

Unification in a λ -Calculus with Intersection Types

Michael Kohlhase

FB Informatik
Universität des Saarlandes
W-6600 Saarbrücken, Germany
kohlhase@cs.uni-sb.de

Frank Pfenning

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
fp@cs.cmu.edu

Abstract

We propose related algorithms for unification and constraint simplification in $\lambda^{\rightarrow\&}$, a refinement of the simply-typed λ -calculus with subtypes and bounded intersection types. $\lambda^{\rightarrow\&}$ is intended as the basis of a logical framework in order to achieve more succinct and declarative axiomatizations of deductive systems than possible with the simply-typed λ -calculus. The unification and constraint simplification algorithms described here lay the groundwork for a mechanization of such frameworks as constraint logic programming languages and theorem provers.

1 Introduction

The motivation for our work comes from the area of *logical frameworks*. A logical framework is a meta-language for the specification and implementation of deductive systems as they arise in logic and the study of programming languages. Examples of such frameworks are LF [?], hereditary Harrop formulae [6], and ALF [?]. All these frameworks are based on some type theory. They have been used as the basis for the logic-independent theorem prover Isabelle [8] and the logic programming languages λ Prolog [7] and Elf [9]. Extensive experiments in logic and the theory of programming languages have been carried out in these implementations.

In a recent paper [?] the second author has proposed a refinement of the

type theory λ^{Π} underlying the LF logical framework in order to simplify the presentation of many deductive systems. This refinement, $\lambda^{\Pi\&}$, incorporates subtypes and intersection types. In addition to a more natural reflection of informal mathematical practice in the specifications of languages and deductive systems, we also believe that refinement types can be beneficial to execution in logic programming languages such as λ Prolog or Elf and to search in theorem provers such as Isabelle. In first-order languages the potential of order-sorted type structures have long been realized (see, for example, [10, ?, ?]) and we see our work as a natural extension of these efforts.

At the heart of theorem proving or logic programming lies unification. In this paper we present 3 related algorithms for unification and constraint simplification for $\lambda^{\rightarrow\&}$, a refinement of the simply-typed λ -calculus with subtypes and bounded intersection types. It builds upon Huet’s algorithm for the simply-typed λ -calculus [3] and extensions to related languages by Nipkow and Qian [?] and the first author [4, 5], although the systems are incomparable in terms of their expressive power. The motivating example below should help to illustrate the differences. A λ -calculus with simple subtypes as considered by [?] contains no intersections and type labels on λ -abstractions are not interpreted as bounds. This means that the necessary sort computations for our algorithm are significantly more complex. The system considered by the first author in [5] permits so-called term declarations which lead to an undecidable type-checking problem and is thus in some ways more general. On the other hand, in order to model intersection types, one would have to add an infinite collection of new types and infinitely many new term declarations to a given signature.

The rest of this paper is structured as follows. We begin with a motivating example in the context of λ Prolog, followed by the definition of $\lambda^{\rightarrow\&}$. We then discuss general (pre)-unification in $\lambda^{\rightarrow\&}$. We conclude the paper by presenting transformations for unification restricted to higher-order patterns in the sense of Miller [?].

2 A Motivating Example

The value of subsorting (sometimes called subtyping) for first-order logic programming languages has long been recognized and extensively investigated (see, for example, [10, ?, ?]). Type systems with subsorts afford a concise, yet declaratively correct formulation of many programs. Furthermore, many programming errors manifest themselves as type errors at

compile-time. Current higher-order logic programming languages with a term language including λ -abstractions (such as λ Prolog or Elf) are based on simple, polymorphic, or dependent type disciplines, but do not incorporate a notion of subtyping. This can be traced in part to a lack of a uniform and accurate type system that combines function types (which classify λ -abstractions) with subtyping. As a motivating example we consider the classification of legal goals and programs in λ Prolog. We begin with the language of formulae.

$$\begin{array}{l} \textit{Simple Types} \quad A ::= a \mid o \mid A_1 \rightarrow A_2 \\ \textit{Formulae} \quad F ::= At \mid F_1 \wedge F_2 \mid F_1 \supset F_2 \mid F_1 \vee F_2 \mid \forall x: A.F \mid \exists x: A.F \end{array}$$

Here o is the distinguished type for formulae and a stands for explicitly declared type constants. An atom At has the normal form $h M_1 \dots M_n$ for a variable or non-logical constant h , where the type of the whole expression must be o . Programs and goals must be restricted in order to guarantee that the *uniform proof property* is satisfied for the resulting logic and goal-directed search will be complete. The restriction we show here is to higher-order hereditary Harrop formulae [6]. The notation At_r stands for rigid atoms, that is, atoms whose head must be a constant.

$$\begin{array}{l} \textit{Programs} \quad D ::= At_r \mid D_1 \wedge D_2 \mid G \supset D \mid \forall x: A.D \\ \textit{Goals} \quad G ::= At \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G \mid \exists x: A.G \mid \forall x: A.G \end{array}$$

There is one further restriction: subterms of At_r may not contain implications. If this restriction were relaxed, then for constants q and r of type o and p of type $o \rightarrow o$, the goal $pG \wedge G$ has a proof given the program

$$p((q \vee r) \supset (r \vee q)),$$

but it has no uniform proof. The difficulty is that the subterm $(q \vee r) \supset (r \vee q)$ is not a legal goal, but becomes a goal after the instantiation of the variable G . Unfortunately this restriction also rules out programs such as meta-interpreters which cannot lead to a failure of the uniform proof property. In order to circumvent this problem, we would like to distinguish legal goals g and legal programs d as *refinements* of the type o of formulae. This leads to the following refinement declarations

$$\begin{array}{l} g ::= o \\ d ::= o \end{array}$$

But what then is the type of conjunction, for example? For one, it maps two goals to a goal, but is also maps two programs to a program. Of course, it

also still maps two arbitrary formulae to a formula. Similar considerations apply to implication and we have

$$\begin{array}{ll}
\wedge : o \rightarrow o \rightarrow o & \supset : o \rightarrow o \rightarrow o \\
\wedge : g \rightarrow g \rightarrow g & \supset : d \rightarrow g \rightarrow g \\
\wedge : d \rightarrow d \rightarrow d & \supset : g \rightarrow d \rightarrow d
\end{array}$$

The declarations for the remaining logical constants can now easily be added; we omit them here for the sake of brevity. Multiple typings are not limited to constants: the function $\lambda x: o. \lambda y: o. y \wedge x$, for example, should intuitively have precisely the same types as \wedge . In order to see this consider the result of applying this function to two goals G_1 and G_2 : the result will always be a goal (namely $G_2 \wedge G_1$). But this fact cannot be expressed in a system of simple subtypes (such as the one considered in [?]): we need to add intersection types, written as $A_1 \& A_2$. With the system we propose below we can infer, for example,

$$\begin{array}{l}
(\lambda x: o. \lambda y: o. y \wedge x) : (o \rightarrow o \rightarrow o) \& (g \rightarrow g \rightarrow g) \& (d \rightarrow d \rightarrow d) \\
(\lambda x: o. \lambda y: o. y \supset x) : (o \rightarrow o \rightarrow o) \& (g \rightarrow d \rightarrow g) \& (d \rightarrow g \rightarrow d)
\end{array}$$

In contrast to other calculi, the type labels on λ -abstractions here must be considered as bounds on the sorts of possible instantiations of the variable. Thus, in order to type-check a λ -expression $\lambda x: A. M$ we must analyze M for every subtype of A . In order to guarantee that this process remains finitary and principal types exist, we distinguish between *proper types* and *sorts*. Proper types (such as o in our example) divide terms into disjoint collections. Sorts (such as g and d) refine the type structure by classifying terms (which must already possess the same proper type) more accurately than is possible with proper types alone. We thus refer to this type system as a system of *refinement types*. Note that sorts may not necessarily be disjoint. For example, every (rigid) atom is both a legal goal and a legal program. Thus a new predicate *pred* on integers should be declared with

$$pred : int \rightarrow (g \& d)$$

or we could add another sort a of atoms and declare it a subsort of both d and g :

$$\begin{array}{lll}
a :: o & a \leq g & a \leq d \\
pred : int \rightarrow a
\end{array}$$

With this type structure, we can now safely program with higher-order predicates without undue restrictions. The counterexample above now fails, since

G is a variable of sort g and the argument $(q \vee r) \supset (r \vee q)$ does not have sort g (only type o). On the other hand, safe usage in a meta-interpreter such as

$$\forall D: d. \forall G: g. (\text{hyp}(D) \supset \text{solve}(G)) \supset \text{solve}(D \supset G)$$

is permitted if we have the typings $\text{solve}: g \rightarrow g$ and $\text{hyp}: d \rightarrow d$.

We have many other examples where refinement types are beneficial in higher-order logic programming. For example, in the higher-order representation of natural deductions [1] one can distinguish normal forms as a refinement of arbitrary derivations instead of explicitly encoding two different representations. In the implementation of functional languages [?] refinement types can distinguish values from arbitrary expressions instead of leaving this distinction implicit. The interested reader is referred to [?] for further examples and discussion.

3 Basic Definitions

The syntax of $\lambda^{\rightarrow \&}$ is that of the simply-typed λ -calculus augmented with the intersection operator $\&$ for types. The main change in the language concerns signatures, where we drop the restriction that each constant be declared at most once. We furthermore add *refinement declarations* $a_1 :: a_2$ which declares sort a_1 as a refinement of type a_2 and *subsort declarations* $a_1 \leq a_2$ which declares that a_1 is a subsort of a_2 . The inference rules for valid signatures guarantee certain consistency properties between multiple declarations.

$$\begin{array}{ll} \text{Types} & A ::= a \mid A_1 \rightarrow A_2 \mid A_1 \& A_2 \\ \text{Objects} & M ::= c \mid x \mid \lambda x: A. M \mid M_1 M_2 \\ \text{Contexts} & \Gamma ::= \cdot \mid \Gamma, x: A \\ \text{Signatures} & \Sigma ::= \cdot \mid \Sigma, a: \text{Type} \mid \Sigma, c: A \mid \Sigma, a_1 :: a_2 \mid \Sigma, a_1 \leq a_2 \end{array}$$

We use a and b to range over type constants, c to range over object constants, and x , y , and z to range over object variables. We also restrict contexts so that each variable is declared at most once. Since we also identify α -convertible terms, this does not essentially restrict the inference rules below. We will call $A \& B$ the *intersection* of A and B , but refer to A and B as its *conjuncts*.

Our system is more restrictive than customary formulations of intersection types (see, for example, [?, ?, ?]). The validity judgments below introduce a distinction between *proper types* and *sorts*. Proper types behave

essentially like simple types and do not contain intersections. Sorts further refine proper types by enabling a more precise classification of terms, but sorts can only be intersected or compared if they refine the same proper type. In the context of a functional language as in [2], this leads to a decidable type inference problem. Here we are more concerned with the fact that the adequacy of representations in the logical framework is preserved. For this it is vital that we do not extend the language of λ -terms, but only the language of types that classify them. Thus the type labels in λ -abstractions are restricted to proper types. For type-checking, a type label A acts as a bound and the body of the λ -term is analyzed for each sort B that refines A . By the restrictions sketched above only finitely many such sorts B exist up to a simple syntactic equivalence. For further discussion and some examples the interested reader is referred to [?].

3.1 Judgments

The validity judgments have the following form. Here, `Type` is a special token to allow a uniform presentation of the validity judgments for types and objects.

$$\begin{array}{ll}
\vdash \Sigma \text{ Sig} & \Sigma \text{ is a valid signature} \\
\vdash_{\Sigma} \Gamma \text{ Ctx} & \Gamma \text{ is a valid context} \\
\vdash_{\Sigma} A: \text{Type} & A \text{ is a valid type} \\
\Gamma \vdash_{\Sigma} M: A & M \text{ is a valid object of type } A
\end{array}$$

We also need some auxiliary judgments. In particular,

$$\begin{array}{ll}
\vdash_{\Sigma} A :: B & A \text{ refines } B \\
\vdash_{\Sigma} A \leq B & A \text{ is a subsort of } B
\end{array}$$

We begin with the *refinement judgment* for types.

$$\frac{\vdash_{\Sigma} A_1 :: B_1 \quad \vdash_{\Sigma} A_2 :: B_2}{\vdash_{\Sigma} A_1 \rightarrow A_2 :: B_1 \rightarrow B_2} \qquad \frac{\vdash_{\Sigma} A_1 :: B \quad \vdash_{\Sigma} A_2 :: B}{\vdash_{\Sigma} A_1 \& A_2 :: B}$$

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

Note that the refinement relation is neither transitive nor reflexive. The conditions on valid signatures will guarantee that exactly one of the last

two rules 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 A there exists at most one B such that $\vdash_{\Sigma} A :: B$. We call a type A such that $\vdash_{\Sigma} A :: A$ a *proper type*.

The next set of rules defines the *valid signatures*.

$$\begin{array}{c}
\frac{}{\vdash \cdot \text{Sig}} \qquad \frac{\vdash \Sigma \text{ Sig} \quad a \text{ not declared in } \Sigma}{\vdash \Sigma, a: \text{Type Sig}} \\
\\
\frac{\vdash \Sigma \text{ Sig} \quad \vdash_{\Sigma} A: \text{Type} \quad \vdash_{\Sigma} A :: A' \quad \vdash_{\Sigma} A_i :: A' \text{ for every } c: A_i \text{ in } \Sigma}{\vdash \Sigma, c: A \text{ Sig}} \quad (1) \\
\\
\frac{\vdash \Sigma \text{ Sig} \quad a_2: \text{Type in } \Sigma \quad a_1 \text{ not declared in } \Sigma}{\vdash \Sigma, a_1 :: a_2 \text{ Sig}} \\
\\
\frac{\vdash \Sigma \text{ Sig} \quad a_1 :: a_3 \text{ in } \Sigma \quad a_2 :: a_3 \text{ in } \Sigma}{\vdash \Sigma, a_1 \leq a_2 \text{ Sig}}
\end{array}$$

The rule (1) for constant declarations enforces that in a valid signature, all types A_i declared for a given constant c refine the same proper type A' . *Valid contexts* are straightforward, just as in the simply-typed λ -calculus.

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

The rules for *valid types* enforce that all type constants are declared and that sorts can only be intersected if they refine a common proper type.

$$\frac{a: \text{Type in } \Sigma}{\vdash_{\Sigma} a: \text{Type}} \quad \frac{a :: b \text{ in } \Sigma}{\vdash_{\Sigma} a: \text{Type}} \quad \frac{\vdash_{\Sigma} A_1: \text{Type} \quad \vdash_{\Sigma} A_2: \text{Type}}{\vdash_{\Sigma} A_1 \rightarrow A_2: \text{Type}} \\
\\
\frac{\vdash_{\Sigma} A_1: \text{Type} \quad \vdash_{\Sigma} A_2: \text{Type} \quad \vdash_{\Sigma} A_1 :: B \quad \vdash_{\Sigma} A_2 :: B}{\vdash_{\Sigma} A_1 \& A_2: \text{Type}}$$

Subtyping is contravariant in the domain sort, as expected. The rules guarantee that we can only compare sorts that refine a common proper type.

$$\begin{array}{c}
\frac{a \leq b \text{ in } \Sigma}{\vdash_{\Sigma} a \leq b} \quad \frac{\vdash_{\Sigma} A_1 :: B \quad \vdash_{\Sigma} A_2 :: B}{\vdash_{\Sigma} A_1 \& A_2 \leq A_1} \quad \frac{\vdash_{\Sigma} A_1 :: B \quad \vdash_{\Sigma} A_2 :: B}{\vdash_{\Sigma} A_1 \& A_2 \leq A_2} \\
\\
\frac{\vdash_{\Sigma} A \leq B_1 \quad \vdash_{\Sigma} A \leq B_2}{\vdash_{\Sigma} A \leq B_1 \& B_2} \quad \frac{\vdash_{\Sigma} A \rightarrow B_1 :: C \quad \vdash_{\Sigma} A \rightarrow B_2 :: C}{\vdash_{\Sigma} (A \rightarrow B_1) \& (A \rightarrow B_2) \leq A \rightarrow (B_1 \& B_2)} \\
\\
\frac{\vdash_{\Sigma} B_1 \leq A_1 \quad \vdash_{\Sigma} A_2 \leq B_2}{\vdash_{\Sigma} A_1 \rightarrow A_2 \leq B_1 \rightarrow B_2} \\
\\
\frac{\vdash_{\Sigma} A :: B}{\vdash_{\Sigma} A \leq A} \quad \frac{\vdash_{\Sigma} A \leq B \quad \vdash_{\Sigma} B \leq C}{\vdash_{\Sigma} A \leq C}
\end{array}$$

We introduce a partial equivalence relation \sim on types by defining $\vdash_{\Sigma} A \sim B$ as an abbreviation for $\vdash_{\Sigma} A \leq B$ and $\vdash_{\Sigma} B \leq A$. It is easy to verify that (with respect to a valid signature) any proper type has only finitely many refinements up to \sim equivalence.

Lemma 3.1 (Basic Properties of Sorts) *We implicitly assume that both sides of each of the equivalences below refine the same type.*

- (i) $A \& B \sim B \& A$, (ii) $A \& (B \& C) \sim (A \& B) \& C$,
- (iii) $A \& A \sim A$, (iv) $(A \rightarrow B) \& (A \rightarrow C) \sim A \rightarrow B \& C$.

In the rules for *valid objects* we see that the type label of a λ -abstraction must be a proper type and that the body of the λ -expression may be analyzed

for every sort which refines this type.

$$\begin{array}{c}
\frac{x: A \text{ in } \Gamma}{\Gamma \vdash_{\Sigma} x: A} \qquad \frac{c: A \text{ in } \Sigma}{\Gamma \vdash_{\Sigma} c: A} \\
\\
\frac{\Gamma \vdash_{\Sigma} M: A_1 \quad \Gamma \vdash_{\Sigma} M: A_2}{\Gamma \vdash_{\Sigma} M: A_1 \& A_2} \qquad \frac{\Gamma \vdash_{\Sigma} M: A \quad \vdash_{\Sigma} A \leq B}{\Gamma \vdash_{\Sigma} M: B} \\
\\
\frac{\Gamma \vdash_{\Sigma} M_1: A_2 \rightarrow A_1 \quad \Gamma \vdash_{\Sigma} M_2: A_2}{\Gamma \vdash_{\Sigma} M_1 M_2: A_1} \qquad \frac{\vdash_{\Sigma} B :: A \quad \Gamma, x: B \vdash_{\Sigma} M: C}{\Gamma \vdash_{\Sigma} \lambda x: A.M: B \rightarrow C}
\end{array}$$

3.2 Algorithmic Judgments

The judgments given above are declarative and it is not immediately obvious, for example, if the subsorting or typing judgments are decidable. Following Pierce [?], we formulate new versions of these judgments which directly embody an algorithm for deciding subsorting and synthesizing a minimal type for an object.

We start with the algorithmic version of the subtype judgment, $\vdash_{\Sigma} A \sqsubseteq B$. It requires an auxiliary operator \bullet on types that is used to uncurry function types.

$$\begin{array}{c}
\frac{}{\vdash_{\Sigma} a \sqsubseteq a} \qquad \frac{a \leq a' \text{ in } \Sigma \quad \vdash_{\Sigma} a' \sqsubseteq b}{\vdash_{\Sigma} a \sqsubseteq b} \\
\\
\frac{\vdash_{\Sigma} A_1 \sqsubseteq B \rightarrow a}{\vdash_{\Sigma} A_1 \& A_2 \sqsubseteq B \rightarrow a} \qquad \frac{\vdash_{\Sigma} A_2 \sqsubseteq B \rightarrow a}{\vdash_{\Sigma} A_1 \& A_2 \sqsubseteq B \rightarrow a} \\
\\
\frac{\vdash_{\Sigma} A \sqsubseteq B \bullet C_1 \rightarrow C_2}{\vdash_{\Sigma} A \sqsubseteq B \rightarrow (C_1 \rightarrow C_2)} \qquad \frac{\vdash_{\Sigma} B_1 \sqsubseteq A_1 \quad \vdash_{\Sigma} A_2 \sqsubseteq B_2 \rightarrow a}{\vdash_{\Sigma} A_1 \rightarrow A_2 \sqsubseteq B_1 \bullet B_2 \rightarrow a} \\
\\
\frac{\vdash_{\Sigma} A \sqsubseteq B \rightarrow C_1 \quad \vdash_{\Sigma} A \sqsubseteq B \rightarrow C_2}{\vdash_{\Sigma} A \sqsubseteq B \rightarrow C_1 \& C_2}
\end{array}$$

Theorem 3.2 *The judgment $\vdash_{\Sigma} A \sqsubseteq B$ is effectively decidable. Furthermore, if A and B are types not containing the \bullet operator such that $\vdash_{\Sigma} A :: C$ and $\vdash_{\Sigma} B :: C$ for some C , then $\vdash_{\Sigma} A \leq B$, iff $\vdash_{\Sigma} A \sqsubseteq B$.*

Proof: By an interpretation into Pierce's system [?]. \square

The second judgment expresses that M has *minimal* type A , written as $\Gamma \vdash_{\Sigma} M \in A$. For the purposes of this system and the remainder of the paper, it is convenient to treat intersection as an operator on multiple arguments and occasionally a set of arguments. This is admissible in view of the basic properties of $\&$ (cf. Lemma 3.1).

$$\frac{x: A \text{ in } \Gamma}{\Gamma \vdash_{\Sigma} x \in A} \qquad \frac{}{\Gamma \vdash_{\Sigma} c \in \&\{A|c: A \text{ in } \Sigma\}}$$

$$\frac{\Gamma \vdash_{\Sigma} M_1 \in \&_i(B_i \rightarrow C_i) \quad \Gamma \vdash_{\Sigma} M_2 \in A}{\Gamma \vdash_{\Sigma} M_1 M_2 \in \&\{C_i|\vdash_{\Sigma} A \sqsubseteq B_i\}}$$

$$\frac{}{\Gamma \vdash_{\Sigma} \lambda x: A.M \in \&\{B \rightarrow C|\vdash_{\Sigma} B :: A; \Gamma, x: B \vdash_{\Sigma} M \in C\}}$$

The intersection operator applied to an empty set is undefined. The last rule could lead to an infinite intersection, but there are only finitely many refinements of a proper type up to \sim . Thus only finitely many conjuncts contribute to the intersection and we can operationalize the rule by assuming a fixed algorithm for enumerating refinements of a proper type. This inference system is now syntax-directed, and it is therefore immediate that the judgment $\Gamma \vdash_{\Sigma} M \in A$ is decidable.

Theorem 3.3 *Given a valid signature Σ , a context Γ valid in Σ and types A and B valid in Σ . Then $\Gamma \vdash_{\Sigma} M: A$ iff $\Gamma \vdash_{\Sigma} M \in B$ and $\vdash_{\Sigma} B \leq A$.*

Proof: Again, via an interpretation into the system of Pierce. It is crucial for this interpretation that the number of \sim -equivalence classes of sorts refining a proper type is finite. \square

Lemma 3.4 *Let $\vdash_{\Sigma} A :: A_1 \rightarrow A_2$, then $\Gamma \vdash_{\Sigma} M: A$, iff $\Gamma \vdash_{\Sigma} \lambda x: A_1.Mx: A$.*

Proof: Via completeness of the algorithmic judgments. \square

We will write $M \equiv N$, if M and N are convertible by $\beta\eta$ -conversions.

Lemma 3.5 (Normal Form Lemma) *Let M be a term such that $\Gamma \vdash_{\Sigma} M: C$. Then there is a long normal form $N = \lambda x_1: F_1 \dots x_n: F_n \cdot h N_1 \dots N_m$, such that h is a constant or variable and $M \equiv N$. As usual we call h the head of M .*

4 General Bindings and Type Constraints

The notion of a general binding is central to all higher-order unification algorithms. In contrast to the simply typed λ -calculus, general bindings for terms in $\lambda^{\rightarrow \&}$ are *not* unique up to the choice of the new variables. Therefore we obtain additional nondeterminism in the imitation and projection steps. However, in contrast to the case with full term declarations [5], we have that type-erasures of all general bindings are unique and the types of the new variables only depend on the types of the binding and its head. Therefore we will handle the nondeterminism by introducing type variables (which we denote by α) for the unification algorithm and delay the computation of the actual type information for the new variables into type constraints. In order to simplify the notation we write \overline{A} for the proper type B such that $\vdash_{\Sigma} A :: B$. We also assume in the following that all types are valid, thus \overline{A} always exists and is unique.

Definition 4.1 (General Binding) Let h be a constant or variable with $\Gamma \vdash_{\Sigma} h \in A$ and $C \sim \&_{j \leq l} C_{1j} \longrightarrow \dots \longrightarrow C_{kj} \longrightarrow D_j$ a sort. Then the *general binding G of type C with the head h* is the term

$$G = \lambda x_1: \overline{C_{1j}} \dots \lambda x_k: \overline{C_{kj}} \cdot h [y_1 x_1 \dots x_k] \dots [y_n x_1 \dots x_k]$$

where $y_i: \&_{j \leq l} C_{1j} \longrightarrow \dots \longrightarrow C_{kj} \longrightarrow \alpha_i^j$ and the $\{\alpha_i^j | 1 \leq i \leq n\}$ are solutions of the type constraint \mathcal{SC}_C^h defined below. As Lemma 4.3 will show, G is a most general term with head h and sort C .

If h is a constant or a free variable we write the general binding as \mathcal{G}_C^h and call it a *general imitation binding*. If h is the bound variable x_i , then we write \mathcal{G}_C^i and call it the *general i -projection binding*. In this case we write the type constraint as \mathcal{SC}_C^i .

In order for $\Gamma \vdash_{\Sigma} \mathcal{G}_C^h: C$ to hold we have to guarantee that for all $1 \leq j \leq l$

$$\Gamma, x_1: C_{1j}, \dots, x_k: C_{kj} \vdash_{\Sigma} h [y_1 x_1 \dots x_k] \dots [y_n x_1 \dots x_k]: D_j.$$

This in turn requires that h at least map the α_i^j into D_j . These considerations together with the co- and contravariance of \leq explain the type constraints

$$\begin{aligned} \mathcal{SC}_C^h &= A \leq \&_{1 \leq j \leq l} \alpha_1^j \longrightarrow \dots \longrightarrow \alpha_n^j \longrightarrow D_j, \\ \mathcal{SC}_C^i &= \bigwedge_{1 \leq j \leq l} C_{ij} \leq \alpha_1^j \longrightarrow \dots \longrightarrow \alpha_n^j \longrightarrow D_j \end{aligned}$$

Definition 4.2 We will call a substitution $\sigma = [M_1/x_1, \dots, M_n/x_n]$ *well-typed* in a context Γ , iff $\Gamma \vdash_{\Sigma} x_i: A_i$ implies $\Gamma \vdash_{\Sigma} M_i: A_i$. Let W be a set of variables. Then we write $\sigma = \theta[W]$ if for all $x \in W$, $\sigma(x) = \theta(x)$, and $\sigma \leq \theta[W]$ if there exists a well-typed substitution ρ such that $\rho \circ \sigma = \theta[W]$.

Lemma 4.3 (General Binding Lemma) *If $\Gamma \vdash M: C$ and the head of M is h , then there exists a general binding G of type C with the head h and a well-typed substitution θ , such that $\theta(G) \equiv M$.*

Proof: By lemma 3.5 we can assume M to be in normal form, that is, M has the form $\lambda x_1: F_1 \dots \lambda x_k: F_k \cdot h N_1 \dots N_n$. We only treat the case where h is not a bound variable — the other case is similar. Let $C = \&_{j \leq l} C_{1j} \longrightarrow \dots \longrightarrow C_{kj} \longrightarrow D_j$ and $\Gamma \vdash_{\Sigma} M: C$. Then $\Gamma, x_1: C_{1j}, \dots, x_k: C_{kj} \vdash h N_1 \dots N_n: D_j$ for all $1 \leq j \leq l$.

Now let $A = \&_{j \leq m} A_{1j} \rightarrow \dots \rightarrow A_{nj} \rightarrow B_j$ and $\Gamma \vdash_{\Sigma} h \in A$. Thus, since $h N_1 \dots N_n$ is well-typed and of type D_j , there is a $k \leq m$, such that $\vdash_{\Sigma} B_k \sqsubseteq D_j$ and there are types F_i^j , such that $\Gamma \vdash_{\Sigma} N_i \in F_i^j$ and $\vdash_{\Sigma} F_i^j \sqsubseteq A_{jk}$. We can easily verify that $\Gamma \vdash_{\Sigma} A_{1k} \rightarrow \dots \rightarrow A_{nk} \rightarrow B_k \sqsubseteq F_1^j \rightarrow \dots \rightarrow F_n^j \rightarrow D_j$ for all j , so the type assignment $[F_i^j/\alpha_i^j]$ is a solution of \mathcal{SC}_C^h .

Now let G be the general binding for the head h and the type C

$$G = \lambda x_1: \overline{C_{1j}} \dots \lambda x_k: \overline{C_{kj}} \cdot h [y_1 x_1 \dots x_k] \dots [y_n x_1 \dots x_k]$$

such that $y_i: \&_{j \leq l} C_{1j} \longrightarrow \dots \longrightarrow C_{kj} \longrightarrow \alpha_i^j$. We note that $\overline{C_{ij}}$ is just F_i and define $\theta(y_r) = \lambda x_1: F_1 \dots \lambda x_k: F_k \cdot N_r$. Then θ is well-typed in the context Γ and we furthermore have that $\theta(G) \equiv \lambda x_1: F_1 \dots \lambda x_k: F_k \cdot h N_1 \dots N_n$. \square

Lemma 4.4 *If $\theta = [M/x] \cup \theta'$ then there exists a general binding G and a substitution ρ , such that $\theta = [M/x] \cup \rho \cup \theta'[\text{dom}(\theta)] = \rho \circ [G/x] \cup \theta'[\text{dom}(\theta)]$.*

Proof sketch: Directly from Lemma 4.3. \square

Example 4.5 Let

$$\begin{aligned} \Sigma &= B: \text{Type}, T :: B, F :: B, \wedge: T \longrightarrow T \longrightarrow T, \wedge: T \longrightarrow F \longrightarrow F, \\ &\wedge: F \longrightarrow T \longrightarrow F, \wedge: F \longrightarrow F \longrightarrow F. \end{aligned}$$

Then

$$\begin{aligned} \mathcal{G}_{T \rightarrow F \& F \rightarrow T}^\wedge &= \lambda x: B. \wedge [y_1 x][y_2 x] \\ \mathcal{SC}_{T \rightarrow F \& F \rightarrow T}^\wedge &= T \rightarrow T \rightarrow T \& T \rightarrow F \rightarrow F \& F \rightarrow T \rightarrow F \\ &\quad \& F \rightarrow F \rightarrow F \leq \alpha_1^1 \rightarrow \alpha_2^1 \rightarrow F \& \alpha_1^2 \rightarrow \alpha_2^2 \rightarrow T \end{aligned}$$

The constraints on the types of y_1 and y_2 have the following three solutions

$$\begin{aligned} y_1: F \rightarrow T \& T \rightarrow F, \quad y_2: F \rightarrow T \& T \rightarrow T \\ y_1: F \rightarrow T \& T \rightarrow F, \quad y_2: F \rightarrow T \& T \rightarrow F \\ y_1: F \rightarrow T \& T \rightarrow T, \quad y_2: F \rightarrow T \& T \rightarrow F \end{aligned}$$

5 General Unification and Pre-Unification

Building upon the notion of general binding and type constraint simplification we give a set of transformations for general unification and pre-unification, which we will prove correct and complete with the methods of [11].

Definition 5.1 (Unification Problem) A *unification problem* is a formula in the language

$$F ::= M \doteq N \mid \exists x: A.F \mid \forall u: A.F \mid \exists \alpha :: A.F \mid F_1 \wedge F_2 \mid \top \mid A_1 \leq A_2$$

where the types A may now contain type variables α . We will call all subformulae of F of the form $A_1 \leq A_2$ *type constraints*. The refinement judgment is extended in the obvious way (assuming α refines A in the scope of $\exists \alpha :: A$) and we require that for each type constraint $A_1 \leq A_2$ there is a proper type A such that $A_1 :: A$ and $A_2 :: A$.

Since we have defined type variables to range only over the (finite) set of refinements of a given type, the set of solutions of a type constraint is effectively computable by a generate-and-test approach. It is clear, however, that this is not a viable implementation strategy. A more reasonable constraint simplifier can be derived from the algorithmic rules for subtyping, but we leave the details to a future paper.

In order to simplify the presentation of the algorithm, we assume that all unification formulae are in $\exists\forall$ -form. Each formula is equivalent to one in this form by raising [?]. We will refer to the universally quantified variables as *parameters* and use the meta-variables u and v to range over parameters. Note that they may not occur in the substitution terms for *existential variables*, which we denote by x , y , and z . We also use h to stand for either a

constant or a parameter. Furthermore, we fix the signature Σ and omit the context Γ and simply write $M \in A$ for the judgment $\Gamma \vdash_{\Sigma} M \in A$ when the context Γ can be recovered from the place in which M appears.

Definition 5.2 (Provability) The basic judgment is $\Gamma \Vdash F$ (F is provable) is defined by the following inference rules.

$$\begin{array}{c}
\frac{\Gamma \vdash_{\Sigma} M: A \quad M \equiv N \quad \Gamma \vdash_{\Sigma} N: A}{\Gamma \Vdash M \doteq N} \qquad \frac{}{\Gamma \Vdash \top} \qquad \frac{\Gamma \Vdash F \quad \Gamma \Vdash G}{\Gamma \Vdash F \wedge G} \\
\\
\frac{\Gamma \vdash_{\Sigma} M: A \quad \Gamma \Vdash [M/x]F}{\Vdash \exists x: A. F} \qquad \frac{\vdash_{\Sigma} B :: A \quad \Gamma \Vdash [B/\alpha]F}{\Vdash \exists \alpha :: A. F} \\
\\
\frac{\Gamma, u: A \Vdash F}{\Gamma \Vdash \forall u: A. F} \qquad \frac{\vdash_{\Sigma} A \leq B}{\Gamma \Vdash A \leq B}
\end{array}$$

We call a substitution σ for the existential variables in a unification formula F *legal for F* if it is well-typed and no parameters occur in the instantiation terms for σ . Note that a proof of a formula F in $\exists\forall$ form uniquely determines a legal substitution σ for the existential variables in F . Conversely, any ground instance of a legal unifier for the equations in the matrix of F uniquely determines a proof for F . In slight abuse of notation, we will thus call such a σ a *unifier for F* . The notion of most general unifier is extended similarly to formulae.

Definition 5.3 (Solved Form) A unification formula F is in *solved form* if it contains no sort constraints and all of its equations are in solved form, i.e., of the form $x \doteq M$, such that $x \in A$, $M \in B$ and $B \leq A$, neither x nor any parameter u is free in M , and x is not free elsewhere in F . It is easy to show that if F is in solved form with matrix $x_1 \doteq M_1 \wedge \dots \wedge x_n \doteq M_n$ then $\sigma_F = [M_1/x_1, \dots, M_n/x_n]$ is a most general unifier for F .

Note that a formula in solved form is not necessarily provable from the empty context, since some sorts may be empty. However we feel that the nonemptiness of sorts should be treated in the deduction system that uses the unification algorithm, rather than in unification itself (cf. Theorem 5.5).

Definition 5.4 (Transformations for General Unification) In the description of the rules, we use F^- to stand for the matrix of F . We omit the obvious versions of the rules where the equations on the left-hand side are reversed.

trivial $M \doteq M \implies \top$.

decompose $hM_1 \dots M_n \doteq hN_1 \dots N_n \implies M_1 \doteq N_1 \wedge \dots \wedge M_n \doteq N_n$.

merge $x \doteq M$ such that $x \in A$, $M \in B$ and $B \leq A$, then $F^- \implies x \doteq M \wedge [M/x]F^-$, if neither x nor any parameter u occurs in M and x occurs elsewhere in F^- .

imitate $xM_1 \dots M_m \doteq cN_1 \dots N_n \implies xM_1 \dots M_m \doteq cN_1 \dots N_n \wedge x \doteq \mathcal{G}_A^c \wedge \mathcal{SC}_A^c$, where $x \in A$.

***i*-project** $xM_1 \dots M_m \doteq N \implies xM_1 \dots M_m \doteq N \wedge x \doteq \mathcal{G}_A^i \wedge \mathcal{SC}_A^i$ where $x \in A$.

guess $xM_1 \dots M_m \doteq yN_1 \dots N_n \implies xM_1 \dots M_m \doteq yN_1 \dots N_n \wedge x \doteq \mathcal{G}_A^h \wedge \mathcal{SC}_A^h$ if h is some constant or existential variable and $x \in A$.

simplify constraint type constraints can be simplified by any sound constraint simplifier.

lam-lam $\lambda v: A.M \doteq \lambda v: A.N \implies \forall v: A.M \doteq N$. Note that the type labels on both sides must be the same for the equation to be valid (they are proper types, not sorts).

lam-term $\lambda v: A.M \doteq N \implies \forall v: A.M \doteq N v$ where N is not a λ -abstraction.

Furthermore we require the structural rules that deal with quantifier exchange from [?] and rules to erase \top from a conjunction.

These transformations (and those of the following unification algorithms) can be employed by very different algorithms, depending on the strategy involved in constraint simplification. Solving type constraints eagerly after each imitation and projection step amounts to separate imitation rules for each solution.

For a realistic implementation it seems advantageous to pass the type constraints along and wait for more information in form of further instantiation. Such further instantiation might be provided by further imitation

steps. An implementation of the algorithm would also add rules to identify failure due to non-applicability of rules early and yield a more efficient algorithm.

The soundness of the transformations can readily be established from the soundness of the constraint simplifier and lemmata 4.3 and 3.4 by using the techniques from [11]. Now we will turn to the completeness of the transformations.

Theorem 5.5 (Completeness) *For any unifier θ of a unification formula F there exists a sequence of transformations for general well-typed unification from F to S in solved form, such that $\sigma_S \leq \theta[X]$ where X is the set of existential variables in F .*

Proof sketch: We define a variant of the transformations from Definition 5.4 that operate on a pair (θ, F) , where F is a unification formula and θ is a substitution. For $\theta = [M/x] \cup \theta'$ let G be the general binding and ρ the substitution guaranteed by Lemma 4.4. The transformations **imitate**, **project** and **guess** are of the form $(\theta, F) \Longrightarrow (\theta' \cup \rho \cup [M/x], F \wedge x \doteq G \wedge SC)$, where ρ is as in Lemma 4.4.

Obviously we get a subsystem of that defined in Definition 5.4, if we restrict this variant to the unification formulae. Furthermore in contrast to the unrestricted system it can be shown that all sequences of transformations in this system must terminate with an irreducible pair (θ, F) . On the other hand by close inspection of the transformations using lemma 4.3, we can see that irreducible pairs are in solved form. Thus we get the completeness result from Definition 5.3. \square

The notion of pre-unification is of interest to automated theorem proving, since pre-unifiers can always be extended to unifiers (see Lemma 5.7) and the pre-unification problem is often tractable. We will only state the definitions and the completeness result.

Definition 5.6 (Transformations for General Pre-Unification) The transformations for pre-unification are the same as those for general unification except that the **guess** rule is dropped.

These rules are applied to an initial unification problem, until it is in *pre-solved form*, that is all equations are in either in solved form or the heads of both sides of the equation are existential variables.

Lemma 5.7 *Pre-solved unification problems are always unifiable.*

Proof: Let F be a unification problem in pre-solved form and let E be an equation $xM_1 \dots M_m \doteq yN_1 \dots N_n$ in F where x and y are existential variables. Then the substitution

$$\sigma = [[\lambda x_1: \overline{A_{j_1}} \dots x_n: \overline{A_{j_n}} \cdot z] / x, [\lambda y_1: \overline{C_{j_1}} \dots y_n: \overline{C_{j_m}} \cdot z] / y]$$

where x, y are existential variables with $x \in \&_{j \leq k} A_{j_1} \longrightarrow \dots \longrightarrow A_{j_n} \longrightarrow B_j$, $y \in \&_{j \leq l} C_{j_1} \longrightarrow \dots \longrightarrow C_{j_m} \longrightarrow D_j$ and $z: \&_{j \leq k} B_j \& \&_{j \leq l} D_j$ unifies E . This idea can be extended to solve all equations of this form simultaneously. \square

Theorem 5.8 (Completeness of Pre-Unification) *For any pre-unifier θ of a unification formula F there exists a sequence of transformations for general pre-unification from F to a pre-solved form S , such that $\sigma_S \leq \theta[X]$ where X is the set of existential variables in F .*

Corollary 5.9 *If F is a closed unification problem and F is transformed into a (pre-)solved form S by a sequence of transformations for general unification or pre-unification and all sorts of variables that are existentially bound in S are nonempty, then $\Vdash F$.*

6 Unification Restricted to Patterns

We now specialize the algorithm from Section 5 to patterns, which are defined just as in the simply-typed λ -calculus: any occurrence of an existential variable x in

$$\exists x_1: A_1 \dots \exists x_q: A_q \forall u_1: B_1 \dots \forall u_p: B_p \cdot F$$

must have the form $x u_{\varphi(1)} \dots u_{\varphi(n)}$, where φ is a partial permutation from n into p , i.e., an injective mapping from $1, \dots, n$ to $1, \dots, p$. The transformations for pattern unifications are those of Definition 5.4 where the rule **guess** is replaced by the following transformations.

Definition 6.1 (Transformations for Pattern Unification) Here x and y are existential variables where $x: C = \&_{j \leq k} C_{1j} \rightarrow \dots \rightarrow C_{nj} \rightarrow D_j$.

var-var-same $x u_{\varphi(1)} \dots u_{\varphi(n)} \doteq x u_{\psi(1)} \dots u_{\psi(n)}$ is transformed into

$$x \doteq \lambda v_1: B_{\varphi(1)} \dots \lambda v_n: B_{\varphi(n)} \cdot x' v_{\rho(1)} \dots v_{\rho(l)},$$

where ρ is a partial permutation satisfying: there exists a k such that $\rho(k) = \varphi(i)$ iff $\varphi(i) = \psi(i)$ and $x': \&_{j \leq k} C_{\rho(1)j} \rightarrow \dots \rightarrow C_{\rho(l)j} \rightarrow D_j$ is a new existential variable.

var-var-diff $x u_{\varphi(1)} \dots u_{\varphi(n)} \doteq y u_{\psi(1)} \dots u_{\psi(m)}$. Then let φ' and ψ' be partial permutations satisfying: there exists a k such that $\varphi'(k) = i$ and $\psi'(k) = j$ iff $\varphi(i) = \psi(j)$. We transform into

$$\begin{aligned} x &\doteq \lambda v_1: B_{\varphi(1)} \dots \lambda v_n: B_{\varphi(n)} \cdot z v_{\varphi'(1)} \dots v_{\varphi'(l)} \quad \wedge \\ y &\doteq \lambda v_1: B_{\psi(1)} \dots \lambda v_m: B_{\psi(m)} \cdot z v_{\psi'(1)} \dots v_{\psi'(l)}, \quad \text{where} \\ z &: \&_{j \leq k} C_{\psi'(1)j} \rightarrow \dots \rightarrow C_{\psi'(l)j} \longrightarrow D_j \end{aligned}$$

Note that in the case of higher-order patterns the use of the rules **project** and **imitate** are deterministic, that is, all but the imitation or one projection immediately lead to failure. The sort constraints, however, may still have multiple solutions.

Theorem 6.2 (Completeness of Pattern Unification) *Let F be a closed unification problem where all objects are higher-order patterns. Then the transformations of pattern unification always terminate and either*

1. *yield a unification problem S in solved form and σ_S is a unifier for F . Furthermore, if all sorts of existentially bound variables in S are nonempty, then $\Vdash F$.*

or 2. *yield a unification problem where none of the transformations are applicable and F is not provable.*

7 Conclusion and Further Work

The unification algorithms presented here can serve as a basis for practical implementations of theorem provers or logic programming languages which incorporate a notion of subtype and intersection type. Standard techniques should be applicable to achieve the same efficiency as current implementations of higher-order unification or pattern unification whenever no subtype or refinement declarations are made. The presence of sort conditions potentially leads to an explosion of the search space during unification. In theorem proving this merely shifts work from logical inferences to unification where it is handled algorithmically, and we expect it to improve overall performance. In logic programming many sort computations can be shown to be redundant at compile-time, given that the goal is always maintained in well-sorted form. This situation is familiar from (first-order) order-sorted logic programming and we believe that such static analysis is necessary to obtain a practical system.

We would also like to extend the algorithm to $\lambda^{\Pi\&}$, a type theory with intersection and dependent types proposed in [?]. An extension of the language Elf [9] along these lines would be based on a constraint solver (rather than a unification or pre-unification algorithm) that solves pattern unification problems, but maintains other equations and sort conditions as constraints. The principal question in this context is when and to what extent sort computation should lead to branching during the computation. This will depend in large part upon the results of experimentation with a prototype implementation.

Finally, we would like to consider relaxing some of the restrictions of the current system without disturbing its basic properties. For example, it may be possible to admit arbitrary type labels in abstractions if we also add conversion rules that relabel abstractions with compatible types.

Acknowledgments. The first author was supported by the “Sonderforschungsbereich 314, Künstliche Intelligenz” of the Deutsche Forschungsgemeinschaft (DFG) and the “Studienstiftung des deutschen Volkes”. The second author was supported in part by the U.S. Air Force under Contract F33615-90-C-1465, ARPA Order No. 7597.

References

- [1] 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.
- [2] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*. ACM Press, June 1991. To appear. Also available as Ergo report 91-097.
- [3] Gérard Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
- [4] Michael Kohlhase. Order-sorted type theory I: Unification. SEKI-Report SR-91-18 (SFB), Universität des Saarlandes, Saarbrücken, 1991.
- [5] Michael Kohlhase. Beweissysteme mit Logiken höherer Stufe. In Karl Hans Bläsius and Hans-Jürgen Bürckert, editors, *Deduktionssysteme, Automatisierung des Logischen Denkens*, chapter 6, pages 213–238. R. Oldenbourg Verlag, 2 edition, 1992.

- [6] 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.
- [7] Gopalan Nadathur and Dale Miller. An overview of λ Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.
- [8] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user’s manual. Technical Report 189, Computer Laboratory, University of Cambridge, January 1990.
- [9] Frank Pfenning. Logic programming in the LF logical framework. In Gérard P. Huet and Gordon D. Plotkin, editors, *Logical Frameworks*. Cambridge University Press, 1991.
- [10] Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Universität Kaiserslautern, 1989.
- [11] Wayne Snyder. *A Proof Theory for General Unification*. Progress in Computer Science and Applied Logic. Birkhäuser, 1991.