Refinement Types as Proof Irrelevance

William Lovas and Frank Pfenning

Carnegie Mellon University
Pittsburgh, PA 15213, USA
wlovas@cs.cmu.edu, fp@cs.cmu.edu

Abstract. Refinement types sharpen systems of simple and dependent types by offering expressive means to more precisely classify well-typed terms. Proof irrelevance provides a mechanism for selectively hiding the identities of terms in type theories. In this paper, we show that refinement types can be interpreted as predicates using proof irrelevance in the context of the logical framework LF, establishing a uniform relationship between two previously studied concepts in type theory. The interpretation and its correctness proof are surprisingly complex, lending credence to the idea that refinement types are a fundamental construct rather than just a convenient surface syntax for certain uses of proof irrelevance.

1 Introduction

Refinement type systems seek to extend type theories with more expressive means of classifying terms. Refinements typically take the form of an added layer of *sorts* above the usual layer of types: types express very basic well-formedness criteria while sorts specify precise properties of terms using technology like subsorting and intersection sorts. Refinement types have been profitably employed in functional languages like ML [7, 4], and they are a topic of much recent and ongoing research [6, 5, 17].

In recent work [10], we developed a system of refinement types for the logical framework LF [8]. An essential guiding principle was to restrict attention to canonical forms using bidirectional typing [16]. Under the canonical forms methodology, features which typically complicate a type system's metatheory could be expressed cleanly and simply. For example, treating intersection introduction as a *checking* rule and intersection elimination as a *synthesis* rule avoided any issues relating to intersection type inference, and restricting typing to canonical forms led to subtyping needing to be defined only at base type.

A simple example of refinement types in LF is the natural numbers with refinements standing for even and odd numbers:

$nat : \mathsf{type}.$
$z : nat.$
$s : nat \to nat.$

$even \sqsubset nat. \quad odd \sqsubset nat.$
$z :: even.$
$s :: even \to odd \ \wedge \ odd \to even.$

In the above, $even \sqsubset nat$ declares *even* as a refinement of the type *nat*, and the declarations using "::" give more precise sorts for the constructors z and s. Note that since the successor function satisfies two unrelated properties, we give two refinements for it using an intersection sort.

In this paper, we exhibit an interpretation of LF refinement types which we refer to as the "subset interpretation", since a sort refining a type is interpreted as a predicate embodying the refinement, and the set of terms having that sort is simply the subset of terms of the refined type that also satisfy the predicate. For example, under the subset interpretation, we translate the refinements *even* and *odd* to predicates on natural numbers, or one-place judgments following the LF judgments-as-types principle [8]. The refinement declarations for z and s turn into constructors for proofs of these predicates.

$even : nat \to \mathsf{type}$. $odd : nat \to \mathsf{type}$.
$\widehat{z}$: $even\ z$.
$\widehat{s}_1 : \Pi x{:}nat.\ even\ x \to odd\ (s\ x)$.
$\widehat{s}_2 : \Pi x{:}nat.\ odd\ x \to even\ (s\ x)$.

The successor function's two unrelated sorts translate to proof constructors for two different predicates.

We show that our interpretation is correct by proving, for instance, that a term N has sort S if and only if its translation $\widehat{N}$ has type $\widehat{S}(N)$, where $\widehat{S}(-)$ is the translation of the sort S into a type family representing a predicate; thus, an adequate encoding using refinement types remains adequate after translation. The chief complication in proving correctness is the dependency of types on terms, which forces us to deal with a *coherence* problem [2, 13].

Normally, subset interpretations are not subject to the issue of coherence—that is, of ensuring that the interpretation of a judgment is independent of its derivation—since the terms in the target of the translation are *the same* as the terms in the source, just with the stipulation that a certain property hold of them. The proofs of these properties are computationally immaterial, so they may simply be ignored at runtime. But the presence of full dependent types in LF means that the interpretation of a sort might depend on these proofs, and in turn the derivability of certain typing judgments in the target might depend on the identities of these proofs. Enter proof irrelevance: our primary tool for coping with coherence.

Proof irrelevance is a technique used in type theories to selectively hide the identities of certain terms representing proofs of propositions [11, 1]. One typical use of proof irrelevance is to render the typechecking of subset types [3, 14] decidable. A subset type $\{x{:}A \mid B(x)\}$ represents the set of terms of type A which also satisfy B; typechecking is undecidable because to determine if a term M has this type, you must search for a proof of $B(M)$. One might attempt to recover decidability by using a dependent sum $\Sigma x{:}A.\, B(x)$, representing the set of terms M of type A paired with proofs of $B(M)$; typechecking is decidable, since a proof of $B(M)$ is provided, but equality of terms is overly fine-grained: if there are two proofs of $B(M)$, the two pairs will be considered unequal. Using proof irrelevance, one can find a middle ground with the type $\Sigma x{:}A.\, [B(x)]$,

where $[-]$ represents the proof irrelevance modality. Type checking is decidable for such terms, since a proof of the property B is always given, but the identity of that proof is ignored, so all pairs with the same first component will be considered equal.

Our situation with the subset interpretation is similar: we would like to represent proofs of sort-checking judgments without depending on the identities of those proofs. By carefully using proof irrelevance to hide the identities of sort-checking proofs, we are able to show our translation sound and complete, preserving the adequacy of representations.

We begin the remainder of the paper by extending our example above to demonstrate the coherence issues that arise in the subset interpretation (Section 2). After that, we review the formal treatment of refinement types (Section 3) and proof irrelevance (Section 4) in the context of the logical framework LF, and then we discuss our translation and its correctness criteria in detail (Section 5). We conclude by highlighting some broader implications (Section 6).

2 Extended (Counter-)Example

Coherence arises in the subset interpretation due to the presence of dependent types. To show what can go wrong, we extend our example from the introduction to make use of dependency. Our uniform translation in Section 5 will be different in some details, but the essential ideas are unchanged.

Consider a judgment *double* which relates any natural number to its doubling.

$double : nat \to nat \to \mathsf{type}.$
$dbl/z : double\ z\ z.$
$dbl/s : \Pi N{:}nat.\ \Pi N2{:}nat.\ double\ N\ N2 \to double\ (s\ N)\ (s\ (s\ N2)).$

Using *refinement kinds*, or *classes*, we can express the property that the second subject of any doubling relation is always even, no matter what properties hold of the first subject. We do so by defining a sort *double** which is isomorphic to *double*, but has a more precise class.

$double^* \sqsubset double :: \top \to even \to \mathsf{sort}.$
$dbl/z :: double^*\ z\ z.$
$dbl/s :: \Pi N{::}\top.\ \Pi N2{::}even.\ double^*\ N\ N2 \to double^*\ (s\ N)\ (s\ (s\ N2)).$

The sort $\top$ represents a natural number with no special properties. Successfully sort-checking the declarations for dbl/z and dbl/s demonstrates that whenever $double^*\ M\ N$ is inhabited, the second argument, N, is even.

There is a crucial difference between refinements like *even* or *odd* and refinements like *double**: while *even* and *odd* denote particular subsets of the natural numbers, the inhabitants of the refinement $double^*\ x\ y$ are identical to those of the ordinary type $double\ x\ y$. What is important is not whether a particular instance $double^*\ x\ y$ is inhabited, but rather whether it is *well-formed at all*.

For this reason, we separate the *formation* of a dependent refinement type family from its *inhabitation*. Following this idea, the refinement *double** translates as follows:

$\widehat{double}* : nat \to nat \to \mathsf{type}.$
$\widehat{double}*/i : \Pi x{:}nat.\ \Pi y{:}nat.\ even\ y \to \widehat{double}*\ x\ y.$
$double* : \Pi x{:}nat.\ \Pi y{:}nat.\ \widehat{double}*\ x\ y \div\!\!\to double\ x\ y \to \mathsf{type}.$

There are three declarations after translation:

- a *formation family*, $\widehat{double}*$, which is inhabited exactly when a particular instance of $double*$ is well-formed (e.g. $\widehat{double}*\ z\ z$ will be inhabited, since $double*\ z\ z$ is well-formed),
- a *constructor* for the formation family, $\widehat{double}*/i$, which builds such proofs of well-formedness (e.g. $\widehat{double}*/i\ z\ z\ \widehat{z}$ will be a proof that $double*\ z\ z$ is well-formed), and
- a *predicate family*, $double*$, which for any x and y, will be inhabited by proofs that a given derivation of $double\ x\ y$ has refinement $double*\ x\ y$, provided that $double*\ x\ y$ is well-formed.

In the predicate family $double*$, the proof of well-formedness is made irrelevant using a proof-irrelevant function space $A \div\!\!\to B$, representing functions from A to B that are insensitive to the identity of their argument. Using irrelevance ensures that a given sort has a unique translation, up to equivalence. We elaborate on this below.

The final component, the predicate family $double*$, is populated by constants generated from the refinement declarations. We write arguments in irrelevant position in [*square brackets*].

$\widehat{dbl/z} : double*\ z\ z$
$\quad [\ \widehat{double}*/i\ z\ z\ \widehat{z}\ ]$
$\quad dbl/z.$
$\widehat{dbl/s} : \Pi N{:}nat.\ \Pi N2{:}nat.\ \Pi \widehat{N2}{:}even\ N2.\ \Pi D{:}double\ N\ N2.$
$\quad double*\ N\ N2\ [\ \widehat{double}*/i\ N\ N2\ \widehat{N2}\ ]\ D$
$\to double*\ (s\ N)\ (s\ (s\ N2))$
$\quad [\ \widehat{double}*/i\ (s\ N)\ (s\ (s\ N2))\ (\widehat{s}_2\ (s\ N2)\ (\widehat{s}_1\ N2\ \widehat{N2}))\ ]$
$\quad (dbl/s\ N\ N2\ D).$

As is evident even from this short and abbreviated example, the interpretation leads to a significant blowup in the size and complexity of a signature, underscoring the importance of a primitive understanding of refinement types.

Note that the formation argument of $double*$ above is always made irrelevant as stipulated by its type. What if we hadn't made the proofs of formation irrelevant? Then if there were more than one proof that $double*\ x\ y$ were well-formed for a given x and y, a soundness problem could arise. To see how, imagine extending the above example with a sort distinguishing zero as a refinement.

$zero \sqsubset nat.$
$z :: even\ \wedge\ zero.$

As with *even* and *odd*, the sort *zero* turns into a predicate. Now that z has two sorts, it translates to two proof constructors.

$zero : nat \to \mathsf{type}.$
$\widehat{z}_1 : even\ z.$
$\widehat{z}_2 : zero\ z.$

Next, we can observe that zero always doubles to itself and augment the declaration of *double** using an intersection class:

$$double^* \sqsubset double :: \top \to even \to \mathsf{sort} \\ \wedge\ zero \to zero \to \mathsf{sort}.$$

After translation, since there are potentially two ways for $double^*\ x\ y$ to be well-formed, there are two introduction constants for the formation family.

$\widehat{double^*}/i_1 : \Pi x{:}nat.\ \Pi y{:}nat.\ even\ y \to \widehat{double^*}\ x\ y.$
$\widehat{double^*}/i_2 : \Pi x{:}nat.\ zero\ x \to \Pi y{:}nat.\ zero\ y \to \widehat{double^*}\ x\ y.$

The declarations for $\widehat{double^*}$ and $double^*$ remain the same.

Now recall the refinement declaration for doubling zero,

$dbl/z :: double^*\ z\ z\ ,$

and observe that it is valid for two reasons, since $double^*\ z\ z$ is well-formed for two reasons. Consequently, after translation, there will be two proofs inhabiting the formation family $\widehat{double^*}\ z\ z$, but only one of them will be used in the translation of the dbl/z declaration. Supposing it is the first one, we'll have

$\widehat{dbl/z} : double^*\ z\ z\ [\ \widehat{double^*}/i_1\ z\ z\ \widehat{z}_1\ ]\ dbl/z\ ,$

but our soundness criterion will still require that the constant $\widehat{dbl/z}$ check at the type $double^*\ z\ z\ [\ \widehat{double^*}/i_2\ z\ \widehat{z}_2\ z\ \widehat{z}_2\ ]\ dbl/z$, the other possibility. The apparent mismatch is resolved by the fact that the formation proofs are irrelevant, and so the two types are considered equal.

Following the intuition given above, we may formally describe our translation and prove it correct. But first, we take a brief detour to review prior work on refinement types and proof irrelevance in LF.

3 Refinement Types

Refinement types give means of more precisely characterizing well-typed terms. Systems of refinement types usually sit on top of ordinary type systems, allowing the programmer to specify precise properties of programs already known to be well-typed. This *refinement restriction* is what allows refinement type systems to employ powerful features like subtyping and intersection and union types without overly complicating the system's metatheory.

Although traditionally treated in the context of functional programming [7, 6, 4, 5], recent work has shown how refinement types can be added to the logical

$$K ::= \mathsf{type} \mid \Pi x{:}A.\, K \quad \text{(kinds)} \qquad L ::= \mathsf{sort} \mid \Pi x{::}S.\, L \mid \top \mid L_1 \wedge L_2 \quad \text{(classes)}$$
$$A ::= P \mid \Pi x{:}A_1.\, A_2 \quad \text{(types)} \qquad S ::= Q \mid \Pi x{::}S_1.\, S_2 \mid \top \mid S_1 \wedge S_2 \quad \text{(sorts)}$$
$$P ::= a \mid P\ N \quad \text{(base types)} \qquad Q ::= s \mid Q\ N \quad \text{(base sorts)}$$

Fig. 1. Syntax of LF types and kinds and LFR sorts and classes.

framework LF [10], making it easier to adequately represent languages and logics with certain forms of judgmental inclusion and to declare and check precise properties about relations over such languages. Above, in Sections 1 and 2, we saw a simple example involving even and odd natural numbers and properties of the doubling relation. Here, we briefly recapitulate some of the details of the formal development to set the stage for what is to come.

LF with refinement types, or LFR, is specified using the methodology of *canonical forms*, pioneered by Watkins, et al. [16] in the definition of the Concurrent Logical Framework, CLF. Following this methodology, we consider only canonical forms, or terms that are β-normal and η-long. Terms are syntactically restricted to the β-normal ones via a separation into *atomic* and *normal* terms:

$$R ::= x \mid c \mid R\ N \qquad \text{(atomic terms)}$$
$$M, N ::= R \mid \lambda x.\, N \qquad \text{(normal terms)}$$

These terms are typed bidirectionally with a synthesis judgment $\Gamma \vdash R \Rightarrow A$ and a checking judgment $\Gamma \vdash N \Leftarrow A$. All judgments are relative to an implicit signature Σ, which declares types and kinds for term and type constants.[1]

$$\Sigma ::= \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \qquad \text{(LF declarations)}$$

In extending to LFR, we add three new forms of declaration: refinement declarations, constant sorting declarations, and subsorting declarations.

$$\cdots \mid \Sigma, s{\sqsubset}a{::}L \mid \Sigma, c{::}S \mid \Sigma, s_1{\leq}s_2 \qquad \text{(LFR declarations)}$$

Sorts S refine types A, written $\Gamma \vdash S \sqsubset A$; this judgment is defined compositionally, with the base case fulfilled by refinement declarations $s{\sqsubset}a{::}L$. Similarly, classes L refine kinds K, written $\Gamma \vdash L \sqsubset K$ and also defined compositionally, with the base class "sort" refining the base kind "type". See Fig. 1 for the syntax.

Sorting, like typing, proceeds bidirectionally with $\Gamma \vdash R \Rightarrow S$ and $\Gamma \vdash N \Leftarrow S$. The refinement restriction is embodied by the idea that we only sort check a term N at sort S provided that N already checks at type A, where S refines A.

The key rule to ensuring that normal terms are maximally η-expanded is the **switch** rule for checking an atomic term, so named because it is the rule where we switch modes from checking to synthesis:

$$\frac{\Gamma \vdash R \Rightarrow P' \qquad P' = P}{\Gamma \vdash R \Leftarrow P}\ (\textbf{switch})\ .$$

[1] As usual, we write the signature explicitly on the turnstile only when necessary.

The rule is restricted to base types, P, forcing all variables and constants to be fully applied. Note that although **switch** is the analogue of the usual "conversion rule", since terms are canonical (β-normal, η-long), the equality $P' = P$ is just α-equivalence—we need not worry about β- or η-conversions.

This key rule is changed only slightly for sort-checking: equality of base types P' and P becomes subsorting between base sorts Q' and Q.

$$\frac{\Gamma \vdash R \Rightarrow Q' \qquad Q' \leq Q}{\Gamma \vdash R \Leftarrow Q}\ (\textbf{switch-sub})\ .$$

The **switch-sub** rule is in fact the *only* rule that appeals to subsorting: under the canonical forms methodology, subsorting need only be defined on base sorts, where it is simply the reflexive, transitive closure of the relation declared in the signature, extended through applications to identical arguments.

Aside from the change to the **switch** rule, the only other change from typing to sorting is the addition of rules for introducing and eliminating intersections. Following the usual pattern in bidirectional typing, the introduction rules are *checking* rules and the elimination rules are *synthesis* rules.

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

$$\frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_1}\ (\wedge\textbf{-E}_1) \qquad \frac{\Gamma \vdash R \Rightarrow S_1 \wedge S_2}{\Gamma \vdash R \Rightarrow S_2}\ (\wedge\textbf{-E}_2) \qquad (\text{no } \top\textbf{-E})$$

To maintain terms in canonical form, we must also replace the usual syntactic substitution $[M/x]\,N$ with the *hereditary substitution* $[M/x]_A\,N$ which hereditarily contracts any β-redexes substitution might have created. Hereditary substitution is indexed by the putative type A of the variable x in order to facilitate an early proof of decidability. For the purposes of this paper, though, we will simply write $[M/x]\,N$ for hereditary substitution, since we have no need for ordinary substitution.

In addition to sort-checking being decidable, LFR enjoys the usual Substitution and Expansion Principles: canonical terms may be substituted for variables, and every atomic term can be η-expanded to a canonical one.

$$\eta_P(R) = R \qquad\qquad \eta_{\Pi x{:}A.\,B}(R) = \lambda x.\,\eta_B(R\ \eta_A(x))$$

Principle (Substitution). If $\Gamma_{\mathrm{L}}, x{::}S, \Gamma_{\mathrm{R}} \vdash N \Leftarrow T$ and $\Gamma_{\mathrm{L}} \vdash M \Leftarrow S$, then $\Gamma_{\mathrm{L}}, [M/x]\,\Gamma_{\mathrm{R}} \vdash [M/x]\,N \Leftarrow [M/x]\,T$.

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

4 Proof Irrelevance

When constructive type theory is used as a foundation for verified functional programming, we notice that many parts of proofs are *computationally irrelevant*,

that is, their structure does not affect the returned value we are interested in. The role of these proofs is only to guarantee that the returned value satisfies the desired specification. For example, from a proof of $\forall x{:}A.\, \exists y{:}B.\, C(x,y)$ we may choose to extract a function $f : A \to B$ such that $C(x, f(x))$ holds for every $x{:}A$, but ignore the proof that this is the case. The proof must be present, but its identity is irrelevant. Proof-checking in this scenario has to ascertain that such a proof is indeed not needed to compute the relevant result.

A similar issue arises when a type theory such as λ^{Π} is used as a logical framework. For example, assume we would like to have an adequate representation of prime numbers, that is, to have a bijection between prime numbers p and closed terms $M : \mathit{primenum}$. It is relatively easy to define a type family $\mathit{prime} : \mathit{nat} \to \mathsf{type}$ such that there exists a closed $M : \mathit{prime}\ N$ if and only if N is prime. Then $\mathit{primenum} = \Sigma n{:}\mathit{nat}.\, \mathit{prime}\ n$ is a candidate (with members $\langle N, M\rangle$), but it is not actually in bijective correspondence with prime numbers unless the proof M that a number is prime is always unique. Again, we need the existence of M, but would like to ignore its identity. This can be achieved with *subset types* [3, 14] $\{x{:}\mathit{nat} \mid \mathit{prime}(x)\}$ whose members are just the prime numbers p, but if the restricting predicate is undecidable then type-checking would be undecidable, which is not acceptable for a logical framework.

For LF, we further note that Σ is not available as a type constructor, so we instead introduce a new type *primenum* with exactly one constructor, $\mathit{primenum}/i$:

$\mathit{primenum} : \mathsf{type}.$
$\mathit{primenum}/i : \Pi N{:}\mathit{nat}.\ \mathit{prime}\ N \mathrel{\dot{\to}} \mathit{primenum}.$

Here the second arrow $\dot{\to}$ represents a function that ignores the identity of its argument. The inhabitants of *primenum*, all of the form $\mathit{primenum}/i\ N\ [M]$, are now in bijective correspondence with prime numbers since $\mathit{primenum}/i\ N\ [M] = \mathit{primenum}/i\ N\ [M']$ for all M and M'.

In the extension of LF with proof irrelevance [11, 12], or LFI, we have a new form of hypothesis $x \dot{:} A$ (x has type A, but the identity of x should be irrelevant). In the non-dependent case (the only one important for the purposes of this paper), such an assumption is introduced by a λ-abstraction:

$$\frac{\Gamma, x \dot{:} A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.\, M \Leftarrow A \mathrel{\dot{\to}} B}\ .$$

We can use such variables only in places where their identity doesn't matter, e.g., in the second argument to the constructor $\mathit{primenum}/i$ in the prime number example. More generally, we can only use it in arguments to constructor functions that do not care about the identity of their argument:

$$\frac{\Gamma \vdash R \Rightarrow A \mathrel{\dot{\to}} B \qquad \Gamma^{\oplus} \vdash N \Leftarrow A}{\Gamma \vdash R\ [N] \Rightarrow B}\ .$$

Here, $\Gamma^{\oplus}$ is the *promotion* operator which converts any assumption $x \dot{:} A$ to $x{:}A$, thereby making x usable in N. Note that there is no direct way to use an assumption $x \dot{:} A$.

Judgment:	Result:
$\Gamma \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}_{\text{f}}(-)$	Type of proofs of the formation family
$K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}_{\text{p}}(-,-)$	Kind of the predicate family
$K \overset{\leq}{\rightsquigarrow} \widehat{K}_{\text{s}}(-,-,-,-,-)$	Type of coercions between families of kind K
$\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}(-)$	Metafunction representing predicate
$\Gamma \vdash Q \sqsubset P \Rightarrow L \rightsquigarrow \widehat{Q}$	Proof that Q is well-formed
$\Gamma \vdash N \Leftarrow S \rightsquigarrow \widehat{N}$	Proof that N has sort S
$\Gamma \vdash R \Rightarrow S \rightsquigarrow \widehat{R}$	Proof that R has sort S
$\Gamma \vdash Q_1 \leq Q_2 \rightsquigarrow F(-,-)$	Metacoercion from proofs of Q_1 to proofs of Q_2
$Q_1 \leq Q_2 \rightsquigarrow \widehat{Q_1 - Q_2}$	Coercion from proofs of Q_1 to proofs of Q_2
$\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}$	Translated context
$\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}$	Translated signature

Table 1. Judgments of the translation.

The underlying definitional equality "=" (usually just α-conversion on canonical forms) is extended so that $R\ [N] = R'\ [N']$ if $R = R'$, no matter what N and N' are.

The substitution principle (shown here only in its simplest, non-dependent form) captures the proper typing as well as the irrelevance of assumptions $x \div A$:

Principle (Irrelevant Substitution). If $\Gamma, x \div A \vdash N \Leftarrow B$ and $\Gamma^{\oplus} \vdash M \Leftarrow A$ then $\Gamma \vdash [M/x]\,N \Leftarrow B$ and $[M/x]\,N = N$ (under definitional equality).

5 Interpretation

5.1 Overview

We interpret LFR into LFI by representing sorts as predicates and derivations of sorting as proofs of those predicates. The translation is derivation-directed and compositional: for each judgment $\Gamma \vdash \mathcal{J}$, there is a corresponding judgment $\Gamma \vdash \mathcal{J} \rightsquigarrow X$ whose rules mimic the rules of $\Gamma \vdash \mathcal{J}$. The syntactic class of X and its precise interpretation vary from judgment to judgment (for reference, the various forms are listed in Table 1), but a great deal of insight can be had by examining the specific cases of sort formation and sort checking.

Sort formation is embodied by the refinement judgment $\Gamma \vdash S \sqsubset A$. The corresponding translation judgment has the form $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$, in which $\widehat{S}$ is a meta-level function representing the sort S as a predicate. Sort checking $\Gamma \vdash N \Leftarrow S$ becomes term translation $\Gamma \vdash N \Leftarrow S \rightsquigarrow \widehat{N}$. Since $\widehat{N}$ represents a proof that N has sort S, we should expect that $\widehat{N} \Leftarrow \widehat{S}(N)$.[2]

[2] Under an appropriate context, briefly discussed below.

For example, take the rule $\wedge$**-I** of intersection introduction. The corresponding translation rule represents the two independent derivations as a pair of proofs. Accordingly, the intersection sort formation rule yields a product of predicates.[3]

$$\frac{\Gamma \vdash N \Leftarrow S_1 \rightsquigarrow \widehat{N_1} \quad \Gamma \vdash N \Leftarrow S_2 \rightsquigarrow \widehat{N_2}}{\Gamma \vdash N \Leftarrow S_1 \wedge S_2 \rightsquigarrow \langle \widehat{N_1}, \widehat{N_2} \rangle} \qquad \frac{\Gamma \vdash S_1 \sqsubset A \rightsquigarrow \widehat{S_1} \quad \Gamma \vdash S_2 \sqsubset A \rightsquigarrow \widehat{S_2}}{\Gamma \vdash S_1 \wedge S_2 \sqsubset A \rightsquigarrow \boldsymbol{\lambda} N.\ \widehat{S_1}(N) \times \widehat{S_2}(N)}$$

We use a bold $\boldsymbol{\lambda}$ for meta-level abstraction and bold (parens) for meta-level application. Our translation is similar in spirit to Liquori and Ronchi Della Rocca's $\Lambda^{\mathsf{t}}_{\wedge}$ [9], a Church-style type system for intersections in which derivations are explicitly represented as proofs and intersections as products.

At the top-level, we are interested in checking entire signatures, so it is also instructive to examine the rules for translating the LFR declarations. As we saw with *even* and *odd* in Section 1, sorting declarations for constants turn into proof constructor declarations.

$$\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad c{:}A \in \Sigma \qquad \cdot \vdash_\Sigma S \sqsubset A \rightsquigarrow \widehat{S}}{\vdash \Sigma,\ c{::}S\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma},\ \widehat{c}{:}\widehat{S}(\eta_A(c))}$$

This matches our intuitions: the proof constructor $\widehat{c}$ witnesses the fact that the constant c satisfies property S. Since our predicates expect terms in canonical form, we η-expand the constant. The translation on contexts is similar.

As we saw with *double** in Section 2, a refinement declaration turns into three declarations: one for the *formation family*, one for the *proof constructor* for the formation family, and one for the *predicate family*.

$$\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad a{:}K \in \Sigma \qquad \cdot \vdash_\Sigma L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}_{\mathrm{f}} \qquad K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}_{\mathrm{p}}}{\vdash \Sigma, s{\sqsubset}a{::}L\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma},\ \widehat{s}{:}K,\ \widehat{s}/i{:}\widehat{L}_{\mathrm{f}}(\widehat{s}),\ s{:}\widehat{K}_{\mathrm{p}}(\widehat{s}, a)}$$

The class formation judgment $\Gamma \vdash L \sqsubset K \overset{\text{form}}{\rightsquigarrow} \widehat{L}_{\mathrm{f}}$ yields a metafunction describing the type of proofs of formation family, while an auxiliary kind translation judgment $K \overset{\text{pred}}{\rightsquigarrow} \widehat{K}_{\mathrm{p}}$ yields a metafunction describing the kind of the predicate family. Recall from our earlier example that the kind of the formation family is the same as the kind of the refined type, in this case K.

The metafunction $\widehat{L}_{\mathrm{f}}$ takes as input the formation family so far (initially just $\widehat{s}$) and the translation derivation adds arguments on the way up. At the base case it returns the formation family itself.

The metafunction $\widehat{K}_{\mathrm{p}}$ takes two arguments: one for the formation family so far (initially $\widehat{s}$) and one for the refined type so far (initially a). This translation is characterized by its behavior on the base kind, type:

$$\frac{}{\mathsf{type} \overset{\text{pred}}{\rightsquigarrow} \boldsymbol{\lambda}(Q_{\mathrm{f}}, P).\ Q_{\mathrm{f}} \doteqdot P \to \mathsf{type}}$$

[3] For compositionality's sake, we target an extension of LFI with product and unit types. Such an extension is orthogonal to the addition of proof irrelevance, and has been studied by many people over the years, including Schürmann [15].

The kind of the predicate family for a base sort Q refining P is essentially a one-place judgment on terms of type P, along with an irrelevant argument belonging to the formation family of Q. In light of this, we can make sense of the rule for translating base sorts:

$$\frac{\Gamma \vdash Q \sqsubset P' \Rightarrow L \rightsquigarrow \widehat{Q} \qquad P' = P \qquad L = \mathsf{sort}}{\Gamma \vdash Q \sqsubset P \rightsquigarrow \boldsymbol{\lambda} N.\ Q\ [\widehat{Q}]\ N}$$

The class synthesis translation judgment $\Gamma \vdash Q \sqsubset P \Rightarrow L \rightsquigarrow \widehat{Q}$ yields a proof of Q's formation family; thus the predicate for a base sort Q, given an argument N, is simply the predicate family Q applied to an irrelevant proof $\widehat{Q}$ that Q is well-formed and the argument itself, N.

We postpone a discussion of subsorting declarations $s_1 \leq s_2$ until after a brief review of some metatheoretic results.

5.2 Correctness

Soundness theorems tell us that the result of a translation is well-formed. Even more importantly than telling us that our translation is on some level correct, they serve as an independent means of understanding the translation. In a sense, a soundness theorem can be read as the meta-level type of a translation judgment, and just as types serve as an organizing principle for the practicing programmer, so too do soundness theorems serve the thoughtful theoretician. We mention a few such theorems, then, not only to demonstrate the sensibility of our translation, but also to aid the reader in understanding its purpose.

In what follows, $\mathrm{form}(Q)$ represents the formation family for a base sort Q.

$$\mathrm{form}(s) = \widehat{s} \qquad\qquad \mathrm{form}(Q\ N) = \mathrm{form}(Q)\ N$$

Theorem 1 (Soundness). *Suppose* $\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}$ *and* $\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}$. *Then:*

1. *If* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\Gamma \vdash N \Leftarrow S \rightsquigarrow \widehat{N}$, *then* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{N} \Leftarrow \widehat{S}(N)$.
2. *If* $\Gamma \vdash R \Rightarrow S \rightsquigarrow \widehat{R}$, *then* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{R} \Rightarrow \widehat{S}(\eta_A(R))$ *(for some* A *and* $\widehat{S}$*).*
3. *If* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\Gamma \vdash N \Leftarrow A$, *then* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{S}(N) \Leftarrow \mathsf{type}$.
4. *If* $\Gamma \vdash Q \sqsubset P \Rightarrow L \rightsquigarrow \widehat{Q}$, *then for some* K, $\widehat{L}_{\mathrm{f}}$, *and* $\widehat{K}_{\mathrm{p}}$,
   - $\Gamma \vdash L \sqsubset K \overset{\mathrm{form}}{\rightsquigarrow} \widehat{L}_{\mathrm{f}}$ *and* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{Q} \Rightarrow \widehat{L}_{\mathrm{f}}(\,\mathrm{form}(Q))$, *and*
   - $K \overset{\mathrm{pred}}{\rightsquigarrow} \widehat{K}_{\mathrm{p}}$ *and* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} Q \Rightarrow \widehat{K}_{\mathrm{p}}(\,\mathrm{form}(Q), P)$.
5. *If* $\Gamma \vdash L \sqsubset K \overset{\mathrm{form}}{\rightsquigarrow} \widehat{L}_{\mathrm{f}}$ *and* $\Gamma \vdash P \Rightarrow K$, *then* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{L}_{\mathrm{f}}(P) \Leftarrow \mathsf{type}$.
6. *If* $K \overset{\mathrm{pred}}{\rightsquigarrow} \widehat{K}_{\mathrm{p}}$, $\Gamma \vdash Q_{\mathrm{f}} \Rightarrow K$, *and* $\Gamma \vdash P \Rightarrow K$, *then* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{K}_{\mathrm{p}}(Q_{\mathrm{f}}, P) \Leftarrow \mathsf{kind}$.

The soundness theorems are each proven by induction on the theorem's main input derivation. Several clauses must be proved mutually, and not all theorems are shown here. The proofs appeal to several key lemmas.

Lemma 1 (Erasure). *If* $\Gamma \vdash \mathcal{J} \rightsquigarrow X$*, then* $\Gamma \vdash \mathcal{J}$*.*

Lemma 2 (Reconstruction). *If* $\Gamma \vdash \mathcal{J}$*, then for some* X*,* $\Gamma \vdash \mathcal{J} \rightsquigarrow X$*.*

Erasure and Reconstruction substantiate the claim that our translation is derivation-directed by allowing us to move freely between translation judgments and ordinary ones. Using Erasure and Reconstruction, we can leverage all of the LFR metatheory without reproving it for translation judgments. For example, several cases require us to substitute into a translation derivation: we can apply Erasure, appeal to LFR's Substitution Theorem, and invoke Reconstruction to get the output we require.

But since Reconstruction only gives us *some* output X, we may not know that it is the one that suits our needs. Therefore, we usually require another lemma, Compositionality, to tell us that the translation commutes with substitution. There are several such lemmas; we show here the one for sort translation.

Lemma 3 (Compositionality). *Let* σ *denote* $[M/x]$*. If* $\Gamma, x{::}\_ \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\Gamma \vdash \sigma S \sqsubset \sigma A \rightsquigarrow \widehat{S}'$*, then* $\sigma\widehat{S}(N) = \widehat{S}'(\sigma N)$*.*

Completeness theorems tell us that our target is not too rich: that everything we find evidence of in the codomain of the translation actually holds true in its domain. While important for establishing general correctness, completeness theorems are not quite so nice to look at as soundness theorems, so we give here only the cases for terms.

Theorem 2 (Completeness). *Suppose* $\vdash \Gamma\ \mathsf{ctx} \rightsquigarrow \widehat{\Gamma}$ *and* $\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}$*. Then:*

1. *If* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$ *and* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{N} \Leftarrow \widehat{S}(N)$*, then* $\Gamma \vdash N \Leftarrow S$*.*
2. *If* $\widehat{\Gamma} \vdash_{\widehat{\Sigma}} \widehat{R} \Rightarrow B$*, then* $\Gamma \vdash S \sqsubset A \rightsquigarrow \widehat{S}$*,* $B = \widehat{S}(\eta_A(R))$*, and* $\Gamma \vdash R \Rightarrow S$ *(for some* S*,* A*,* $\widehat{S}$*, and* R*).*

In stating Completeness, we syntactically isolate the set of terms that could represent proofs using, e.g., metavariables $\widehat{R}$ and $\widehat{N}$. Completeness is proven by induction over the structure of these terms.

Adequacy of a representation is generally shown by exhibiting a compositional bijection between informal entities and terms of certain LFR sorts. Since we have undertaken a subset interpretation, the set of terms of any LFR sort are unchanged by translation, and so any bijective correspondence between those terms and informal entities remains after translation. Furthermore, soundness and completeness tell us that our interpretation preserves and reflects the derivability of any refinement type judgments over those terms. Thus, we have achieved our main goal: any adequate LFR representation can be translated to an adequate LFI representation.

5.3 Subsorting

We now return to the question of how the translation handles subsorting. Recall that an LFR signature can include subsorting declarations between sort family

constants, $s_1 \leq s_2$. We require both sort constants to refine the same type constant a and to have the same class L. The rule for translating such declarations creates a coercion constant $s_1\text{-}s_2$.

$$\frac{\vdash \Sigma\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma} \qquad s_1 \sqsubset a :: L \in \Sigma \qquad s_2 \sqsubset a :: L \in \Sigma \qquad a : K \in \Sigma \qquad K \overset{\leq}{\rightsquigarrow} \widehat{K}_{\mathrm{s}}}{\vdash \Sigma, s_1 \leq s_2\ \mathsf{sig} \rightsquigarrow \widehat{\Sigma}, s_1\text{-}s_2 : \widehat{K}_{\mathrm{s}}(a, \widehat{s_1}, s_1, \widehat{s_2}, s_2)}$$

The auxiliary judgment $K \overset{\leq}{\rightsquigarrow} \widehat{K}_{\mathrm{s}}$ yields a metafunction describing the type of coercions between sorts that refine a type family of kind K. The metafunction $\widehat{K}_{\mathrm{s}}$ takes five arguments: the refined type, the formation family and predicate family for the domain of the coercion, and the formation family and predicate family for the codomain of the coercion. As before, the translation derivation builds up a spine of arguments on the way up towards the leaves. At the base kind type, it outputs the type of the coercion:

$$\mathsf{type} \overset{\leq}{\rightsquigarrow} \boldsymbol{\lambda}(P, Q_{1\mathrm{f}}, Q_1, Q_{2\mathrm{f}}, Q_2).\ \Pi f_1 : Q_{1\mathrm{f}}.\ \Pi f_2 : Q_{2\mathrm{f}}.\ \Pi x : P.\ Q_1\ [f_1]\ x \to Q_2\ [f_2]\ x$$

Essentially, this is the type of coercions, given x, from proofs of $Q_1\ x$ to proofs of $Q_2\ x$, but of course we must pass Q_1 and Q_2 proof that they are well-formed, so the coercion requires those proofs as inputs as well.

How do these coercions work? Recall from Section 3 that subsorting need only be defined at base sorts Q, and there, it is simply the application-compatible, reflexive, transitive closure of the declared relation. For the purposes of the translation, we give an equivalent algorithmic formulation of subsorting. Following the inspiration of bidirectional typing, we give two judgments: a *checking* judgment that takes two base sorts as inputs and a *synthesis* judgment that takes one base sort as input and outputs another base sort that is one step higher in the subsort hierarchy.

The synthesis judgment constructs a coercion from the new coercion constants in the signature.

$$\frac{s_1 \leq s_2 \in \Sigma}{s_1 \leq s_2 \rightsquigarrow s_1\text{-}s_2} \qquad \frac{Q_1 \leq Q_2 \rightsquigarrow \widehat{Q_1\text{-}Q_2}}{Q_1\ N \leq Q_2\ N \rightsquigarrow \widehat{Q_1\text{-}Q_2}\ N}$$

The checking judgment, on the other hand, constructs a *meta-level* coercion between the two sorts. It is defined by two rules: a rule of reflexivity and a rule to climb the subsort hierarchy.

$$\frac{Q_1 = Q_2}{\Gamma \vdash Q_1 \leq Q_2 \rightsquigarrow \boldsymbol{\lambda}(R, R_1).\ R_1}\ (\textbf{refl})$$

$$\frac{\begin{array}{cc} Q_1 \leq Q' \rightsquigarrow \widehat{Q_1\text{-}Q'} & \Gamma \vdash Q_1 \sqsubset P \Rightarrow \mathsf{sort} \rightsquigarrow \widehat{Q_1} \\ \Gamma \vdash Q' \leq Q_2 \rightsquigarrow F & \Gamma \vdash Q' \sqsubset P \Rightarrow \mathsf{sort} \rightsquigarrow \widehat{Q'} \end{array}}{\Gamma \vdash Q_1 \leq Q_2 \rightsquigarrow \boldsymbol{\lambda}(R, R_1).\ F(R, \widehat{Q_1\text{-}Q'}\ \widehat{Q_1}\ \widehat{Q'}\ R\ R_1)}\ (\textbf{climb})$$

The reflexivity rule's metacoercion simply returns the proof it is given, while the climb rule composes the actual coercion $\widehat{Q_1 - Q'}$ with the metacoercion F. Two extra premises generate the necessary formation proofs.

We can now end where we started, with the **switch** rule. Using the metacoercion generated by subsort checking, we can construct the proof we need for soundness.

$$\frac{\Gamma \vdash R \Rightarrow Q' \rightsquigarrow \widehat{R} \qquad \Gamma \vdash Q' \leq Q \rightsquigarrow F}{\Gamma \vdash R \Leftarrow Q \rightsquigarrow F(R, \widehat{R})} \ (\textbf{switch-sub})$$

6 Conclusion

Logical frameworks are metalanguages specifically designed so that common concepts and notations in logic and the theory of programming languages can be represented elegantly and concisely. LF [8] intrinsically supports α-conversion, capture-avoiding substitution, and hypothetical and parametric judgments, but as with any such enterprise, certain patterns fall out of its scope and must be encoded indirectly. One pattern is the ability to form regular subsets of types already defined. This is addressed in LF extended with type refinement (LFR) [10]. Another pattern is to ignore the identities of proofs, relying only on their existence. This is addressed in LF extended with proof irrelevance (LFI) [11, 12]. In this paper we have shown that the former can be mapped to the latter in a bijective manner, preserving adequacy theorems for LFR representations in LFI.

In the methodology of logical frameworks research, it is important to understand the cost of such a translation: how much more complicated are encodings in the target framework, and how much more difficult is it to work with them? We cannot measure this cost precisely, but we hope it is evident from the definition of the translation and the examples that the price is considerable. Even if in special cases more direct encodings are possible, we believe our general translation could not be simplified much, given the explicit goal to preserve the adequacy of representations. Other translations from programming languages, such as coercion interpretations where sorts are translated to distinct types and subsorting to coercions, appear even more complex because adequacy depends on certain functional equalities between coercions. Our preliminary conclusion is that refinement types in logical frameworks provide elegant and immediate representations that are not easy to simulate without them, providing a solid argument for their inclusion in the next generation of frameworks.

Refinement types have been also been proposed for functional programming [7, 4], most recently in conjunction with a limited form of dependent types [5]. Proof irrelevance is already integrated in this setting, and also available in general type theories such as NuPrl or Coq. One can ask the same question here: Can we simply eliminate refinement types and just work with dependent types and proof irrelevance? The results in this paper lend support to the conjecture that this can be accomplished by a uniform translation. On the other hand, just as here, it seems there would likely be a high cost in terms of brevity in order

to maintain a bijection between well-sorted data in the source and dependently well-typed data in the target of the translation.

Acknowledgements. Thanks to Jason Reed for many fruitful discussions on the topic of proof irrelevance.

References

1. Awodey, S., Bauer, A.: Propositions as [types]. Journal of Logic and Computation **14**(4) (2004) 447–471
2. Breazu-Tannen, V., Coquand, T., Gunter, C.A., Scedrov, A.: Inheritance as implicit coercion. Information and Computation **93**(1) (July 1991) 172–221
3. Constable, R.L., et al.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall, Englewood Cliffs, New Jersey (1986)
4. Davies, R.: Practical Refinement-Type Checking. PhD thesis, Carnegie Mellon University (May 2005) Available as Technical Report CMU-CS-05-110.
5. Dunfield, J.: A Unified System of Type Refinements. PhD thesis, Carnegie Mellon University (August 2007) Available as Technical Report CMU-CS-07-129.
6. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In Leroy, X., ed.: ACM Symp. Principles of Programming Languages (POPL '04), Venice, Italy (January 2004) 281–292
7. Freeman, T.: Refinement Types for ML. PhD thesis, Carnegie Mellon University (March 1994) Available as Technical Report CMU-CS-94-110.
8. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the Association for Computing Machinery **40**(1) (January 1993) 143–184
9. Liquori, L., Ronchi Della Rocca, S.: Intersection-types à la Church. Information and Computation **205**(9) (2007) 1371–1386
10. Lovas, W., Pfenning, F.: A bidirectional refinement type system for LF. Electronic Notes in Theoretical Computer Science **196** (January 2008) 113–128
11. Pfenning, F.: Intensionality, extensionality, and proof irrelevance in modal type theory. In Halpern, J., ed.: Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01), Boston, Massachusetts, IEEE Computer Society Press (June 2001) 221–230
12. Reed, J., Pfenning, F.: Proof irrelevance in a logical framework. Unpublished draft (July 2008)
13. Reynolds, J.C.: The coherence of languages with intersection types. In Ito, T., Meyer, A.R., eds.: Theoretical Aspects of Computer Software. Volume 526 of Lecture Notes in Computer Science., Berlin, Springer-Verlag (1991) 675–700
14. Salvesen, A., Smith, J.M.: The strength of the subset type in Martin-Löf's type theory. In: Proceedings of LICS'88, IEEE Computer Society Press (1988) 384–391
15. Schürmann, C.: Towards practical functional programming with logical frameworks. Unpublished, available at `http://cs-www.cs.yale.edu/homes/carsten/delphin/` (July 2003)
16. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: 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.
17. Zeilberger, N.: Refinement types and computational duality. In: PLPV '09: Proceedings of the 3rd workshop on Programming Languages Meets Program Verification, New York, NY, USA, ACM (2009) 15–26