Technical Report

UCAM-CL-TR-210
ISSN 1476-2986

Number 210

# Higher-order unification, polymorphism, and subsorts

Tobias Nipkow

ISSN 1476-2986

# Higher-Order Unification, Polymorphism, and Subsorts

Tobias Nipkow*
University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
England
Tobias.Nipkow@cl.cam.ac.uk

**Abstract**

This paper analyzes the problems that arise in extending Huet's higher-order unification algorithm from the simply typed $\lambda$-calculus to one with type variables. A simple, incomplete, but in practice very useful extension to Huet's algorithm is discussed. This extension takes an abstract view of types. As a particular instance we explore a type system with ML-style polymorphism enriched with a notion of *sorts*. Sorts are partially ordered and classify types, thus giving rise to an order-sorted algebra of types. *Type classes* in the functional language Haskell can be understood as sorts in this sense. Sufficient conditions on the sort structure to ensure the existence of principal types are discussed. Finally we suggest a new type system for the $\lambda$-calculus which may pave the way to a complete unification algorithm for polymorphic terms.

*Research supported by ESPRIT BRA 3245, *Logical Frameworks.*

## 1 Introduction

Huet's algorithm for higher-order unification in the simply typed $\lambda$-calculus [11] forms the core of a number of theorem provers based on higher-order logic [1, 17, 14]. In this paper we discuss extensions to this unification algorithm which permit more expressive type systems: ML-style polymorphism enriched with a notion of sorts constraining type variables. It turns out that the step from the simply typed $\lambda$-calculus considered by Huet to one with type variables is nontrivial. The unification algorithm we analyze is in fact incomplete. A complete extension remains an open problem which is briefly considered at the end. On a more positive note, the introduction of sorts is completely orthogonal to the higher-order unification algorithm and causes no problems whatsoever.

It has to be stressed that this is not the first extension of Huet's algorithm to a polymorphic type system. Nadathur [14] reports on an implementation of the programming language $\lambda$Prolog which features polymorphism. His extensions seem to be very similar to the ones analyzed in this paper. In particular he faces the same problem with completeness that we have. However, the author is not aware of a *formal* treatment of these issues, in particular not in connection with an order-sorted type system.

All the features described in this paper have recently been implemented in the generic theorem prover Isabelle. Isabelle is generic in the sense that it can be parameterized with the intended object-logic. Its meta-logic is a fragment of higher-order logic and its inference mechanism is based on higher-order unification. Detailed descriptions of the monomorphic version of Isabelle can be found in [18, 19]. Before we go into technical details, the motivation for the extensions to the type system in presented.

Although ML-style polymorphism hardly needs to be motivated in a programming language context, some simple examples from the realm of generic theorem provers seem appropriate. The basic reason for the introduction of polymorphism is the desire to shift type-checking in many-sorted logics from the object to the meta level. A typical example is the definition of an ordinary many-sorted first-order logic. We need to model a type of formulae *form*, and an unbounded number of different types of terms like natural numbers, reals, lists etc. In a polymorphic type system, the operation of equality between terms has type $\alpha \rightarrow \alpha \rightarrow form$ where $\alpha$ is a type variable. The built-in type checker ensures that equality always compares terms of the same type.

In a monomorphic type system, we would have to declare a new equality $=_s$ of type $s \rightarrow s \rightarrow form$ for each new type of terms $s$, and give the same inference rules over and over again. This is clearly impractical. Alternatively, we can formalize types as part of the object-logic. In this case terms have to be decorated with their types, the type checking rules are part of the inference system and type checking is part of a proof. This is the approach taken in all logics defined in the monomorphic version of Isabelle. If a logic's type system is sufficiently expressive to become undecidable, for example Intuitionistic Type Theory [12], this is in fact the only possible approach. For simpler systems, however, ML-polymorphism is clearly preferable.

Having adopted polymorphism, we need to tame its power. Initially one might be tempted to declare equality and universal quantification as follows:

$$\begin{aligned} =&: \alpha \to \alpha \to form \\ \forall&: (\alpha \to form) \to form \end{aligned}$$

The intention is that $\alpha$ ranges only over different types of terms, but certainly not over formulae or arbitrary function types (if the logic is supposed to be first-order). However, there is nothing in this declaration to enforce those constraints. As a consequence, some rather surprising inferences are possible. In first-order logic $\forall$-elimination is

$$\frac{\forall x.\ P(x)}{P(t)}$$

where $x : \alpha$, $t : \alpha$, $P : \alpha \to form$ and $\alpha$ are variables. Using the substitution $\{\alpha \mapsto form, t \mapsto Q, P \mapsto \lambda x.x\}$ we obtain the derived rule

$$\frac{\forall x.\ x}{Q}.$$

In a first order-logic, $\forall x.\ x$ is ill-formed. This formula could only arise because of the instantiation of $\alpha$ by $form$. In a higher-order logic, $\forall x.\ x$ is in fact a valid formula, usually identified with falsity, and the rule expresses *ex falso quodlibet*.

In order to avoid such pitfalls, Isabelle's types are classified by so called *sorts*. The sort of a type variable defines the subset of types the variable ranges over, thus prohibiting undesirable instantiations. Details are given in Section 4. Note that a similar mechanism for restricted polymorphism is embodied in ML's *equality types* and, in a more general form, in Haskell's *type classes* [24, 16].

After fixing the basic notation in Section 2, Section 3 presents the extension to Huet's algorithm, taking an abstract view of types. Section 4 focuses on the type system, introducing polymorphism, sorts and subsorts. Finally, we speculate on a way to obtain a complete unification algorithm for the polymorphic case.

## 2 Preliminaries

The reader should be familiar with the basic notations and facts of equational logic as found for example in Dershowitz and Jouannaud [4]. We are broadly consistent with their notation, except that we treat substitutions as ordinary functions. Function composition is denoted by $\circ$: $(f \circ g)(x) = f(g(x))$. The rest of this section presents the notation for the typed $\lambda$-calculus with $\alpha$, $\beta$ and $\eta$ equality. For more details see [23, 10].

For the time being we take an abstract view of types with just enough detail to describe the unification process. Section 4 gives the full description of the intended type system. This separation is possible because the unification of terms and types proceeds largely independently.

The set $\mathcal{T}$ of all *types* is a subset of the free algebra over some set of type constructors, including the function-space constructor "$\to$", generated by a set of type variables $\mathcal{V}_\mathcal{T}$. The idea is that the types themselves form a sorted algebra and $\mathcal{T}$ is the set of all "well-sorted", i.e. well-formed types. Types whose outermost constructor is "$\rightarrow$" are called *function types*, all other types are called *base types*. The letters $\sigma$, $\tau$ and $\gamma$ represent types. Function types associate to the right: $\sigma \rightarrow \tau \rightarrow \gamma$ means $\sigma \rightarrow (\tau \rightarrow \gamma)$. Instead of $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$ we write $\overline{\sigma_m} \rightarrow \tau$. The latter form is used only if $\tau$ is a base type. The type variables occurring in a type $\sigma$ are denoted by $\mathcal{V}_\mathcal{T}(\sigma)$.

$\mathcal{T}$ comes with a set of *type substitutions* $S_\mathcal{T}$ which is some subset of the set of all possible mappings from $\mathcal{V}_\mathcal{T}$ to $\mathcal{T}$ and contains $\{\}$, the empty map. Again, the sorted nature of types has to be taken into account. Elements of $S_\mathcal{T}$ are denoted by $\Theta$ and $\Delta$. The quasi-order $\leq$ on terms and substitutions is defined in the usual way. Correspondingly, there is a unification function $\mathcal{U} : \mathcal{T} \times \mathcal{T} \rightarrow 2^{S_\mathcal{T}}$ which returns a complete set of unifiers: for any two $\sigma, \tau \in \mathcal{T}$ and any $\Delta \in S_\mathcal{T}$ with $\Delta(\sigma) = \Delta(\tau)$ there is a $\Theta \in \mathcal{U}(\sigma, \tau)$ such that $\Theta \leq \Delta\ [\mathcal{V}_\mathcal{T}(\sigma, \tau)]$. For simplicity we assume that all variables in the range of a substitution returned by $\mathcal{U}$ are "new".

Terms are generated from a set of *free variables* $V$, a set of *bound variables* $X$, and a set of *constants* $C$ by $\lambda$-abstraction and application. Free variables are denoted by $F$, $G$, and $H$, bound variables by $x$, $y$ and $z$, and constants by $a$, $b$, and $c$. Terms are denoted by $s$, $t$, and $u$. The inductive definition of *typed terms* in a context $\Gamma$, which is just a mapping from $X$ to $\mathcal{T}$, is as follows:

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \Gamma \vdash F^\tau : \tau \qquad \Gamma \vdash c^\tau : \tau$$

$$\frac{\Gamma \vdash s : \sigma \rightarrow \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash (s\ t) : \tau} \qquad \frac{\Gamma \circ \{x \mapsto \sigma\} \vdash s : \tau}{\Gamma \vdash (\lambda x^\sigma . s) : \sigma \rightarrow \tau}$$

Note that free variables and constants on their own are not legal terms. They have to be tagged with some type, as in the second and third formation rule above. In particular we assume that all occurrences of a free variable in a term are decorated with the same type. There is no corresponding requirement for constants because they can be polymorphic. A term $t$ is *well-typed* if $\{\} \vdash t : \tau$ holds for some $\tau \in \mathcal{T}$. Instead of $\{\} \vdash t : \tau$ we simply write $t : \tau$. In the sequel we consider only well-typed terms, which means in particular that there are no "loose" bound variables: $x$, for example, is illegal. Instead of $\lambda x_1 . \ldots \lambda x_n . s$ we write $\lambda x_1, \ldots, x_n . s$ or just $\lambda \overline{x_n} . s$. Similarly we write $t(u_1, \ldots, u_n)$ or just $t(\overline{u_n})$ instead of $(\ldots(t\ u_1)\ldots)u_n$. The free variables in $t$ are denoted by $\mathcal{V}(t)$, the type variables by $\mathcal{V}_\mathcal{T}(t)$. Type decorations are omitted if they are not important.

We assume $\alpha$, $\beta$ and $\eta$ conversion on terms. Relying on the strong normalization property of the typed $\lambda$-calculus we assume that terms are in $\beta$-normal form. We also ignore $\alpha$ conversion by working with $\alpha$-equivalence classes of terms, using the generic bound variable names $x$ and $y$. Similarly $\eta$-conversion is ignored by working with the *$\eta$-expanded form* of terms [23]. The $\eta$-expanded form of the $\beta$-normal form of a term $t$ is denoted by $t{\downarrow}$.

A term $\lambda \overline{x_m} . s(\overline{u_k})$ in $\beta$-normal form is called *rigid* if $s \in X \cup C$, and *flexible* if $s \in V$.

Substitutions on terms are defined as mappings from free variables to $\lambda$-terms in the usual way. They are denoted by $\theta$ and $\delta$. Applying a type substitution to a term means applying it to all type decorations in the term.

A *unifier* of two terms $s$ and $t$ is a pair of substitutions $\langle\Theta, \theta\rangle$ on types and terms respectively such that $\theta(\Theta(s))$ and $\theta(\Theta(t))$ are equivalent modulo $\alpha$, $\beta$ and $\eta$-conversion.

## 3 Higher-Order Unification

The starting point for most work on higher-order unification is Huet's algorithm [11] which enumerates a complete and minimal set of unifiers with respect to the *simply typed* $\lambda$-calculus [10]. As higher-order unification is undecidable in general [9], this is the best one can hope for. Snyder and Gallier [23] have recently reformulated Huet's algorithm in terms of inference rules, which simplifies the presentation. Their version will be our reference point.

Isabelle was originally based on the simply typed $\lambda$-calculus and used Huet's algorithm. A recent extension permits *polymorphic* constants in the ML sense. As a consequence, terms may now contain type variables as well as term variables, both of which may need to be instantiated during the unification process. This is a significant departure from the original problem.

**Example 3.1** Let $\tau$ be some base type and $\alpha$ a type variable, let $a : \tau$ be a constant and $G : \alpha$ and $F : \alpha \rightarrow \tau$ be two variables. The following matching problem is given:

$$F(G) = a \tag{1}$$

Let us first look at (some of) the infinitely many solutions to this problem. We know that any instantiation of $\alpha$ must be of the form

$$\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \gamma$$

for suitable types $\tau_1, \ldots, \tau_n$ and $\gamma$. In particular we may assume that $\gamma$ is not a function type. Without any assumptions about $\alpha$ we get the solution $F = \lambda x.a$. In terms of Huet's algorithm this is the solution obtained after a single imitation step. In fact, for any instantiation of $\alpha$ where $\gamma \neq \tau$, this is the only solution.

If we also assume that $\gamma = \tau$, we get the following infinite set of solutions, indexed by $n$:

$$F = \lambda x.x(A_1(x), \ldots, A_n(x)), \quad G = \lambda x_1, \ldots, x_n.a$$

The $A_i$ are new free function variables.

Further instantiations of $\alpha$ yield yet more solutions. For example $\alpha = \tau \rightarrow \tau$ alone has an infinite set of independent solutions:

$$F = \lambda x.x^k(a), \quad G = \lambda x.x$$

is a solution for any natural number $k$, where $x^k$ is the $k$-fold composition of $x$.

This shows quite drastically that type variables introduce a new degree of freedom into unification problems. Different type instantiations give rise to completely independent sets of solutions with greatly varying cardinality (finite vs. infinite). This is not in itself surprising, but it raises the question how this new freedom can be incorporated into the unification algorithm. We shall try to "lift" Huet's algorithm to this new setting by replacing equality tests on types by unification. The following example shows why such a simple-minded extension is problematic. You need to be familiar with Huet's algorithm to follow it.

**Example 3.2** We return to the unification problem in Example 3.1. As mentioned above, a single imitation step finds the solution $F = \lambda x.a$ which requires no type instantiations. Projection, however, is different. $F$ can be a projection only if the *result type* of its argument $G$ is $\tau$. This means that projection is applicable for any instance of $\alpha$ of the form $\tau_1 \to \cdots \to \tau_n \to \tau$. Unfortunately, the set of these types does not have a finite representation in terms of function types and type variables. Hence we can either try all possible type instantiations (which is complete but completely impractical) or sacrifice completeness for efficiency. In the latter case we pick the simplest type instantiation that permits projection. In this example it means unifying $\alpha$ and $\tau$. This results in the instantiation of $\alpha$ by $\tau$ and the single additional solution

$$F = \lambda x.x(A(x)), \quad G = \lambda x.a.$$

This is obviously incomplete. In particular, if (1) is just one in a set of equations to be solved simultaneously, some of the other equations may well require $\alpha$ to be instantiated by a different type. In that case we would find no solution unless we also backtrack over possible type instantiations.

Despite its incompleteness, the solution outlined in the above example has been adopted in both $\lambda$Prolog and Isabelle. In the sequel we present a formal treatment of this extension to Huet's algorithm.

## 3.1 The Algorithm

This section formalizes the idea of "lifting" Huet's algorithm. Our treatment is very close to that of Snyder and Gallier [23]. In fact, parts of our algorithm are identical to the one they present. The price we pay is the incompleteness discussed above.

The unification algorithm is presented as a collection of conditional rewrite rules on pairs $\langle\Theta, D\rangle$, where $\Theta$ is a type substitution and $D$ a multiset of unification problems $s =^? t$. $D$ is called a *system* in the sequel. We assume that all occurrences of a free variable in $D$ are decorated with the same type, a property that is preserved by the inference rules below. Note that $\Theta$ merely records and accumulates the type substitutions, whereas $D$ represents both the current multiset of unification problems and the term substitutions obtained so far.

$$\frac{s : \sigma \quad t : \tau \quad \sigma \neq \tau \quad \Delta \in \mathcal{U}(\sigma, \tau)}{\langle\Theta, \{s =^? t\} \cup D\rangle \Longrightarrow \langle\Delta \circ \Theta, \Delta(\{s =^? t\} \cup D)\rangle} \qquad \text{(T)}$$

$$\langle \Theta, \{s \overset{?}{=} s\} \cup D\rangle \Longrightarrow \langle \Theta, D\rangle \tag{D}$$

$$\frac{t \in X \cup C}{\langle \Theta, \{\lambda\overline{x_k}.t(\overline{s_n}) =^? \lambda\overline{x_l}.F(\overline{u_m})\} \cup D\rangle \Longrightarrow \langle \Theta, \{\lambda\overline{x_l}.F(\overline{u_m}) =^? \lambda\overline{x_k}.t(\overline{s_n})\} \cup D\rangle} \tag{O}$$

$$\frac{F \notin \mathcal{V}(t) \quad F \in \mathcal{V}(D)}{\langle \Theta, \{\lambda\overline{x_k}.F(\overline{x_k}) =^? t\} \cup D\rangle \Longrightarrow \langle \Theta, \{\lambda\overline{x_k}.F(\overline{x_k}) =^? t\} \cup \{F \mapsto t\}(D)\rangle} \tag{V}$$

$$\frac{\sigma \neq \tau \quad \Delta \in \mathcal{U}(\sigma,\tau)}{\begin{array}{c}\langle \Theta, \{\lambda\overline{x_k}.c^\sigma(\overline{s_m}) =^? \lambda\overline{x_l}.c^\tau(\overline{u_n})\} \cup D\rangle \Longrightarrow \\ \langle \Delta \circ \Theta, \Delta(\{\lambda\overline{x_k}.c^\sigma(\overline{s_m}) =^? \lambda\overline{x_l}.c^\tau(\overline{u_n})\} \cup D)\rangle\end{array}} \tag{T$_S$}$$

$$\frac{t \in X \cup C}{\begin{array}{c}\langle \Theta, \{\lambda\overline{x_k}.t(\overline{s_n}) =^? \lambda\overline{x_k}.t(\overline{u_n})\} \cup D\rangle \Longrightarrow \\ \langle \Theta, \{\lambda\overline{x_k}.s_i =^? \lambda\overline{x_k}.u_i \mid i \in \{1,\ldots,n\}\} \cup D\rangle\end{array}} \tag{S}$$

$$\frac{\sigma = \overline{\sigma_{m+i}} \to \sigma' \quad \tau = \overline{\tau_{n+j}} \to \sigma'}{\begin{array}{c}\langle \Theta, \{\lambda\overline{x_k}.F^\sigma(\overline{s_m}) =^? \lambda\overline{x_l}.c^\tau(\overline{u_n})\} \cup D\rangle \Longrightarrow \\ \langle \Theta, \{F =^? \lambda\overline{x_{m+i}}.c^\tau(\overline{H_{n+j}(\overline{x_{m+i}})}), \lambda\overline{x_k}.F^\sigma(\overline{s_m}) =^? \lambda\overline{x_l}.c^\tau(\overline{u_n})\} \cup D\rangle\end{array}} \tag{I}$$

$$\frac{t \in X \cup C \quad \sigma = \overline{\sigma_{m+i}} \to \sigma' \quad \sigma_j = \overline{\tau_p} \to \tau \quad \sigma' \neq \tau \quad \Delta \in \mathcal{U}(\sigma',\tau)}{\begin{array}{c}\langle \Theta, \{\lambda\overline{x_k}.F^\sigma(\overline{s_m}) =^? \lambda\overline{x_l}.t(\overline{u_n})\} \cup D\rangle \Longrightarrow \\ \langle \Delta \circ \Theta, \Delta(\{\lambda\overline{x_k}.F^\sigma(\overline{s_m}) =^? \lambda\overline{x_l}.t(\overline{u_n})\} \cup D)\rangle\end{array}} \tag{T$_P$}$$

$$\frac{t \in X \cup C \quad \sigma = \overline{\sigma_{m+i}} \to \tau \quad \sigma_j = \overline{\tau_p} \to \tau}{\begin{array}{c}\langle \Theta, \{\lambda\overline{x_k}.F^\sigma(\overline{s_m}) =^? \lambda\overline{x_l}.t(\overline{u_n})\} \cup D\rangle \Longrightarrow \\ \langle \Theta, \{F^\sigma =^? \lambda\overline{x_{m+i}}.x_j(\overline{H_p(\overline{x_{m+i}})}), \lambda\overline{x_k}.F^\sigma(\overline{s_m}) =^? \lambda\overline{x_l}.t(\overline{u_n})\} \cup D\rangle\end{array}} \tag{P}$$

The rules (T), (T$_S$), and (T$_P$) unify types and are new. Rule (O) orients rigid-flexible pairs. The remaining rules unify terms and are almost identical to the ones given by Snyder and Gallier. (T) unifies the types of two terms. (T$_S$) unifies the types of two rigid heads. (T$_P$) unifies the result type of a rigid term with the result type of one of its arguments. (T$_S$) and (T$_P$) may need to be applied before (S) and (P) respectively in order to make types meet. We assume that (I) and (P) are immediately followed by (V), eliminating $F$. The free variables $H_r$ in (I) and (P) are assumed to be new.

Huet's insight, which turned higher-order unification from a mere curiosity into computational reality, was that flexible-flexible pairs need not be solved. This simplifies the algorithm but requires some new terminology. The following definitions are broadly consistent with the literature.

A system $D$ is in *presolved* form if for every unification problem $s =^? t$ in $D$

- either $s = \lambda\overline{x_m}.F(\overline{x_m})$ and $F$ occurs neither in $t$ nor in the rest of $D$, or both $s$ and $t$ are flexible, and
- $s$ and $t$ have the same type.

If $D$ is in presolved form, define

$$\vec{D} \;=\; \{F \mapsto t \mid (\lambda\overline{x_k}.F(\overline{x_k}) \overset{?}{=} t) \in D\}.$$

Like Gallier and Snyder [23] we define $\cong$ to be the least congruence on well-typed terms of the same type containing all flexible-flexible pairs. The pair $\langle\Theta, \theta\rangle$ is a *preunifier* of $D$ if $\theta(\Theta(s))\!\downarrow \cong \theta(\Theta(t))\!\downarrow$ for all pairs $s =^? t$ in $D$.

The next simple lemma shows that the terminology is consistent in that presolved forms give rise to preunifiers which in turn extend to unifiers. The precise statement requires some more notation. With every base type $\sigma$ we associate some arbitrary but fixed free variable $F_\sigma$, and with every type $\tau = \overline{\sigma_m} \to \sigma$ a term $\hat{u}_\tau = \lambda\overline{x_m^{\sigma_m}}.F_\sigma$ where the $x_i$ are all distinct.

**Lemma 3.1** *If $D$ is in presolved form, $\langle\{\}, \vec{D}\rangle$ is a preunifier of $D$. If $\langle\Theta, \theta\rangle$ is a preunifier of $D$, then $\langle\Theta, \theta \cup \xi\rangle$ is a unifier of $D$, where*

$$\xi = \{F \mapsto \hat{u}_\sigma \mid F^\sigma \in \mathcal{V}(D) - dom(\theta)\}$$

The following soundness and completeness theorems can be proved along the same lines as their counterparts in [11, 23]:

**Theorem 3.1** *If $\langle\Theta, D\rangle \Longrightarrow^* \langle\Theta', D'\rangle$ and $D'$ is in presolved form, then $\langle\Theta', \vec{D'}\rangle$ is a preunifier of $D$.*

**Theorem 3.2** *Let $\langle\Delta, \delta\rangle$ be a unifier of a system $D$ such that no type in the range of $\Delta$ is a function type. Then there exists a presolved form $\langle\Theta, D'\rangle$ such that $\langle\{\}, D\rangle \Longrightarrow^* \langle\Theta, D'\rangle$, $\Theta \leq \Delta\ [\mathcal{V}_T(D)]$, $\vec{D'} \leq \delta\ [\mathcal{V}(D)]$, and $\delta$ is a unifier of all flexible-flexible pairs in $D'$.*

The completeness theorem is rather more conservative than the actual algorithm. In many practical cases it will find all solutions, although some of them require type variables to be instantiated by function types.

As with all unification algorithms expressed as rewrite rules on collections of unification problems, there are two kinds of nondeterminism during execution: the choice between different transformations that apply ("don't know") and between different unification problems they can be applied to ("don't care"). Ideally, all strategies for selecting unification problems should be equivalent with respect to completeness. For example Huet [11] and Elliot [5] show this quite explicitly for their algorithms. This result also holds with respect to the limited completeness theorem above. However, our algorithm is in general not just incomplete but also sensitive to the selection of unification problems. The point is that ($\mathrm{T}_P$) commits to a particular type instantiation out of an infinite set, thus reducing the solution space. Hence the application of ($\mathrm{T}_P$) should be delayed as long as possible in order to minimize incompleteness. More precisely, one can give an operational characterization of completeness: the above set of transformation rules enumerates a complete set of unifiers for a particular unification problem if ($T_P$) need not be applied in such a way that either $\gamma$ or $\tau$ are type variables. This means that no solutions are lost if the application of ($T_P$) to a particular unification problem can always be delayed until the types are sufficiently instantiated.

## 3.2 Optimizations

In this section we briefly consider some simple optimizations of the above rather high-level algorithm. First we look at issues connected with conversion of $\lambda$-terms. In the description above, we have abstracted from all three conversion rules by considering $\alpha$-equivalence classes of $\eta$-expanded $\beta$-normal forms. Here are some remarks on how to minimize the work of obtaining these normal forms:

$\alpha$ If De Bruijn notation [3] is used, $\alpha$-conversion can be ignored completely.

$\beta$ The question here is mainly how much of the normal form to compute when. As we can see from the rules, full $\beta$-normal form is not required — head-normal form will do. If all terms are in $\beta$-normal form initially, only the application of (V) entails further normalization.

$\eta$ In contrast to unification for the simply typed $\lambda$-calculus, our transformation rules do not leave terms in $\eta$-expanded form because type-variables may become instantiated. Fortunately, only (S) requires the $\eta$-expansion of the head, thus minimizing the need for $\eta$-expansion. In Isabelle we have gone so far as banning the instantiation of type variables by function types during the unification process, thus removing the need for $\eta$-expansion completely (provided the input is in $\eta$-expanded form). However, preliminary experiments suggest that in practice this is a rather drastic restriction, requiring frequent explicit type instantiations by the user.

Further important optimizations concern the order in which rules are applied. In contrast to ($T_P$), (T) and ($T_S$) should be performed as soon as possible, to detect nonunifiability by type-clashes. It is in fact sufficient to apply (T) to the input because the remaining rules maintain the invariant that the two terms of a unification problem have the same type. The rules for unification of terms, simplification (S), imitation (I), and projection (P), should be tried in the order embodied in Huet's algorithm.

## 4 Polymorphism and Subsorts

This section fills in the details left open by the rather general discussion of types in Section 2. As indicated in that section and motivated in the introduction, type variables need to be constrained in order to capture certain restrictions, for example the first-order nature of some proof system. In Isabelle, type variables are constrained by introducing a third level into the system: so far we had terms, and types which qualify terms. Now we add *sorts* which qualify types. This is very reminiscent of *generalized type systems* [2], a relationship that needs further investigation. In contrast to generalized type systems we have a partial order on the sorts, indicating that a subclass of types identified by one sort is contained in the subclass identified by the another sort. This embodies the central ideas behind the notion of *type classes* in the functional language Haskell [24], a relationship that is made precise in [16]. Note that we are dealing with coercion on the level of sorts, not of types. Hence the system is quite different from OBJ [6], where we have a partial order between types[1].

The step from ML-style polymorphism to one with partially ordered sorts leads from a single-sorted to an *order-sorted* algebra of types. Therefore we need some basic vocabulary of order-sorted types. After that we look at order-sorted unification, which fulfills two functions in Isabelle: it is required for type checking [16] and for higher-order unification of polymorphic terms as described in Section 3.

The following definitions are consistent with [21, 22, 25]. Note that the naming conventions established in Section 2 do not all carry over. This is important to keep in mind since the order-sorted *terms* discussed below are the *types* in the $\lambda$-calculus discussed in Section 2.

An *order-sorted signature* consists of a set of sorts $S$, a partial order $\leq$ on $S$, and a set of function declarations $f : (\overline{s_n})s$. Given a fixed signature and an $S$-sorted set $V = \bigcup_{s \in S} V_s$ of disjoint sets of variables, a set of typed terms is defined inductively:

$$\frac{x \in V_s \quad s \leq s'}{x : s'} \qquad\qquad \frac{f : (\overline{s_n})s \quad s \leq s' \quad \forall i.\ t_i : s_i}{f(\overline{t_n}) : s'}$$

The definitions of substitutions, unifiers, complete sets of unifiers etc. are straightforward generalizations of the unsorted case and can be found for example in [21]. In the sequel the ordering $\leq$ is extended from $S$ to $S^*$ in the canonical componentwise way.

A signature is called *regular* [22] if every term has a least sort. Regularity is decidable for finite signatures:

**Theorem 4.1** (Smolka et al. [22, 25]) *A signature $(S, \leq, \Sigma)$ is regular iff for every $f \in \Sigma$ and $w \in S^*$ the set $\{s \mid \exists w' \geq w.\ f : (w')s\}$ either is empty or contains a least element.*

Regularity is important because it implies that order-sorted unification is finitary. Order-sorted unification in non-regular signatures may be infinitary [22].

As in the section on higher-order unification, the algorithm is presented as a rewrite system on multisets of equations. The following rules are similar to those in Jouannaud and Kirchner's survey [8]. With respect to some fixed signature define

$$\begin{aligned} s_1 \wedge s_2 &= max\{s \mid s \leq s_1 \wedge s \leq s_2\} \\ D(f, s) &= max\{w \mid \exists s'.\ f : (w)s' \wedge s' \leq s\} \end{aligned}$$

[1]Which happen to be called *sorts* in OBJ, an unfortunate confusion.

If $E = \{x_1 =^? t_1, \ldots, x_n =^? t_n\}$ such that all $x_i$ are distinct and no $x_i$ occurs in one of the $t_j$, $E$ is in *solved form* and $\vec{E} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a most general unifier of $E$.

$$\{t \stackrel{?}{=} t\} \cup E \Longrightarrow E$$

$$\{f(\overline{r_n}) \stackrel{?}{=} f(\overline{t_n})\} \cup E \Longrightarrow \{r_1 \stackrel{?}{=} t_1, \ldots, r_n \stackrel{?}{=} t_n\} \cup E$$

$$\frac{t \notin V}{\{t =^? x\} \cup E \Longrightarrow \{x =^? t\} \cup E}$$

$$\frac{x \in V_s \cap \mathcal{V}(E) \quad x \notin \mathcal{V}(t) \quad t : s}{\{x =^? t\} \cup E \Longrightarrow \{x =^? t\} \cup \{x \mapsto t\}(E)}$$

$$\frac{x \in V_s \quad y \in V_{s'} \quad s < s'}{\{x =^? y\} \cup E \Longrightarrow \{y =^? x\} \cup E}$$

$$\frac{x \in V_s \quad y \in V_{s'} \quad s \not\leq s' \quad s' \not\leq s \quad z \in V_{s''} \quad s'' \in s \wedge s' \quad z \notin \{x, y\} \cup \mathcal{V}(E)}{\{x =^? y\} \cup E \Longrightarrow \{x =^? z, y =^? z\} \cup E}$$

$$\frac{x \in V_s \quad n > 0 \quad \overline{s_n} \in D(f, s) \quad \forall i.\ x_i \in V_{s_i} \wedge x_i \notin \{x\} \cup \mathcal{V}(t_1, \ldots, t_n, E)}{\{x =^? f(\overline{t_n})\} \cup E \Longrightarrow \{x =^? f(\overline{x_n}), x_1 =^? t_1, \ldots, x_n =^? t_n\} \cup E}$$

This algorithm is less deterministic but essentially equivalent to the ones given by Schmidt-Schauß [20] and Waldmann [25]. Adapting their results one can easily show:

**Theorem 4.2** *For any regular signature the set of all $\vec{S}$ such that $E \Longrightarrow^* S$ and $S$ is in solved form is a complete set of unifiers for $E$.*

It follows that in finite regular signatures, where $s \wedge s'$ and $D(f, s)$ must be finite, finite complete sets of unifiers always exist.

The remainder of this section focuses on the existence of single most general unifiers. Since our terms represent types, this happens to coincide with the existence of *principal types*. A lack of principal types is undesirable for two reasons: expressions may have any finite number of incomparable types, making them rather hard to comprehend from a user's point of view; finite sets of type unifiers increase the search space for higher-order unification.

Waldmann [25] characterizes *unitary signatures*, i.e. those leading to unitary unification problems[2]. He calls a partial order $(S, \leq)$ *downward complete* iff the set $s \wedge s'$ contains at most one element.

**Theorem 4.3** (Waldmann [25]) *A regular signature is unitary iff it is downward complete and for all $s$, $w$, and $f : (w^0)s^0$ such that $w \leq w^0$ and $s \leq s^0$ the set*

$$W(f, w, s) = \{w' \mid \exists \overline{w}, \overline{s}.\ f : (\overline{w})\overline{s} \wedge \overline{s} \leq s \wedge w' \leq w \wedge w' \leq \overline{w}\}$$

*either is empty or contains a greatest element.*

[2]The characterization results by Meseguer et. al. [13] do not immediately apply to our situation because they admit unsorted variables in their unification problems.

Unfortunately, the definition of the set $W(f, w, s)$ is not very intuitive, difficult to check, and hence unsuitable as a guideline for users.

The programming language Haskell [7] solves this problem by imposing a number of severe context conditions which are sufficient to guarantee principal types. Although they are not necessary, they make a lot of sense from a programming language point of view. Their essence is captured in the next lemma, which follows easily from Lemma 4.2 below.

**Lemma 4.1** *A signature is unitary if it is finite, regular, downward complete,*

- injective*:* $f : (w)s$ *and* $f(w')s$ *imply* $w = w'$*, and*
- subsort reflecting*:* $f : (w')s'$ *and* $s' \leq s$ *imply the existence of* $w$ *such that* $w' \leq w$ *and* $f : (w)s$*.*

Isabelle uses a slightly weaker criterion called *coregularity.* A signature is *coregular* if for all $f$ and $s$ the set $D(f, s)$ contains at most one element. The following lemma goes back to Schmidt-Schauß and is also given by Smolka et al. [22].

**Lemma 4.2** *Signatures which are finite, regular, downward complete, and coregular are unitary.*

This lemma follows directly from the correctness of the unification algorithm above because downward completeness and coregularity eliminate all essential nondeterminism from that algorithm.

It is easy to see that injectivity and subsort reflection together imply coregularity but that coregularity implies neither. Nevertheless, the two criteria are equivalent in the following sense: given a coregular signature $\Sigma$, there exists an injective and subsort reflecting signature $\Sigma'$ on the same sort structure but with declarations

$$\{f : (w)s \mid D_\Sigma(f, s) = \{w\}\}$$

such that $t : s$ holds w.r.t. $\Sigma$ iff it holds w.r.t. $\Sigma'$. This means that the sorting judgements derivable w.r.t. $\Sigma$ and $\Sigma'$ are identical. Hence $\Sigma$ and $\Sigma'$ are interchangeable for all intents and purposes.

## 5 Speculation

The algorithm for higher-order unification in the presence of type variables presented in Section 3 is incomplete. The source of this incompleteness is our inability, given a type $\sigma$, to find a finite representation for the set of all types $\overline{\sigma_n} \to \sigma$. We propose a radical extension to the type system which overcomes this limitation. We introduce a "pseudo"-product type $*$ and a unit-type 1 which are related to $\to$ as follows:

$$\begin{array}{rclrcl}
\alpha * \beta \to \gamma &=& \alpha \to \beta \to \gamma & \alpha \to \beta * \gamma &=& (\alpha \to \beta) * (\alpha \to \gamma) \\
\alpha * 1 &=& \alpha & 1 * \alpha &=& \alpha \\
1 \to \alpha &=& \alpha & \alpha \to 1 &=& 1 \\
(\alpha * \beta) * \gamma &=& \alpha * (\beta * \gamma) & & &
\end{array}$$

Read from left to right, this is a terminating and confluent rewriting system. Normal forms are products of simple types. Notice that this system, together with commutativity of $*$, axiomatizes isomorphism in Cartesian Closed Categories. The unification problem for the latter and related systems, but not for the one above, has already been considered by Narendran et al. [15].

The point of this new type system is that the set of all types $\overline{\sigma_n} \to \sigma$ can be represented by $\alpha \to \sigma$, where $\alpha$ is a type variable. To obtain $\overline{\sigma_n} \to \sigma$, simply instantiate $\alpha$ with $\sigma_1 * \cdots * \sigma_n$.

The new type constructors $*$ and $1$ are accompanied by two new term constructors $\langle \_, \_ \rangle$ and $\langle \rangle$ and a number of equalities between $\lambda$-terms containing pairs and units.

Whether this approach leads to a complete higher-order unification algorithm for polymorphic $\lambda$-terms over these new types is not clear yet, in particular since the unification problem for the types themselves is now highly nontrivial (it contains associative unification as a special case).

## Acknowledgements

The author wishes to thank Larry Paulson and David Wolfram for many discussions on higher-order unification and polymorphism, Uwe Waldmann for his patient explanations of order-sorted unification, and Annette Schumann for proof reading.

## References

[1] P.B. Andrews, D.A. Miller, E.L. Cohen, F. Pfenning. Automating Higher-Order Logic. In *Automated Theorem Proving: After 25 Years*, AMS Contemporary Mathematics Series 29 (1984), 169–192.

[2] H. Barendregt. Introduction to Generalised Type Systems. To appear in *J. Functional Programming.*

[3] N. G. de Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae* **34** (1972), 381–392.

[4] N. Dershowitz, J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen (editor), *Handbook of Theoretical Computer Science, Vol B: Formal Methods and Semantics*, North-Holland, to appear.

[5] C. Elliot. Higher-Order Unification with Dependent Function Types. *Proc. Rewriting Techniques and Applications*, LNCS 355 (1989), 121–136.

[6] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer. Principles of OBJ2. *Proc. 12th ACM Symp. Principles of Programming Languages* (1985), 52–66.

[7] P. Hudak, P. Wadler (Eds.). *Report on the Programming Language Haskell.* Version 1.0, April 1990.

[8] J.-P. Jouannaud, C. Kirchner. *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification.* Technical report, March 1990.

[9] W. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science* **13** (1981), 225-230.

[10] J.R. Hindley, J.P. Seldin. *Introduction to Combinators and $\lambda$-Calculus*, Cambridge University Press (1986).

[11] G.P. Huet. A Unification Algorithm for Typed $\lambda$-Calculus. *Theoretical Computer Science* **1** (1975), 27–57.

[12] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis (1984).

[13] J. Meseguer, J.A. Goguen, G. Smolka. Order-Sorted Unification. *J. Symbolic Computation* **8** (1989), 383–413.

[14] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD Thesis, University of Pennsylvania (1987).

[15] P. Narendran, F. Pfenning, R. Statman. *On the Unification Problem for Cartesian Closed Categories.* Ergo Report 89–082, School of Computer Science, Carnegie Mellon University, September 1989.

[16] T. Nipkow, G. Snelting. *Type Classes and Overloading Resolution via Order-Sorted Unification.* Tech. Rep. 200, University of Cambridge, Computer Laboratory, August 1990.

[17] L.C. Paulson. Natural Deduction as Higher-Order Resolution. *J. Logic Programming* **3** (1986), 237–258.

[18] L.C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi (editor), *Logic and Computer Science*, Academic Press (1990), 361–385.

[19] L.C. Paulson, T. Nipkow: *Isabelle Tutorial and User's Manual*, Tech. Rep. 189, University of Cambridge, Computer Laboratory, January 1990.

[20] M. Schmidt-Schauß. A Many-Sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation. *Proc. 9th Int. Joint Conf. Artificial Intelligence* (1985), 1162–1168.

[21] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, LNCS 395 (1989).

[22] G. Smolka, W. Nutt, J.A. Goguen, J. Meseguer. Order-Sorted Equational Computation. In H. Aït-Kaci and M. Nivat (eds.), *Resolution of Equations in Algebraic Structures Vol. 2*, Academic Press (1989), 297–367.

[23] W. Snyder, J. Gallier. Higher-Order Unification Revisited: Complete Sets of Transformations. *J. Symbolic Computation* **8** (1989), 101–140.

[24] P. Wadler, S. Blott. How to Make *ad-hoc* Polymorphism Less *ad hoc*. *Proc. 16th ACM Symp. Principles of Programming Languages* (1989), 60–76.

[25] U. Waldmann: *Unification in Order-Sorted Signatures.* Forschungsbericht 298, Universität Dortmund, Fachbereich Informatik (1989).