Intersection Types for a Logical Framework

Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891

Internet: `fp@cs.cmu.edu`

December 1992

Abstract

We propose a refinement of the type theory underlying the LF logical framework by a form of subtypes and intersection types. This refinement preserves desirable features of LF, such as decidability of type-checking, and at the same time considerably simplifies the representations of many deductive systems.

1 Introduction

Over the past two years we have carried out extensive experiments in the application of the LF Logical Framework [HHP93] to represent and implement deductive systems and their metatheory. Such systems arise naturally in the study of logic and the theory of programming languages. For example, we have formalized the operational semantics and type system of Mini-ML and implemented a proof of type preservation [MP91] and the correctness of a compiler to a variant of the Categorical Abstract Machine [HP92]. LF is based on a predicative type theory with dependent types. It has proved to be an excellent language for such formalization efforts, since it allows direct representation of deductions as objects and judgments as types and supports common concepts such as variable binding, substitution, and generic and hypothetical judgments. The logic programming language Elf [Pfe91a] implements LF and gives it an operational interpretation so that LF signatures can be executed as logic programs. It also provides sophisticated term reconstruction, which is important for realistic applications.

Despite its expressive power, certain weaknesses of LF emerged during these experiments. One of these is the absence of any direct form of subtyping. Clearly, this is not a theoretical problem: what is informally presented as subtyping can be encoded either via explicit coercions or via auxiliary judgments as we will illustrate below. In practice, however, this becomes a significant burden and one has the feeling that the framework *should* support simple reasoning about subtypes.

An obvious candidate for an extension of the type system are *subset types* as they are used for example in Martin-Löf type theory [SS88]. In a logical framework, however, they are problematic, because they lead to an undecidable type-checking problem. The methodology of LF reduces proof checking in the object language to type checking in the meta-language (the LF type theory), and thus decidability is paramount. Looking elsewhere, we find an extensive body of work on *order-sorted* first-order calculi and their use in logic programming and automated theorem proving (see, for example, [Smo89, SS89]). However, it is not clear how to generalize these calculi to logics or type theories with higher-order functions, although recently some interesting work in this direction has begun [Koh91, Koh92, NQ92]. Similar systems of simple subtypes have been used in programming languages, in particular in connection with record types and object-oriented programming, but such systems are not expressive enough for our purposes. More promising are enhancements of simple subtypes with *intersection types* [CDCV81], which have been applied to programming languages [Rey91] and recently also in type theory [Hay91]. General decidability of type-checking or inference in such calculi is problematic, but under certain restrictions type checking is decidable and principal types exist [Rey88, FP91, CG92].

In this paper we tie together ideas from these threads of research and propose a refinement of the LF type theory by a version of bounded intersection types, or *refinement types*, as we call them. The resulting type theory $\lambda^{\Pi\&}$ allows more direct encodings of deductive systems in many examples. We show that it has a decidable type-checking problem and is thus useful as a logical framework. We have not yet implemented this system, but experience with a related implementation of refinement types for ML [FP91] and the current Elf term reconstruction algorithm leads us to believe that type-checking will be practical.

The system we propose is relevant not only to LF and its Elf implementation, but could be directly applied to λProlog [MNPS91] or Isabelle [PN90] with similar benefits.

In future work, we plan to consider the operational aspects of this type theory so that it can be fully embedded into the current Elf implementation. This includes unification, or rather, solving of constraints in the style described in [Pfe91a, Pfe91b], type reconstruction, and search. Based on experience from first-order logic programming we conjecture that subtyping constraints can lead to improved operational behavior of many programs.

2 Two Motivating Examples

In this section we give two prototypical examples which motivate our extension of the LF type theory. Space only permits a rather sketchy discussion of these examples; the interested reader may find additional explanation in the indicated references.

Hereditary Harrop Formulas. Here we consider, as an object logic, the language of hereditary Harrop formulas [MNPS91], a fragment of logic suitable as a basis for a logic programming language. For the sake of brevity we restrict ourselves to the propositional formulas.

$$\textit{Formulas} \quad F \quad ::= \quad A \mid F_1 \wedge F_2 \mid F_1 \supset F_2 \mid F_1 \vee F_2$$

Here A ranges over atomic formulas. We now define legal program and goal formulas.

$$\begin{array}{lll} \textit{Programs} & D & ::= \quad A \mid D_1 \wedge D_2 \mid G \supset D \\ \textit{Goals} & G & ::= \quad A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G \end{array}$$

How do we represent these definitions in LF? The definition of formulas given here in the concrete syntax of Elf, is straightforward.

```
form : type.

=>   : form -> form -> form.  %infix right 10 =>
||   : form -> form -> form.  %infix right 12 ||
&&   : form -> form -> form.  %infix right 14 &&
```

Atomic formulas are not explicitly declared, but we assume that declarations for predicate constants are added to this basic signature as they are introduced. The next question is how to represent programs and goals. Here we can go two ways: one is to introduce explicit judgments *atom* F, *prog* F, and *goal* F which can be used to prove that a given formula F is either an atom, program, or goal. That is, showing only the rules for programs:

```
atom : form -> type.
goal : form -> type.
prog : form -> type.

p_atom : atom A -> prog A.
p_imp  : goal A -> prog B -> prog (A => B).
p_and  : prog A -> prog B -> prog (A && B).
```

Here, free variables in a declaration are implicitly Π-quantified.

A judgment, such as $P \vdash G$ (program P entails goal G) must now carry explicit evidence that the constituents D and G are in fact legal programs and goals. We call this judgment *solve* $P\,G$, indicating its use as a logic program. It requires *backchain* as an auxiliary judgment. `{x:A} K` is Elf's concrete syntax for $\Pi x{:}A.\ K$.

```
solve     : {P:form} prog P -> {G:form} goal G -> type.
backchain : {P:form} prog P -> {A:form} atom A -> {G:form} goal G -> type.
```

The rules defining these judgments lead to a very awkward and inefficient implementation of proof search, since *solve* is now a type family indexed by four arguments instead of only two.

Another possibility is to declare separate types for programs and goals. Unfortunately, this means that we have to introduce separate instances of the shared connectives, and the connection to an overarching language of formulas is lost and would have to be axiomatized separately.

Both alternatives illustrate general techniques available within the LF type theory. While feasible for relatively small examples, they become very difficult to manage for larger examples and obscure the representations greatly compared to the relative simplicity of the informal definition. In contrast, with refinement types we can declare a type of formulas and then atoms, programs, and goals as subtypes.

Natural Deductions in Normal Form. The next example illustrates that we often want to make subtype distinctions at the level of deductions and not only at the level of syntax. We follow the usual representation of natural deduction in LF [HHP93] and Felty's trick to enforce normal forms [Fel89]. We restrict ourselves to the purely implicational fragment.

$$\dfrac{\begin{array}{c}\overline{A}^{\,x}\\ \vdots\\ B\end{array}}{A \supset B}\supset\mathrm{I}^x \qquad\qquad \dfrac{A \supset B \qquad A}{B}\supset\mathrm{E}$$

The deduction in the premise of the implication introduction rule discharges the hypothesis A labelled x and is represented as a function from deductions of A to deductions of B. The derivability judgment is represented by the family *pf* which is indexed by a formula.

```
o   : type.
imp : o -> o -> o.

pf  : o -> type.

impi : (pf A -> pf B) -> pf (imp A B).
impe : pf (imp A B) -> pf A -> pf B.
```

Again, quantifiers over A and B are implicit. A type of the form *pf* A is the type of all natural deductions of A. A natural deduction is *normal* if no introduction of an implication is immediately followed by its elimination. An equivalent formulation essentially says that we can only reason with elimination rules from hypotheses and with introduction rules from the conclusion. We implement this via two judgments, *elim* and *nf*, on deductions. This has the same drawbacks as in the previous example: it is more verbose, and arguments proliferate in judgments which depend on *elim* and *nf*. Here is how this alternative could be written:

```
nf   : pf A -> type.
elim : pf A -> type.

impi_nf   : {Q:pf A -> pf B} ({P:pf A} elim P -> nf (Q P)) -> nf (impi Q).
impe_elim : {P:pf (imp A B)} {Q:pf A} elim P -> nf Q -> elim (impe P Q).
elim_nf   : {P:pf A} elim P -> nf P.
```

Implicit arguments (to *nf*, *elim*, *impi*, and *impe*) and type reconstruction in Elf go a long way towards making this option feasible, but it is still awkward. Felty's solution introduces new families *elim* and *nf* indexed by formulas. Again, the connection to *pf* remains informal and one then has to prove that every normal natural deduction is in fact a natural deduction. Using refinement types, we will be able to declare deductions in normal form as a subtype of natural deductions.

3 The Refinement Type System

In this section we present a refinement of the LF type theory (λ^{Π}) to accomodate commonly used forms of subtypes. We refer to this system as $\lambda^{\Pi\&}$. We have to ensure that the basic, necessary properties of the LF type theory are not destroyed: in particular, we need to preserve decidability of type-checking and the adequacy of encodings. These requirements have led us to a number of basic design decision which we review here before the technical development. The examples will draw upon Section 2.

Sorts and Proper Types. Semantically, a *sort* may be best thought of as describing a subset of a *proper type* as it exists in LF. This extends through the type hierarchy in straightforward fashion; for example, the sort ($\mathit{elim}\, A \to \mathit{nf}\, B$) will describe a subset of the functions of type ($\mathit{pf}\ A \to \mathit{pf}\, B$), namely those that map a deduction of A by elimination rules to a normal form deduction of B. Thus we think of sorts as a *refinement* of the structure of types, and similary for sort families indexed by objects. Sorts are not distinguished syntactically, but via a new form of declaration that specifies a sort refining a type. For example, $\mathit{goal} :: \mathit{form}$ declares the sort *goal* of legal goals as a refinement of the type *form* of formulas.

Subsorts and Intersection Types. The space of sorts that refine a given proper type must possess structure to be useful. We thus introduce new declarations of the form $a \leq a'$ that specify that sort a is a subsort of sort a'. This will only be considered well-formed when both a and a' refine some proper type b. At the level of functions, simple subsorting is insufficient, since a given λ-expression may have a number of different sorts. For example, $(\lambda x{:}\mathit{pf}\ A.\ x)$ has type $\mathit{pf}\ A \to \mathit{pf}\ A$, and also sorts $\mathit{elim}\,A \to \mathit{elim}\,A$ and $\mathit{nf}\ A \to \mathit{nf}\ A$. In order to express all these properties directly we use intersection types:

$$(\lambda x{:}\mathit{pf}\ A.\ x) : (\mathit{elim}\,A \to \mathit{elim}\,A)\,\&(\mathit{nf}\ A \to \mathit{nf}\ A)\,\&(\mathit{pf}\ A \to \mathit{pf}\ A).$$

Again, in keeping with the basic refinement philosophy, sorts may only be conjoined if they refine a common type ($\mathit{pf}\ A \to \mathit{pf}\ A$, in this example).

Objects. We also make a basic decision not to change the space of objects, but merely to classify them more accurately than in λ^{Π}. This may seem rather drastic insofar as types occur in objects (labelling λ's) and one might thus expect them to change as the language of types changes. Through the typing rules we enforce that λ-abstractions are labelled by proper types. The typing rules then allow analysis of the body of the term $\lambda x{:}A.\ M$ for every sort that refines the type A. This restriction may not be necessary to obtain a decidable system, but it affords a tremendous simplification of the meta-theory of our calculus without effecting its expressiveness in any essential way. It is also consistent with the philosphy behind refinement types. Note that such a restriction is only reasonable because the functions we may form at the level of objects do not constitute a general purpose, functional programming language, but only serve to represent binding and scope at various levels (variable binding, hypotheses, *etc.*).

3.1 Syntax

We maintain LF's three levels and augment families and kinds by intersections. Objects and contexts remain basically the same, although we have eliminated family-level abstractions $\lambda x{:}A_1.\ A_2$, since they do not occur in practice or in normal forms and would complicate our meta-theory.

$$\begin{array}{llcl}
\textit{Kinds} & K & ::= & \mathrm{Type} \mid \Pi x{:}A.\ K \mid K_1 \,\&\, K_2 \\
\textit{Families} & A & ::= & a \mid A\,M \mid \Pi x{:}A_1.\ A_2 \mid A_1 \,\&\, A_2 \\
\textit{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\,M_2 \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}$$

Signatures may now contain two additional forms of declarations: refinement declarations $a_1 :: a_2$ and subsort declarations $a_1 \leq a_2$.

$$\textit{Signatures}\quad \Sigma \quad ::= \quad \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \mid \Sigma, a_1 :: a_2 \mid \Sigma, a_1 \leq a_2$$

We now also drop the restriction that a constant may be declared at most once in a signature (where $a{:}K$, $a_1 :: a_2$, and $c{:}A$ declare a, a_1, and c, respectively). Instead we impose other validity conditions in the next section. As usual, we consider α-convertible terms to be identical.

3.2 Judgments

In our approach, it is extremely important that sorts and sort families can be recognized, and that a sort refines a unique type. Thus we begin by defining the refinement judgment. Since it must be applied uniformly through all levels (kinds, families, objects) with essentially the same rules, we use the meta-variables U and V to range over terms from any of the three levels and d to range over object-level or family-level constants. For an instance of a rule schema to be valid it must be sensible according to the stratification imposed above. Variable occurring in the terms involved in this judgment are treated uniformly, so we omit the context here.

$$\frac{}{\vdash_\Sigma \mathrm{Type} :: \mathrm{Type}} \qquad \frac{\vdash_\Sigma U_1 :: V_1 \qquad \vdash_\Sigma U_2 :: V_2}{\vdash_\Sigma \Pi x{:}U_1.\, U_2 :: \Pi x{:}V_1.\, V_2}$$

$$\frac{\vdash_\Sigma U_1 :: V_1 \qquad \vdash_\Sigma U_2 :: V_2}{\vdash_\Sigma U_1\, U_2 :: V_1\, V_2} \qquad \frac{\vdash_\Sigma U_1 :: V \qquad \vdash_\Sigma U_2 :: V}{\vdash_\Sigma U_1 \,\&\, U_2 :: V}$$

$$\frac{}{\vdash_\Sigma x :: x} \qquad \frac{\vdash_\Sigma U_1 :: V_1 \qquad \vdash_\Sigma U_2 :: V_2}{\vdash_\Sigma \lambda x{:}U_1.\, U_2 :: \lambda x{:}V_1.\, V_2}$$

$$\frac{d{:}U \text{ in } \Sigma}{\vdash_\Sigma d :: d} \qquad \frac{a :: a' \text{ in } \Sigma}{\vdash_\Sigma a :: a'}$$

Note that the refinement relation is neither transitive nor reflexive. The conditions on valid signatures will guarantee that exactly one of the last two cases is applicable for any declared constant, and the second only for a unique a'. This implies that in a valid signature Σ for a given U there exists at most one V such that $\vdash_\Sigma U :: V$.

The validity judgments have the following form. Here, Kind is a special token to allow a uniform presentation of the validity judgments at the three levels.

$\vdash \Sigma$ Sig	Σ *is a valid signature*
$\vdash_\Sigma \Gamma$ Ctx	Γ *is a valid context*
$\Gamma \vdash_\Sigma K : \mathrm{Kind}$	K *is a valid kind*
$\Gamma \vdash_\Sigma A : K$	A *is a valid family of kind* K
$\Gamma \vdash_\Sigma M : A$	M *is a valid object of type* A

We also need the auxiliary judgments

$U \equiv V$	U *is* $\beta\eta$*-convertible to* V
$\vdash_\Sigma U \leq V$	U *is a subsort of* V

where the subsorting judgment only applies at the levels of families and kinds. Here are the rules for valid signatures.

$$\frac{}{\vdash \cdot \ \mathrm{Sig}}$$

$$\frac{\vdash \Sigma\ \mathrm{Sig} \qquad \vdash_\Sigma K : \mathrm{Kind} \qquad \vdash_\Sigma K :: K' \qquad \vdash_\Sigma K_i :: K' \text{ for any } a{:}K_i \text{ in } \Sigma \qquad \text{no } a :: a' \text{ in } \Sigma}{\vdash \Sigma, a{:}K\ \mathrm{Sig}}$$

$$\frac{\vdash \Sigma\ \mathrm{Sig} \qquad \vdash_\Sigma A : \mathrm{Type} \qquad \vdash_\Sigma A :: A' \qquad \vdash_\Sigma A_i :: A' \text{ for any } c{:}A_i \text{ in } \Sigma}{\vdash \Sigma, c{:}A\ \mathrm{Sig}}$$

$$\frac{\vdash \Sigma\ \mathrm{Sig} \qquad a_2{:}K \text{ in } \Sigma \qquad a_1 \text{ not declared in } \Sigma}{\vdash \Sigma, a_1 :: a_2\ \mathrm{Sig}}$$

$$\frac{\vdash \Sigma\ \mathrm{Sig} \qquad a_1 :: a_3 \text{ in } \Sigma \qquad a_2 :: a_3 \text{ in } \Sigma}{\vdash \Sigma, a_1 \leq a_2\ \mathrm{Sig}}$$

A declaration of the form $a :: b$ declares a *sort family* a which inherits its kind from the type family b it refines. Valid contexts are straightforward.

$$\frac{}{\vdash_\Sigma \cdot \ \mathrm{Ctx}} \qquad \frac{\vdash_\Sigma \Gamma\ \mathrm{Ctx} \qquad \Gamma \vdash_\Sigma A : \mathrm{Type}}{\vdash_\Sigma \Gamma, x{:}A\ \mathrm{Ctx}}$$

The rules for valid terms are uniform throughout the levels (as long as they apply), so we give them in schematic form for terms. Note that we do not check validity of signatures or contexts at the leaves, but require their validity in the theorems and take care to propagate this property. Where there is no ambiguity we use the usual conventions for the names of meta-variables. Here, S stands for either Type or Kind.

$$\frac{}{\Gamma \vdash_\Sigma \mathrm{Type} : \mathrm{Kind}} \qquad \frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : A}$$

$$\frac{d{:}U \text{ in } \Sigma}{\Gamma \vdash_\Sigma d : U} \qquad \frac{a :: b \text{ in } \Sigma \qquad b{:}K \text{ in } \Sigma}{\Gamma \vdash_\Sigma a : K}$$

$$\frac{\Gamma \vdash_\Sigma U : V_1 \qquad \Gamma \vdash_\Sigma U : V_2}{\Gamma \vdash_\Sigma U : V_1 \,\&\, V_2}(1) \qquad \frac{\Gamma \vdash_\Sigma U : V \qquad \vdash_\Sigma V \leq W \qquad \Gamma \vdash_\Sigma W : S}{\Gamma \vdash_\Sigma U : W}(2)$$

$$\frac{\Gamma \vdash_\Sigma A : \mathrm{Type} \qquad \Gamma, x{:}A \vdash_\Sigma U : S}{\Gamma \vdash_\Sigma \Pi x{:}A.\ U : S} \qquad \frac{\Gamma \vdash_\Sigma U_1 : S \quad \Gamma \vdash_\Sigma U_2 : S \quad \vdash_\Sigma U_1 :: V \quad \vdash_\Sigma U_2 :: V}{\Gamma \vdash_\Sigma U_1 \,\&\, U_2 : S}$$

$$\frac{\Gamma \vdash_\Sigma U : \Pi x{:}A.\ V \qquad \Gamma \vdash_\Sigma M : A}{\Gamma \vdash_\Sigma U\,M : [M/x]V}$$

$$\frac{\vdash_\Sigma B :: A \qquad \Gamma \vdash_\Sigma A : \mathrm{Type} \qquad \Gamma \vdash_\Sigma B : \mathrm{Type} \qquad \Gamma, x{:}B \vdash_\Sigma M : C}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M : \Pi x{:}B.\ C}(3)$$

$$\frac{\Gamma \vdash_\Sigma U : V \qquad V \equiv W \qquad \Gamma \vdash_\Sigma W : S}{\Gamma \vdash_\Sigma U : W}(4)$$

Note that we need a subsorting rule (2) *and* a type conversion rule (4), since we have formulated them as separate judgments which interact very little (formally). In the rule for λ-abstraction (3) one can see that the type label acts as a bound: we can analyze the expression for each sort B which refines A and conjoin the results using the introduction rule for & (1).

Finally, the rules for subsorting. The rules enforce the restriction that sorts and sort families can only be compared if they refine a common type.

$$\frac{\vdash_\Sigma U :: W \qquad \vdash_\Sigma V :: W}{\vdash_\Sigma U \,\&\, V \leq U} \qquad \frac{\vdash_\Sigma U :: W \qquad \vdash_\Sigma V :: W}{\vdash_\Sigma U \,\&\, V \leq V}$$

$$\frac{\vdash_\Sigma U \leq V_1 \qquad \vdash_\Sigma U \leq V_2}{\vdash_\Sigma U \leq V_1 \,\&\, V_2} \qquad \frac{a \leq b \text{ in } \Sigma}{\vdash_\Sigma a \leq b}$$

$$\frac{\vdash_\Sigma \Pi x{:}A.\ U_1 :: W \qquad \vdash_\Sigma \Pi x{:}A.\ U_2 :: W}{\vdash_\Sigma (\Pi x{:}A.\ U_1) \,\&(\Pi x{:}A.\ U_2) \leq (\Pi x{:}A.\ U_1 \,\&\, U_2)}$$

$$\frac{}{\vdash_\Sigma \text{Type} \leq \text{Type}} \qquad \frac{\vdash_\Sigma B \leq A \qquad \vdash_\Sigma U \leq V}{\vdash_\Sigma \Pi x{:}A.\ U \leq \Pi x{:}B.\ V} \qquad \frac{\vdash_\Sigma A \leq B}{\vdash_\Sigma A\, M \leq B\, M}$$

$$\frac{\vdash_\Sigma U :: W}{\vdash_\Sigma U \leq U} \qquad \frac{\vdash_\Sigma U \leq V \qquad \vdash_\Sigma V \leq W}{\vdash_\Sigma U \leq W}$$

The subsorting relationship is contravariant in the domain of a function type, as expected. Indexed sort families may only be compared if the indices are identical, which may require some applications of the type conversion rule (4) in a typing derivation before the subsumption rule (2) can be applied.

3.3 Properties of $\lambda^{\Pi\&}$

We begin by defining a forgetful mapping $\|.\|$ from $\lambda^{\Pi\&}$ to λ^{Π}. It ignores the distinctions introduced by sorts by collapsing them to the type they refine. The result of interpreting a signature Σ is a signature Σ' in λ^{Π} and a substitution σ mapping terms over Σ into terms over Σ'. We use $\sigma(U)$ as a notation for the result of applying σ to U with the special provision that

$$\sigma(U_1 \,\&\, U_2) = \begin{cases} V & \text{if } \sigma(U_1) = \sigma(U_2) = V \\ \text{undefined} & \text{otherwise} \end{cases}$$

The application of σ to a context Γ distributes into the constituent terms. The empty substitution is denoted by $[]$ and the extension of a substitution σ mapping the new constant d to d' is written as $\sigma \oplus [d \mapsto d']$.

$$\begin{array}{rcll}
\| \cdot \| & = & \langle \cdot; [] \rangle & \\
\|\Sigma, d{:}U\| & = & \|\Sigma\| & \text{if } d \text{ declared in } \Sigma \\
\|\Sigma, d{:}U\| & = & \langle \Sigma', d{:}\sigma(U); \sigma \oplus [d \mapsto d] \rangle & \text{if } d \text{ not declared in } \Sigma \text{ and } \langle \Sigma'; \sigma \rangle = \|\Sigma\| \\
\|\Sigma, a_1 :: a_2\| & = & \langle \Sigma'; \sigma \oplus [a_1 \mapsto a_2] \rangle & \text{where } \langle \Sigma'; \sigma \rangle = \|\Sigma\| \\
\|\Sigma, a_1 \leq a_2\| & = & \|\Sigma\| &
\end{array}$$

Lemma 1 *If Σ is valid and $\vdash_\Sigma U \leq V$ then there exists a (unique) W such that $\vdash_\Sigma U :: W$ and $\vdash_\Sigma V :: W$.*

Lemma 2 (Refinement) *Let Σ be a valid signature, Γ be a valid context, and $\|\Sigma\| = \langle \Sigma'; \sigma \rangle$. Then:*

(i) *if $\vdash_\Sigma U :: V$ then $\sigma(U) = \sigma(V)$,* *(ii)* *if $\vdash_\Sigma U \leq V$ then $\sigma(U) = \sigma(V)$,*
(iii) *if $U \equiv V$ then $\sigma(U) \equiv \sigma(V)$,* *(iv)* *if $\Gamma \vdash_\Sigma U : V$ then $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(U) : \sigma(V)$.*

Proof: By straightforward inductions over the derivations of the given judgments, employing uniqueness of bounds and Lemma 1. □

We call a $\lambda^{\Pi\&}$ term *canonical* if it is in long $\beta\eta$-normal form, as in LF.

Lemma 3 *The judgment $U \equiv V$ is decidable on valid terms and every valid term U has a unique equivalent canonical form.*

Proof sketch: The corresponding judgment on LF is decidable on valid LF terms (see, for example, [Geu92]). Equivalence on types and kinds is structural and therefore trivially decidable, except for conversions among the embedded objects. But labels of λ-abstractions are restricted to terms which remain unchanged under the forgetful interpretation, and thus conversions in $\sigma(U)$ and $\sigma(V)$ can be lifted to conversions in U and V. □

The equivalence relation $\cong$ is defined by $U \cong V$ iff $U \leq V$ and $V \leq U$. It is easily shown that this is a congruence. Also, the following properties are easily proved.

Lemma 4 (Basic Properties of Sorts) *We assume implicitly that both sides of each of the equivalences below refine the same type.*

$$\begin{array}{rlrl} (i) & U \,\&\, V \cong V \,\&\, U, & (ii) & U \,\&(V \,\&\, W) \cong (U \,\&\, V) \,\&\, W, \\ (iii) & U \,\&\, U \cong U, & (iv) & (\Pi x{:}A.\, U_1) \,\&(\Pi x{:}A.\, U_2) \cong (\Pi x{:}A.\, U_1 \,\&\, U_2). \end{array}$$

Theorem 5 (Decidability of Subsorting) *The subsorting judgment $\vdash_\Sigma U \leq V$ is decidable for valid signatures Σ.*

Proof sketch: By an interpretation into the subtyping problem for Forsythe, for which a decidability proof has been given by Reynolds [personal communication, 1991]. The proof can be found in [Pie91] in a slightly different form. Each atomic type of the form $a\, M_1 \ldots M_n$ is interpreted as a simple type $\overline{a\, M_1 \ldots M_n}$ which inherits its subsorting property from a. The main observation in the correctness proof of this interpretation is that $A\, M \leq B\, N$ iff $A \leq B$ and $M = N$. □

We call a type A a *minimal type* for the object M in context Γ if A is canonical and for every canonical B such that $\Gamma \vdash_\Sigma M : B$ we have $\vdash_\Sigma A \leq B$. A similar definition applies to minimal kinds.

Theorem 6 (Decidability of $\lambda^{\Pi\&}$) *The validity of signatures and contexts and the typing judgment $\Gamma \vdash_\Sigma U : V$ are decidable. Furthermore, every valid term U has a minimal type or kind.*

Proof sketch: Using the forgetful interpretation and the soundness and completeness of the algorithmic version of LF in [HHP93] we can show that each derivation can be transformed into one which eagerly applies normalization on types, but otherwise requires no type conversion. Secondly we show that applications of the subsorting rule in such a derivation can be pushed up to the leaves, except for λ-abstractions and applications, where we can directly calculate a minimal type from minimal types of the constituents. The completeness of this calculation relies on the fact that only finitely many sorts (modulo $\cong$) refine a given type. □

4 Examples Revisited

Now that the $\lambda^{\Pi\&}$ calculus has been defined, we revisit the earlier examples. We use the concrete syntax `::` for :: and `<:` for $\leq$.

Hereditary Harrop formulas. Following the previous and unchanged definitions of the connectives, we declare atoms, goals, and programs as refinements of formulas. Then we declare sorts for the constructors.

```
atom :: form.  % atoms
goal :: form.  % legal goals
prog :: form.  % legal programs

atom <: goal.  % every atom is a legal goal

=>   : prog -> goal -> goal.
||   : goal -> goal -> goal.
&&   : goal -> goal -> goal.

atom <: prog.  % every atom is a legal program

=>   : goal -> prog -> prog.
&&   : prog -> prog -> prog.
```

The entailment and backchaining judgments can now be declared naturally. Their definition (not shown here) is also simple and intuitive.

```
solve     : prog -> goal -> type.
backchain : prog -> atom -> goal -> type.
```

Normal Natural Deductions. Here, both *elim* and *nf* become sort families which refine *pf*. Following the previous declarations for *pf*, *impi*, and *impe* we complete the definition as follows.

```
nf   :: pf.  % normal form deductions
elim :: pf.  % pure elimination deductions from hypotheses

elim <: nf.  % every elim deduction is in normal form

impi : (elim A -> nf B) -> nf (imp A B).
impe : elim (imp A B) -> nf A -> elim B.
```

Below we show the obvious deduction of $p \supset (q \supset p)$ for parameters p and q. Terms of the form $\lambda x{:}A.\ M$ are written as `[x:A] M` in concrete syntax.

```
  ([p:o] [q:o] impi ([P:pf p] impi ([Q:pf q] P))
: {p:o} {q:o} nf (imp p (imp q p)).
```

These small examples should help to illustrate how refinement types provide a natural and direct means to express subtyping in the context of a logical framework. Many of the case studies of deductive systems in LF that we and others have carried out would benefit similarly.

5 Conclusion and Further Work

We plan to implement the system $\lambda^{\Pi\&}$ as an extension of Elf. However, before this can be done, a unification algorithm and a feasible type reconstruction algorithm need to be developed. The type-checking algorithm which arises out of the proof of Theorem 6 works by bottom-up synthesis and is not practical. However, a top-down type-checking algorithm as in the implementation of refinement types for ML [FP91] promises to be of acceptable efficiency, especially since our language lacks recursion at the level of terms.

We would also like to consider relaxing some of the restrictions currently in place to enforce orthogonality of conversion and subsorting. In particular, it is intuitively appealing to allow sorts (to be interpreted as bounds) in the labels of λ-abstractions, but we believe that this necessitates a form of typed or sorted conversion and our decidability proof no longer applies. One might also consider promotion of sorts to types and demotion of types to sorts which sometimes further economizes representations without making them less intuitive.

Finally, there is the question of adequacy proofs for representations in $\lambda^{\Pi\&}$. The normal form theorem is useful here, but we would also like to describe an interpretation which maps a signature in $\lambda^{\Pi\&}$ into an equivalent signature in λ^{Π}. We conjecture that there is such a mapping which interprets refinement by relativizing Π-quantifiers and subsorting by coercions.

Acknowledgments. We would like to thank Michael Kohlhase for discussions regarding refinement types and Nevin Heintze and Ekkehard Rohwedder for comments on a draft of this extended abstract.

References

[CDCV81] Mario Coppo, Maria Dezani-Ciancaglini, and B. Venneri. Functional character of solvable terms. *Zeitschrift für mathematische Logic und Grundlagen der Mathematik*, 27:45–58, 1981.

[CG92] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In J.-C. Raoult, editor, *17th Colloquium on Trees in Algebra and Programming, Rennes, France*, pages 102–123, Berlin, February 1992. Springer-Verlag LNCS 581.

[Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989. Available as Technical Report MS-CIS-89-53.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[Geu92] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Hay91] Susumu Hayashi. Singleton, union and intersection types for program extraction. In T. Ito and A. R. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Software*, pages 701–730, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.

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

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Koh91] Michael Kohlhase. Order-sorted type theory I: Unification. SEKI Report SR-91-18, Universität des Saarlandes, Saarbrücken, Germany, 1991.

[Koh92] Michael Kohlhase. Unification in order-sorted type theory. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 421–432, St. Petersburg, Russia, July 1992. Springer-Verlag LNAI 624.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[NQ92] Tobias Nipkow and Zhenyu Qian. Reduction and unification in lambda calculi with subtypes. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 66–78, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[Pfe91a] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe91b] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1991. Available as Technical Report CMU–CS–91–205.

[PN90] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, Computer Laboratory, University of Cambridge, January 1990.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software*, pages 675–700, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* Dissertation, Universität Kaiserslautern, May 1989.

[SS88] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland*, pages 384–391. IEEE, July 1988.

[SS89] Manfred Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations.* Springer-Verlag LNAI 395, 1989.