longrightarrow$
type_def_equal ctx sig_t sig_k bnd b a k"

| TY_EQ_TRANS : "[type_def_equal ctx sig_t sig_k bnd a a′ k;

type_def_equal ctx sig_t sig_k bnd a′ b k] $\Longrightarrow$

type_def_equal ctx sig_t sig_k bnd a b k"

| TY_EQ_CNG_CON : "[fconlookup sig_k x = Some k; k_level 0 k] $\Longrightarrow$

type_def_equal ctx sig_t sig_k bnd (FCON x) (FCON x) k"

| TY_EQ_CNG_APP : "[type_def_equal ctx sig_t sig_k bnd a b (KPI c2 c1);

obj_def_equal ctx sig_t sig_k bnd m n c2; k_subst 0 c1 m = c1′] $\Longrightarrow$

type_def_equal ctx sig_t sig_k bnd (FAPP a m) (FAPP b n) c1′"

| TY_EQ_CNG_PI : "[type_def_equal ctx sig_t sig_k bnd a a′ TYPE;

type_def_equal ctx sig_t sig_k (a # bnd) b b′ TYPE] $\Longrightarrow$

type_def_equal ctx sig_t sig_k bnd (FPI a b) (FPI a′ b′) TYPE"

| KIND_EQ_REFL : "validkind ctx sig_t sig_k bnd a $\Longrightarrow$

kind_def_equal ctx sig_t sig_k bnd a a"

| KIND_EQ_SYM : "kind_def_equal ctx sig_t sig_k bnd a b $\Longrightarrow$

kind_def_equal ctx sig_t sig_k bnd b a"

| KIND_EQ_TRANS : "[kind_def_equal ctx sig_t sig_k bnd a b;

kind_def_equal ctx sig_t sig_k bnd b c] $\Longrightarrow$

kind_def_equal ctx sig_t sig_k bnd a c"

| KIND_EQ_CNG_PI : "[type_def_equal ctx sig_t sig_k bnd a a′ TYPE;

kind_def_equal ctx sig_t sig_k (a # bnd) k k′] $\Longrightarrow$

kind_def_equal ctx sig_t sig_k bnd (KPI a k) (KPI a′ k′)"

Appendix B

Canonical HybridLF substitution functions

Substitution on kinds is performed by kind_subst_bv, defined in definition 93.

The ctype_subst_bv function performs substitution of a canonical term for a bound variable on a given canonical type:

Definition 198 (ctype_subst_bv)**.**

$$\begin{array}{l}
\mathsf{ctype_subst_bv}\ 0\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ n'\ t = \mathtt{None}\\
\mathsf{ctype_subst_bv}\ (q+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ n'\ (\mathtt{ATYPE}\ p) =\\
\qquad (\mathtt{case}\ (\mathsf{atype_subst_bv}\ q\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ n'\ p)\ \mathtt{of}\ \mathtt{Some}\ p' \Rightarrow\\
\qquad \mathtt{Some}\ (\mathtt{ATYPE}\ p')\mid \mathtt{None} \Rightarrow \mathtt{None})\\
\mathsf{ctype_subst_bv}\ (q+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ n'\ (\mathtt{PI}\ a2\ a) =\\
\qquad (\mathtt{case}\ \mathsf{ctype_subst_bv}\ q\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ n'\ a2\ \mathtt{of}\ \mathtt{Some}\ a2' \Rightarrow\\
\qquad (\mathtt{case}\ \mathsf{ctype_subst_bv}\ q\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ (n'+1)\ a\ \mathtt{of}\ \mathtt{Some}\ a' \Rightarrow\\
\qquad \mathtt{Some}\ (\mathtt{PI}\ a2'\ a')\mid \mathtt{None} \Rightarrow \mathtt{None})\mid \mathtt{None} \Rightarrow \mathtt{None})
\end{array}$$

The atype_subst_bv function performs substitution of a canonical term for a bound variable on a given atomic type:

Definition 199 (atype_subst_bv).

$\mathsf{atype\_subst\_bv}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ r = \texttt{None}$

$\mathsf{atype\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{FCON}\ a) =$

$\quad (\texttt{if}\ \mathsf{cterm\_level}\ 1\ m\ \texttt{then}\ \texttt{Some}\ (\texttt{FCON}\ a)\ \texttt{else}\ \texttt{None})$

$\mathsf{atype\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{FAPP}\ p\ m') =$

$\quad (\texttt{case}\ \mathsf{atype\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ p\ \texttt{of}\ \texttt{Some}\ p' \ \Rightarrow$

$\quad (\texttt{case}\ \mathsf{cterm\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ m'\ \texttt{of}\ \texttt{Some}\ m'' \ \Rightarrow$

$\quad \texttt{Some}\ (\texttt{FAPP}\ p'\ m'')\ |\ \texttt{None}\ \Rightarrow \texttt{None})\ |\ \texttt{None} \Rightarrow \texttt{None})$

The cterm_subst_bv function performs substitution of a canonical term for a bound variable on a canonical term:

Definition 200 (cterm_subst_bv).

$\mathsf{cterm\_subst\_bv}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ m'' \ = \texttt{None}$

$\mathsf{cterm\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{ATERM}\ r) =$

$\quad (\texttt{case}\ (\mathsf{aterm\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ r)\ \texttt{of}\ \texttt{Some}\ r' \Rightarrow$

$\quad \texttt{Some}\ (\texttt{ATERM}\ r')\ |\ \texttt{None} \Rightarrow (\texttt{case}\ (\mathsf{aterm\_can\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k$

$\quad bnd\ m\ n\ n'\ r)\ \texttt{of}\ \texttt{Some}\ r'' \Rightarrow \texttt{Some}\ r''\ |\ \texttt{None} \Rightarrow \texttt{None}))$

$\mathsf{cterm\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{ABS}\ a\ m') =$

$\quad (\texttt{case}\ \mathsf{ctype\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ a\ \texttt{of}\ \texttt{Some}\ a' \Rightarrow$

$\quad (\texttt{case}\ \mathsf{cterm\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ (a\ \#\ bnd)\ m\ (n+1)\ (n'+1)\ m'$

$\quad \texttt{of}\ \texttt{Some}\ m'' \Rightarrow (\texttt{if}\ \mathsf{cterm\_level}\ 1\ m\ \texttt{then}\ \texttt{Some}\ (\texttt{ABS}\ a'\ m'')\ \texttt{else}\ \texttt{None})$

$\quad |\ \texttt{None} \Rightarrow \texttt{None})\ |\ \texttt{None} \Rightarrow \texttt{None})$

The aterm_can_subst_bv function performs substitution of a canonical term for a bound variable in a given atomic term, with a canonical term as result:

Definition 201 (aterm_can_subst_bv).

$\mathsf{aterm\_can\_subst\_bv}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ r\ = \texttt{None}$

$\mathsf{aterm\_can\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{BND}\ n'') =$

$\quad (\texttt{if}\ n = n''\ \texttt{then Some}\ (\mathsf{cterm\_shift}\ n'\ 0\ m)\ \texttt{else None})$

$\mathsf{aterm\_can\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{APP}\ r\ m2) =$

$\quad (\texttt{case}\ \mathsf{aterm\_can\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ r\ \texttt{of}$

$\quad \texttt{Some}\ (\texttt{ABS}\ t\ m1') \Rightarrow (\texttt{case}\ \mathsf{cterm\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ m2$

$\quad \texttt{of Some}\ m2' \Rightarrow \mathsf{cterm\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m2'\ 0\ 0\ m1'$

$\quad \mid \texttt{None} \Rightarrow \texttt{None}) \mid \texttt{None} \Rightarrow \texttt{None})$

$\mathsf{aterm\_can\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{VAR}\ v) =$

$\quad (\texttt{if}\ \mathsf{cterm\_level}\ 1\ m\ \texttt{then Some}\ (\texttt{ATERM}\ (\texttt{VAR}\ v))\ \texttt{else None})$

$\mathsf{aterm\_can\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{CON}\ c) =$

$\quad (\texttt{if}\ \mathsf{cterm\_level}\ 1\ m\ \texttt{then Some}\ (\texttt{ATERM}\ (\texttt{CON}\ c))\ \texttt{else None})$

The aterm_subst_bv function performs substitution of a canonical term for a bound variable in a given atomic term, with an atomic term as result:

Definition 202 (aterm_subst_bv).

$\mathsf{aterm\_subst\_bv}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ r\ = \texttt{None}$

$\mathsf{aterm\_subst\_bv}\ (q+1) ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{BND}\ b) =$

$\quad (\texttt{if}\ (n \neq b \wedge \mathsf{cterm\_level}\ 1\ m)\ \texttt{then Some}\ (\texttt{BND}\ b)\ \texttt{else None})$

$\mathsf{aterm\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{VAR}\ v) =$

$\quad (\texttt{if}\ \mathsf{cterm\_level}\ 1\ m\ \texttt{then Some}\ (\texttt{VAR}\ v)\ \texttt{else None})$

$\mathsf{aterm\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{CON}\ c) =$

$\quad (\texttt{if}\ \mathsf{cterm\_level}\ 1\ m\ \texttt{then Some}\ (\texttt{CON}\ c)\ \texttt{elseNone})$

$\mathsf{aterm\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ (\texttt{APP}\ r\ m2) =$

$\quad (\texttt{case}\ \mathsf{aterm\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ r\ \texttt{of Some}\ r' \Rightarrow$

$\quad (\texttt{case}\ \mathsf{cterm\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ m2\ \texttt{of Some}\ m2' \Rightarrow$

$\quad \texttt{Some}\ (\texttt{APP}\ r'\ m2') \mid \texttt{None} \Rightarrow \texttt{None}) \mid \texttt{None} \Rightarrow \texttt{None})$

The ctx_subst_bv function performs substitution of a canonical term for a bound variable on a given context:

Definition 203 (ctx_subst_bv).

$\mathsf{ctx\_subst\_bv}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ c\ = \mathtt{None}$

$\mathsf{ctx\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ [] =$

$\quad (\mathtt{if}\ \mathsf{cterm\_level}\ 0\ m\ \mathtt{then}\ \mathtt{Some}\ []\ \mathtt{else}\ \mathtt{None})$

$\mathsf{ctx\_subst\_bv}\ (q+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ ((x,y)\ \#\ xs) =$

$\quad (\mathtt{case}\ \mathsf{ctx\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ xs\ \mathtt{of}\ \mathtt{Some}\ xs' \Rightarrow$

$\quad (\mathtt{case}\ \mathsf{ctype\_subst\_bv}\ q\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ n'\ y\ \mathtt{of}\ \mathtt{Some}\ y' \Rightarrow$

$\quad \mathtt{Some}\ ((x,y')\ \#\ xs')\ |\ \mathtt{None} \Rightarrow \mathtt{None})\ |\ \mathtt{None} \Rightarrow \mathtt{None})$

Substituting for free variables is performed by another set of mutually-defined functions. The first parameter is again a natural number, used to ensure termination. Like the 'subst_bv' functions, the second parameter is a context, the third is the part of the signature containing object constants, the fourth is the part of the signature containing type constants and the fifth is a binding environment. The sixth parameter is the canonical term that is being substituted for the free variable, and the seventh is a natural number corresponding to the number of the variable to substitute for. The eighth parameter is the kind, canonical type, atomic type, canonical term, atomic term or context that substitution is taking place in.

The function kind_subst_fv substitutes a canonical term for a free variable in a given kind:

Definition 204 (kind_subst_fv).

$\mathsf{kind\_subst\_fv}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ t\ = \mathtt{None}$

$\mathsf{kind\_subst\_fv}\ (n'+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ \mathtt{TYPE} = \mathtt{Some}\ \mathtt{TYPE}$

$\mathsf{kind\_subst\_fv}\ (n'+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n$

$\quad (\mathtt{KPI}\ a\ k) = (\mathtt{case}\ \mathsf{ctype\_subst\_fv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ a$

$\quad \mathtt{of}\ \mathtt{Some}\ a' \Rightarrow (\mathtt{case}\ \mathsf{kind\_subst\_fv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ k$

$\quad \mathtt{of}\ \mathtt{Some}\ k' \Rightarrow \mathtt{Some}\ (\mathtt{KPI}\ a'\ k')\ |\ \mathtt{None} \Rightarrow \mathtt{None})\ |\ \mathtt{None} \Rightarrow \mathtt{None})$

The function ctype_subst_fv substitutes a canonical term for a free variable in a given canonical type:

Definition 205 (ctype_subst_fv).

$$\begin{aligned}
&\mathsf{ctype_subst_fv}\ 0\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ c\ = \texttt{None}\\
&\mathsf{ctype_subst_fv}(n'+1)\ ctx\ sig_t\ sig_k\ bndm\ n\ (\texttt{ATYPE}\ p) =\\
&\qquad (\texttt{case}\ \mathsf{atype_subst_fv}\ n'\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ p\ \texttt{of Some}\ p'\\
&\qquad \Rightarrow \texttt{Some}\ (\texttt{ATYPE}\ p')\ |\ \texttt{None} \Rightarrow \texttt{None})\\
&\mathsf{ctype_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\\
&\qquad (\texttt{PI}\ a2\ a) = (\texttt{case}\ \mathsf{ctype_subst_fv}\ n'\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ a2\\
&\qquad \texttt{of Some}\ a2' \Rightarrow (\texttt{case}\ \mathsf{ctype_subst_fv}\ n'\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ a\\
&\qquad \texttt{of Some}\ a' \Rightarrow \texttt{Some}\ (\texttt{PI}\ a2'\ a')\ |\ \texttt{None} \Rightarrow \texttt{None})\ |\ \texttt{None} \Rightarrow \texttt{None})
\end{aligned}$$

atype_subst_fv substitutes a canonical term for a free variable in an atomic type:

Definition 206 (atype_subst_fv).

$$\begin{aligned}
&\mathsf{atype_subst_fv}\ 0\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ at = \texttt{None}\\
&\mathsf{atype_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ (\texttt{FCON}\ a) =\\
&\qquad (\texttt{if}\ \mathsf{cterm_level}\ 0\ m\ \texttt{then Some}\ (\texttt{FCON}\ a)\ \texttt{else None})\\
&\mathsf{atype_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\\
&\qquad (\texttt{FAPP}\ p\ m') = (\texttt{case}\ \mathsf{atype_subst_fv}\ n'\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ p\\
&\qquad \texttt{of Some}\ p' \Rightarrow (\texttt{case}\ \mathsf{cterm_subst_fv}\ n'\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ m'\\
&\qquad \texttt{of Some}\ m'' \Rightarrow \texttt{Some}\ (\texttt{FAPP}\ p'\ m'')\ |\ \texttt{None} \Rightarrow \texttt{None})\ |\ \texttt{None} \Rightarrow \texttt{None})
\end{aligned}$$

cterm_subst_fv substitutes a canonical term for a free variable in a given canonical term:

Definition 207 (cterm_subst_fv).

$\mathsf{cterm\_subst\_fv}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ t = \texttt{None}$”

$\mathsf{cterm\_subst\_fv}\ (n'+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ (\texttt{ATERM}\ r) =$

$(\texttt{case}\ \mathsf{aterm\_subst\_fv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd$

$m\ n\ r\ \texttt{of Some}\ r' \Rightarrow \texttt{Some}\ (\texttt{ATERM}\ r')\ |\ \texttt{None} \Rightarrow (\texttt{case}$

$\mathsf{aterm\_can\_subst\_fv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ r\ \texttt{of Some}\ r'' \Rightarrow$

$\texttt{Some}\ r''\ |\ \texttt{None} \Rightarrow \texttt{None}))$

$\mathsf{cterm\_subst\_fv}\ (n'+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n$

$(\texttt{ABS}\ a\ m') = (\texttt{case}\ \mathsf{ctype\_subst\_fv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ a$

$\texttt{of Some}\ a' \Rightarrow (\texttt{case}\ \mathsf{cterm\_level}\ 0\ m\ \texttt{of True} \Rightarrow$

$(\texttt{case}\ (\mathsf{cterm\_subst\_fv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ m')\ \texttt{of Some}\ m''$

$\Rightarrow \texttt{Some}\ (\texttt{ABS}\ a'\ m'')\ |\ \texttt{None} \Rightarrow \texttt{None})\ |\ False \Rightarrow \texttt{None})$

$|\ \texttt{None} \Rightarrow \texttt{None})$

The function aterm_can_subst_fv substitutes a canonical term for a free variable in an atomic term, resulting in a canonical term:

Definition 208 (aterm_can_subst_fv).

$\mathsf{aterm\_can\_subst\_fv}\ 0\ ctx\ sig\_t\ sig\_k\ bnd\ m\ v\ atm = \texttt{None}$

$\mathsf{aterm\_can\_subst\_fv}\ (n'+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ v\ (\texttt{VAR}\ v') =$

$(\texttt{if}\ v = v'\ \texttt{then Some}\ m\ \texttt{else Some}\ (\texttt{ATERM}\ (\texttt{VAR}\ v')))$

$\mathsf{aterm\_can\_subst\_fv}\ (n'+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ v\ (\texttt{BND}\ b) = \texttt{Some}\ (\texttt{ATERM}\ (\texttt{BND}\ b))$

$\mathsf{aterm\_can\_subst\_fv}\ (n'+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ v\ (\texttt{CON}\ c) = \texttt{Some}\ (\texttt{ATERM}\ (\texttt{CON}\ c))$

$\mathsf{aterm\_can\_subst\_fv}\ (n'+1)\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ (\texttt{APP}\ r\ m2) =$

$(\texttt{case}\ \mathsf{aterm\_can\_subst\_fv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ r\ \texttt{of Some}\ (\texttt{ABS}\ t\ m1')$

$\Rightarrow (\texttt{case}\ \mathsf{cterm\_subst\_fv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd\ m\ n\ m2\ \texttt{of Some}\ m2'$

$\Rightarrow \mathsf{cterm\_subst\_bv}\ n'\ ctx\ sig\_t\ sig\_k\ bnd\ m2'\ 0\ 0\ m1'$

$|\ \texttt{None} \Rightarrow \texttt{None})\ |\ \texttt{None} \Rightarrow \texttt{None})$

aterm_subst_fv substitutes a canonical term for a free variable in an atomic term, resulting in an atomic term:

Definition 209 (aterm_subst_fv).

$$\begin{aligned}
&\mathsf{aterm_subst_fv}\ 0\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ a\ = \mathtt{None}\\
&\mathsf{aterm_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ (\mathtt{BND}\ b) =\\
&\qquad (\mathtt{if}\ (\mathsf{cterm_level}\ 0\ m)\ \mathtt{then}\ \mathtt{Some}\ (\mathtt{BND}\ b)\ \mathtt{else}\ \mathtt{None})\\
&\mathsf{aterm_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ (\mathtt{VAR}\ v) =\\
&\qquad (\mathtt{if}\ (v \neq n)\ \mathtt{then}\ (\mathtt{if}\ \mathsf{cterm_level}\ 0\ m\ \mathtt{then}\ \mathtt{Some}\ (\mathtt{VAR}\ v)\\
&\qquad \mathtt{else}\ \mathtt{None})\ \mathtt{else}\ \mathtt{None})\\
&\mathsf{aterm_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ (\mathtt{CON}\ c) =\\
&\qquad (\mathtt{if}\ (\mathsf{cterm_level}\ 0\ m)\ \mathtt{then}\ \mathtt{Some}\ (\mathtt{CON}\ c)\ \mathtt{else}\ \mathtt{None})\\
&\mathsf{aterm_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ (\mathtt{APP}\ r\ m2) =\\
&\qquad (\mathtt{case}\ \mathsf{aterm_subst_fv}\ n'\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ r\\
&\qquad \mathtt{of}\ \mathtt{Some}\ r' \Rightarrow (\mathtt{case}\ \mathsf{cterm_subst_fv}\ n'\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ m2\\
&\qquad \mathtt{of}\ \mathtt{Some}\ m2' \Rightarrow \mathtt{Some}\ (\mathtt{APP}\ r'\ m2')\mid \mathtt{None} \Rightarrow \mathtt{None})\mid \mathtt{None} \Rightarrow \mathtt{None})
\end{aligned}$$

The function ctx_subst_fv substitutes a canonical term for a free variable in a given context:

Definition 210 (ctx_subst_fv).

$$\begin{aligned}
&\mathsf{ctx_subst_fv}\ 0\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ ct = \mathtt{None}\\
&\mathsf{ctx_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ [] =\\
&\qquad (\mathtt{if}\ \mathsf{cterm_level}\ 0\ m\ \mathtt{then}\ \mathtt{Some}\ []\ \mathtt{else}\ \mathtt{None})\\
&\mathsf{ctx_subst_fv}\ (n'+1)\ ctx\ sig_t\ sig_k\ bnd\ m\ n\ ((x,y)\ \#\ xs) =\\
&\qquad (\mathtt{case}\ (\mathsf{ctx_subst_fv}\ n'\ ctx\ sig_t\ sig_k\\
&\qquad bnd\ m\ n\ xs)\ \mathtt{of}\ \mathtt{Some}\ xs' \Rightarrow (\mathtt{case}\ (\mathsf{ctype_subst_fv}\ n'\ ctx\ sig_t\\
&\qquad sig_k\ bnd\ m\ n\ y)\ \mathtt{of}\ \mathtt{Some}\ y' \Rightarrow \mathtt{Some}\ ((x,y')\ \#\ xs')\\
&\qquad \mid \mathtt{None} \Rightarrow \mathtt{None})\mid \mathtt{None} \Rightarrow \mathtt{None})
\end{aligned}$$

Appendix C

Simply-typed lambda calculus example

```
definition sig_type_option :: "(o_cons × (o_cons, t_cons) ctype
    option) list" where
"sig_type_option = [(empty, Some (ATYPE (FCON tm))),

(unit, Some (ATYPE (FCON tp))),

(arrow, Some (PI (ATYPE (FCON tp)) (PI (ATYPE (FCON tp)) (ATYPE (FCON tp))))),

(app, Some (PI (ATYPE (FCON tm)) (PI (ATYPE (FCON tm)) (ATYPE (FCON tm))))),

(lam, Some (PI (ATYPE (FCON tp)) (PI (PI (ATYPE (FCON tm)) (ATYPE (FCON tm)))
    (ATYPE (FCON tm))))),

(of_lam, (ctype_bind6 (ATYPE (FCON tp))
(λT. (ATYPE (FCON tp)))
(λT. λT2. PI (ATYPE (FCON tm)) (ATYPE (FCON tm)))
(λT. λT2. λE. (ATYPE (FCON tm)))
(λT. λT2. λE. λx. (ATYPE ((FCON of) $$_T (ATERM x) $$_T (ATERM T2))))
(λT. λT2. λE. λx. λa. ATYPE ((FCON of) $$_T ATERM (E $$_O (ATERM x)) $$_T (ATERM T)))
(λT. λT2. λE. λx. λa. λb. ATYPE ((FCON of) $$_T ATERM ((CON lam) $$_O (ATERM T2)
    $$_O (ATERM E)) $$_T (ATERM ((CON arrow) $$_O (ATERM T2) $$_O (ATERM T))))))),
```

```
(of_app, ctype_bind6 (ATYPE (FCON tm))
(λE1. (ATYPE (FCON tm)))
(λE1. λE2.(ATYPE (FCON tp)))
(λE1. λE2. λT. (ATYPE (FCON tp)))
(λE1. λE2. λT. λT2. ATYPE ((FCON of) $$_T (ATERM E1) $$_T (ATERM (CON arrow
    $$_O (ATERM T2) $$_O (ATERM T)))))
(λE1. λE2. λT. λT2. λa. ATYPE ((FCON of) $$_T (ATERM E2) $$_T (ATERM T2)))
(λE1. λE2. λT. λT2. λa. λb.  ATYPE ((FCON of) $$_T (ATERM ((CON app $$_O (ATERM E1)
    $$_O (ATERM E2)))) $$_T (ATERM T2)))),

(val_lam, ctype_bind2 (PI (ATYPE (FCON tm)) (ATYPE (FCON tm)))
(λE. (ATYPE (FCON tp)))
(λE. λT. (ATYPE ((FCON val) $$_T (ATERM ((CON lam) $$_O (ATERM T) $$_O (ATERM E))))))),

(step_app_1, ctype_bind4 ((ATYPE (FCON tm)))
(λE1. (ATYPE (FCON tm)))
(λE1. λE1′. (ATYPE (FCON tm)))
(λE1. λE1′. λE2. (ATYPE ((FCON step) $$_T (ATERM E1) $$_T (ATERM E1′))))
(λE1. λE1′. λE2. λa. (ATYPE ((FCON step) $$_T (ATERM (CON app $$_O (ATERM E1)
    $$_O (ATERM E2))) $$_T (ATERM (CON app $$_O (ATERM E1′) $$_O (ATERM E2))))))),

(step_app_2, ctype_bind5 ((ATYPE (FCON tm)))
(λE1. (ATYPE (FCON tm)))
(λE1. λE2.  (ATYPE (FCON tm)))
(λE1. λE2. λE2′. (ATYPE ((FCON val) $$_T (ATERM E1))))
(λE1. λE2. λE2′. λa. (ATYPE ((FCON step) $$_T (ATERM E2) $$_T (ATERM E2′))))
(λE1. λE2. λE2′. λa. λb. ATYPE ((FCON step) $$_T (ATERM ((CON app) $$_O (ATERM E1)
    $$_O (ATERM E2))) $$_T (ATERM ((CON app) $$_O (ATERM E1) $$_O (ATERM E2′)))))),

(step_app_beta, ctype_bind4 (PI (ATYPE (FCON tm)) (ATYPE (FCON tm)))
(λE. (ATYPE (FCON tm)))
(λE. λE2. (ATYPE (FCON tp)))
```

```
(λE. λE2. λT. ATYPE ((FCON val) $$_T (ATERM E2)))
(λE. λE2. λT. λa. ATYPE ((FCON step) $$_T (ATERM ((CON app) $$_O (ATERM ((CON lam)
    $$_O (ATERM T) $$_O (ATERM E))) $$_O (ATERM E2))) $$_T (ATERM (E $$O (ATERM E2)))))),

(pres_app_1, ctype_bind10 ((ATYPE (FCON tm)))
(λE1. (ATYPE (FCON tm)))
(λE1. λE1′. (ATYPE (FCON tp)))
(λE1. λE1′. λT2. (ATYPE (FCON tp)))
(λE1. λE1′. λT2. λT. ATYPE (FCON step $$_T (ATERM E1) $$_T (ATERM E1′)))
(λE1. λE1′. λT2. λT. λS1. ATYPE (FCON of $$_T (ATERM E1) $$_T (ATERM (CON arrow
    $$_O (ATERM T2) $$_O (ATERM T)))))
(λE1. λE1′. λT2. λT. λS1. λO1. ATYPE (FCON of $$_T (ATERM E1′) $$_T (ATERM (CON arrow
    $$_O (ATERM T2) $$_O (ATERM T)))))
(λE1. λE1′. λT2. λT. λS1. λO1. λO2. (ATYPE (FCON tm)))
(λE1. λE1′. λT2. λT. λS1. λO1. λO2. λE2.  ATYPE (FCON of $$_T (ATERM E2)
    $$_T (ATERM T2)))
(λE1. λE1′. λT2. λT. λS1. λO1. λO2. λE2. λO3. (ATYPE (FCON pres $$_T (ATERM S1)
    $$_T (ATERM O1) $$_T (ATERM O2))))
(λE1. λE1′. λT2. λT. λS1. λO1. λO2. λE2. λO3. λP1. (ATYPE (FCON pres $$_T (ATERM
    (CON step_app_1 $$_O (ATERM E1) $$_O (ATERM E1′) $$_O (ATERM E2) $$_O (ATERM S1)))
    $$_T (ATERM (CON of_app $$_O (ATERM E1) $$_O (ATERM E2) $$_O (ATERM T)
    $$_O (ATERM T2) $$_O (ATERM O1) $$_O (ATERM O3))) $$_T (ATERM (CON of_app
    $$_O (ATERM E1′) $$_O (ATERM E2) $$_O (ATERM T) $$_O (ATERM T2) $$_O (ATERM O2)
    $$_O (ATERM O3))))))),

(pres_app_2, ctype_bind11 ((ATYPE (FCON tm)))
(λE2. (ATYPE (FCON tm)))
(λE2. λE2′. (ATYPE (FCON tp)))
(λE2. λE2′. λT2. (ATYPE (FCON step $$_T (ATERM E2) $$_T (ATERM E2′))))
(λE2. λE2′. λT2. λS1.  (ATYPE (FCON of $$_T (ATERM E2) $$_T (ATERM T2))))
(λE2. λE2′. λT2. λS1. λO1. ATYPE (FCON of $$_T (ATERM E2′) $$_T (ATERM T2)))
(λE2. λE2′. λT2. λS1. λO1. λO2. ATYPE (FCON tm))
(λE2. λE2′. λT2. λS1. λO1. λO2. λE1. ATYPE (FCON tp))
```

$(\lambda$E2. λE2$'$. λT2. λS1. λO1. λO2. λE1. λT. ATYPE (FCON val $\$\$_{\mathrm{T}}$ (ATERM E1)))

$(\lambda$E2. λE2$'$. λT2. λS1. λO1. λO2. λE1. λT. λV1. ATYPE (FCON of $\$\$_{\mathrm{T}}$ (ATERM E1) $\$\$_{\mathrm{T}}$ (ATERM (CON arrow $\$\$_{\mathrm{O}}$ (ATERM T2) $\$\$_{\mathrm{O}}$ (ATERM T)))))

$(\lambda$E2. λE2$'$. λT2. λS1. λO1. λO2. λE1. λT. λV1. λO3. ATYPE (FCON pres $\$\$_{\mathrm{T}}$ (ATERM S1) $\$\$_{\mathrm{T}}$ (ATERM O1) $\$\$_{\mathrm{T}}$ (ATERM O2)))

$(\lambda$E2. λE2$'$. λT2. λS1. λO1. λO2. λE1. λT. λV1. λO3. λP1. ATYPE (FCON pres $\$\$_{\mathrm{T}}$ (ATERM (CON step_app_2 $\$\$_{\mathrm{O}}$ (ATERM S1) $\$\$_{\mathrm{O}}$ (ATERM V1))) $\$\$_{\mathrm{T}}$ (ATERM (CON of_app $\$\$_{\mathrm{O}}$ (ATERM O1) $\$\$_{\mathrm{O}}$ (ATERM O3))) $\$\$_{\mathrm{T}}$ (ATERM (CON of_app $\$\$_{\mathrm{O}}$ (ATERM O2) $\$\$_{\mathrm{O}}$ (ATERM O3)))))),

(pres_app_beta, ctype_bind7 (ATYPE (FCON tp))

$(\lambda$T2. PI (ATYPE (FCON tm)) (ATYPE (FCON tm)))

$(\lambda$T2. λE. ATYPE (FCON tm))

$(\lambda$T2. λE. λE2. ATYPE (FCON tp))

$(\lambda$T2. λE. λE2. λT. ATYPE (FCON val $\$\$_{\mathrm{T}}$ (ATERM E2)))

$(\lambda$T2. λE. λE2. λT. λV. ATYPE (FCON of $\$\$_{\mathrm{T}}$ (ATERM E) $\$\$_{\mathrm{T}}$ (ATERM T2)))

$(\lambda$T2. λE. λE2. λT. λV. λO1. PI (ATYPE (FCON tm)) (PI (ATYPE (FCON of $\$\$_{\mathrm{T}}$ (ATERM (BND 0)) $\$\$_{\mathrm{T}}$ (ATERM T2))) (ATYPE (FCON of $\$\$_{\mathrm{T}}$ (ATERM (E $\$\$_{\mathrm{O}}$ (ATERM (BND 1)))) $\$\$_{\mathrm{T}}$ (ATERM T)))))

$(\lambda$T2. λE. λE2. λT. λV. λO1. λO2. (ATYPE (FCON pres $\$\$_{\mathrm{T}}$ (ATERM (CON step_app_beta $\$\$_{\mathrm{O}}$ (ATERM V))) $\$\$_{\mathrm{T}}$ (ATERM (CON of_app $\$\$_{\mathrm{O}}$ (ATERM O1) $\$\$_{\mathrm{O}}$ (ATERM (CON of_lam $\$\$_{\mathrm{O}}$ (ABS (ATYPE (FCON tm)) (ABS (ATYPE (FCON of $\$\$_{\mathrm{T}}$ (ATERM (BND 0)) $\$\$_{\mathrm{T}}$ (ATERM T2))) (ATERM (O2 $\$\$_{\mathrm{O}}$ (ATERM (BND 1)) $\$\$_{\mathrm{O}}$ (ATERM (BND 0))))))))))))))]”

Bibliography

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich and Steve Zdancewic. *Mechanized Metatheory for the Masses: The PoplMark Challenge*, Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, Volume 3603, Pages 50-65, 2005.

[2] Nicholaas G. de Bruijn. *A Survey of the project Automath.* In J. P. Seldin and J. R. Hindley, editors, To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism, 1980.

[3] Lawrence C. Paulson. *Isabelle: the next 700 theorem provers.* In P. Odifreddi, editor, Logic and Computer Science, 1990.

[4] Arthur Charguéraud. *The Locally Nameless Representation.* In Journal of Automated Reasoning, Volume 49, Issue 3, pp 363-408, 2012.

[5] Randy Pollack, Masahiko Sato and Wilmer Ricciotti. *A Canonical Locally Named Representation of Binding.* In journal of Automated Reasoning, Volume 49, Issue 2, pages 185-207, 2012.

[6] Conor McBride and James McKinna. *Functional Pearl: I Am Not A Number - I Am A Free Variable.* Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, pages 1-9, 2004.

[7] Conor McBride. *Epigram: Practical Programming with Dependent Types.* In Advanced Functional Programming, Lecture Notes in Computer Science Volume 3622, pages 130-170, 2005.

[8] Andrew M. Pitts. *Nominal Logic, a First Order Theory of Names and Binding.* In Information and Computation, volume 186, issue 2, pages 165-193, 2003.

[9] Murdoch J. Gabbay and Andrew M. Pitts. *A New Approach to Abstract Syntax with Variable Binding.* Formal Aspects of Computing, volume 13, issue 3-5, pages 341-363, 2002.

[10] Christian Urban. *Nominal Reasoning Techniques in Isabelle/HOL.* In Journal of Automatic Reasoning, Vol. 40(4), 327-356, 2008.

[11] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantic.* Studies in Logic and the Foundations of Mathematics, North-Holland, 1981.

[12] The Coq Development Team. *The Coq proof assistant: reference manual.* URL: https://coq.inria.fr/distrib/current/refman/, 2014. [Accessed 17th March 2015].

[13] Amy Felty and Dale Miller. *Specifying theorem provers in a higher-order logic programming language.* University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-12.

[14] A. Momigliano, S. J. Ambler, and R. L. Crole. *A Comparison of Formalizations of the Meta-Theory of a Language with Variable Bindings in Isabelle.* In Richard J. Boulton and Paul B. Jackson, editors, Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, pages 267-282, 2001. Report EDI-INF-RR-0046.

[15] S. J. Ambler, R. L. Crole, and A. Momigliano. *Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction.* In Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics, Hampton, VA, USA, volume 2410 of Lecture Notes in Computer Science, pages 13-30. Springer-Verlag, 2002.

[16] Joëlle Despeyroux, Amy Felty, and André Hirschowitz. *Higher-order abstract syntax in Coq.* In Proceedings of the 2nd International Conference on Typed Lambda-Calculi and Applications, TLCA'95, Edinburgh, Volume 902 of Lecture Notes in Computer Science, Springer-Verlag, 1995.

[17] Adam Chlipala. *Parametric higher-order abstract syntax for mechanized semantics.* In ICFP '08 Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, pages 143-156, 2008.

[18] Nicolaas G. de Bruijn. *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem.* Indag. Math volume 34, pages 381-392, 1972.

[19] Alan Martin. *Reasoning Using Higher-Order Abstract Syntax in a Higher-Order Logic Proof Environment: Improvements to Hybrid and a Case Study*. PhD Thesis, University of Ottawa, 2010.

[20] Roy L. Crole. *The Representational Adequacy of Hybrid*. Mathematical Structures in Computer Science, 21(3), pages 585-646, 2011.

[21] Frank Pfenning, *Computation and Deduction*. Draft notes, 2001.

[22] Robert Harper, Furio Honsell and Gordon Plotkin. *A Framework for Defining Logics*. In Journal of the ACM, Volume 40 Issue 1, Pages 143-184, 1993.

[23] Robin Milner. *A theory of type polymorphism in programming*. In Journal of Computer and System Sciences, Volume 17, Pages 348-375, 1978.

[24] Robert Harper and Daniel R. Licata. *Mechanizing metatheory in a logical framework*. In Journal of Functional Programming, Volume 17 Issue 4-5, Pages 613-673, 2007.

[25] Frank Pfenning and Carsten Schuërmann. *System description: Twelf - a meta-logical framework for deductive systems*. In H. Ganzinger (ed.) CADE 1999. LNCS(LNAI), Volume 1632, Pages 202-206, 1999.

[26] Ekkehard Rohwedder and Frank Pfenning. *Mode and Termination Checking for Higher-Order Logic Programs*, In H. R. Nielson (ed.), ESOP 96, LNCS, Volume 1058, Pages 296-310,1996.

[27] Carsten Schürmann and Frank Pfenning. *A Coverage Checking Algorithm for LF*, In D. Basin and B. Wolff (ed.), Theorem Proving in Higher Order Logics, LNCS, Volume 2758, Pages 120-135, 2003.

[28] Carsten Schürmann and Frank Pfenning. *Automated Theorem Proving in a Simple Meta-Logic for LF*, In CADE-15 Proceedings of the 15th International Conference on Automated Deduction, Pages 286-300, 1998.

[29] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*, PhD Thesis, Carnegie-Mellon University, 2000.

[30] Yuting Wang and Gopalan Nadathur. *Towards Extracting Explicit Proofs From Totality Checking In Twelf*, In LFMTP '13 Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages, Pages 55-66, 2013.

[31] *Proving Metatheorems With Twelf.* URL: http://www.twelf.org/wiki/Proving_metatheorems_with_Twelf, 2007 [Accessed 16th March 2015].

[32] Warren D. Goldfarb. *The Undecidability of the Second-Order Unification Problem*, Theoretical Computer Science, Volume 13, Issue 2, Pages 225-230, 1981.

[33] Conal M. Elliott. *Higher-order Unification with Dependent Function Types*, Rewriting Techniques and Applications, Lecture Notes in Computer Science Volume 355, Pages 121-136, 1989.

[34] G. P. Huet. *A Unification Algorithm for Typed λ-calculus*, Theoretical Computer Science, Volume 1, Issue 1, Pages 27-57, 1975.

[35] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD Thesis, Chalmers University of Technology, 2007.

[36] Dale Miller. *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification*, Journal of Logic and Computation, Volume 1, Pages 253-281, 1991.

[37] Jason Reed. *Higher-Order Constraint Simplification in Dependent Type Theory*, Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, Pages 49-56, 2009.

[38] Kevin Watkins, Iliano Cervesato, Frank Pfenning and David Walker. *A Concurrent Logical Framework I: Judgments and Properties.* Carnegie-Mellon University Department of Computer Science Technical Report CMU-CS-02-101, 2003.

[39] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: an introduction.* Cambridge University Press, 2008.