A Bidirectional Refinement Type System for LF

William Lovas[1] Frank Pfenning[2]

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

We present a system of refinement types for LF in the style of recent formulations where only canonical forms are well-typed. Both the usual LF rules and the rules for type refinements are bidirectional, leading to a straightforward proof of decidability of type-checking even in the presence of intersection types. Because we insist on canonical forms, structural rules for subtyping can now be derived rather than being assumed as primitive. We illustrate the expressive power of our system with several examples in the domain of logics and programming languages.

Keywords: LF, refinement types, subtyping, dependent types, intersection types

1 Introduction

LF was created as a framework for defining logics [6]. Since its inception, it has been used to formalize reasoning about a number of deductive systems (see [13] for an introduction). In its most recent incarnation as the Twelf metalogic [14], it has been used to encode and mechanize the metatheory of programming languages that are prohibitively complex to reason about on paper [3,9].

It has long been recognized that some LF encodings would benefit from the addition of a subtyping mechanism to LF [12,2]. In LF encodings, judgements are represented by type families, and many subsyntactic relations and judgemental inclusions can be elegantly represented via subtyping.

Prior work has explored adding subtyping and intersection types to LF via *refinement types* [12]. Many of that system's metatheoretic properties were proven indirectly by translation into other systems, though, giving little insight into a notion of adequacy or an implementation strategy. We present here a refinement type system for LF based on the modern *canonical forms* approach, and by doing so we obtain direct proofs of important properties like decidability.

[1] Email: wlovas@cs.cmu.edu
[2] Email: fp@cs.cmu.edu

In canonical forms-based LF, only β-normal η-long terms are well-typed — the syntax restricts terms to being β-normal, while the typing relation forces them to be η-long. Since standard substitution might introduce redexes even when substituting a normal term into a normal term, it is replaced with a notion of *hereditary substitution* that contracts redexes along the way, yielding another normal term. Since only canonical forms are admitted, type equality is just α-equivalence, and typechecking is manifestly decidable.

Canonical forms are exactly the terms one cares about when adequately encoding a language in LF, so this approach loses no expressivity. Since all terms are normal, there is no notion of reduction, and thus the metatheory need not directly treat properties related to reduction, such as subject reduction, Church-Rosser, or strong normalization. All of the metatheoretic arguments become straightforward structural inductions, once the theorems are stated properly.

By introducing a layer of refinements distinct from the usual layer of types, we prevent subtyping from interfering with our extension's metatheory. We also follow the general philosophy of prior work on refinement types [5,4] in only assigning refined types to terms already well-typed in pure LF, ensuring that our extension is conservative.

In the remainder of the paper, we describe our refinement type system alongside several illustrative examples (Section 2). Then we explore its metatheory and give proof sketches of important results, including decidability (Section 3). We note that our approach leads to subtyping only being defined on atomic types, but we show that subtyping at higher types is already present in our system by proving that the usual declarative rules are sound and complete with respect to an intrinsic notion of subtyping (Section 4). Finally, we discuss some related work (Section 5) and summarize our results (Section 6).

2 System and Examples

We present our system of LF with Refinements, LFR, through several examples. In what follows, R refers to atomic terms and N to normal terms. 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 normal 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-formed 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.

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

Sorts are related to types by a refinement relation, $S \sqsubset A$ ("S refines A"), discussed below. A term of type A can be assigned a sort S only when $S \sqsubset A$. 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

$$\mathit{nat} : \mathsf{type}. \qquad \mathit{zero} : \mathit{nat}. \qquad \mathit{succ} : \mathit{nat} \to \mathit{nat}.$$

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

$$\mathit{even} \sqsubset \mathit{nat}. \qquad \mathit{odd} \sqsubset \mathit{nat}.$$

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:

$$\mathit{zero} :: \mathit{even}.$$

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 *intersection sorts*, written as $S_1 \wedge S_2$.[3]

$$\mathit{succ} :: \mathit{even} \to \mathit{odd} \;\wedge\; \mathit{odd} \to \mathit{even}.$$

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.[4]

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

[3] Intersection has lower precedence than arrow.

[4] 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.

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)**

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

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

$\mathit{pos} \sqsubset \mathit{nat}. \qquad \mathit{succ} :: \cdots \wedge \top \to \mathit{pos}.$

Since we only sort-check well-typed programs and *succ* is declared to have type $\mathit{nat} \to \mathit{nat}$, 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*.

$\mathit{odd} \leq \mathit{pos}.$

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

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.

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.

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

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.

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

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

Note that although LF type systhesis is unique, 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.

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

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

$$\frac{\vdots}{\vdash s\ z \Leftarrow o} \qquad \frac{\vdots}{\vdash s\ z \Leftarrow p}$$
$$\vdash s\ z \Leftarrow o \wedge p$$

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 *double* in LF.

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

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.

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

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.

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

After these declarations, it would be a *sort error* to pose a 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.

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 [16], even for normal terms as we have here [15]. Using techniques such as focusing, we believe that for practical cases they can be analyzed efficiently for the purpose of sort checking. [5]

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
➪ : tp → tp → tp.   % object language function space
%infix right 10 ➪ .
```

[5] The present paper concentrates primarily on decidability, though, not efficiency.

$exp : tp \to \mathsf{type}.$ % 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

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

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*. We posit a front end that can recover this information from the refinement declarations for *val* and *cmp*, avoiding redundancy.

The most interesting declaration is the one for the constant *lam*. The argument type $(val\ A \to cmp\ B)$ 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*.

$eval :: cmp\ A \to val\ A \to \mathsf{type}.$
$ev\text{-}lam :: eval\ (lam\ \lambda x.\, E\ x)\ (lam\ \lambda x.\, E\ x).$
$ev\text{-}app :: \Pi E_1' {::} (val\ A \to cmp\ A).$
$\quad eval\ (app\ E_1\ E_2)\ V$
$\quad \leftarrow eval\ E_1\ (lam\ \lambda x.\, E_1'\ x)$
$\quad \leftarrow eval\ E_2\ V_2$
$\quad \leftarrow eval\ (E_1'\ V_2)\ V.$

Sort checking the above declarations demonstrates that evaluation always returns a value. Moreover, due to the explicit sort given for E_1', 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 recover this restriction automatically—if the front end were to assign E_1' the "more precise" sort $cmp\ A \to cmp\ A$, then the check would be lost.

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 $type : type$.

$term : \mathsf{type}.$ % terms at all levels

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

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

$of :: knd \to hyp \to \mathsf{type} \wedge$
$\quad fam \to knd \to \mathsf{type} \wedge$
$\quad obj \to fam \to \mathsf{type}.$

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

$of\text{-}pi :: of\ (pi\ T_1\ \lambda x.\, T_2\ x)\ tp$
$\quad \leftarrow of\ T_1\ tp$
$\quad \leftarrow (\Pi x{::}term.\, of\ x\ T_1 \to of\ (T_2\ x)\ tp).$
$of\text{-}lm :: of\ (lm\ U_1\ \lambda x.\, T_2\ x)\ (pi\ U_1\ \lambda x.\, U_2\ x)$
$\quad \leftarrow of\ U_1\ tp$
$\quad \leftarrow (\Pi x{::}term.\, of\ x\ U_1 \to of\ (T_2\ x)\ (U_2\ x)).$
$of\text{-}ap :: of\ (ap\ T_1\ T_2)\ (U_1\ T_2)$
$\quad \leftarrow of\ T_1\ (pi\ U_2\ \lambda x.\, U_1\ x)$
$\quad \leftarrow of\ T_2\ U_2.$

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.

3 Metatheory

In this section, we present some metatheoretic results about our framework. These follow a similar pattern as previous work using hereditary substitutions [17,11,7]. To conserve space, we omit proofs that are similar to those from prior work, and

only sketch novel results. We refer the interested reader to the companion technical report [10], which contains complete proofs of all thoerems.

3.1 *Hereditary Substitution*

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 only ones of interest.

First, we note that 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 $(A)^-$.

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

We write $[N/x]^{\mathrm{n}}_A\, M = M'$ as short-hand for $[N/x]^{\mathrm{n}}_{(A)^-}\, M = M'$.

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. The former is just defined compositionally; the latter is defined by two rules:

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

The rule **rn-var** just returns the substitutend N_0 and its putative type index α_0. The rule **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.

Lemma 3.1

(i) *If* $[N/x]^{\mathrm{rr}}_A\, R = R'$, *then the head of* R *is not* x.

(ii) *If* $[N/x]^{\mathrm{rn}}_A\, R = (N', \alpha')$, *then the head of* R *is* x.

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.

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

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.

A hallmark of the canonical forms/hereditary substitution approach is that it allows a decidability proof to be carried out comparitively 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. (See [8] for a typical non-canonical account of LF definitional equality.)

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.

Lemma 3.2 *If* $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$, *then* α' *is a subterm of* α_0.

Proof. By induction on the derivation of $[N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$. In rule **rn-var**, α' is the same as α_0. In rule **rn-β**, our inductive hypothesis tells us that $\alpha_2 \to \alpha_1$ is a subterm of α_0, so α_1 is as well. □

Theorem 3.3 (Decidability of Substitution) *Hereditary substitution is decidable. In particular:*

(i) *Given* N_0, x_0, α_0, *and* R, *either* $\exists R'.\, [N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$, *or* $\nexists R'.\, [N_0/x_0]^{\mathrm{rr}}_{\alpha_0} R = R'$,

(ii) *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* $\nexists (N', \alpha').\, [N_0/x_0]^{\mathrm{rn}}_{\alpha_0} R = (N', \alpha')$,

(iii) *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

Proof. By lexicographic induction on the type subscript α_0, the main subject of the substitution judgement, and the clause number. For each rule defining hereditary substitution, the premises are at a smaller type subscript, or if the same type subscript, then a smaller term, or if the same term, then an earlier clause. The case for rule **rn-β** relies on Lemma 3.2 to know that α_2 is a strict subterm of α_0. □

Theorem 3.4 (Decidability of Subsorting) *Given* Q_1 *and* Q_2, *it is decidable whether or not* $Q_1 \leq Q_2$.

Proof. Since the subsorting relation $Q_1 \leq Q_2$ is just the reflexive, transitive closure of the declared subsorting relation $s_1 \leq s_2$, it suffices to compute this closure and check whether the heads of Q_1 and Q_2 are related by it. □

We prove decidability of typing by exhibiting a deterministic algorithmic system that is equivalent to the original. Instead of synthesizing a single sort for an atomic term, the algorithmic system synthesizes an intersection-free list of sorts, Δ.

$$\Delta ::= \cdot \mid \Delta, Q \mid \Delta, \Pi x{::}S{\sqsubset}A.\, T$$

One can think of Δ as the intersection of all its elements. Instead of applying intersection eliminations, the algorithmic system eagerly breaks down intersections

using a "split" operator, leading to a deterministic "minimal-synthesis" system.

$$\begin{array}{cc} \mathrm{split}(Q) = Q & \mathrm{split}(S_1 \wedge S_2) = \mathrm{split}(S_1), \mathrm{split}(S_2) \\ \mathrm{split}(\Pi x{::}S \sqsubset A.\, T) = \Pi x{::}S \sqsubset A.\, T & \mathrm{split}(\top) = \cdot \end{array}$$

$$\frac{c{::}S \in \Sigma \qquad c{:}A \in \Sigma}{\Gamma \vdash c \Rrightarrow \mathrm{split}(S)} \qquad \frac{x{::}S \sqsubset A \in \Gamma}{\Gamma \vdash x \Rrightarrow \mathrm{split}(S)} \qquad \frac{\Gamma \vdash R \Rrightarrow \Delta \qquad \Gamma \vdash \Delta \mathbin{@} N = \Delta'}{\Gamma \vdash R\, N \Rrightarrow \Delta'}$$

The rule for applications uses an auxiliary judgement $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ which computes the possible types of $R\, N$ given that R synthesizes to all the sorts in Δ. It has two key rules:

$$\frac{}{\Gamma \vdash \cdot \mathbin{@} N = \cdot} \qquad \frac{\Gamma \vdash \Delta \mathbin{@} N = \Delta' \qquad \Gamma \vdash N \Lleftarrow S \qquad [N/x]^{\mathrm{s}}_{A}\, T = T'}{\Gamma \vdash (\Delta, \Pi x{::}S \sqsubset A.\, T) \mathbin{@} N = \Delta', \mathrm{split}(T')}$$

The other rules force the judgement to be defined when neither of the above two rules apply. Finally, to tie everything together, we define a new checking judgement $\Gamma \vdash N \Lleftarrow S$ that makes use of the algorithmic synthesis judgement; it looks just like $\Gamma \vdash N \Leftarrow S$ except for the rule for atomic terms.

$$\frac{\Gamma \vdash R \Rrightarrow \Delta \qquad Q' \in \Delta \qquad Q' \leq Q}{\Gamma \vdash R \Lleftarrow Q}$$

This new algorithmic system is manifestly decidable.

Theorem 3.5 *Algorithmic type checking is decidable. In particular:*

(i) *Given* Γ *and* R*, there is a unique* Δ *such that* $\Gamma \vdash R \Rrightarrow \Delta$*.*

(ii) *Given* Γ*,* N*, and* S*, it is decidable whether or not* $\Gamma \vdash N \Lleftarrow S$*.*

(iii) *Given* Γ*,* Δ*, and* N*, there is a unique* Δ' *such that* $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$*.*

Proof. By induction on the term, R or N, the clause number, and the sort S or the list of sorts Δ. For each rule, the premises are either known to be decidable, or at a smaller term, or if the same term, then an earlier clause, or if the same clause, then either a smaller S or a smaller Δ. □

Note that the algorithmic synthesis system *always* outputs *some* Δ; if the given term has no sort, then the output will be $\cdot$.

It is straightforward to show that the algorithm is sound and complete with respect to the original bidirectional system.

Theorem 3.6 (Soundness of Algorithmic Typing)

(i) *If* $\Gamma \vdash R \Rrightarrow \Delta$*, then for all* $S \in \Delta$*,* $\Gamma \vdash R \Rightarrow S$*.*

(ii) *If* $\Gamma \vdash N \Lleftarrow S$*, then* $\Gamma \vdash N \Leftarrow S$*.*

(iii) *If* $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$*, and for all* $S \in \Delta$*,* $\Gamma \vdash R \Rightarrow S$*, then for all* $S' \in \Delta'$*,* $\Gamma \vdash R\, N \Rightarrow S'$*.*

Proof. By straightforward induction on the given derivation. □

Lemma 3.7 *If* $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ *and* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\Pi x{::}S \sqsubset A.\, T \in \Delta$ *and* $\Gamma \vdash N \Lleftarrow S$ *and* $[N/x]^{\mathrm{s}}_{A}\, T = T'$, *then* $\mathrm{split}(T') \subseteq \Delta'$.

Proof. By straightforward induction on the derivation of $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$. □

Theorem 3.8 (Completeness for Algorithmic Typing)

(i) *If* $\Gamma \vdash R \Rightarrow S$, *then* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathrm{split}(S) \subseteq \Delta$.

(ii) *If* $\Gamma \vdash N \Leftarrow S$, *then* $\Gamma \vdash N \Lleftarrow S$.

Proof. By straightforward induction on the given derivation. In the application case, we make use of the fact that $\Gamma \vdash \Delta \mathbin{@} N = \Delta'$ is always defined and apply Lemma 3.7. □

Decidability theorems and proofs for other syntactic categories' formation judgements are similar, so we omit them.

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.

Theorem 3.9 (Substitution) *If* $\Gamma_{\mathrm{L}} \vdash N_0 \Leftarrow S_0$, *and* $\vdash \Gamma_{\mathrm{L}}, x_0{::}S_0 \sqsubset A_0, \Gamma_{\mathrm{R}}\ \mathsf{ctx}$, *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* $\Gamma_{\mathrm{L}}, \Gamma'_{\mathrm{R}} \vdash N' \Leftarrow S'$, *and similarly for other syntactic categories.*

Proof. The staging of the substitution theorem is somewhat intricate. First, we strengthen its statement to one that does not presuppose the well-formedness of the context or the classifying types, but instead presupposes that substitution is defined on them. This strengthened statement may be proven by induction on $(A_0)^-$ and the derivations being substituted into. In the application case, we require a lemma about how hereditary substitutions compose, analogous to the fact that for ordinary substitution, $[N_0/x_0]\,[N_2/x_2]\,N = [[N_0/x_0]\,N_2/x_2]\,[N_0/x_0]\,N$. □

A more in-depth discussion of the proof of substitution for core canonical LF can be found in [7]. The story for LFR is quite similar, and is detailed in the companion technical report [10].

Theorem 3.10 (Expansion) *If* $\Gamma \vdash S \sqsubset A$ *and* $\Gamma \vdash R \Rightarrow S$, *then* $\Gamma \vdash \eta_A(R) \Leftarrow S$.

Proof. By induction on S. The $\Pi x{:}A_2.\, A_1$ case relies on the auxiliary fact that $[\eta_{A_2}(x)/x]^{\mathrm{s}}_{A_2}\, S_1 = S_1$. □

Corollary 3.11 (Identity) *If* $\Gamma \vdash S \sqsubset A$, *then* $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Leftarrow S$.

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 only typing canonical forms, this rule is limited to atomic sorts Q, and consequently, subsorting need only be defined on atomic sorts.

As it turns out, though, the usual variance principles and structural rules for subsorting at higher sorts are admissible with respect to an intrinsic notion of higher-sort subsorting. The simplest way of formulating this intrinsic notion is as a variant of the identity principle: S is a subtype 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) *The following are equivalent:*

(i) *If* $\Gamma \vdash R \Rightarrow S_1$, *then* $\Gamma \vdash \eta_A(R) \Leftarrow S_2$.

(ii) $\Gamma, x{::}S_1 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$.

(iii) *If* $\Gamma \vdash N \Leftarrow S_1$, *then* $\Gamma \vdash N \Leftarrow S_2$.

(iv) *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* $\Gamma_{\mathrm{L}}, [N_1/x]^{\gamma}_A \Gamma_{\mathrm{R}} \vdash [N_1/x]^{\mathrm{n}}_A N \Leftarrow [N_1/x]^{\mathrm{s}}_A S$.

Proof. Using Identity and Substitution, and the fact that $[N/x]^{\mathrm{n}}_A \eta_A(x) = N$.

i $\Longrightarrow$ *ii*: By rule, $\Gamma, x{::}S_1 \sqsubset A \vdash x \Rightarrow S_1$. By i, $\Gamma, x{::}S_1 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$.

ii $\Longrightarrow$ *iii*: Suppose $\Gamma \vdash N \Leftarrow S_1$. By ii, $\Gamma, x{::}S_1 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$. By Theorem 3.9 (Substitution), $\Gamma \vdash [N/x]^{\mathrm{n}}_A \eta_A(x) \Leftarrow S_2$. Thus, $\Gamma \vdash N \Leftarrow S_2$.

iii $\Longrightarrow$ *iv*: Suppose $\Gamma_{\mathrm{L}}, x{::}S_2 \sqsubset A, \Gamma_{\mathrm{R}} \vdash N \Leftarrow S$ and $\Gamma_{\mathrm{L}} \vdash N_1 \Leftarrow S_1$. By iii, $\Gamma_{\mathrm{L}} \vdash N_1 \Leftarrow S_2$. By Theorem 3.9 (Substitution), $\Gamma_{\mathrm{L}}, [N_1/x]^{\gamma}_A \Gamma_{\mathrm{R}} \vdash [N_1/x]^{\mathrm{n}}_A N \Leftarrow [N_1/x]^{\mathrm{s}}_A S$.

iv $\Longrightarrow$ *i*: Suppose $\Gamma \vdash R \Rightarrow S_1$. By Theorem 3.10 (Expansion), $\Gamma \vdash \eta_A(R) \Leftarrow S_1$. By Corollary 3.11 (Identity), $\Gamma, x{::}S_2 \sqsubset A \vdash \eta_A(x) \Leftarrow S_2$. By iv, $\Gamma \vdash [\eta_A(R)/x]^{\mathrm{n}}_A \eta_A(x) \Leftarrow S_2$. Thus, $\Gamma \vdash \eta_A(R) \Leftarrow S_2$. □

All of the rules in Fig. 1 are sound with respect to this intrinsic notion of subsorting.

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

Proof. By induction, making use of the alternate formulations given by Theorem 4.1. □

The soundness of the rules in Fig. 1 demonstrates that any subsumption relationship you might want to capture with them is already captured by our checking and synthesis rules. More interesting is the fact that the usual rules are *complete* with respect to our intrinsic notion. Space limitations preclude more than a brief overview here; the companion technical report contains a detailed account.

We demonstrate completeness by appeal to an algorithmic subtyping system very similar to the algorithmic typing system from Section 3.2. This system is characterized by two judgements: $\Delta \leq S$ and $\Delta \mathbin{@} (N :: \Delta_1) = \Delta_2$. With the

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

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

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

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

Fig. 1. Derived structural rules for subsorting.

appropriate definition, we can prove the following by induction on the type A and the derivation $\mathcal{E}$.

Theorem 4.3 *Suppose* $\Gamma \vdash R \Rightarrow A$. *Then:*

(i) *If* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathcal{E} :: \Gamma \vdash \eta_A(R) \Lleftarrow S$, *then* $\Delta \leq S$.

(ii) *If* $\Gamma \vdash R \Rrightarrow \Delta$ *and* $\mathcal{E} :: \Gamma \vdash \Delta_0 \mathbin{@} \eta_A(R) = \Delta'$, *then* $\Delta_0 \mathbin{@} (\eta_A(R) :: \Delta) = \Delta'$.

From this and Theorem 3.8 we obtain a completeness theorem:

Theorem 4.4 *If* $\Gamma, x{::}S \sqsubset A \vdash \eta_A(x) \Lleftarrow T$, *then* $\mathrm{split}(S) \leq T$.

Finally, we can complete the triangle by showing that the algorithmic formulation of subtyping implies the original declarative formulation:

Theorem 4.5 *If* $\mathrm{split}(S) \leq T$, *then* $S \leq T$.

5 Related Work

The most closely related work is [12], which also sought to extend LF with refinement types. We improve upon that work by intrinsically supporting a notion of canonical form. Also closely related in Aspinall and Compagnoni's work on subtyping and dependent types [2,1]. The primary shortcoming of their work is its lack of intersection types, which are essential for even the simplest of our examples.

6 Summary

In summary, we have exhibited a variant of the logical framework LF with a notion of subtyping based on refinement types. We have demonstrated the expressive power of this extension through a number of realistic examples, and we have shown several metatheoretic properties critical to its utility as a logical framework, including decidability of typechecking.

Our development was drastically simplified by the decision to admit only canonical forms. One effect of this choice was that subsorting was only required to

be judgementally defined at base sorts; higher-sort subsorting was derived through an η-expansion-based definition which we showed sound and complete with respect to the usual structural subsorting rules.

There are a number of avenues of future exploration. For one, it is unclear how subsorting and intersection sorts will interact with the typical features of a metalogical framework, including type reconstruction, unification, and proof search, to name a few; these questions will have to be answered before refinement types can be integrated into a practical implementation. It is also worthwhile to consider adapting the refinement system to more expressive frameworks, like the Linear Logical Framework on the Concurrent Logical Framework.

References

[1] 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.

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

[3] Karl Crary. Toward a foundational typed assembly language. In G. Morrisett, editor, *Proceedings of the 30th Annual Symposium on Principles of Programming Languages (POPL '03)*, pages 198–212, New Orleans, Louisiana, January 2003. ACM Press.

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

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

[6] 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.

[7] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 2007. To appear. Available from `http://www.cs.cmu.edu/~drl/`.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

[14] 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.

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

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

[17] 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.