Thesis Proposal: Refinement Types for LF

William Lovas (wlovas@cs.cmu.edu)

October 7, 2008

Abstract

The logical framework LF and its implementation as the Twelf metalogic provide both a practical system and a proven methodology for representing deductive systems and their metatheory in a machine-checkable way. An extension of LF with refinement types provides a convenient means for representing certain kinds of judgemental inclusions in an intrinsic manner. I propose to carry out such an extension in full, adapting as much of the Twelf metatheory engine as possible to the new system, and I intend to argue that the extension is both useful and practical.

1 Introduction

Thesis: Refinement types are a useful and practical extension to the LF logical framework.

The logical framework LF [HHP93] and its metalogic Twelf [PS99] can be used to encode and reason about a wide variety of logics, languages, and other deductive systems in formal, machine-checkable way. The key representation strategy used to encode deductive systems in LF is the *judgements-as-types* principle. Under judgements-as-types, the deductive system's judgements are represented as LF type families, and derivations of evident judgements are represented as LF terms of particular types.

Recent studies have shown that ML-like languages can profitably be extended with a notion of subtyping called *refinement types*. A refinement type discipline uses an extra layer of term classification above the usual type system to more accurately capture certain properties of terms. Refinements, also known as *sorts*, are usually only checked after normal typechecking succeeds, resulting in a system that admits no more terms as well-typed but one that classifies more precisely the terms that *do* typecheck.

In this work, I intend to show that such a refinement type system can also profitably be added to LF. Under LF's judgements-as-types representation methodology, refinement types represent more precise judgements than those represented by ordinary types, with subtyping between refinements acting as a kind of judgemental inclusion. Things previously tricky to encode in LF, like

subsyntactic relations between terms of an object language, become trivial to encode with the availability of refinement types. Refinement types are a truly *useful* addition to the LF representation language.

Of course, expressive power always comes at a cost, so I also intend to show that the cost of refinement types is in fact quite small. Since a logical framework like LF is usually one of the few "trusted components" in formal, machine-checkable proofs, it is critical that its own metatheory rest on solid foundations, and I will show that the addition of refinement types to LF does not overly complicate its metatheory. In fact, I hope to show that refinement types are an excellent match for LF: modern canonical forms-only presentations of LF make use of bidirectional typechecking similar to that which has been used in recent refinement type systems, and since LF is pure, no issues relating to effects arise with the addition of refinements. Modern techniques make refinement types a natural extension to LF, rather than a technical tour-de-force: refinement types are a *practical* addition to LF.

To demonstrate that adding refinement types to LF is both useful and practical, I intend to exhibit an extension of LF with refinement types called LFR, to work out important details of its meta(meta)theory, and to exhibit several example programs that highlight the utility of refinement types for mechanized metatheory.

To that end, much work has already been done. The first half of this proposal reviews the completed work, which has focused primarily on specifying the type theory underlying LFR and proving important metatheoretic properties. I first describe the LFR system by way of a series of illustrative examples (Section 2), and then I present a high-level overview of certain metatheoretic properties crucial to any logical framework, namely decidability of typechecking and the identity and substitution principles (Section 3). After that, I observe that LFR as specified only appeals to subsorting at base sorts, but I go on to explain how, surprisingly, higher-sort subsorting is an implicit feature of the system by virtue of its canonical forms presentation (Section 4).

The rest of the proposal describes what more should be done to convincingly demonstrate the utility and practicality of refinements for LF. Core components include coming to a fuller understanding of LFR's representation methodology and specifying a unification algorithm suitable for type reconstruction and coverage checking (described in Section 5). There are also several additional directions for further study that could be worthwhile time permitting (described in Section 6), including an operational interpretation, a prototype implementation, and various other useful components for metalogical reasoning. The proposal concludes with a discussion of related work (Section 7) and a rough timeline for completion (Section 8).

2 System and Examples

The design of the LFR type theory is heavily inspired by the canonical forms-style presentations of logical frameworks popularized by Watkins in the spec-

ification of the Concurrent Logical Framework CLF [WCPW02, WCPW04]; Harper and Licata recently presented core LF in this style [HL07]. In the canonical forms-style, only β-normal, η-long ("canonical") forms are admitted as well-formed, and therefore ordinary substitution must be replaced by "hereditary substitution", a partial function which hereditarily computes a canonical form.

LFR is best understood through several examples. These examples not only serve to motivate the design of the type theory, but also they begin to suggest why refinement types are a useful addition to LF. In what follows, R refers to atomic terms and N to normal terms. As in previous work on refinement types, the refinement system does not change the underlying term structure, so our atomic and normal terms are exactly the terms from canonical presentations of LF.

$$\begin{array}{rl} R ::= c \mid x \mid R\ N & \text{atomic terms} \\ N, M ::= R \mid \lambda x.\, N & \text{normal terms} \end{array}$$

In this style of presentation, typing is defined bidirectionally by two judgements: $R \Rightarrow A$, which says atomic term R *synthesizes* type A, and $N \Leftarrow A$, which says normal term N *checks* against type A. Since λ-abstractions are always checked against a given type, they need not be decorated with their domain types.

Types are similarly stratified into atomic and normal types.

$$\begin{array}{rl} P ::= a \mid P\ N & \text{atomic type families} \\ A, B ::= P \mid \Pi x{:}A.\, B & \text{normal type families} \end{array}$$

The operation of hereditary substitution, written $[N/x]_A$, is a partial function which computes the canonical form of the standard capture-avoiding substitution of N for x. It is indexed by the putative type of x, A, to ensure termination, but neither the variable x nor the substituted term N are required to bear any relation to this type index for the operation to be defined. We show in Section 3 that when N and x *do* have type A, hereditary substitution is a total function on well-typed terms.

Our layer of refinements uses metavariables Q for atomic sorts and S for normal sorts. These mirror the definition of types above, except for the addition of intersection and "top" sorts.

$$\begin{array}{rl} Q ::= s \mid Q\ N & \text{atomic sort families} \\ S, T ::= Q \mid \Pi x{::}S \sqsubset A.\, T \mid \top \mid S_1 \wedge S_2 & \text{normal sort families} \end{array}$$

Sorts are related to types by a refinement relation, $S \sqsubset A$ ("S refines A"), discussed below. We only sort-check well-typed terms, and a term of type A can be assigned a sort S only when $S \sqsubset A$. These constraints are collectively referred to as the "refinement restriction". We occasionally omit the "$\sqsubset A$" from function sorts when it is clear from context.

2.1 Example: Natural Numbers

For the first running example we will use the natural numbers in unary notation. In LF, they would be specified as follows

nat : type.
$zero$: nat.
$succ$: $nat \to nat$.

Suppose we would like to distinguish the odd and the even numbers as refinements of the type of all numbers.

$even \sqsubset nat$.
$odd \sqsubset nat$.

The form of the declaration is $s \sqsubset a$ where a is a type family already declared and s is a new sort family. Sorts headed by s are declared in this way to refine types headed by a. The relation $S \sqsubset A$ is extended through the whole sort hierarchy in a compositional way.

Next we declare the sorts of the constructors. For zero, this is easy:

$zero :: even$.

The general form of this declaration is $c :: S$, where c is a constant already declared in the form $c : A$, and where $S \sqsubset A$. The declaration for the successor is slightly more difficult, because it maps even numbers to odd numbers and vice versa. In order to capture both properties simultaneously we need to use an *intersection sort*, written as $S_1 \wedge S_2$.[1]

$succ :: even \to odd \ \wedge \ odd \to even$.

In order for an intersection to be well-formed, both components must refine the same type. The nullary intersection $\top$ can refine any type, and represents the maximal refinement of that type.[2]

$$\frac{s \sqsubset a \in \Sigma}{s\ N_1 \ldots N_k \sqsubset a\ N_1 \ldots N_k} \qquad \frac{S \sqsubset A \qquad T \sqsubset B}{\Pi x{::}S.\, T \sqsubset \Pi x{:}A.\, B} \qquad \frac{S_1 \sqsubset A \qquad S_2 \sqsubset A}{S_1 \wedge S_2 \sqsubset A} \qquad \frac{}{\top \sqsubset A}$$

To show that the declaration for $succ$ is well-formed, we establish that $even \to odd \wedge odd \to even \ \sqsubset \ nat \to nat$.

The *refinement relation* $S \sqsubset A$ should not be confused with the usual *subtyping relation*. Although each is a kind of subset relation[3], they are quite different:

[1] Intersection has lower precedence than arrow.

[2] As usual in LF, we use $A \to B$ as shorthand for the dependent type $\Pi x{:}A.\, B$ when x does not occur in B.

[3] It may help to recall the interpretation of $S \sqsubset A$: for a term to be judged to have sort S, it must already have been judged to have type A for some A such that $S \sqsubset A$. Thus, the refinement relation represents an inclusion "by fiat": every term with sort S is also a term of sort A, by invariant. By contrast, subsorting $S_1 \leq S_2$ is a more standard sort of inclusion: every term with sort S_1 is also a term of sort S_2, by subsumption (see Section 4).

subtyping relates two types, is contravariant in the domains of function types, and is transitive, while refinement relates a sort to a type, so it does not make sense to consider its variance or whether it is transitive. We will discuss our notion of subtyping, subsorting, below and in Section 4.

Now suppose that we also wish to distinguish the strictly positive natural numbers. We can do this by introducing a sort *pos* refining *nat* and declaring that the successor function yields a *pos* when applied to anything, using the maximal sort.

$$pos \sqsubset nat.$$
$$succ :: \cdots \wedge \top \rightarrow pos.$$

Since we only sort-check well-typed programs and *succ* is declared to have type $nat \rightarrow nat$, the sort $\top$ here acts as a sort-level reflection of the entire *nat* type.

We can specify that all odds are positive by declaring *odd* to be a subsort of *pos*.

$$odd \leq pos.$$

Although any ground instance of *odd* is evidently *pos*, we need the subsorting declaration to establish that variables of sort *odd* are also *pos*.

Putting it all together, we have the following:

$$even \sqsubset nat. \quad odd \sqsubset nat. \quad pos \sqsubset nat.$$
$$odd \leq pos.$$
$$zero :: even.$$
$$succ :: even \rightarrow odd \ \wedge \ odd \rightarrow even \ \wedge \ \top \rightarrow pos.$$

Now we should be able to verify that, for example, *succ* (*succ zero*) $\Leftarrow$ *even*. To explain how, we analogize with pure canonical LF. Recall that atomic types have the form $a\ N_1 \ldots N_k$ for a type family a and are denoted by P. Arbitrary types A are either atomic (P) or (dependent) function types ($\Pi x{:}A.\, B$). Canonical terms are then characterized by the rules shown in the left column above.

There are two typing judgements, $N \Leftarrow A$ which means that N checks against A (both given) and $R \Rightarrow A$ which means that R synthesizes type A (R given as input, A produced as output). Both take place in a context Γ assigning types to variables. To force terms to be η-long, the rule for checking an atomic term R only checks it at an atomic type P. It does so by synthesizing a type P' and comparing it to the given type P. In canonical LF, all types are already canonical, so this comparison is just α-equality.

On the right-hand side we have shown the corresponding rules for sorts. First, note that the format of the context Γ is slightly different, because it declares sorts for variables, not just types. The rules for functions and applications are straightforward analogues to the rules in ordinary LF. The rule **switch** for checking atomic terms R at atomic sorts Q replaces the equality check with a subsorting check and is the only place where we appeal to subsorting (defined below). For applications, we use the type A that refines the type S as the index parameter of the hereditary substitution.

Canonical LF	LF with Refinements
$\dfrac{\Gamma, x{:}A \vdash N \Leftarrow B}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{:}A.\, B}$	$\dfrac{\Gamma, x{::}S \sqsubset A \vdash N \Leftarrow T}{\Gamma \vdash \lambda x.\, N \Leftarrow \Pi x{::}S \sqsubset A.\, T}$ **(Π-I)**
$\dfrac{\Gamma \vdash R \Rightarrow P' \qquad P' = P}{\Gamma \vdash R \Leftarrow P}$	$\dfrac{\Gamma \vdash R \Rightarrow Q' \qquad Q' \leq Q}{\Gamma \vdash R \Leftarrow Q}$ **(switch)**
$\dfrac{x{:}A \in \Gamma}{\Gamma \vdash x \Rightarrow A}$ $\qquad$ $\dfrac{c{:}A \in \Sigma}{\Gamma \vdash c \Rightarrow A}$	$\dfrac{x{::}S \sqsubset A \in \Gamma}{\Gamma \vdash x \Rightarrow S}$ **(var)** $\qquad$ $\dfrac{c :: S \in \Sigma}{\Gamma \vdash c \Rightarrow S}$ **(const)**
$\dfrac{\Gamma \vdash R \Rightarrow \Pi x{:}A.\, B \qquad \Gamma \vdash N \Leftarrow A}{\Gamma \vdash R\, N \Rightarrow [N/x]_A\, B}$	$\dfrac{\Gamma \vdash R \Rightarrow \Pi x{::}S \sqsubset A.\, T \qquad \Gamma \vdash N \Leftarrow S}{\Gamma \vdash R\, N \Rightarrow [N/x]_A\, T}$ **(Π-E)**

Subsorting is exceedingly simple: it only needs to be defined on atomic sorts, and is just the reflexive and transitive closure of the declared subsorting relationship. For dependent sort families, the indices must be equal.

$$\frac{s_1 \leq s_2 \in \Sigma}{s_1\, N_1 \ldots N_k \leq s_2\, N_1 \ldots N_k} \qquad \frac{}{Q \leq Q} \qquad \frac{Q_1 \leq Q' \qquad Q' \leq Q_2}{Q_1 \leq Q_2}$$

The sorting rules do not yet treat intersections. In line with the general bidirectional nature of the system, the introduction rules are part of the *checking* judgement, and the elimination rules are part of the *synthesis* judgement. Binary intersection $S_1 \wedge S_2$ has one introduction and two eliminations, while nullary intersection $\top$ has just one introduction.

$$\frac{\Gamma \vdash N \Leftarrow S_1 \qquad \Gamma \vdash N \Leftarrow S_2}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2}\ (\wedge\textbf{-I}) \qquad \frac{}{\Gamma \vdash N \Leftarrow \top}\ (\top\textbf{-I})$$

$$\frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_1}\ (\wedge\textbf{-E}_1) \qquad \frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_2}\ (\wedge\textbf{-E}_2)$$

Note that although canonical forms-style LF type synthesis is unique, LFR sort synthesis is not, due to the intersection elimination rules.

Now we can see how these rules generate a deduction of *succ* (*succ zero*) $\Leftarrow$ *even*. The context is always empty and therefore omitted. To save space, we abbreviate *even* as *e*, *odd* as *o*, *pos* as *p*, *zero* as *z*, and *succ* as *s*, and we omit

reflexive uses of subsorting.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash s \Rightarrow e \to o \wedge (o \to e \wedge \top \to p)}}{\vdash s \Rightarrow o \to e \wedge \top \to p}}{\vdash s \Rightarrow o \to e} \qquad \dfrac{\dfrac{\dfrac{}{\vdash s \Rightarrow e \to o \wedge (\ldots)}}{\vdash s \Rightarrow e \to o} \quad \dfrac{}{\vdash z \Rightarrow e}}{\vdash s\, z \Rightarrow o} \quad \dfrac{}{\vdash z \Leftarrow e}}{\vdash s\, z \Leftarrow o}}{\vdash s\,(s\, z) \Rightarrow e}}{\vdash s\,(s\, z) \Leftarrow e}$$

Using the $\wedge$**-I** rule, we can check that *succ zero* is both odd and positive:

$$\dfrac{\vdots \quad \vdots}{\vdash s\, z \Leftarrow o \quad \vdash s\, z \Leftarrow p}{\vdash s\, z \Leftarrow o \wedge p}$$

Each remaining subgoal now proceeds similarly to the above example.

To illustrate the use of sorts with non-trivial type *families*, consider the definition of the *double* relation in LF.

$$\begin{array}{l} \mathit{double} : \mathit{nat} \to \mathit{nat} \to \mathsf{type}. \\ \mathit{dbl\text{-}zero} : \mathit{double}\ \mathit{zero}\ \mathit{zero}. \\ \mathit{dbl\text{-}succ} : \Pi X{:}\mathit{nat}.\, \Pi Y{:}\mathit{nat}.\, \mathit{double}\, X\, Y \to \mathit{double}\,(\mathit{succ}\, X)\,(\mathit{succ}\,(\mathit{succ}\, Y)). \end{array}$$

With sorts, we can now directly express the property that the second argument to *double* must be even. But to do so, we require a notion analogous to *kinds* that may contain sort information. We call these *classes* and denote them by L.

$$\begin{array}{lll} K ::= \mathsf{type} \mid \Pi x{:}A.\, K & & \text{kinds} \\ L ::= \mathsf{sort} \mid \Pi x{::}S \sqsubset A.\, L \mid \top \mid L_1 \wedge L_2 & & \text{classes} \end{array}$$

Classes L mirror kinds K, and they have a refinement relation $L \sqsubset K$ similar to $S \sqsubset A$. (We elide the rules here.) Now, the general form of the $s \sqsubset a$ declaration is $s \sqsubset a :: L$, where $a : K$ and $L \sqsubset K$; this declares sort constant s to refine type constant a and to have class L.

We reuse the type name *double* as a sort, as no ambiguity can result. As before, we use $\top$ to represent a *nat* with no additional restrictions.

$$\begin{array}{l} \mathit{double} \sqsubset \mathit{double} :: \top \to \mathit{even} \to \mathsf{sort}. \\ \mathit{dbl\text{-}zero} :: \mathit{double}\ \mathit{zero}\ \mathit{zero}. \\ \mathit{dbl\text{-}succ} :: \Pi X{::}\top.\, \Pi Y{::}\mathit{even}.\, \mathit{double}\, X\, Y \to \mathit{double}\,(\mathit{succ}\, X)\,(\mathit{succ}\,(\mathit{succ}\, Y)). \end{array}$$

After these declarations, it would be a *sort error* to pose an Elf-like logic programming query such as "?- *double X* (*succ* (*succ* (*succ zero*)))." before any search is ever attempted. In LF, queries like this could fail after a long search or even not terminate, depending on the search strategy. One of the important motivations for considering sorts for LF is to avoid uncontrolled search in favor of decidable static properties whenever possible.

The tradeoff for such precision is that now sort checking itself is non-deterministic and has to perform search because of the choice between the two intersection elimination rules. As Reynolds has shown, this non-determinism causes intersection type checking to be PSPACE-hard [Rey96], even for normal terms as we have here [Rey89]. As discussed in Section 5, one element of this proposal is to determine the practicality of LFR sort checking by seeking heuristics that help in common cases.

2.2 A Second Example: The λ-Calculus

As a second example, we use an intrinsically typed version of the call-by-value simply-typed λ-calculus. This means every object language expression is indexed by its object language type. We use sorts to distinguish the set of *values* from the set of arbitrary *computations*. While this can be encoded in LF in a variety of ways, it is significantly more cumbersome.

tp : type. % the type of object language types
$\Rightarrow$: $tp \rightarrow tp \rightarrow tp$. % object language function space
%infix *right* 10 $\Rightarrow$.

exp : $tp \rightarrow$ type. % the type of expressions
$cmp \sqsubset exp$. % the sort of computations
$val \sqsubset exp$. % the sort of values

$val \leq cmp$. % every value is a (trivial) computation

$lam :: (val\ A \rightarrow cmp\ B) \rightarrow val\ (A \Rightarrow B)$.
$app :: cmp\ (A \Rightarrow B) \rightarrow cmp\ A \rightarrow cmp\ B$.

In the last two declarations, we follow Twelf convention and leave the quantification over A and B implicit, to be inferred by type reconstruction. Also, we did not explicitly declare a type for *lam* and *app*, since a practical front end should be able to recover this information from the refinement declarations for *val* and *cmp*, avoiding redundancy. Part of the proposed work is to specify a front end that does this kind of type reconstruction.

The most interesting declaration is the one for the constant *lam*. The argument type ($val\ A \rightarrow cmp\ B$) indicates that *lam* binds a variable which stands for a value of type A and the body is an arbitrary computation of type B. The result type $val\ (A \Rightarrow B)$ indicates that any λ-abstraction is a value. Now we have, for example (parametrically in A and B): $A{::}\top\sqsubset tp, B{::}\top\sqsubset tp \vdash lam\ \lambda x.\, lam\ \lambda y.\, x \Leftarrow val\ (A \Rightarrow (B \Rightarrow A))$.

Now we can express that evaluation must always returns a value. Since the declarations below are intended to represent a logic program, we follow the logic programming convention of reversing the arrows in the declaration of *ev-app*.

$eval :: cmp\ A \rightarrow val\ A \rightarrow$ sort.

ev-lam :: *eval* (*lam* $\lambda x.\, E\ x$) (*lam* $\lambda x.\, E\ x$).
ev-app :: *eval* (*app* $E_1\ E_2$) V
$\quad \leftarrow$ *eval* E_1 (*lam* $\lambda x.\, E'_1\ x$)
$\quad \leftarrow$ *eval* $E_2\ V_2$
$\quad \leftarrow$ *eval* $(E'_1\ V_2)\ V$.

Sort checking the above declarations demonstrates that evaluation always returns a value. Moreover, if type reconstruction gives E'_1 the "most general" sort *val* $A \rightarrow$ *cmp* A, the declarations also ensure that the language is indeed call-by-value: it would be a sort error to ever substitute a computation for a *lam*-bound variable, for example, by evaluating $(E'_1\ E_2)$ instead of $(E'_1\ V_2)$ in the *ev-app* rule. An interesting question for future work is whether type reconstruction can always find such a "most general" sort for implicitly quantified metavariables.

A side note: through the use of sort families indexed by object language types, the sort checking not only guarantees that the language is call-by-value and that evaluation, if it succeeds, will always return a value, but also that the object language type of the result remains the same (type preservation).

2.3 A Final Example: The Calculus of Constructions

As a final example, we present the Calculus of Constructions. Usually, there is a great deal of redundancy in its presentation because of repeated constructs at the level of objects, families, and kinds. Using sorts, we can enforce the stratification and write typing rules that are as simple as if we assumed the infamous *type* : *type*.

term : type. % terms at all levels

hyp $\sqsubset$ *term*. % hyperkinds (the classifier of "kind")
knd $\sqsubset$ *term*. % kinds
fam $\sqsubset$ *term*. % families
obj $\sqsubset$ *term*. % objects

tp :: *hyp* $\wedge$ *knd*.
pi :: *fam* $\rightarrow$ (*obj* $\rightarrow$ *fam*) $\rightarrow$ *fam* $\wedge$ % dependent function types, $\Pi x{:}A.\, B$
$\quad$ *fam* $\rightarrow$ (*obj* $\rightarrow$ *knd*) $\rightarrow$ *knd* $\wedge$ % type family kinds, $\Pi x{:}A.\, K$
$\quad$ *knd* $\rightarrow$ (*fam* $\rightarrow$ *fam*) $\rightarrow$ *fam* $\wedge$ % polymorphic function types, $\forall \alpha{:}K.\, A$
$\quad$ *knd* $\rightarrow$ (*fam* $\rightarrow$ *knd*) $\rightarrow$ *knd*. % type operator kinds, $\Pi \alpha{:}K_1.\, K_2$
lm :: *fam* $\rightarrow$ (*obj* $\rightarrow$ *obj*) $\rightarrow$ *obj* $\wedge$ % functions, $\lambda x{:}A.\, M$
$\quad$ *fam* $\rightarrow$ (*obj* $\rightarrow$ *fam*) $\rightarrow$ *fam* $\wedge$ % type families, $\lambda x{:}A.\, B$
$\quad$ *knd* $\rightarrow$ (*fam* $\rightarrow$ *obj*) $\rightarrow$ *obj* $\wedge$ % polymorphic abstractions, $\Lambda \alpha{:}K.\, M$
$\quad$ *knd* $\rightarrow$ (*fam* $\rightarrow$ *fam*) $\rightarrow$ *fam*. % type operators, $\lambda \alpha{:}K.\, A$
ap :: *obj* $\rightarrow$ *obj* $\rightarrow$ *obj* $\wedge$ % ordinary application, $M\ N$
$\quad$ *fam* $\rightarrow$ *obj* $\rightarrow$ *fam* $\wedge$ % type family application, $A\ M$
$\quad$ *obj* $\rightarrow$ *fam* $\rightarrow$ *obj* $\wedge$ % polymorphic instantiation, $M\ [A]$
$\quad$ *fam* $\rightarrow$ *fam* $\rightarrow$ *fam*. % type operator instantiation, $A\ B$

The typing rules can now be given non-redundantly, illustrating the implicit overloading afforded by the use of intersections. We omit the type conversion rule and auxiliary judgements for brevity.

$$
\begin{array}{l}
\mathit{of} :: \mathit{knd} \to \mathit{hyp} \to \mathsf{sort} \wedge \\
\quad \mathit{fam} \to \mathit{knd} \to \mathsf{sort} \wedge \\
\quad \mathit{obj} \to \mathit{fam} \to \mathsf{sort}.
\end{array}
$$

$$
\mathit{of\text{-}tp} :: \mathit{of}\ \mathit{tp}\ \mathit{tp}.
$$

$$
\begin{array}{l}
\mathit{of\text{-}pi} :: \mathit{of}\ (\mathit{pi}\ T_1\ \lambda x.\, T_2\ x)\ \mathit{tp} \\
\qquad \leftarrow \mathit{of}\ T_1\ \mathit{tp} \\
\qquad \leftarrow (\Pi x{:}\mathit{term}.\, \mathit{of}\ x\ T_1 \to \mathit{of}\ (T_2\ x)\ \mathit{tp}). \\
\mathit{of\text{-}lm} :: \mathit{of}\ (\mathit{lm}\ U_1\ \lambda x.\, T_2\ x)\ (\mathit{pi}\ U_1\ \lambda x.\, U_2\ x) \\
\qquad \leftarrow \mathit{of}\ U_1\ \mathit{tp} \\
\qquad \leftarrow (\Pi x{:}\mathit{term}.\, \mathit{of}\ x\ U_1 \to \mathit{of}\ (T_2\ x)\ (U_2\ x)). \\
\mathit{of\text{-}ap} :: \mathit{of}\ (\mathit{ap}\ T_1\ T_2)\ (U_1\ T_2) \\
\qquad \leftarrow \mathit{of}\ T_1\ (\mathit{pi}\ U_2\ \lambda x.\, U_1\ x) \\
\qquad \leftarrow \mathit{of}\ T_2\ U_2.
\end{array}
$$

Intersection types also provide a degree of modularity: by deleting some conjuncts from the declarations of *pi*, *lm*, and *ap* above, we can obtain an encoding of any point on the λ-cube.

Note, though, that in the rules *of-pi* and *of-lm*, we omit the sort annotation from the Π-bound variable, leaving it to be inferred by type reconstruction: each declaration actually turns into an intersection of declarations, and the type of the variable is different for each one. So the price of modularity is more complicated type reconstruction, part of the proposed work described in Section 5.

3 Metatheory

In this section, we present some metatheoretic results about our framework. These follow a similar pattern as previous work using hereditary substitutions [WCPW02, NPP07, HL07]. The following is only a high-level whirlwind tour of the LFR metatheory: we elide nearly all proofs and auxiliary lemmas. Details of the development can be found in the LFR technical report [LP07].

3.1 Hereditary Substitution

Recall that we replace ordinary capture-avoiding substitution with *hereditary substitution*, $[N/x]_A$, an operation which contracts any redexes it might create in the course of substitution. The operation is indexed by the putative type of N and x to facilitate a proof of termination. In fact, the type index on hereditary substitution need only be a simple type to ensure termination. To that end, we

denote simple types by α and define an erasure to simple types $(A)^-$.

$$\alpha ::= a \mid \alpha_1 \to \alpha_2 \qquad (a\ N_1 \ldots N_k)^- = a \qquad (\Pi x{:}A.\, B)^- = (A)^- \to (B)^-$$

For clarity, we also index hereditary substitutions by the syntactic category on which they operate, so for example we have $[N/x]^{\mathrm{n}}_A M = M'$ and $[N/x]^{\mathrm{s}}_A S = S'$. We write $[N/x]^{\mathrm{n}}_A M = M'$ as short-hand for $[N/x]^{\mathrm{n}}_{(A)^-} M = M'$.

Hereditary substitution is defined judgementally by inference rules. The only place β-redexes might be introduced is when substituting a normal term N into an atomic term R: N might be a λ-abstraction, and the variable being substituted for may occur at the head of R. Therefore, the judgements defining substitution into atomic terms are the most interesting ones.

We denote substitution into atomic terms by two judgements: $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$, for when the head of R is *not* x, and $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$, for when the head of R *is* x, where α' is the simple type of the output N'. The former is just defined compositionally; the latter is defined by two rules:

$$\frac{}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} x_0 = (N_0, \alpha_0)}\ \textbf{(subst-rn-var)}$$

$$\frac{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R_1 = (\lambda x.\, N_1, \alpha_2 \to \alpha_1) \qquad [N_0/x_0]^{\mathrm{n}}_{\alpha_0} N_2 = N_2' \qquad [N_2'/x]^{\mathrm{n}}_{\alpha_2} N_1 = N_1'}{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R_1\ N_2 = (N_1', \alpha_1)}\ \textbf{(subst-rn-}\beta\textbf{)}$$

The rule **subst-rn-var** just returns the substitutend N_0 and its putative type index α_0. The rule **subst-rn-**β applies when the result of substituting into the head of an application is a λ-abstraction; it avoids creating a redex by hereditarily substituting into the body of the abstraction.

A simple lemma establishes that these two judgements are mutually exclusive by examining the head of the input atomic term.

$$\mathrm{head}(x) = x \qquad \mathrm{head}(c) = c \qquad \mathrm{head}(R\ N) = \mathrm{head}(R)$$

Lemma 3.1.

1. *If* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$, *then* $\mathrm{head}(R) \neq x_0$.
2. *If* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$, *then* $\mathrm{head}(R) = x_0$.

Proof. By induction over the given derivation. □

Substitution into normal terms has two rules for atomic terms R, one which calls the "rr" judgement and one which calls the "rn" judgement.

$$\frac{[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0} R = R'}\ \textbf{(subst-n-atom)} \qquad \frac{[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (R', a')}{[N_0/x_0]^{\mathrm{n}}_{\alpha_0} R = R'}\ \textbf{(subst-n-atom-norm)}$$

Note that the latter rule requires both the term and the type returned by the "rn" judgement to be atomic.

Every other syntactic category's substitution judgement is defined compositionally.

Although we have only defined hereditary substitution relationally, it is easy to show that it is in fact a partial function by proving that there only ever exists one "output" for a given set of "inputs".

Theorem 3.2 (Functionality of Substitution)**.** *Hereditary substitution is a functional relation. In particular:*

1. *If* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R_1$ *and* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R_2$*, then* $R_1 = R_2$*,*
2. *If* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N_1, \alpha_1)$ *and* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N_2, \alpha_2)$*, then* $N_1 = N_2$ *and* $\alpha_1 = \alpha_2$*,*
3. *If* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N_1$ *and* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N_2$*, then* $N_1 = N_2$*,*

and similarly for other syntactic categories.

Additionally, it is worth noting that hereditary substitution so-defined behaves just as "ordinary" substitution on terms that do not contain the distinguished free variable.

Theorem 3.3 (Trivial Substitution)**.** *Hereditary substitution for a non-occurring variable has no effect.*

1. *If* $x_0 \notin \mathrm{FV}(R)$*, then* $[N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R$*,*
2. *If* $x_0 \notin \mathrm{FV}(N)$*, then* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N$*,*

and similarly for other syntactic categories.

3.2 Decidability

A hallmark of the canonical forms/hereditary substitution approach is that it allows a decidability proof to be carried out comparatively early, before proving anything about the behavior of substitution, and without dealing with any complications introduced by β/η-conversions inside types. Ordinarily in a dependently typed calculus, one must first prove a substitution theorem before proving typechecking decidable, since typechecking relies on type equality, type equality relies on β/η-conversion, and β/η-conversions rely on substitution preserving well-formedness. (See for example [HP05] for a typical non-canonical account of LF definitional equality.)

In contrast, if only canonical forms are permitted, then type equality is just α-convertibility, so one only needs to show *decidability* of substitution in order to show decidability of typechecking. Since LF encodings represent judgements as type families and proof-checking as typechecking, it is comforting to have a decidability proof that relies on so few assumptions.

Theorem 3.4 (Decidability of Substitution)**.** *Hereditary substitution is decidable. In particular:*

1. *Given* N_0, x_0, α_0, *and* R, *either* $\exists R'.\, [N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$, *or* $\nexists R'.\, [N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$,
2. *Given* N_0, x_0, α_0, *and* R, *either* $\exists (N', \alpha').\, [N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$, *or* $\nexists (N', \alpha').\, [N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$,
3. *Given* N_0, x_0, α_0, *and* N, *either* $\exists N'.\, [N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N'$, *or* $\nexists N'.\, [N_0/x_0]^{\mathrm{n}}_{\alpha_0} N = N'$,

and similarly for other syntactic categories.

Theorem 3.5 (Decidability of Subsorting)**.** *Given* Q_1 *and* Q_2, *either* $Q_1 \leq Q_2$ *or* $Q_1 \not\leq Q_2$.

Proving decidability of the given typing rules directly is technically tricky due to a combination of the bidirectional character of natural deduction and our non-deterministic intersection synthesis rules: since the intersection elimination rules are not directed by the structure of the term, there's nothing obvious to induct on. Intuitively, though, the checking rules always decrease the size of the sort on the way up a derivation while the synthesis rules always decrease the size of the sort on the way down, making both judgements decidable.

To make this argument precise, we give an alternative deterministic formulation of the typing rules that is sound and complete with respect to the original rules; the intersection eliminations are delayed as long as possible and no information is discarded during synthesis. Details of the alternative system and its soundness and completeness can be found in the LFR technical report [LP07].

Theorem 3.6 (Decidability of Sort Checking)**.** *Sort checking is decidable. In particular:*

1. *Given* Γ, N, *and* S, *either* $\Gamma \vdash N \Leftarrow S$ *or* $\Gamma \nvdash N \Leftarrow S$,
2. *Given* Γ, S, *and* A, *either* $\Gamma \vdash S \sqsubset A$ *or* $\Gamma \nvdash S \sqsubset A$, *and*
3. *Given* Σ, *either* $\vdash \Sigma\ \mathsf{sig}$ *or* $\nvdash \Sigma\ \mathsf{sig}$.

3.3 Identity and Substitution Principles

Since well-typed terms in our framework must be canonical, that is β-normal and η-long, it is non-trivial to prove $S \to S$ for non-atomic S, or to compose proofs of $S_1 \to S_2$ and $S_2 \to S_3$. The Identity and Substitution principles ensure that our type theory makes logical sense by demonstrating the reflexivity and transitivity of entailment. Reflexivity is witnessed by η-expansion, while transitivity is witnessed by hereditary substitution.

The Identity Principle effectively says that synthesizing (atomic) objects can be made to serve as checking (normal) objects. The Substitution Principle dually says that checking objects may stand in for synthesizing assumptions, that is, variables. The Substitution Theorem also tells us that if all of its subjects are

well-formed, a heriditary substitution exists—a non-trivial fact, since hereditary substitution is in general a partial operation. In this way, the Substitution Theorem is effectively a normalization theorem.

Theorem 3.7 (Expansion)**.** *If* $\Gamma \vdash S \sqsubset A$ *and* $\Gamma \vdash R \Rightarrow S$*, then* $\Gamma \vdash \eta_A(R) \Leftarrow S$*.*

Theorem 3.8 (Identity)**.** *If* $\Gamma \vdash S \sqsubset A$*, then* $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow S$*.*

Theorem 3.9 (Substitution)**.** *Suppose* $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$ *. Then:*

1. *If*
   - $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}}\ \mathsf{ctx}$ *, and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}} \vdash S \sqsubset A$ *, and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ *,*

   then
   - $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$ *and* $\vdash \Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}}\ \mathsf{ctx}$ *, and*
   - $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S'$ *and* $[N_0/x_0]^{\mathrm{a}}_{A_0}\, A = A'$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash S' \sqsubset A'$ *, and*
   - $[N_0/x_0]^{\mathrm{n}}_{A_0}\, N = N'$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$ *,*

2. *If*
   - $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}}\ \mathsf{ctx}$ *and*
   - $\Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}} \vdash R \Rightarrow S$ *,*

   then
   - $[N_0/x_0]^{\gamma}_{A_0}\, \Gamma_{\mathrm{R}} = \Gamma'_{\mathrm{R}}$ *and* $\vdash \Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}}\ \mathsf{ctx}$ *, and* $[N_0/x_0]^{\mathrm{s}}_{A_0}\, S = S'$ *, and either*
     - $[N_0/x_0]^{\mathrm{rr}}_{A_0}\, R = R'$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash R' \Rightarrow S'$ *, or*
     - $[N_0/x_0]^{\mathrm{rn}}_{A_0}\, R = (N', \alpha')$ *and* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$ *,*

and similarly for other syntactic categories.

The proofs of the Identity and Substitution principles use only elementary methods like structural induction—in particular, they require nothing akin logical relations. Despite their proof-theoretic simplicity, though, they must be very carefully staged. Substitution as stated does not admit a straightforward inductive proof: instead, it is broken up into several "Proto-Substitution" theorems which do not require well-formedness of all their subjects, but instead assume just those properties that are required for each step of the proof.

For instance, Proto-Substitution for normal terms $\Gamma \vdash N \Leftarrow S$ does not require the context Γ or the classifying sort S to be well-formed, but it instead merely requires that hereditary substitution be defined on them. This is sufficient to show that hereditary substitution is defined on the term N and that its substitution instance has the substituted sort in the substituted context.

After proving Proto-Substitution for terms, we can go on to show Proto-Substitution for sorts and types. Using this, we can show Proto-Substitution for contexts. Finally, we have enough to prove the stated Substitution Theorem as a direct corollary.

The Expansion principle's proof is a straightforward structural induction, with one appeal to Substitution to determine that hereditary substitution is defined on a well-formed sort. The Identity principle follows as a direct corollary.

Along the way, we require two lemmas, one about how substitutions compose and another about how they commute with η-expansions. To prove the Substitution theorem, we need a lemma analogous to the property of non-hereditary substitutions that $[N_0/x_0]\,[N_2/x_2]\,N = [[N_0/x_0]\,N_2/x_2]\,[N_0/x_0]\,N$. Hereditary substitutions enjoy a similar property, with an additional proviso about definedness: if the three "inner" substitutions are defined, then the two "outer" ones are also defined, and equal. For the Expansion theorem, we need to know that substitution of variable's η-expansion for itself acts as an identity substitution.[4]

Lemma 3.10 (Composition of Substitutions)**.** *Suppose* $[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N_2 = N_2^{\backprime}$ *and* $x_2 \notin \mathrm{FV}(N_0)$*. Then if*

$$[N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N = N^{\backprime} \text{ and } [N_2/x_2]^{\mathrm{n}}_{\alpha_2}\, N = N',$$

then for some $N^{\backprime\prime}$*,*

$$[N_2^{\backprime}/x_2]^{\mathrm{n}}_{\alpha_2}\, N^{\backprime} = N^{\backprime\prime} \text{ and } [N_0/x_0]^{\mathrm{n}}_{\alpha_0}\, N' = N^{\backprime\prime}.$$

and similarly for other syntactic categories.

Lemma 3.11 (Commutativity of Substitution and η-expansion)**.** *Substitution commutes with η-expansion. In particular:*

1. *If* $[\eta_\alpha(x)/x]^{\mathrm{n}}_{\alpha}\, N = N'$*, then* $N' = N$ *,*
2. *If* $[N/x]^{\mathrm{n}}_{\alpha}\, \eta_\alpha(x) = N'$*, then* $N' = N$ *,*

and similarly for other syntactic categories.

4 Subsorting at Higher Sorts

Our bidirectional typing discipline limits subsorting checks to a single rule, the **switch** rule when we switch modes from checking to synthesis. Since we insist on typing only canonical forms, this rule is limited to checking at atomic sorts Q, and consequently, subsorting need only be defined on atomic sorts. These observations naturally lead one to ask, what is the status of higher-sort subsorting in LFR? How do our intuitions about things like structural rules, variance, and distributivity—in particular, the rules shown in Figure 1—fit into the LFR picture?

[4]The categorically-minded reader might think of the composition lemma as the associativity of $\circ$ and commutativity as the right- and left-unit laws for $\circ$, where $\circ$ in the category represents substitution, as usual.

$$\boxed{S_1 \leq S_2}$$

$$\frac{}{S \leq S}\ \textbf{(refl)} \qquad \frac{S_1 \leq S_2 \qquad S_2 \leq S_3}{S_1 \leq S_3}\ \textbf{(trans)} \qquad \frac{S_2 \leq S_1 \qquad T_1 \leq T_2}{\Pi x{::}S_1.\, T_1 \leq \Pi x{::}S_2.\, T_2}\ \textbf{(S-}\Pi\textbf{)}$$

$$\frac{}{S \leq \top}\ (\top\textbf{-R}) \quad \frac{T \leq S_1 \qquad T \leq S_2}{T \leq S_1 \wedge S_2}\ (\wedge\textbf{-R}) \quad \frac{S_1 \leq T}{S_1 \wedge S_2 \leq T}\ (\wedge\textbf{-L}_1) \quad \frac{S_2 \leq T}{S_1 \wedge S_2 \leq T}\ (\wedge\textbf{-L}_2)$$

$$\frac{}{\top \leq \Pi x{::}S.\, \top}\ (\top/\Pi\textbf{-dist})$$

$$\frac{}{(\Pi x{::}S.\, T_1) \wedge (\Pi x{::}S.\, T_2) \leq \Pi x{::}S.\, (T_1 \wedge T_2)}\ (\wedge/\Pi\textbf{-dist})$$

Figure 1: Derived rules for subsorting at higher sorts.

It turns out that despite not *explicitly* including subsorting at higher sorts, LFR *implicitly* includes an intrinsic notion of higher-sort subsorting through the η-expansion associated with canonical forms. The simplest way of formulating this intrinsic notion is as a variant of the identity principle: S is taken to be a subsort of T if $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$. This notion is equivalent to a number of other alternate formulations, including a subsumption-based formulation and a substitution-based formulation.

Theorem 4.1 (Alternate Formulations of Subsorting)**.** *Suppose that for some* Γ_0, $\Gamma_0 \vdash S_1 \sqsubset A$ *and* $\Gamma_0 \vdash S_2 \sqsubset A$*, and define:*

1. $S_1 \leq_1 S_2 \stackrel{def}{=}$ *for all* Γ *and* R*: if* $\Gamma \vdash R \Rightarrow S_1$*, then* $\Gamma \vdash \eta_A(R) \Leftarrow S_2$.
2. $S_1 \leq_2 S_2 \stackrel{def}{=}$ *for all* Γ*:* $\Gamma, x{::}S_1 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$.
3. $S_1 \leq_3 S_2 \stackrel{def}{=}$ *for all* Γ *and* N*: if* $\Gamma \vdash N \Leftarrow S_1$*, then* $\Gamma \vdash N \Leftarrow S_2$.
4. $S_1 \leq_4 S_2 \stackrel{def}{=}$ *for all* Γ_{L}, Γ_{R}, N, *and* S*: if* $\Gamma_{\mathrm{L}}, x{::}S_2 \sqsubset A, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ *then* $\Gamma_{\mathrm{L}}, x{::}S_1 \sqsubset A, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$
5. $S_1 \leq_5 S_2 \stackrel{def}{=}$ *for all* Γ_{L}, Γ_{R}, N, S, *and* N_1*: if* $\Gamma_{\mathrm{L}}, x{::}S_2 \sqsubset A, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ *and* $\Gamma_{\mathrm{L}} \vdash N_1 \Leftarrow S_1$*, then* $\Gamma_{\mathrm{L}}, [N_1/x]^{\gamma}_A \Gamma_{\mathrm{R}} \vdash [N_1/x]^{\mathrm{n}}_A N \Leftarrow [N_1/x]^{\mathrm{s}}_A S$.

Then, $S_1 \leq_1 S_2 \iff S_1 \leq_2 S_2 \iff \cdots \iff S_1 \leq_5 S_2$.

Proof. Each implication $1 \Longrightarrow 2 \Longrightarrow \cdots \Longrightarrow 5 \Longrightarrow 1$ follows directly from the Identity and Substitution principles along with Lemma 3.11, the Commutativity of Substitution with η-expansion. □

If we take "subsorting as η-expansion" to be our *model* of subsorting, we can show the "usual" presentation in Figure 1 to be both sound and complete with respect to this model. In other words, subsorting as η-expansion *really is* subsorting (soundness), and it is *no more than* subsorting (completeness). Alternatively, we can say that completeness demonstrates that there are no subsorting rules missing from the usual declarative presentation: Figure 1 accounts for everything covered intrinsically by η-expansion.

Theorem 4.2 (Soundness of Declarative Subsorting)**.** *If $S \leq T$, then $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$.*

Theorem 4.3 (Completeness of Declarative Subsorting)**.** *If $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow T$, then $S \leq T$.*

Soundness is a straightforward inductive argument. The proof of completeness is considerably more intricate. We demonstrate completeness via a detour through an algorithmic subsorting system very similar to the algorithmic typing system used to show decidability, mentioned above in Section 3.2. To show completeness, we show that intrinsic subsorting implies algorithmic subsorting and that algorithmic subsorting implies declarative subsorting; the composition of these theorems is our desired completeness result.

5 Proposed Work

The main core of the proposed work revolves around understanding enough of LFR that it would be possible to build a practical logical framework suitable for mechanized metatheory, similar to the Twelf system [PS99]. In particular, this entails understanding LFR representation methodology, solving a suitable fragment of higher-order unification for LFR terms, and determining a useful enough type reconstruction algorithm.

5.1 Representation Methodology

An important step in designing a new logical framework is to understand its representation methodology. The original LF was founded on the principle of *judgements as types*, in which a deductive system's judgements are represented as LF type families. The linear logical framework LLF was designed to make use of *state as linear hypotheses* [CP02], where the evolution of a stateful system is represented as a context of linear resources. Most recently, Watkins et al. described the concurrent logical framework CLF and its guiding principle of *concurrent computations as monadic expressions* [WCPW02, WCPW04], where possible reorderings of concurrent computations are captured by an equality relation modding out by permutative conversions of monadic binding.

A complete understanding of representation methodology entails an understanding of the notion of *adequacy*. Normally, one's LF formalizations are backed by an adequacy theorem, which exhibits an isomorphism between constructs

of one's deductive system and canonical forms of certain LF type families; then formal results about an encoding automatically represent results about the informal system. What kind of isomorphism must we exhibit to show an LFR encoding adequate? Is the nature of the isomorphism changed by the fact that synthesized sorts are not unique, as synthesized types are in LF? Does our interpretation that terms are typechecked before they are sort-checked allow us to leverage an LF adequacy theorem when proving an LFR adequacy theorem for an extended system?

What is the representation methodology embodied by the design of LFR? Our *sorts*, through the intrinsic interpretation of subsorting, represent a form of judgemental inclusion, so perhaps it should be something like *judgemental inclusion as refinement types*. In order to better characterize the general methods of representing deductive systems in LFR, I propose to create a suite of example encodings from a variety of domains representative of the kinds of formalizations I expect LFR users to carry out. Not only would a suite of examples elucidate a representation methodology, but also it would serve to demonstrate the utility of LFR—although the examples from Section 2 suggest the utility of LFR, they only begin to scratch the surface.

Furthermore, in order to understand the meaning of adequacy for LFR encodings, I propose to do several careful proofs of adequacy, in the style of Harper and Licata [HL07], for a representative subset of this suite of examples.

5.2 Subset Interpretation

The subset interpretation is a translation of LFR signatures and derivations into ordinary LF signatures and derivations. Sorts are represented as predicates on type families, and sort declarations for constants turn into predicate declarations for those constants. The running example of natural numbers illustrates the key ideas succinctly. Recall the signature:

$$
\begin{array}{l}
\mathit{nat} : \mathsf{type}. \\
\mathit{zero} : \mathit{nat}. \\
\mathit{succ} : \mathit{nat} \to \mathit{nat}.
\end{array}
$$

$$
\begin{array}{l}
\mathit{even}, \mathit{odd}, \mathit{pos} \sqsubset \mathit{nat}. \\
\mathit{zero} :: \mathit{even}. \\
\mathit{succ} :: \mathit{even} \to \mathit{odd} \\
\quad \wedge\ \mathit{odd} \to \mathit{even} \\
\quad \wedge\ \top \to \mathit{pos}.
\end{array}
$$

Under the subset interpretation, *even*, *odd*, and *pos* are represented as predicates over natural numbers, i.e. unary type families, and the two sort declarations for *zero* and *succ* turn into declarations of ways to prove those predicates for given natural numbers.

$$
\begin{array}{l}
\mathit{nat} : \mathsf{type}. \\
\mathit{zero} : \mathit{nat}.
\end{array}
$$

$succ : nat \to nat.$

$even, odd, pos : nat \to \mathsf{type}.$

$p{-}zero : even\ zero.$
$p{-}succ : (\Pi x{:}nat.\ even\ x \to odd\ (succ\ x))$
$\quad \times (\Pi x{:}nat.\ odd\ x \to even\ (succ\ x))$
$\quad \times (\Pi x{:}nat.\ \mathbf{1} \to pos\ (succ\ x)).$

As you can see, the sort declaration for the constant *zero* turned into a constructor for predicates that hold of *zero* and the sort declaration for the constant *succ* turned into a constructor for predicates that hold of natural numbers constructed using *succ*. For compositionality's sake it is convenient to let our target language extend LF with products and unit. This is really no extension at all, though: products can always be eliminated by distributing them over arrows until they reach the top-level of the signature, where they can then be split into multiple declarations.

A compositional translation following these principles is relatively straightforward to carry out, if slightly tedious. The translation works over sorting derivations: a derivation of $\Gamma \vdash S \sqsubset A$ turns into a type with a hole, $\hat{S}(-)$, and a derivation of $\Gamma \vdash N \Leftarrow S$ turns into a proof $\hat{N} : \hat{S}(N)$ representing evidence that the term N has property S. Intersections of sorts become products of proofs, and the trivial sort $\top$ becomes the trivial unit type.

As with any translation based on non-unique derivations, we must be careful to ensure coherence [BTCGS91, Rey91]: any two translations of the same judgement should be equal. Although working with a canonical forms-only presentation obviates any of the usual reasoning about β/η-equality, we have introduced something new to the game: equality of evidences.

As it turns out, the naïve subset interpretation is *not* coherent: some equalities that hold in the source might *not* hold after translation. Since LF typing depends upon term equality, some terms which typechecked in the source might not typecheck in the translation. A slightly contrived example illustrates the phenomenon:

$a : \mathsf{type}.$
$c : a.$

$s \sqsubset a.$
$c :: s \wedge s.$ % note nondeterminism!

$eq : a \to a \to \mathsf{type}.$ % equality on a's
$eq/i : \Pi x{:}a.\ eq\ x\ x.$

$seq \sqsubset eq :: s \to s \to \mathsf{sort}.$ % equality on s's
$eq/i :: \Pi x{::}s.\ seq\ x\ x.$

Under this signature, we can derive both $\cdot \vdash \mathit{seq}\ c\ c \sqsubset \mathit{eq}\ c\ c$ and $\cdot \vdash \mathit{eq/i}\ c \Leftarrow \mathit{seq}\ c\ c$. But beware, for both derivations will contain subderivations of $\cdot \vdash c \Leftarrow s$, and there are *two* derivations that $\cdot \vdash c \Leftarrow s$—one derivation uses the first conjunct of c's sort declaration, the other uses the second. The signature's translation under the subset interpretation is as follows:

a : type.
$c : a$.
$s : a \to$ type.
$p{-}c : s\ c \times s\ c$.

$\mathit{eq} : a \to a \to$ type.
$\mathit{eq/i} : \Pi x{:}a.\ \mathit{eq}\ x\ x$.

$\mathit{seq} : \Pi x{:}a.\ s\ x \to \Pi y{:}a.\ s\ y \to \mathit{eq}\ x\ y \to$ type.
$p{-}\mathit{eq/i} : \Pi x{:}a.\ \Pi p x{:}s\ x.\ \mathit{seq}\ x\ p x\ x\ p x\ (\mathit{eq/i}\ x)$.

Owing to the nondeterminism discussed above, there are two translations of $\cdot \vdash c \Leftarrow s$, namely $\cdot \vdash \pi_1\ p{-}c \Leftarrow s\ c$ and $\cdot \vdash \pi_2\ p{-}c \Leftarrow s\ c$. Consequently, there are several translations for the sort $\cdot \vdash \mathit{seq}\ c\ c \sqsubset \mathit{eq}\ c\ c$; one of them is $\mathit{seq}\ c\ (\pi_1\ p{-}c)\ c\ (\pi_2\ p{-}c)\ (-)$, where we use $(-)$ to represent the hole of type $\mathit{eq}\ c\ c$. We also have that the term $\cdot \vdash \mathit{eq/i}\ c \Leftarrow \mathit{seq}\ c\ c$ translates to the proof $p{-}\mathit{eq/i}\ c\ (\pi_1\ p{-}c)$. But it is impossible to derive $\cdot \vdash p{-}\mathit{eq/i}\ c\ (\pi_1\ p{-}c) \Leftarrow \mathit{seq}\ c\ (\pi_1\ p{-}c)\ c\ (\pi_2\ p{-}c)\ (\mathit{eq/i}\ c)$, as our soundness criterion above would require—the sort uses two different translations for c, and the signature requires them to be equal.

The solution to the coherence problem *proof irrelevance* [Pfe01]. If we specify in our translation that proofs of sorting predicates appear in irrelevant positions, then we obtain exactly the equalities we need for the translation to be coherent. Proof irrelevance provides the essential ingredient: for a term to belong to a sort, we require a proof that it has that sort, but *we don't care which one*: any proof will suffice, and all such proofs should be treated as equal.

To date, I have exhibited several candidate subset interpretations, differing chiefly in minor syntactic details. I propose to settle on one presentation as canonical and to carry out the requisite proof that it is sound and complete in the appropriate sense, along the way demonstrating that by treating sorting proofs as irrelevant, we obtain all the desired equalities in the target of the translation.

5.3 Coercion Interpretation

An alternative interpretation of LFR into some form of LF without refinements is the *coercion interpretation*. Under this interpretation, sorts are represented as types, objects with sorts are represented as objects with types, and inclusions between sorts are represented as coercion functions. The coercion interpretation corresponds to so-called "intrinsic encodings" in LF, where for instance, values

in a λ-calculus are represented as a separate type family and an explicit coercion from values to expressions is assumed. The LFR signature

exp : type.
$val \sqsubset exp$.

lam : $(exp \to exp) \to exp$.
app : $exp \to exp \to exp$.

lam :: $(val \to exp) \to val$.

might be translated to the LF signature

exp : type.
val : type.
$val2exp$: $val \to exp$.

lam : $(exp \to exp) \to exp$.
app : $exp \to exp \to exp$.

$vlam$: $(val \to exp) \to val$.

The issues here are subtler than in the subset interpretation. We would like the terms $val2exp$ ($vlam\ \lambda x.val2exp\ x$) and $lam\ \lambda x.x$ to be equal, since their sources were equal before translation. This is another manifestation of the coherence problem, but perhaps one more closely related to coercive interpretations of subtyping for functional languages [BTCGS91].

The coercion interpretation has been studied previously by Chaudhuri (personal communication). In his studies, he came to believe that a variation on the usual notion of proof irrelevance might handle the coherence problem. The precise characterization of this notion was never completed successfully, however, so no viable coercion interpretation has been exhibited to date.

It is possible that using the modern canonical forms-style presentation of LFR, the relevant issues will be more transparent than they were in previous studies. Therefore, I propose to carry out a brief study of the coercion interpretation, not only to see if its character becomes clearer when viewed through a more modern lens, but also to glean any useful knowledge it may have to offer about the relation between LFR and LF without refinements: such knowledge will be invaluable when trying to extend LFR with the metatheoretic capabilities of Twelf. It may turn out that this branch of study yields no useful insights or artifacts, but I believe it is worth exploring at least briefly.

5.4 Contextual Modal Formulation

Contextual Modal Type Theory [NPP07] provides an account of logic programming *metavariables*, an integral component to an understanding of unification. Thus, before embarking on a study of metalogical reasoning in LFR, I propose to reformulate Contextual Modal Type Theory in the presence of refinement

types. I do not anticipate any great difficulties in the extension to sorts, especially since CMTT is already presented in a modern canonical forms style. Nonetheless, an account of metavariables is a necessary first step to turning LFR into a usable metalogical framework.

5.5 Unification

Unification lies at the heart of a metalogical framework: it is an integral component of logic programming, type reconstruction, pattern matching, and coverage checking, at least. In short, unification is the engine that makes formal reasoning possible. I therefore propose to examine the problem of unification in the presence of refinement types.

Many of the essential difficulties of unification in the presence of intersection types have been studied in the simply-typed case by Kohlhase and Pfenning [KP93]. Kohlhase and Pfenning primarily studied problem of unifiability, though, without necessarily seeking most general unifiers. It remains therefore not only to adapt their techniques to the dependent case, but also to bring their presentation more into line with modern views of pattern unification.

The main problem of unification with intersection types is choosing the types of generated metavariables. Consider the following fragment, which defines a type of boolean formulas and refinements for true formulas and false formulas.

$$
\begin{array}{l}
\mathit{bool} : \mathsf{type}. \\
\\
t \sqsubset \mathit{bool}. \\
f \sqsubset \mathit{bool}. \\
\\
\mathit{true} :: t. \\
\mathit{false} :: f. \\
\\
\mathit{and} :: t \to t \to t \\
\quad \wedge\ t \to f \to f \\
\quad \wedge\ f \to t \to f \\
\quad \wedge\ f \to f \to f.
\end{array}
$$

As you can see, the sort given for *and* specifies its truth table completely, giving a great deal of type information for the unification algorithm to work with. Suppose that in this signature we are presented with a unification problem $X{::}t \to f; y{::}t \vdash X\ y \doteq \mathit{and}\ M\ N : f$. Following standard pattern unification techniques, we can set X to $\lambda b.\mathit{and}\ (X_1\ b)\ (X_2\ b)$, with one of

$$
\begin{array}{l}
X_1 :: t \to t, X_2 :: t \to f, \text{or} \\
X_1 :: t \to f, X_2 :: t \to t, \text{or} \\
X_1 :: t \to f, X_2 :: t \to f.
\end{array}
$$

The question is how do we choose which of these incomparable metavariable typings to proceed with? Following Kohlhase, we can make up type variables

to represent the codomain types of X_1 and X_2 and adding a subtyping constraint to be solved later. In particular, for this problem, we can set

$$X_1 :: t \to \alpha_1, X_2 :: t \to \alpha_2$$

and include the constraint

$$t \to f \to f \wedge f \to t \to f \wedge f \to f \to f \;\leq\; \alpha_1 \to \alpha_2 \to f$$

Each conjunct in the left-hand side represents one of the possible typings given above; the two argument positions of each conjunct represent the codomain types of X_1 and X_2, respectively; and the final f represents the type of the original unification problem. Inverting the subtyping constraint, we can see that it is equivalent to asserting that either α_1 is t and α_2 is f, or that α_1 is f and α_2 is t, or that α_1 and α_2 are both f — in other words, exactly the *dis*junction we wanted to specify above.

Since in LFR, we only have primitive subtyping at base types, this may not be precisely the right strategy for encoding subtyping constraints, but it is comforting nonetheless to know that the essential ingredients of intersection unification have been studied before. Ideally, we would like to exhibit a unification algorithm that cleanly separates term unification from refinement type checking, and to prove that such an algorithm is still complete.

5.6 Type Reconstruction

Some form of type reconstruction is essential for making a logical framework practical for mechanized metatheory. In Twelf, type reconstruction is mainly concerned with quantifying over and determining the types of implicit metavariables. Reconstruction can be tricky because the implicit metavariables in a clause may not always be syntactically evident due to dependencies.

Twelf solves the problem using a two-pass algorithm that first computes simple types to determine which metavariables require quantification and second uses pattern unification to solve for the types of the metavariables [PS99]. Although the general problem is undecidable [Gol81], Twelf's algorithm always terminates, at the very least with an error saying that the original source requires more annotations to ensure successful reconstruction. Anecdotal experience suggests that its algorithm is quite good for many kinds of examples that arise in practice.

The chief difficulty in adapting Twelf-style type reconstruction to LFR is that inferred sorts may not be unique. By analogy with unification-based inference returning a most-general unifier, we conjecture that the correct behavior is for inference to choose the most general sort that is type correct, if a most general sort exists. Consider the doubling relation example from above:

double $\sqsubset$ *double* :: $\top \to$ *even* $\to$ sort.
dbl−zero :: *double zero zero*.
dbl−succ :: *double X Y* $\to$ *double* (*succ X*) (*succ* (*succ Y*)).

In the last line, the sort of X is unconstrained: the most general sort that makes the clause well-sorted is $X :: \top$. The metavariable Y, however, must be given sort *even* for the clause to be well-sorted, and this sort is the only such sort. Therefore, after reconstruction, the clause reads as follows:

$$\begin{aligned}&\mathit{dbl{-}succ} :: \Pi X{::}\top.\ \Pi Y{::}\mathit{even}.\\ &\qquad \mathit{double}\ X\ Y \rightarrow \mathit{double}\ (\mathit{succ}\ X)\ (\mathit{succ}\ (\mathit{succ}\ Y)).\end{aligned}$$

A slightly more interesting example comes from the evaluation relation for the simply-typed lambda calculus discussed above.

$$\begin{aligned}&\mathit{eval} :: \mathit{cmp}\ A \rightarrow \mathit{val}\ A \rightarrow \mathsf{sort}.\\ &\mathit{ev{-}app} :: \mathit{eval}\ (\mathit{app}\ E1\ E2)\ V\\ &\qquad \leftarrow \mathit{eval}\ E1\ (\mathit{lam}\ \lambda x.E1'\ x)\\ &\qquad \leftarrow \mathit{eval}\ E2\ V2\\ &\qquad \leftarrow \mathit{eval}\ (E1'\ V2)\ V.\end{aligned}$$

Most of the metavariable sorts here are straightforward consequences of propagating type information. For $E1'$ though, there is a choice: it could be given sort $\mathit{val}\ A \rightarrow \mathit{cmp}\ A$ or sort $\mathit{cmp}\ A \rightarrow \mathit{cmp}\ A$. By the above claim that we should assign it the most general sort, we should give it sort $\mathit{val}\ A \rightarrow \mathit{cmp}\ A$, since $\mathit{cmp}\ A \rightarrow \mathit{cmp}\ A \leq \mathit{val}\ A \rightarrow \mathit{cmp}\ A$. Indeed, this is not only the sort that gives us the most precise typing guarantees (as discussed above), but also the sort that helps coverage checking the most (as discussed below)

An open question is what should be done when a metavariable could be assigned several incomparable sorts. Consider two equality relations on natural numbers, a "smart" equality *smart−eq* and a "dumb" equality *dumb−eq*, with a coercion from the former to the latter:

$$\begin{aligned}&\mathit{equal} : \mathit{nat} \rightarrow \mathit{nat} \rightarrow \mathsf{type}.\\ &\mathit{smart{-}eq} \sqsubset \mathit{equal} :: \mathit{even} \rightarrow \mathit{even} \rightarrow \mathsf{sort} \wedge \mathit{odd} \rightarrow \mathit{odd} \rightarrow \mathsf{sort}.\\ &\mathit{dumb{-}eq}\ \sqsubset \mathit{equal} :: \top \rightarrow \top \rightarrow \mathsf{sort}.\\ \\ &\mathit{coerce} :: \mathit{smart{-}eq}\ X\ Y \rightarrow \mathit{dumb{-}eq}\ X\ Y.\end{aligned}$$

To make the last clause well-sorted, there are two possibilities: either X and Y must both be *odd* or they must both be *even*. Therefore, reconstruction should convert it to the following clause:

$$\begin{aligned}&\mathit{coerce} :: \Pi X{::}\mathit{even}.\ \Pi Y{::}\mathit{even}.\ \mathit{smart{-}eq}\ X\ Y \rightarrow \mathit{dumb{-}eq}\ X\ Y\\ &\qquad \wedge\ \Pi X{::}\mathit{odd}.\ \Pi Y{::}\mathit{odd}.\ \mathit{smart{-}eq}\ X\ Y \rightarrow \mathit{dumb{-}eq}\ X\ Y.\end{aligned}$$

It is unclear exactly how type reconstruction should proceed to come up with this clause, and part of the proposed work is to closely examine examples like this to come up with a practical type reconstruction algorithm for LFR. (The encoding of the Calculus of Constructions in Section 2 is another good example of this phenomenon.)

Another motivation of LFR type reconstruction is to avoid redundancy between LF typing declarations and LFR sorting declarations. For sorts that do not include the maximal sort $\top$, it is clear what sort they refine, and so

type reconstruction could automatically come up with such "obvious" typing declarations. For instance, if we specify

nat : type.
$even \sqsubset nat$.
$odd \sqsubset nat$.

$zero :: even$.
$succ :: even \rightarrow odd \wedge odd \rightarrow even$.

then we should not need to include declarations

$zero : nat$.
$succ : nat \rightarrow nat$.

since we can see that they must exist for the given sorting declarations to be well-formed. Even if our sorting declaration for s included some conjuncts with $\top$'s in them, as in

$pos \sqsubset nat$.

$succ :: even \rightarrow odd \wedge odd \rightarrow even \wedge \top \rightarrow pos$.

the typing declarations are still inferrable since at least one conjunct makes them obvious.

I propose to study several examples along these lines to come up with (a) a suitable concrete syntax for LFR signatures and (b) a practical type reconstruction algorithm for LFR signatures. Both should be on par with Twelf's own syntax and reconstruction in terms of convenience and practicality.

5.7 Coverage Checking

Coverage checking is the problem of determining whether a set of pattern clauses cover all possible instances of a type family [SP03], and it is a crucial component of metatheorem checking in a metalogical framework such as Twelf. As with type reconstruction, the general problem is undecidable, but experience with Twelf shows that many practical examples are tractable with an incomplete solution. Owing to its importance to mechanized metatheory, I propose to study the problem of coverage checking for LFR type families.

I expect the issues to be similar to the issues in core LF coverage checking, but sometimes sorts provide more precise information than types alone. Recall the intrinsic encoding of the simply-typed lambda calculus from above, with its evaluation judgement,

$eval :: cmp\ A \rightarrow val\ A \rightarrow$ sort.
$ev{-}app :: eval\ (app\ E1\ E2)\ V$
$\quad \leftarrow eval\ E1\ (lam\ \lambda x.E1'\ x)$
$\quad \leftarrow eval\ E2\ V2$
$\quad \leftarrow eval\ (E1'\ V2)\ V$.

Under this signature, coverage checking should be able to determine that for every closed $e :: cmp\ A$ there exists a closed $v :: val\ A$ such that $eval\ e\ v$, or in other words, that the sort family $eval\ e\ v$ covers all inputs of sort $cmp\ A$ and all outputs of sort $val\ A$.

An affirmative answer from the proposed coverage checking algorithm would amount to an intrinsic progress theorem for the given evaluation judgement by guaranteeing that for every input expression, proof search for an evaluation derivation would never fail due to an unmatched case. By leveraging sort information, coverage checking can prove a property that normally requires an entire metatheorem; this is similar to how simply sort checking the *eval* declarations can "prove" the metatheorem that evaluation always returns a value.

Initial investigations suggest that coverage checking should be relatively straightforward given a "sorted unification" algorithm as proposed above. There is some indication though that choices made during type reconstruction could have a non-trivial effect. If for example, as discussed above, type reconstruction chose to give *E1'* the sort $cmp\ A \rightarrow cmp\ A$ instead of the more general $val\ A \rightarrow cmp\ A$, output coverage checking would fail on the second subgoal of rule *ev−app*. A careful study should reveal interesting details.

6 Additional Directions

In addition to the proposal core, there are a number of research avenues that would be interesting and useful, provided enough time remains after completion of the core components discussed above. In this section, I outline several such avenues, including a prototype implementation of LFR and explorations into termination and uniqueness checking for LFR logic programs, two key components of a usable metatheory-checking engine.

6.1 Implementation

It would be nice to have some sort of an implementation in order to play with the examples already given and those proposed. Such an implementation could fall anywhere in between a formalization of LFR and some of its metatheory in Twelf and a full-scale ML implementation of a Twelf-style metareasoning environment including elements discussed below, but more likely on the smaller end of the scale, like a proof-of-concept prototype of some of the algorithms proposed.

At the very least it would be worthwhile to formalize in Twelf the basic metatheory of some fragment of LFR, since arguments similar to those given in Section 3 have been reconstructed several times by several different people over the last few years but never actually formalized. A formal artifact would benefit future researchers of canonical forms-based logical frameworks. (Crary's formalization of the singleton metatheory an SML internal language [LCH07] would be a suitable starting point, since it is based on the canonical forms

methodology, but many of the complexities introduced by singletons could be ignored.)

6.2 Operational Semantics

Twelf is endowed with an operational semantics based on logic programming proof search, and this semantics is the main vehicle for establishing metatheoretic results about encoded object languages. It is unclear how the addition of refinements affects this operational semantics, beyond the changes required to unification already discussed above. But it seems worthwhile to investigate, since an operational semantics would be a key component in a realistic metalogical framework based on LFR.

6.3 Termination Checking

Logic programs in Twelf can be checked for termination, which ensures that any recursive calls occur at smaller arguments in a well-founded order. Coverage and termination checking together provide a means for determining when a relation encoded as a logic program is total, and thus represents a proof of an object language metatheorem. An account of termination checking for the LFR operational semantics would be another step towards a full account of metareasoning about LFR encodings of deductive systems.

6.4 Uniqueness Checking

Uniqueness checking is the problem of determining whether a relation encoded as a logic program is functional, i.e. that given its inputs, its outputs are unique, if they exist. A great deal of metatheoretic reasoning depends on knowing the functionality of certain relations, and current Twelf users are required to encode this reasoning directly in the form of tedious uniqueness and equality lemmas. For this reason, Anderson and Pfenning [AP04] undertook a study of uniqueness checking for Twelf.

It has been suggested that the addition of sort information might make uniqueness checking easier. It would therefore be worthwhile to study the interaction of sorts and uniqueness, in order to pave the way for future metareasoning engines based on LFR.

7 Related Work

7.1 Automath

One would be remiss to propose a thesis about LF without acknowledging its roots in the Automath tradition [dB94b]. A great many of the ideas and methodologies used for representing systems in LF owe to early work by de Bruijn. LF has been compared quite closely to the AUT-QE system in particular [NG94].

De Bruijn's treatise on the "Mathematical Vernacular" [dB94a] describes a system MV inspired by Automath but attempting to adhere more closely to accepted mathematical practice; similarly, one can see LFR as being (heavily) inspired by LF, but with an attempt to make a stronger connection to mathematical practice. MV distinguishes between *substantives*, the classes to which objects might belong, and *archetypes*, the largest possible substantive to which an object belongs. Its substantives are somewhat analogous to our *sorts*, and its archetypes to our *types*. In a similar manner to how MV's archetypes are "never mentioned in the language rules", our sorting rules nearly never mention the types that sorts refine: in effect, sorting can really be seen as a separate matter from typing.

7.2 Regular tree types and order-sorted logics for logic programming

Types naturally arise in logic programming and automated theorem proving as a way to curtail meaningless search. For example, given a clause $\forall n.\, nat(n) \to \cdots \to prime(n)$, one may end up searching for a proof of $nat(\text{Peter}) \to \cdots \to prime(\text{Peter})$ after instantiating n; even though this search will never succeed, since it is not the case that $nat(\text{Peter})$, it would be better to avoid such meaningless search in the first place. This observation leads to the introduction of *order-sorted logics*, from which we borrow the term "sort". The clause above might be rewritten as $\forall n : nat.\ \cdots \to prime(n)$, capturing the appropriate constraint statically.

One class of types that has proved particularly profitable is the *regular tree types* [DZ92] (see [YFS92] for example), so-called due to their connection with regular tree grammars. A key property that makes the regular tree types useful is the existence of computable intersections, and this property eventually led to the introduction of intersections in the context of refinement type systems for functional languages, described below.

Among order-sorted logics are systems with "term declarations" [SS89]; several people have studied the problem of unification in such systems [SA93, Koh94]. Term declarations have the form $\forall x_1{::}S_1 \ldots \forall x_n{::}S_n.\, M : S$, meaning that in any context extending $x_1{::}S_1, \ldots, x_n{::}S_n$ the term M can be judged to have sort S. For instance, one might declare

$$\forall x{::}even.\, succ\ x :: odd.$$
$$\forall x{::}odd.\, succ\ x :: even.$$

to achieve roughly the same effect as our $succ :: even \to odd \wedge odd \to even$. One problem with such systems is that they fail to give first-class status to the notion that a term can have multiple sorts, like our intersection sorts do. Furthermore, the typechecking problem for systems with term declarations is tricky at best, since it requires the use of higher-order matching, a problem whose status was until recently open and for which no practical implementation currently exists.

7.3 Intersection types

Intersection types were introduced by Coppo et al. [CDCV81] to describe a type theory in which types are preserved not only by reduction but also by convertibility, i.e. in which subject *expansion* holds in addition to subject *reduction*. Since β-conversion preserves types, they were able to precisely characterize the class of normalizing terms as those that have a non-trivial (essentially, non-$\top$) type.

Later, Reynolds used intersection types to simplify the design of the programming language Forsythe [Rey96], e.g. by representing n-ary records as n-ary intersections of single-field records. Although Reynolds's setting of imperative programming was vastly different from our world of logical frameworks, many of his core motivations were similar to ours: namely, intersections can be used to capture multiple properties of individual terms. Furthermore, despite his working under assumptions and constraints quite different from ours, one can see shades of many of his ideas reflected in our development.

7.4 Refinement types for functional languages

The utility of regular tree types in logic programming led Freeman to investigate them for functional languages [Fre94]. Freeman studied a system of refinement types for a fragment of ML based on ideas relating to regular trees, but intersection types were also a crucial addition for many of his examples. His focus was on maintaining decidable inference with minimal declarations, but ultimately the theory fell prey to algorithmic efficiency issues.

Later, Davies sought to tame the complexities and make refinement types practical for Standard ML [Dav05]. Davies's key decision was to abandon pure inference of unannotated programs in favor of a bidirectional type system and minimally annotated programs. The focus then was on minimizing the annotation burden, a task somewhat alleviated by the apparent positive benefits of annotations during program construction and their interpretation as machine-checked documentation. Along the way, Davies discovered a curious interaction of intersection types with effects quite similar to the interaction of polymorphism with effects; the solution was to impose a *value restriction* similar to the one familiar from ML-style let-polymorphism [DP00]. Additionally, the system had to be weakened by the removal of the $\wedge/\rightarrow$ distributivity subtyping rule, which could allow users to circumvent the value restriction.

Dunfield unified Davies work with a form of dependent typing inspired by Xi's Dependent ML [XP99], and extended the resulting system with the "indefinite" union and existential types [Dun07]. Dunfield abandoned the refinement restriction, studying a type system with arbitrary intersections and unions directly at the type level, but he maintained a bidirectional typing discipline. To properly account for the indefinite property types, he extended bidirectional checking with an evaluation order-directed "tridirectional rule" [DP04].

All of this work was in the context of functional programming, and thus quite different from our work in logical frameworks. Obviously, since LF has no side effects—indeed no reduction at all!—we have no need of Davies's

value restriction, and since we treat only negative types, we have no need of Dunfield's tridirectional rule. But there are still some similarities that help guide the present work, such as Freeman's and Davies's refinement restriction and Davies's and Dunfield's bidirectional typing.

Subtler differences arise from different assumptions about the world at typechecking time. The work on functional languages is all concerned with typing closed terms, and datatypes embody a closed-world assumption, both of which can be leveraged to reason about things like the emptiness of the intersection $\mathit{even} \wedge \mathit{odd}$. In our setting, though, we have an inherently open-ended signature and we work under non-empty contexts: it is impossible to show that $\mathit{even} \wedge \mathit{odd}$ is empty because one might always have an assumption of type $\mathit{even} \wedge \mathit{odd}$.[5] However, it is exactly this open-endedness, coupled with LF's very weak, purely representational function space, that allows us to show such strong theorems as the soundness and completeness of subtyping via η-expansion, a theorem one would not expect to hold generally in the presence of a large, computational function space.

7.5 Subtyping in dependent type theories

Pfenning described in a workshop paper a proposed extension of LF with refinement types [Pfe93]. The present work can be seen as a reconstruction, reformulation, and extension of his ideas, with a focus on canonical forms, decidability, and good proof-theoretic properties.

Aspinall and Compagnoni [AC01] studied a type theory $\lambda \mathrm{P}_{\leq}$ with both dependent types and subtyping, but they treated subtyping directly rather than introducing a refinement layer. Their chief difficulty was breaking the cycle that arises between subtyping, kinding, and typing in order to show decidability, which they did by splitting ordinary β-reduction into two levels, one that reduces terms and one that reduces types. In our setting, the restriction of attention to canonical forms obviates the need to consider β-reduction and its properties (e.g. subject reduction, Church-Rosser, etc.) at the cost of a more involved Substitution theorem, an arguably simpler development.

Aspinall [Asp00] also studied an unconventional system of subtyping with dependent types using "power types", a type-level analogue of power sets. Aspinall's system λ_{Power} has uniform "subtyping" at all levels since power "types" can in fact classify type families; although the system remains predicative, this generalization complicates the system's metatheory. Aspinall's use of a "rough typing" judgement in formulating the metatheory of λ_{Power} is somewhat related to our use of simple types in the metatheory of hereditary substitution and η-expansion.

Both Aspinall and Compagnoni's $\lambda \mathrm{P}_{\leq}$ and Aspinall's λ_{Power} are more general than LFR in a certain sense, since they allow subtyping declarations between

[5] One might imagine extending LFR with declarations of the form $\mathit{even} \wedge \mathit{odd} \leq \mathit{empty}$ to allow the user to capture this property explicitly. As currently specified, LFR does not give the user the ability to define any arbitrary subtyping lattice since it excludes such declarations.

atomic families whose arities and indices might be different. So far in the development of LFR, no examples have wanted for such declarations. The primary shortcoming of both $\lambda P_{\leq}$ and λ_{Power} is their lack of intersection types, which are essential for even the simplest of our examples.

Kopylov [Kop03] studied a dependent intersection $\bigwedge x{::}A.\, B$, a generalization of ordinary intersection $A \wedge B$ where the second type may depend on the element that has both types.[6] His motivation was finding a simple way to define dependent records in NuPRL in terms of only single-field records (following Reynolds's trick in the design of Forsythe [Rey96]). It is tempting to consider a dependent intersection sort $\bigwedge x{::}S{\sqsubset}A.\, T$ generalizing our $S \wedge T$, but it turns out not to fit in the refinement framework: the sorts S and T must both refine the same type A, but this precludes T from depending on x; in other words, a dependent intersection would always be degenerate.

7.6 Coercive subtyping in logical frameworks

A number of people have investigated "coercive subtyping" [Luo99] for logical frameworks, in which subtyping declarations induce certain identity coercions which are inserted implicitly at typechecking time. Coercive subtyping has been implemented in several logical frameworks and proof assistants, notably Coq [Saï97]. Coercive subtyping is undoubtedly related to the proposed coercion interpretation of LFR.

8 Plan

The material proposed in Section 5 follows a logical progression.

1. **Motivation:** Representation methodology
2. **Type theory:** Subset interpretation, coercion interpretation, contextual modal formulation
3. **Algorithms:** Unification
4. **Applications:** Type reconstruction, coverage checking

I therefore propose to proceed in the order presented, for the most part. Of course, a strict division between phases is unlikely, since there is some overlap, but the organization above gives a clear accounting for exactly what contribution each part of the work will make.

References

[AC01] David Aspinall and Adriana B. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1-2):273–309, 2001.

[6] Kopylov wrote $x{:}A \cap B$ and $A \cap B$.

[AP04] Penny Anderson and Frank Pfenning. Verifying uniqueness in a logical framework. In K.Slind, A.Bunker, and G.Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'04)*, pages 18–33, Park City, Utah, September 2004. Springer LNCS 3223.

[Asp00] David Aspinall. Subtyping with power types. In Peter Clote and Helmut Schwichtenberg, editors, *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2000.

[BTCGS91] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 27(2-6):45–58, 1981.

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002.

[Dav05] Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, May 2005. Available as Technical Report CMU-CS-05-110.

[dB94a] N. G. de Bruijn. The Mathematical Vernacular, a language for mathematics with typed sets. In Nederpelt et al. [NGdV94], pages 865–935.

[dB94b] N. G. de Bruijn. A survey of the project Automath. In Nederpelt et al. [NGdV94], pages 141–161.

[DP00] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proceedings of the Fifth International Conference on Functional Programming (ICFP'00)*, pages 198–208, Montreal, Canada, September 2000. ACM Press.

[DP04] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In Xavier Leroy, editor, *ACM Symp. Principles of Programming Languages (POPL '04)*, pages 281–292, Venice, Italy, January 2004.

[Dun07] Joshua Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, August 2007. Available as Technical Report CMU-CS-07-129.

[DZ92] Philip W. Dart and Justin Zobel. A regular type language for logic programs. In Pfenning [Pfe92], pages 157–187.

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, March 1994. Available as Technical Report CMU-CS-94-110.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HL07] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

[HP05] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *Transactions on Computational Logic*, 6:61–101, January 2005.

[Koh94] Michael Kohlhase. Unification in a sorted lambda-calculus with term declarations and function sorts. In Bernhard Nebel and Leonie S. Dreschler-Fischer, editors, *Proceedings of the 18th Annual German Conference on Artificial Intelligence (KI-94)*, volume 861 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 1994.

[Kop03] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS '03)*, pages 86–95. IEEE Computer Society, 2003.

[KP93] Michael Kohlhase and Frank Pfenning. Unification in a λ-calculus with intersection types. In Dale Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 488–505, Vancouver, Canada, October 1993. MIT Press.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In Matthias Felleisen, editor, *Proceedings of the 34th Annual Symposium on Principles of Programming Languages (POPL '07)*, pages 173–184, Nice, France, January 2007. ACM Press.

[LP07] William Lovas and Frank Pfenning. A bidirectional refinement type system for LF. Technical Report CMU-CS-07-127, Department of Computer Science, Carnegie Mellon University, 2007. In preparation.

[Luo99] Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

[NG94] R. P. Nederpelt and J. H. Geuvers. Twenty-five years of Automath research. In Nederpelt et al. [NGdV94], pages 3–54.

[NGdV94] R. D. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected Papers on Automath*. Number 133 in Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1994.

[NPP07] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007. To appear.

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[Pfe01] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Rey89] John C. Reynolds. Even normal forms can be hard to type. Unpublished, marked Carnegie Mellon University, December 1, 1989.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Berlin, 1991. Springer-Verlag.

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 28, 1996.

[SA93] Rolf Socher-Ambrosius. Unification in order-sorted logic with term declarations. In *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning (LPAR '93)*, volume 698 of *Lecture Notes in Computer Science*, pages 301–308. Springer, 1993.

[Saï97] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL '97)*, pages 292–301. ACM Press, 1997.

[SP03] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.

[SS89] Manfred Schmidt-Schauß. *Computational Aspects of an Order-sorted Logic with Term Declarations*. Number 395 in Lecture Notes in Computer Science. Springer, 1989.

[WCPW02] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[WCPW04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, pages 355–377. Springer-Verlag LNCS 3085, 2004. Revised selected papers from the *Third International Workshop on Types for Proofs and Programs*, Torino, Italy, April 2003.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.

[YFS92] Eyal Yardeni, Thom Frühwirth, and Ehud Shapiro. Polymorphically typed logic programs. In Pfenning [Pfe92], pages 63–90.