Order-Sorted Polymorphism in Isabelle*

Tobias Nipkow†
Institut für Informatik
TU München
Postfach 20 24 20
8000 München 2
Germany

April 1992

Abstract

ML-style polymorphism can be generalized from a single-sorted algebra of types to an order-sorted one by adding a partially ordered layer of "sorts" on top of the types. Type inference proceeds as in the Hindley/Milner system, except that order-sorted unification of types is used. The resulting system has been implemented in Isabelle to permit type variables to range over user-definable subsets of all types. Order-sorted polymorphism allows a simple specification of type restrictions in many logical systems. It accommodates user-defined parametric overloading and allows for a limited form of abstract axiomatic reasoning. It can also explain type inference with Standard ML's equality types and Haskell's type classes.

1 Introduction

This paper describes a recent extension of the generic theorem prover Isabelle. The extension, namely the introduction of "order-sorted polymorphism", was motivated by the desire to obtain simple axiomatizations of typed object-logics. The rest of the introduction is devoted to Isabelle and the motivation for its extension. Section 2 introduces the new type system by way of examples: a series of object-logic definitions is shown which demonstrate the various aspects of the type system. Section 3 describes the underlying technicalities, including type inference and higher-order unification. It is followed by a section of a more speculative nature which discusses order-sorted polymorphism as a module facility for abstract axiomatic reasoning. Section 5 sketches how to rephrase Standard ML's type system in terms of order-sorted polymorphism.

Isabelle is a generic interactive theorem prover developed by Lawrence Paulson [21, 22]. It is generic in the sense that it can be parameterized with the intended object-logic. Its meta-logic is intuitionistic higher-order logic and its inference mechanism is based on higher-order unification. In order to distinguish the old and the new version of Isabelle I will use the qualifiers "90" and "91", respectively.

The presentation of an object-logic in Isabelle-90 consists of

*Isabelle is written in Standard ML. The source files and documentation can be obtained free of charge by contacting the author.

†E-mail: `Tobias.Nipkow@Informatik.TU-Muenchen.De`. Research supported by ESPRIT BRA 3245, *Logical Frameworks*, and carried out at the Computer Laboratory, University of Cambridge.

- a signature extension, introducing the new types and constants describing the syntax of formulae, and
- new axioms describing the inference rules of the logic.

In the sequel I use an OBJ-inspired [7] syntax which is very close to the format actually used in the implementation. As a tiny example, consider the following definition of propositional logic.

$$\begin{array}{rcl} \mathcal{P}rop & = & \textbf{types}\ form \\ & & \textbf{consts}\ \_\wedge\_ \ :\ form \to form \to form \\ & & \qquad\qquad\ \neg\ :\ form \to form \\ & & \qquad\qquad\ [\_]\ :\ form \to prop \\ & & \textbf{rules}\ [P] \Longrightarrow [Q] \Longrightarrow [P \wedge Q] \\ & & \qquad\ ([\neg P] \Longrightarrow [P]) \Longrightarrow [P] \end{array}$$

The only new type is $form$, the type of formulae. The only connectives are conjunction ($\wedge$) and negation ($\neg$). In Isabelle-90, constants can be of base type or of arbitrary function type over the base types. Although the absence of a nullary constant, for example $true$, means there are no closed formulae, expressions like $P \wedge Q$, where P and Q are variables, are legal, provided both P and Q are of type $form$. Variables need not be declared and their type is inferred automatically: writing $P \wedge Q$ forces P and Q to have type $form$. Finally there is the constant $[\_]$ which maps $form$ to the predefined type $prop$ of propositions. The proposition $[P]$ should be read as "The formula P is provable". This coercion is necessary because object-level inference rules are written as meta-level axioms, i.e. expressions of type $prop$. For example the first rule above is the linear form of the text-book inference

$$\frac{P \quad Q}{P \wedge Q}$$

using the pre-defined meta-level implication $\Longrightarrow$ of type $prop \to prop \to prop$. Hence it is impossible simply to write $P \Longrightarrow Q \Longrightarrow P \wedge Q$. In the sequel square brackets are dropped. This improves readability and is in line with actual Isabelle practice, where parser and pretty-printer take care of such matters.

This is an intentionally fragmentary presentation of propositional logic — many rules and connectives are missing. Throughout the paper I adopt the convention of presenting only as much of a logic as is necessary to make certain points.

We can now extend propositional logic to a single sorted first-order logic by adding a new type of terms and a quantifier:

$$\begin{array}{rcl} \mathcal{P}red & = & \mathcal{P}rop\ +\ \textbf{types}\ term \\ & & \qquad\qquad\ \textbf{consts}\ \forall\ :\ (term \to form) \to form \\ & & \qquad\qquad\ \textbf{rules}\ \forall(P) \Longrightarrow P(t) \end{array}$$

Because Isabelle is based on higher-order logic, its expressions are simply typed λ-terms. Quantifiers can thus be expressed as functionals. This means that internally the formula $\forall x.P(x)$, where P is of type $term \to form$, is represented as $\forall(\lambda x.P(x))$, which is equivalent to $\forall(P)$. Again, parser and pretty-printer permit these different concrete syntaxes to coexist. The one inference rule shown is $\forall$-*elimination* or *specialization*.

This definition of predicate logic is fine as far as it goes, but has one major drawback: it is single-sorted, i.e. there is only a single type $term$ of individuals. Any attempt to do many-sorted logic on top of it is as prone to type errors as programming in an untyped language. If we were to extend $\mathcal{P}red$ further by two constants $succ : term \to term$ and $[] : term$ representing the successor function on natural numbers and the empty list, the (in our understanding) meaningless expression $succ([])$ is well-formed and of type $term$. There are several ways to deal with the problem of representing typed object-logics in Isabelle.

The generic representation of typed logics in Isabelle, and in any other such system, is to introduce types and type checking into the object-language. This means there is a new type of object-level types and the inference rules are augmented to ensure that only well-typed theorems can be deduced. Under this approach, the definition of $\mathcal{P}red$ might look as follows:

$$
\begin{array}{rcll}
\mathcal{P}red & = & \mathcal{P}rop \; + & \textbf{types}\; term, type \\
 & & & \textbf{consts} \quad \forall : type \to (term \to form) \to form \\
 & & & \qquad\quad {}_{-} \in {}_{-} : term \to type \to prop \\
 & & & \textbf{rules}\; t \in T \Longrightarrow \forall(T,P) \Longrightarrow P(t)
\end{array}
$$

The universal quantifier has acquired a new argument, namely the type that the bound variable is ranging over. The judgement $\in$ is introduced to form explicit typing assertions at the object-level. The rule of specialization is guarded by the expression $t \in T$ which makes sure that only elements of the right type are substituted for the bound variable.

A further extension with new object-level types, say the natural numbers, might look like this:

$$
\begin{array}{rcll}
\mathcal{N}at & = & \mathcal{P}red \; + & \textbf{consts} \quad nat : type \\
 & & & \qquad\qquad\quad 0 : term \\
 & & & \qquad\qquad succ : term \to term \\
 & & & \textbf{rules} \;\; 0 \in nat \\
 & & & \qquad\quad n \in nat \Longrightarrow succ(n) \in nat
\end{array}
$$

In general this explicit axiomatization of typing rules is the only way to proceed, in particular if the type system is undecidable, for example Intuitionistic Type Theory [15]. On the other hand, it is frustrating to deal with object-logic typing judgements for simple systems like many-sorted predicate logics. Not only does it clutter up the specification of the logic, but it also introduces additional type checking obligations during a proof. Although these proof obligations can be dealt with automatically by tactics, the incurred overhead is often significant. This is particularly annoying because Isabelle has a built-in type inference system which is quite capable of dealing with many sorted logics, provided the meta-logic type system is used.

The idea of identifying meta-logic and object-logic types is the key to inheriting Isabelle's built-in facilities. Of course it presupposes that the object-logic type system is weak enough. It fails for logics with dependent types or subtypes, neither of which can be modelled by the simple meta-types. In fact, in Isabelle-90 the idea does not work at all, as we shall now see. It is straightforward to declare different meta-types like nat and $list$ and constants $succ : nat \to nat$ and $[] : list$, thus preventing expressions like $succ([])$. However, we now need a quantifier for each of these types, i.e. $\forall_{nat} : (nat \to form) \to form$ and $\forall_{list} : (list \to form) \to form$, and the corresponding rules for each of them. This may just about be feasible for a small number of fixed types but is completely impractical in general.

In order to make this approach work, we need polymorphism, which is exactly what Isabelle-91 and this paper are all about. However, a naïve adoption of ML-style polymorphism is dangerous, as the following example shows:

$$
\begin{array}{rcll}
\mathcal{P}red & = & \mathcal{P}rop \; + & \textbf{consts}\; \forall : (\alpha \to form) \to form \\
 & & & \textbf{rules}\; \forall x.P(x) \Longrightarrow P(t)
\end{array}
$$

The universal quantifier has a polymorphic type. The type variable α is implicitly universally quantified at the level of types, as is the usual practice. Type inference determines the following typings of the variables in the specialization rule: $x : \alpha$, $P : \alpha \to form$ and $t : \alpha$. Unfortunately, this specification is too general, at least for first-order logic. The intention is that α ranges only over different types of "individuals", but certainly not over formulae or arbitrary function types. However, there is nothing in this declaration to enforce those constraints. As a consequence, some rather surprising inferences are possible. Using the substitution $\{\alpha \mapsto form, P \mapsto \lambda x.x\}$, specialization degenerates to $\forall x.x \Longrightarrow t$. In a first order-logic, $\forall x.x$ is simply ill-formed. This formula could only arise because of the instantiation of α by $form$. In a higher-order logic, $\forall x.x$ is a perfectly legitimate formula, usually identified with falsity, and the rule expresses *ex falso quodlibet*.

2 Order-Sorted Polymorphism by Example

Having demonstrated the need for tighter control over instantiations of type variables, this section describes a simple extension of the Hindley/Milner type discipline which achieves just that without sacrificing any of the properties that make Hindley/Milner polymorphism attractive. The basic idea is to introduce an additional layer of *sorts* on top of the types, together with a new judgement $\tau : s$, where τ is a type and s a sort. This means in particular that type variables have a sort, thus restricting their instantiations to types of the required sort. Sorts denote subsets of the set of all types. In order to specify hierarchies of such subsets, sorts can be partially ordered, thus leading to an order-sorted algebra of types. It must be stressed that there is no notion of subtypes involved. This is in contrast to, for example, OBJ [7], where the types[1] themselves are partially ordered.

Although this notion of sorts is superficially similar to the notion of "kinds" in the Edinburgh Logical Framework [9], the reader will see that it is qualitatively quite different, not least because of the partial order on sorts. It is much more closely related to the Haskell notion of "classes" [10, 20].

2.1 Specifying Logics

We will now introduce the different features of order-sorted polymorphism by example, going through a series of logic definitions in Isabelle-91, starting with Isabelle's meta-logic. As we are only interested in the type system, none of the inference rules are shown.

$$
\begin{array}{rcll}
\mathcal{M}eta & = & \textbf{sorts}\ \top & \\
 & & \textbf{types}\ \_ \to \_ & : \ (\top, \top)\top \\
 & & \qquad\qquad prop & : \ \top \\
 & & \textbf{consts}\qquad \bigwedge & : \ (\alpha_\top \to prop) \to prop \\
 & & \qquad\qquad \_ \Longrightarrow \_ & : \ prop \to prop \to prop \\
 & & \qquad\qquad \_ \equiv \_ & : \ \alpha_\top \to \alpha_\top \to prop
\end{array}
$$

Since $\mathcal{M}eta$ is not an extension of anything else, we start from ground zero, i.e. no sorts, types etc. are known. The **sorts** section, not present in Isabelle-90, introduces a single new sort $\top$. The intention is that $\top$ should be the sort of all types. The **types** section is different from its Isabelle-90 counterpart in that it does not just list the types but also gives them an arity. Here

[1] called *sorts* in that context, making the terminology somewhat confusing.

there are two types, namely the base type *prop* of sort $\top$ and the function type constructor $\rightarrow$ which takes two arguments of sort $\top$ and returns a result of sort $\top$. The notation $t : (s_1, \ldots, s_n)s$ was chosen instead of something like $t : s_1 \rightarrow \cdots \rightarrow s_n \rightarrow s$ in order not to overload $\rightarrow$. Note that with respect to the current type signature, all types built from *prop* and $\rightarrow$, i.e. all ground types, are indeed of sort $\top$.

The **consts** section introduces the three basic connectives of the meta-logic: universal quantification, implication, and equality. The type of $\bigwedge$ is very general, i.e. it is possible to quantify over *prop* and over arbitrary function types. Thus we have defined an impredicative meta-logic. A predicative variant could have been defined as follows:

$$\begin{array}{rcl} \mathcal{M}eta' & = & \textbf{sorts}\ \top \\ & & \qquad\quad i < \top \\ & & \textbf{types}\ \_ \rightarrow \_ \ :\ (\top,\top)\top \\ & & \qquad\quad prop\ :\ \top \\ & & \textbf{consts}\qquad \bigwedge\ :\ (\alpha_i \rightarrow prop) \rightarrow prop \\ & & \qquad\quad \_ \Longrightarrow \_\ :\ prop \rightarrow prop \rightarrow prop \\ & & \qquad\qquad \_ \equiv \_\ :\ \alpha_\top \rightarrow \alpha_\top \rightarrow prop \end{array}$$

Here we introduced a subsort i of "individuals". So far, no type lies in i, but the intention is that types introduced in object-logics should reside there if they are to be quantified over. Meta-level quantification over *prop* or function types is excluded because neither are of sort i. Thus we have arrived at a predicative meta-logic.

Object-logics are defined as extensions of the meta-logic. In order to demonstrate the full flexibility of order-sorted polymorphism we look at definitions of first-, second-, and higher-order logic.

$$\begin{array}{rcl} \mathcal{FOL} & = & \mathcal{M}eta + \textbf{sorts}\ i < \top \\ & & \qquad\qquad \textbf{types}\ form : \top \\ & & \qquad\qquad \textbf{consts}\ True\ :\ form \\ & & \qquad\qquad\qquad\quad \_=\_\ :\ \alpha_i \rightarrow \alpha_i \rightarrow form \\ & & \qquad\qquad\qquad\qquad \forall\ :\ (\alpha_i \rightarrow form) \rightarrow form \end{array}$$

Quantification over and equality between formulae or function types is ruled out because neither has sort i. For example $True = True$ is not well-typed. The sort i of "individuals" is initially empty but further extensions may change this:

$$\begin{array}{rcl} \mathcal{N}at & = & \mathcal{FOL} + \textbf{types}\ nat : i \\ & & \qquad\qquad \textbf{consts}\qquad 0\ :\ nat \\ & & \qquad\qquad\qquad\quad succ\ :\ nat \rightarrow nat \end{array}$$

The formula $\forall x.x = 0$ is legal, with x having the inferred type *nat* which is of sort i. On the other hand $\forall f.f(0) = 0$ is ill-typed because f must have type $nat \rightarrow nat$ which is not of sort i.

A more liberal dose of quantification is granted in second-order logic:

$$\begin{array}{rcl} \mathcal{SOL} & = & \mathcal{M}eta + \textbf{sorts}\ q\ <\ \top \\ & & \qquad\qquad\qquad i\ <\ q \\ & & \qquad\qquad\qquad p\ <\ q \\ & & \qquad\quad \textbf{types}\ form\ :\ p \\ & & \qquad\qquad\qquad \_ \rightarrow \_\ :\ (i,p)p \\ & & \qquad\quad \textbf{consts}\ \forall : (\alpha_q \rightarrow form) \rightarrow form \end{array}$$

This type structure is more involved. The sort q denotes all types that one can quantify over. These are all individuals (hence $i < q$) and all n-ary predicates of type $\tau_1 \to \cdots \to \tau_n \to form$, where each τ_j is a type of sort i, such as nat. The latter set of types is called p and there are two declarations which generate types in p: $form$ is in p, corresponding to the case $n = 0$, and if $\tau : i$ and $\sigma : p$, then $\tau \to \sigma$ is also in p. It should be clear that this defines exactly the intended set of predicate types.

$\mathcal{SOL}$ is interesting because it presents a new feature of the type system, namely overloading: the function type constructor has two arities, $(\top, \top)\top$, which is inherited from $\mathcal{M}eta$, and $(i,p)p$. Neither subsumes the other: $form \to form : \top$ and $nat \to form : p$ require the first and second arity respectively.

Another example of overloading is found in the definition of higher-order logic:

$$\begin{array}{rcll} \mathcal{HOL} & = & \mathcal{M}eta \; + & \textbf{sorts}\; i < \top \\ & & & \textbf{types}\; form \;:\; i \\ & & & \qquad\quad \_ \to \_ \;:\; (i,i)i \\ & & & \textbf{consts}\; \forall : (\alpha_i \to form) \to form \end{array}$$

Here we need only the subsort i of individuals which contains $form$ and any function type over individuals.

It should be stressed that the declaration of sorts, types and constants resembles the assertion of axioms: there is no way that Isabelle could check the "correctness" of these declarations. For example, it is possible to extend $\mathcal{FOL}$ with a type declaration $form : i$, thus enabling the quantification over formulae and stepping outside first-order logic.

2.2 Overloading

The flavour of all the examples so far is that of parametric polymorphism. This section shows how order-sorted polymorphism can also be used to specify *ad-hoc* polymorphism or *overloading.* Suppose we would like to use the symbols $+$ and $\leq$ at more than one type, for example both natural numbers and strings (where $+$ might denote concatenation). Isabelle's type system does not allow the simultaneous declaration

$$\begin{array}{rrl} \textbf{consts} & \_+\_ & : \; nat \to nat \to nat \\ & \_+\_ & : \; string \to string \to string \end{array}$$

However, $+$ and $\leq$ can be declared once and for all as polymorphic operators:

$$\begin{array}{rcll} \mathcal{OFOL} & = & \mathcal{FOL} \; + & \textbf{sorts}\; a \;<\; i \\ & & & \qquad\quad o \;<\; i \\ & & & \textbf{consts}\; \_+\_ \;:\; \alpha_a \to \alpha_a \to \alpha_a \\ & & & \qquad\quad\;\; \_\leq\_ \;:\; \alpha_o \to \alpha_o \to form \end{array}$$

This requires some explanation. The two sorts a and o are meant to represent those types which provide addition $(+)$ and an ordering $(\leq)$ respectively. More precisely, any type τ of sort a automatically gives rise to a constant $+ : \tau \to \tau \to \tau$, and similarly for $\leq$. At the moment there are no such types, but it is still possible to state some generic rules for these operations, for example:

$$\begin{array}{rl} \textbf{rules} & x \leq x \\ & x + y = y + x \end{array}$$

Of course such polymorphic assertion may be a bit rash: commutativity of $+$ rules out its use for string or list concatenation. A more general treatment of overloaded operations satisfying certain laws can be found in Section 4.

So far we only have generic operations but no instances. The latter are created by asserting that some type is of sort a or o:

$$
\begin{array}{lcll}
\mathcal{N}at & = & \mathcal{OFOL} + & \textbf{types}\ nat\ :\ a \\
 & & & \phantom{\textbf{types}}\ nat\ :\ o \\
 & & & \textbf{consts}\quad 0\ :\ nat \\
 & & & \phantom{\textbf{consts}}\ succ\ :\ nat \to nat \\
 & & & \textbf{rules}\ 0 + n = n \\
 & & & \phantom{\textbf{rules}}\ succ(m) + n = succ(m+n) \\
 & & & \phantom{\textbf{rules}}\ 0 \le n \\
 & & & \phantom{\textbf{rules}}\ succ(m) \le succ(n)\ \leftrightarrow\ m \le n \\
 & & & \phantom{\textbf{rules}}\ \neg(succ(m) \le 0)
\end{array}
$$

We could further enrich $\mathcal{N}at$ with strings and define a suitable concatenation and ordering on strings, provided strings are also declared to be of sorts a and o. However, that would create a problem: what is the type of x in

$$(x \le x) \wedge (x + y = y + x) \quad ? \tag{1}$$

Morally one would expect it to be α_a and α_o simultaneously, but there is no sort which subsumes both a and o. Hence this formula is ill-typed in the context of $\mathcal{OFOL}$. In the presence of both nat and $string$, however, there are two valid, and in fact most general typings: $x : int$ and $x : string$. Hence we have lost the principal type property, i.e. expressions fail to have most general types. In Isabelle this situation cannot arise because the system will only admit those type and sort declarations which preserve the principal type property. Exactly which restrictions Isabelle enforces is discussed in the next section. The problem with $\mathcal{N}at$ is that nat does not have a least sort. Hence it is necessary to declare a new sort ao which lies below both a and o, i.e. ao is the intersection of the types in a and o: $\mathcal{OFOL}$ should contain a third sort clause $ao < a, o$, and both nat and $string$ should be declared to be of sort ao. The variable x in (1) now has type ao. In general this leads to an exponential blow-up in the number of sorts required, as one has to span the whole powerset of basic sorts. A more refined type system than the one implemented in Isabelle would automatically operate on the powerset level and x in (1) would be assigned the type $\alpha_{\{a,o\}}$. This is precisely what happens in Haskell [10, 20]. A similar type system, where the sort names are identified with the names of the overloaded operators, i.e. $a \cong +$ and $o \cong \le$, was first proposed by Stefan Kaes [14] who also noted the connection with order-sorted unification.

It should have become clear that the two uses of order-sorted polymorphism in sections 2.1 and 2.2 correspond to parametric and ad-hoc polymorphism respectively. In Section 2.1 specific types like nat inherit both the names *and* the properties of generic operations like quantification and equality: $\forall$ on natural numbers behaves exactly like $\forall$ on any other type of sort i. In Section 2.2, on the other hand, $+$ is declared without any properties attached, and nat inherits only the name, but defines its own laws for $+$, which may be quite different from those for $+$ on strings.

3 Order-Sorted Types and Unification

The examples of the previous section should have prepared the ground for the general notion of order-sorted polymorphism which this section presents in a largely Isabelle-independent way. We begin technicalities by summarizing and restating some basic notions of order-sorted algebras (see, for example, [24]) in terms of types.

Fix a set of *sorts* S and a set of *type constructors* T. An *(order-sorted) type signature* is a pair $(\leq, \Sigma)$ where $\leq$ is a partial order on S and $\Sigma \subseteq T \times S^* \times S$ is a set of declarations. In Isabelle-91 these two components are declared in the **sorts** and **types** sections respectively. In the sequel let $(\leq, \Sigma)$ be some arbitrary but fixed type signature and let $X = \bigcup_{s \in S} X_s$ be an S-sorted family of disjoint sets of *variables*. Instead of $(t, w, s) \in \Sigma$ we write $t : (w)s$. The relation $\leq$ extends to S^* in the componentwise way.

The set of *sorted types* is inductively defined as follows:

$$\frac{\alpha \in X_s}{\alpha : s} \qquad \frac{t : (\overline{s_n})s \quad \forall i.\ \tau_i : s_i}{t(\overline{\tau_n}) : s} \qquad \frac{\tau : s \quad s \leq s'}{\tau : s'}$$

In the sequel τ denotes arbitrary types, α and β stand for elements of X, and we write α_s to indicate that $\alpha \in X_s$. Note that "$\rightarrow$" is just one among many other type constructors and needs no special treatment.

3.1 Order-Sorted Unification

In Isabelle-91, the principal operation on types is unification. It is required both for type inference and resolution. The definitions of substitutions, unifiers, complete sets of unifiers etc. are straightforward generalizations of the unsorted case and can be found in the general literature on order-sorted unification [23, 24, 16].

Order-sorted unification differs from unsorted unification in the cardinality of minimal complete sets of unifiers. In the unsorted case it is at most 1 (called *unitary*), in the order-sorted case it depends on the structure of $\leq$ and Σ and may be infinite. Isabelle-91 is restricted to unitary type signatures. Thus the principal type property for the language of terms is guaranteed and the nondeterminism of higher-order unification does not increase further. Although there are precise characterizations of unitary type signatures [28], the implementation relies on two sufficient but not necessary criteria, regularity and coregularity, which lead to particularly simple algorithms.

A type signature is *regular* if every type has a least sort. Regularity is decidable for finite signatures:

Theorem 3.1 (Smolka *et al.* [24]) *A signature $(\leq, \Sigma)$ is regular iff for every $t \in T$ and $w \in S^*$ the set $\{s \mid \exists w' \geq w.\ t : (w')s\}$ either is empty or contains a least element.*

Regularity is essential because order-sorted unification in non-regular signatures may be infinitary [24]. To arrive at unitary signatures we need two additional properties.

$$\begin{aligned} s_1 \sqcap s_2 &= \{s \mid s \leq s_1 \land s \leq s_2\} \\ D(t, s) &= \{w \mid \exists s'.\ t : (w)s' \land s' \leq s\} \end{aligned}$$

A partial ordering is called *downward complete* if any two elements with a lower bound have a greatest lower bound, i.e. if $s_1 \sqcap s_2$ either is empty or contains a greatest element. A signature is *coregular* if for all $t \in T$ and all $s \in S$ the set $D(t,s)$ either is empty or contains a greatest element. The finitary version of the following lemma goes back to Schmidt-Schauß and is also given by Smolka *et al.* [24].

Lemma 3.2 *Every regular, downward complete, and coregular signature is unitary.*

Isabelle-91 requires all type signatures to be regular, downward complete, and coregular, thus enforcing unitary unification. Comparing this lemma with the results of Waldmann [28] it turns out that only coregularity is unnecessarily strong. However it leads to a very simple algorithm described below. This algorithm is a specialization of known algorithms to downward complete and coregular signatures. Its correctness and completeness can easily be established by means of program transformation.

The algorithm is expressed by rewrite rules on pairs $\langle E, \theta\rangle$, where E is the list[2] of pairs $\tau =^? \tau'$, the problems yet to be solved, and θ is a substitution, the fragment of the solution computed so far. Solving such a pair means rewriting it to normal form. If the normal form is $\langle [], \theta\rangle$, then θ is the solution to the initial problem. Otherwise the initial problem has no solution.

$$
\begin{array}{rcll}
\langle(\alpha =^? \alpha) :: E, \theta\rangle & \Longrightarrow & \langle E, \theta\rangle & \\
\langle(\tau =^? \alpha) :: E, \theta\rangle & \Longrightarrow & \langle(\alpha =^? \tau) :: E, \theta\rangle & \text{if } \tau \notin X \\
\langle(\alpha_s =^? \tau) :: E, \theta\rangle & \Longrightarrow & \langle\theta'(E), W(\tau, s) \circ \theta' \circ \theta\rangle & \text{if } \alpha \text{ does not occur in } \tau \\
 & & & \text{and } \theta' = \{\alpha \mapsto \tau\} \\
\langle(t(\overline{\tau_n}) =^? t(\overline{\tau'_n})) :: E, \theta\rangle & \Longrightarrow & \langle[\tau_1 =^? \tau'_1, \ldots, \tau_n =^? \tau'_n]@E, \theta\rangle &
\end{array}
$$

The only interesting case is the variable-term one. In contrast to unsorted unification it is not possible simply to map α_s to τ because τ might not be of sort s. However, there may exist a *weakening*, i.e. a sort-decreasing mapping from X to X, which forces the sort of τ down to s. $W(\tau, s)$ computes the most general weakening θ such that $\theta(\tau)$ is of sort s. If there is a weakening at all, coregularity implies the existence of a most general one.

The presentation of W is simplified by bringing it into tail recursive form. The call $\overline{W}(ps, \theta)$, where ps is a list of type-sort pairs and none of the variables in the domain of θ occur in ps, computes the most general instance θ' of θ such that $\theta'(\tau) : s$ for every pair (τ, s) in ps.

$$
\begin{array}{rcl}
W(\tau, s) & = & \overline{W}([(\tau, s)], \{\}) \\
\overline{W}([], \theta) & = & \theta \\
\overline{W}((\alpha_s, s') :: ps, \theta) & = & \left\{ \begin{array}{ll} \overline{W}(ps, \theta) & \text{if } s \leq s' \\ \overline{W}(\theta'(ps), \theta' \circ \theta) & \text{where } \theta' = \{\alpha \mapsto \beta_{glb(s,s')}\} \text{ and } \beta \text{ is new} \end{array} \right. \\
\overline{W}((t(\overline{\tau_n}), s) :: ps, \theta) & = & \overline{W}([(\tau_1, s_1), \ldots, (\tau_n, s_n)]@ps, \theta) \quad \text{where } (s_1, \ldots, s_n) = dom(t, s)
\end{array}
$$

$\overline{W}$ relies on the two partial functions *glb* and *dom* which are defined as follows: $glb(s_1, s_2) = s$ if $max(s_1 \sqcap s_2) = \{s\}$ and $dom(t, s) = w$ if $max(D(t, s)) = \{w\}$; otherwise they are undefined. Due to downward completeness and coregularity both functions can be stored in pre-computed tables. Table lookup at undefined positions results in failure of the unification algorithm.

3.2 Terms and Type Inference

Isabelle is based on the intuitionistic fragment of Church's Higher-Order Logic [1] and hence its terms are typed λ-terms. The grammar for "raw", i.e. as yet untyped, terms is shown

[2] Standard ML list notation is used.

Identifiers	x			Sorts	s		
Expressions	e	$=$	x	Types	τ	$=$	$\alpha_s \mid t(\tau_1, \ldots, \tau_n)$
		$\mid$	$(e_0\ e_1)$	Type schemes	σ	$=$	$\tau \mid \forall \alpha_s.\sigma$
		$\mid$	$\lambda x.e$				
		$\mid$	$\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1$				

Figure 1: Mini-ML expressions and types

$$\text{ASM} \qquad \frac{A(x) \succeq \tau}{A \vdash x : \tau}$$

$$\text{APP} \qquad \frac{A \vdash e_0 : \tau \to \tau' \qquad A \vdash e_1 : \tau}{A \vdash (e_0\ e_1) : \tau'}$$

$$\text{ABS} \qquad \frac{A + \{x \mapsto \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \to \tau'}$$

$$\text{LET} \qquad \frac{A \vdash e_0 : \tau \qquad \mathcal{TV}(\tau) - \mathcal{TV}(A) = \{\alpha_{s_1}, \ldots, \alpha_{s_k}\} \qquad A + \{x \mapsto \forall \overline{\alpha_{s_k}}.\tau\} \vdash e_1 : \tau'}{A \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 : \tau'}$$

Figure 2: Type inference for Mini-ML

in Figure 1. Let us call this language *Mini-ML* [2]. It goes beyond what is implemented in Isabelle by featuring a **let**-clause. Dealing with this richer language prepares the ground for the applications described in Section 5. To cope with **let**-expressions, the language of types introduced in the previous subsection has to be enriched with a universal quantifier, leading to the distinction between unquantified *types* and quantified *type schemes* due to Damas and Milner [4]. The rules of type inference are shown in Figure 2. The free variables in a type scheme are denoted by $\mathcal{TV}(\sigma)$. A is a finite mapping from identifiers to type schemes and $\mathcal{TV}(A)$ denotes the free type variables in the range of A. These rules are almost identical to the system DM' due to Clément *et al.* [2], except for the definition of generic instantiation. In contrast to unsorted polymorphism, where type schemes may be instantiated by arbitrary types, order-sorted polymorphism requires that the instantiation is sort-correct:

$$\forall \alpha_{s_1}, \ldots, \alpha_{s_n}.\tau \ \succeq \ \tau[\tau_1/\alpha_{s_1}, \ldots, \tau_n/\alpha_{s_n}]$$

if each τ_i is of sort s_i with respect to the type signature, which, as usual, remains implicit and fixed. Unsorted polymorphism can be recovered by having just a single sort, say $\top$, and exactly one declaration $t : (\top, \ldots, \top)\top$ for each type constructor.

The above inference rules can be turned into a type inference algorithm simply be interpreting them as Horn clauses and using Prolog technology. In particular this means that guessing the right instantiation in ASM is replaced by unification. More precisely, generic instantiation simply peels off all universally quantified type variables and replaces them by *logical* variables in the Prolog sense. The required instantiation is found by order-sorted unification of types during resolution. Having restricted ourselves to unitary type signatures, there is at most one most-general unifier and hence principal types exist. A functional implementation of type inference is obtained if Milner's algorithm $\mathcal{W}$ [17] is used with order-sorted unification.

In Isabelle, type inference takes place with respect to some *theory* containing sorts, types and constants. The **sorts** and **types** section determines the type signature and the **consts** declarations the initial environment A.

3.3 Higher-Order Unification

We now turn from the static analysis to Isabelle's equivalent of execution, higher-order resolution, which is the heart of Isabelle's inference engine. Its central component is an implementation of Huet's higher-order unification procedure [11]. In contrast to type inference, which extended very smoothly to order-sorted polymorphism, the same cannot be said for higher-order unification. The reason is that type inference in Isabelle-90 already dealt with unsorted polymorphism, whereas its unification procedure was a direct implementation of Huet's algorithm which is designed for simply typed terms. The difficulties encountered in extending it to cope with type variables are solely due to polymorphism in its most basic form, i.e. the presence of type variables; the order-sorted twist is again orthogonal and remains hidden in the type unification algorithm.

3.3.1 The Problem

The basic problem arises because terms may now contain type variables. The syntax of terms given in Figure 1 is an impoverished version of the actual Isabelle implementation where every variable and constant is tagged with its type, which may contain type variables. This type information is essential during unification when both term and type variables need to be instantiated. A simple example demonstrates the difficulty.

Let τ be some base type and α a type variable, let $c : \tau$ be a constant and $G : \alpha$ and $F : \alpha \to \tau$ be two term variables. The following matching problem is given:

$$F(G) \stackrel{?}{=} c \tag{2}$$

Let us first look at (some of) the infinitely many solutions to this problem. Every instantiation of α must be of the form $\tau_1 \to \cdots \to \tau_n \to \tau'$ for suitable types $\tau_1, \ldots, \tau_n$ and τ', where τ' is not a function type. Without any assumptions about α we get the solution $F = \lambda x.c$. In terms of Huet's algorithm this is the solution obtained after a single imitation step. In fact, for any instantiation of α where $\tau' \neq \tau$, this is the only solution.

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

$$F = \lambda x.x(H_1(x), \ldots, H_n(x)), \quad G = \lambda x_1, \ldots, x_n.c$$

The H_i are new free function variables.

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

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

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 should be dealt with. One impractical solution is to enumerate all possible type instantiations. Isabelle-91, just like λProlog [19], is more pragmatic: only the simplest type instantiation is tried. This approach is obviously incomplete. In the above example only the two solutions $\{F \mapsto \lambda x.c\}$ and $\{F \mapsto \lambda x.x, G \mapsto c, \alpha \mapsto \tau\}$ are found. Nevertheless it is sufficient in practice for reasons discussed at the end of the next section.

3.3.2 The Algorithm

This section presents the actual higher-order unification algorithm, proves its soundness and a limited form of completeness. The description is fairly terse and familiarity with Huet's original algorithm [11] is helpful. To facilitate the discussion we introduce a bit of notation.

Terms are slightly different from those in Figure 1: there is no **let**-construct (it can be replaced by abstraction plus application) and type inference has attached type information to the constants and variables. Terms are generated from a set of (term) variables V and a set of constants C by λ-abstraction and application. Free variables are denoted by F, G, and H, bound variables by x, y and z, constants by c, atoms ($C \cup V$) by a and b, and terms by s, t, and u. Variables and constants are tagged by superscripting: F^σ, c^τ etc. Note that proper type schemes cannot occur in a term and σ denotes a type in the sequel. Type decorations are omitted if they are unimportant. The free term variables in a term t are denoted by $\mathcal{FV}(t)$.

We use the following abbreviations: $\overline{\tau_n} \to \tau$ stands for $\tau_1 \to \cdots \to \tau_n \to \tau$ where τ is not a function type; $\lambda\overline{x_n}.s$ stands for $\lambda x_1.\ldots\lambda x_n.s$; $a(\overline{u_n})$ stands for $(\ldots(a\ u_1)\ldots)u_n$. A term $\lambda\overline{x_k}.a(\overline{s_n})$ is called *rigid* if $a \in C \cup \{\overline{x_k}\}$ and *flexible* otherwise.

Much of Huet's algorithm can be "lifted" to terms with type variables by replacing equality tests by unification of types. Before we present the actual algorithm we show where this näive lifting breaks down and how we deal with that situation. The problem occurs in the projection step of Huet's algorithm. Let the unification problem $\lambda\overline{x_k}.F^{\overline{\tau_n}\to\tau}(\overline{s_n}) =^? t$ be given where t is rigid and $\tau_i = \overline{\sigma_m} \to \sigma$. Then F can be a projection on its ith argument provided the result types of σ and τ coincide. If neither σ nor τ are type variables, their result types are σ and τ and need only be unified. If however, for example, σ is a type variable, it could be instantiated by any function type with the same result type as τ, leading to the the infinite branching shown in example (2) above. Isabelle-91 unifies σ and τ regardless of whether they are type variables or not and produces the single projection substitution $\{F \mapsto \lambda\overline{y_n}.y_i(\ldots)\}$. In example 2 this leads to the second solution $\alpha = \tau$, $F = \lambda x.x$ and $G = c$.

Let $\mathcal{U}(\tau, \tau')$ denote a complete set of unifiers of two types. In case the (implicit) order-sorted type signature is unitary, the result is a singleton set and can be computed by the algorithm in Section 3.1 above.

In addition to substitutions for type variables, denoted by θ, there are also substitutions for term variables, denoted by Θ. Applying a type substitution to a term means applying it to all type decorations in the term.

We have the following notation for both kinds of substitutions: $\mathcal{D}om(\theta)$ denotes the domain of θ; $\theta_2 \circ \theta_1$ is the composition $\lambda\tau.\theta_2(\theta_1(\tau))$; if W is a set of variables then $\theta_1 = \theta_2\ [W]$ means $\theta_1\alpha = \theta_2\alpha$ for all $\alpha \in W$ and $\theta_1 \leq \theta_2\ [W]$ means that there is a θ_3 such that $\theta_3 \circ \theta_1 = \theta_2\ [W]$.

A *unifier* of two terms s and t is a pair of substitutions $\langle\Theta, \theta\rangle$ on terms and types respectively such that $\Theta(\theta(s))$ and $\Theta(\theta(t))$ are equivalent modulo α, β and η-conversion. A unifier of a *system* S, i.e. a multiset of unordered pairs (unification problems) $s =^? t$, is defined analogously and is denoted by $\mathcal{U}(S)$. A system S is *presolved* if every element $(s =^? t) \in S$ is either

- a *flex-flex pair*, i.e. both s and t are flexible, or
- *solved*, i.e. $s = \lambda\overline{x_k}.F(\overline{x_k})$ and $F \notin \mathcal{FV}(t) \cup \mathcal{FV}(S - \{s =^? t\})$,

and both s and t have the same type. If S is presolved, define

$$\vec{S} = \{F \mapsto t \mid (\lambda\overline{x_k}.F(\overline{x_k}) \stackrel{?}{=} t) \in S\}.$$

A pair $\langle\Theta, \theta\rangle$ is a *pre-unifier* of a system S if $\mathcal{U}(\Theta(\theta(S))) \neq \{\}$. The terminology is consistent: if S is presolved, $\langle\vec{S}, id\rangle$ is a pre-unifier of S.

Huet's insight, which transformed higher-order unification from a mere curiosity into computational reality, was that flex-flex pairs need not be solved. This leads to a pre-unification algorithm where the result is a substitution (the solved pairs) together with a set of satisfiable constraints (the flex-flex pairs). Our formulation of the pre-unification algorithm is very close to that of Snyder and Gallier [25]. We use rewrite rules $S \stackrel{\theta}{\Longrightarrow} S'$ between systems. The set of solutions for a system S is given by those θ and presolved T such that $S \stackrel{\theta*}{\Longrightarrow} T$. The solution to the initial problem is the type substitution θ, the term substitution $\vec{T}$ and the remaining flex-flex pairs of T.

The actual pre-unification algorithm is given in Figure 3 as a collection of conditional rewrite rules. The substitution θ records the type instantiation computed by each transformation, whereas S represents both the current system and the term substitutions obtained so far. $S \Longrightarrow T$ abbreviates $S \stackrel{id}{\Longrightarrow} T$. If $S_{i-1} \stackrel{\theta_i}{\Longrightarrow} S_i$, $i = 1 \dots n$, we write $S_0 \stackrel{\theta*}{\Longrightarrow} S_n$ where $\theta = \theta_n \circ \cdots \circ \theta_1$.

The rules (D), (V), (S) ("Simplification"), (I) ("Imitation") and (P) ("Projection") are very similar to the ones given by Snyder and Gallier, except that terms need only be in β-normal form, not in η-expanded form. The rules (T), (T_S), and (T_P) unify types and are new. (T) unifies the types of two terms, thus ensuring in particular that s and t can be written as $\lambda\overline{x_k}.a(\overline{s_m})$ and $\lambda\overline{x_l}.b(\overline{u_n})$. ($\mathrm{T}_S$) unifies the types of two rigid heads. The combination of (T) and (T_S) prepares the ground for the application of (S). (T_P) prepares the flexible term in a flex-rigid pair for the application of (P) by adjusting types. (I) and (P) are immediately followed by (V), eliminating F. The free variables H in (I) and (P) are new. Note that terms are automatically put into head normal form after each transformation and that α-conversion remains implicit.

Soundness of the transformation rules relies on the following straightforward lemma:

Lemma 3.3 *If $S \stackrel{\theta}{\Longrightarrow} S'$ and $\langle\Theta', \theta'\rangle \in \mathcal{U}(S')$, then $\langle\Theta', \theta' \circ \theta\rangle \in \mathcal{U}(S)$.*

As a direct consequence we obtain the actual soundness theorem:

Theorem 3.4 *If $S \stackrel{\theta*}{\Longrightarrow} T$ and T is presolved, then $\langle\vec{T}, \theta\rangle$ is a pre-unifier of S.*

To prove completeness we follow Snyder and Gallier in defining a rewrite relation on triples $\langle\Theta, \theta, S\rangle$:

$$\langle\Theta, \theta, S\rangle \Longrightarrow \langle\Theta, \theta, T\rangle$$

if $S \Longrightarrow T$ via (D), (V) or (S).

$$\langle\Theta, \theta, S\rangle \Longrightarrow \langle\Theta \cup \Delta, \theta, T\rangle$$

$$\frac{s:\sigma \quad t:\tau \quad \sigma \neq \tau \quad \theta \in \mathcal{U}(\sigma,\tau)}{\{s \stackrel{?}{=} t\} \cup S \stackrel{\theta}{\Longrightarrow} \theta(\{s \stackrel{?}{=} t\} \cup S)} \tag{T}$$

$$\{s \stackrel{?}{=} s\} \cup S \Longrightarrow S \tag{D}$$

$$\frac{F \in \mathcal{FV}(S) - \mathcal{FV}(t)}{\{\lambda\overline{x_k}.F(\overline{x_k}) \stackrel{?}{=} t\} \cup S \Longrightarrow \{\lambda\overline{x_k}.F(\overline{x_k}) \stackrel{?}{=} t\} \cup \{F \mapsto t\}(S)} \tag{V}$$

$$\frac{\sigma \neq \tau \quad \theta \in \mathcal{U}(\sigma,\tau)}{\{\lambda\overline{x_k}.c^\sigma(\overline{s_m}) \stackrel{?}{=} \lambda\overline{x_l}.c^\tau(\overline{u_n})\} \cup S \stackrel{\theta}{\Longrightarrow} \theta(\{\lambda\overline{x_k}.c^\sigma(\overline{s_m}) \stackrel{?}{=} \lambda\overline{x_l}.c^\tau(\overline{u_n})\} \cup S)} \tag{T_S}$$

$$\frac{a \in C \cup \{\overline{x_k}\}}{\{\lambda\overline{x_k}.a^\tau(\overline{s_n}) \stackrel{?}{=} \lambda\overline{x_k}.a^\tau(\overline{u_n})\} \cup S \Longrightarrow \{\lambda\overline{x_k}.s_i \stackrel{?}{=} \lambda\overline{x_k}.u_i \mid i = 1 \dots n\} \cup S} \tag{S}$$

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

$$\frac{a \in C \cup \{\overline{x_l}\} \quad \sigma = \overline{\sigma_{m+i}} \to \sigma' \quad \sigma_j = \overline{\tau_p} \to \tau \quad \sigma' \neq \tau \quad \theta \in \mathcal{U}(\sigma',\tau)}{\{\lambda\overline{x_k}.F^\sigma(\overline{s_m}) \stackrel{?}{=} \lambda\overline{x_l}.a(\overline{u_n})\} \cup S \stackrel{\theta}{\Longrightarrow} \theta(\{\lambda\overline{x_k}.F^\sigma(\overline{s_m}) \stackrel{?}{=} \lambda\overline{x_l}.a(\overline{u_n})\} \cup S)} \tag{T_P}$$

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

Figure 3: Transformations for Pre-Unification

if either $S \Longrightarrow T$ via (I), $\Theta(F) = \lambda\overline{x_{m+i}}.c(\overline{t_{n+j}})$ and $\Delta = \{H_h \mapsto \lambda\overline{x_{m+i}}.t_h \mid h = 1 \dots n+j\}$, or $S \Longrightarrow T$ via (P), $\Theta(F) = \lambda\overline{x_{m+i}}.x_j(\overline{t_p})$ and $\Delta = \{H_h \mapsto \lambda\overline{x_{m+i}}.t_h \mid h = 1 \dots p\}$.

$$\langle\Theta, \theta_0, S\rangle \Longrightarrow \langle\Theta, \theta', T\rangle$$

if $S \stackrel{\theta}{\Longrightarrow} T$ via (T), (T_S) or (T_P) and $\theta_0 = \theta' \circ \theta\ [\mathcal{TV}(S)]$.

The idea is that if $\langle\Theta, \theta\rangle \in \mathcal{U}(S)$ then $\langle\Theta, \theta\rangle$ can be used to guide the search for a unifier of S. The following lemmas lead up to the final completeness theorem. Their proofs are very similar to those of the corresponding results in [25].

Lemma 3.5 *If $\langle\Theta, \theta_0\rangle \in \mathcal{U}(S)$ and $\langle\Theta, \theta_0, S\rangle \Longrightarrow \langle\Theta', \theta', S'\rangle$, then $\langle\Theta', \theta'\rangle \in \mathcal{U}(S')$, $\Theta = \Theta'\ [\mathcal{FV}(S)]$ and there is a θ such that $S \stackrel{\theta}{\Longrightarrow} S'$ and $\theta_0 = \theta' \circ \theta\ [\mathcal{TV}(S)]$.*

A substitution θ is called *basic* if there is no α such that $\theta\alpha$ is a function type.

Lemma 3.6 *Let the underlying type signature be regular. If θ_0 is basic, $\langle\Theta, \theta_0\rangle \in \mathcal{U}(S)$ and S is not presolved, then there are Θ' and θ' such that θ' is basic and $\langle\Theta, \theta_0, S\rangle \Longrightarrow \langle\Theta', \theta', S'\rangle$.*

Proof The proof proceeds by exhaustive case analysis pretty much as in [25]. The only unusual bit is to show that θ' is again basic. This is non-obvious if $\theta_0 = \theta' \circ \theta\ [\mathcal{TV}(S)]$ where $\theta \in \mathcal{U}(\sigma, \tau)$.

W.l.o.g. we may assume that the domain of θ' is minimal, in particular $\mathcal{D}om(\theta') \cap \mathcal{D}om(\theta) = \{\}$. Let $\beta \in \mathcal{D}om(\theta')$ and hence $\beta \notin \mathcal{D}om(\theta)$. We distinguish two cases. If $\beta \in \mathcal{TV}(S)$ then $\theta'\beta = \theta'\theta\beta = \theta_0\beta$ and hence $\theta'\beta$ is not a function type. If $\beta \notin \mathcal{TV}(S)$ then $\beta \in \mathcal{TV}(\theta\alpha)$ for some $\alpha \in \mathcal{D}om(\theta) \cap \mathcal{TV}(S)$ — otherwise the domain of θ' is not minimal. Examining the order-sorted unification algorithm for regular signatures given, for example, by Jouannaud and Kirchner [13] we see that $\mathcal{U}(\sigma, \tau)$ returns only substitutions θ with the property that all new variables in the range of θ are introduced by weakening. Formally: if $\beta \in \mathcal{TV}(\theta\alpha) - \mathcal{TV}(\sigma, \tau)$ then $\theta\alpha = \beta$. In particular this is obvious for the algorithm in Section 3.1. This means that $\theta'\beta = \theta'\theta\alpha = \theta_0\alpha$ and hence that $\theta'\beta$ is again not a function type. □

Lemma 3.7 *The relation $\Longrightarrow$ on triples is terminating.*

Proof The proof is by a decreasing complexity measure $\langle M, n, i_T + i_S + i_P \rangle$ where M and n are as in [25], i.e. M is the sum of the sizes of $\Theta(F)$ for all unsolved $F \in \mathcal{FV}(S)$, and n is the sum of the sizes of all terms in S. In addition we have i_T, i_S and i_P which are the number of unification problems where the rules (T), (T_S) and (T_P) are applicable. The application of each of these rules decreases the corresponding counter, cannot increase the other two counters, and leaves M and n unchanged. □

It is now routine to prove completeness from the above lemmas:

Theorem 3.8 *Let the underlying type signature be regular and let θ_0 be basic. If $\langle \Theta, \theta_0 \rangle \in \mathcal{U}(S)$ then there is a presolved T such that $S \stackrel{\theta *}{\Longrightarrow} T$, $\vec{T} \leq \Theta\ [\mathcal{FV}(S)]$, $\theta \leq \theta_0\ [\mathcal{TV}(S)]$, and $\langle \Theta, \theta_0 \rangle$ is a pre-unifier of the flex-flex pairs in T.*

Note that the final condition is necessary to ensure that the remaining flex-flex constraints are consistent with the initial solution $\langle \Theta, \theta_0 \rangle$.

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. But even in its limited form it has some interesting consequences. For example it implies that higher-order unification is complete for Isabelle's encoding of first-order logic ($\mathcal{FOL}$) because the sort system prevents type variables from ranging over function types. In higher-order logic ($\mathcal{HOL}$), however, there are examples of proofs that are not found automatically because of the above incompleteness. Such situations occur infrequently and can always be remedied by user-supplied explicit type instantiations.

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 Elliott [6] 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 (T_P) commits to a particular type instantiation out of an infinite set, thus reducing the solution space. Hence the application of (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 σ' or τ are type variables. This means that no solution is lost if the application of (T_P) to a particular unification problem can always be delayed until the types are sufficiently instantiated.

3.4 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 λ-terms.

α If De Bruijn notation [5] is used, α-conversion can be ignored completely.

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

η In contrast to unification for the simply typed λ-calculus, our transformation rules do not leave terms in η-expanded form because type-variables may become instantiated. Fortunately, only (S) requires the η-expansion of the head.

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. Rules (S), (I) and (P) should be tried in the order embodied in Huet's algorithm.

4 Single-Sorted Modules

This section discusses some more speculative application of order-sorted polymorphism as a module system. The aim is to support abstract reasoning of the following kind: every group of index 2 is abelian, the powerset algebra with symmetric difference forms a group of index 2, hence symmetric difference is commutative. This is a very heavy tool for deriving commutativity of symmetric difference but it illustrates the point nicely. In general this technique only pays off if the abstract theory provides enough interesting theorems.

The abstract theory of groups can be axiomatized as follows:

$$
\begin{array}{rclll}
\mathcal{G} & = & \mathcal{HOL} \; + & \textbf{sorts} \; grp < i & \\
 & & & \textbf{consts} \quad e \; : \; \alpha_{grp} & \\
 & & & \qquad\quad\; i \; : \; \alpha_{grp} \to \alpha_{grp} & \\
 & & & \qquad\quad m \; : \; \alpha_{grp} \to \alpha_{grp} \to \alpha_{grp} & \\
 & & & \textbf{rules} \qquad m(e,x) \; = \; x & \\
 & & & \qquad\quad m(i(x),x) \; = \; e & \\
 & & & \quad m(m(x,y),z) \; = \; m(x,m(y,z)) &
\end{array}
$$

Instead of a particular type grp, we have declared a sort grp which shall represent the collection of all groups. The operations have become polymorphic, because they must be defined on each individual group.

Within $\mathcal{G}$ one can easily derive consequences of group axioms such as the implication

$$\forall x.m(x,x) = e \implies \forall x,y.m(x,y) = m(y,x) \tag{3}$$

In general one is given an abstract theory A and a concrete theory T which satisfies all the axioms of A, and one would like a mechanism to transfer theorems proved generically in A to T. It is possible to treat this situation in the framework of order-sorted polymorphism provided that the theory abstraction talks only about a single type. Examples include classical algebra up to fields, but exclude vector spaces which involve both an abelian group and a field at the same time. A theory A with a single abstract type t translates into an Isabelle theory with a new sort t, no new type, and the polymorphic version of all operations, restricted to types of sort t. Hierarchies of theories are modelled via the subsort relation, e.g. $rng < grp$.

Coming back to the example, we would like to identify specific theories as groups and carry over consequences such as (3). A particular instance of a group is the type $form$ with exclusive-or as multiplication, identity as inverse, and falsity as unit. This can be expressed very simply as

$$\begin{array}{rcll} \mathcal{G}_{form} & = & \mathcal{G} + & \textbf{types}\ form : grp \\ & & & \textbf{rules} \quad \begin{array}{rcl} e & \equiv & False \\ i(x) & \equiv & x \\ m(x,y) & \equiv & \neg(x = y) \end{array} \end{array}$$

Note that since $form$ is already known, $form : grp$ is simply considered as an additional declaration for an old type.

Within $\mathcal{G}_{form}$ it is possible to specialize (3) by replacing e and m by their respective definitions. The premise $\forall x. \neg(x = x) = False$ is easily shown to be a theorem and we are left with the consequence that exclusive-or is commutative: $\forall x, y. \neg(x = y) = \neg(y = x)$.

The only problem with this approach is its unsoundness: the declaration $form : grp$ is not accompanied by any check that the given interpretation of the group operations actually induces a group structure on $form$, i.e. that $\neg(False = x) = x$, $\neg(x = x) = False$, and $\neg(\neg(x = y) = z) = \neg(x = \neg(y = z))$.

The soundness problem can be avoided by the introduction of two new constructs into Isabelle's theory building language: the declaration and instantiation of abstract theories. The concrete syntax might look somewhat like this:

$$\begin{array}{rcll} \mathcal{G} & = & \mathcal{HOL} + & \textbf{class}\ grp < i \\ & & & \textbf{consts}\ \ldots \\ & & & \textbf{rules}\ \ldots \\ G_{form} & = & \mathcal{G} + & \textbf{instance}\ form : grp \\ & & & \textbf{definitions}\ e \equiv \ldots \\ & & & \textbf{theorems}\ \neg(False = x) = x, \ldots \end{array}$$

It is no coincidence that this resembles type classes and their instantiation in the programming language Haskell [10]. In the conclusion to their paper [27], Wadler and Blott notice that type classes are very similar to theories of the OBJ-variety, except that, Haskell being a programming language, the properties are omitted. In the above extension to Isabelle they have returned via the **rules** and **theorems** section. Note that **theorems** are qualitatively different from the rest in that they are not text but actual Isabelle theorems. They need to be supplied as witnesses to the claim that under the given definitions $form$ is indeed a group.

The meaning of **class** and **instance** declarations can be explained by a transformation into core-Isabelle: **class** corresponds to **sorts**, **instance** to **types**, and **definitions** to **rules**. In addition, there are a number of well-formedness conditions that have to be met. The most important one is that the rules for sorts introduced via **class** are in some sense "closed": further extensions of a theory containing the declaration of a class c cannot add new rules constraining the constants introduced in c. We want to make sure that whenever we talk about groups, they have exactly those properties listed in the corresponding class declaration. The overall correctness criterion that needs to be satisfied is the following: any theorem provable for c should also be provable for any instance of c.

Classes are not completely closed as they can be extended hierarchically. Abelian groups can be introduced as a subclass of groups:

$$\begin{array}{rcl} \mathcal{AG} & = & \mathcal{G} + \textbf{class}\ agrp < grp \\ & & \qquad\ \textbf{rules}\ m(x,y) = m(y,x) \end{array}$$

At the moment the application of order-sorted polymorphism as a module system is still in its infancy: no serious attempt has been made to evaluate its usefulness and the **class** and **instance** constructs have not been implemented. Should these ideas prove successful, it may be desirable to move to classes with multiple parameters to cover abstract many-sorted theories like vector spaces. This goes beyond order-sorted polymorphism but could be supported by the notion of *qualified types* as developed by Mark Jones [12]. Although all these extensions are unlikely to provide a general solution to the problem of abstract theories, they should be very convenient for those applications that are within their range. This is analogues to the function of classes in Haskell: they cannot completely replace a module system like that of Standard ML [18] but for many simple problems they are easier to use than modules.

5 Applications to Standard ML

Although order-sorted polymorphism was initially designed to express restricted polymorphism in Isabelle, it subsequently turned out to be equally suitable as a type system for the functional programming languages Standard ML [18] and Haskell [10]. In this section we explore the application to the notion of equality types in Standard ML. The treatment of Haskell's *type classes* via order-sorted polymorphism is more involved and can be found elsewhere [20].

Standard ML is a complex language. We analyze only a fragment of its core by reducing it to Mini-ML as introduced in Section 2. This reduction is obviously feasible for the term language, but it remains to be seen how the type system can be reduced to order-sorted polymorphism. The benefits of this reduction are a uniform type inference algorithm and sound criteria for the existence of principal types.

Order-sorted polymorphism is ideally suited to model Standard ML's [18] *equality types*. They partition the set of all types into those that admit equality and those that do not. The following type signature yields a precise definition of this distinction:

$$\begin{array}{rcl} \mathcal{ML}_e & = & \textbf{sorts}\ \top \\ & & \qquad\quad e < \top \\ & & \textbf{types}\ \begin{array}{rcl} \texttt{bool} & : & e \\ \texttt{int} & : & e \\ \_*\_ & : & (\top,\top)\top \\ \_*\_ & : & (e,e)e \\ \_\texttt{->}\_ & : & (\top,\top)\top \\ \_\ \texttt{ref} & : & (\top)e \end{array} \\ & & \textbf{consts}\ \_\texttt{=}\_ : \alpha_e\ \texttt{->}\ \alpha_e\ \texttt{->}\ \texttt{bool} \end{array}$$

This expresses quite succinctly that booleans, integers, and all reference types admit equality, products admit equality iff both components do, and no function type admits equality. Hence

e is exactly the set of equality types according to Standard ML. Similarly, the notion of generalization $\succeq$ is precisely the one in Standard ML. Therefore type inference with equality types is just a special case of type inference in Mini-ML with respect to $\mathcal{ML}_e$. It is a straightforward exercise to check that $\mathcal{ML}_e$ (and the extensions below) satisfy the requirements of Lemma 3.2. Hence every typable term has a principal type.

In terms of Standard ML, $\mathcal{ML}_e$ forms part of the "initial static basis" of the language. This static basis changes with the introduction of new user-defined types. The sorts and their ordering $\leq$ remain fixed as in **sorts** above. The set of declarations is initially the one shown in **types** above but is enriched with every new data type declaration `datatype` $(\overline{\alpha_n})t$ `=` ... which automatically adds the declaration $t : (\top, \ldots, \top)\top$. In case t *admits equality* in the sense of Standard ML [18], the declaration $t : (e, \ldots, e)e$ is also added. Gunter *et al.* [8] give a detailed analysis of the semantics of data types and equality types. They show that if t admits equality, there is a least subset of the arguments of t that need to be of sort e for the result to be of sort e.

Standard ML's other extension to ordinary polymorphism, *imperative types*, cannot be explained completely with order-sorted polymorphism alone. However, it is not difficult to recast Tofte's treatment of imperative types [26] in an order-sorted framework. For example the sort-structure becomes the following 4-element lattice

$$\begin{array}{ccc} & \top & \\ / & & \backslash \\ e & & i \\ \backslash & & / \\ & ei & \end{array}$$

where i is the sort of *imperative* types and ei the intersection of e and i. The rule that a type is imperative iff all its type variables are imperative can also be formulated as an order-sorted type signature. However, the actual type checking discipline for imperative types [18, 26] does not quite fit the standard Damas-Milner scheme: when type checking **let** $x = e_1$ **in** e_2, where the dynamic execution of e_1 might allocate a new reference, only *non-imperative* type variables in the type of e_1 can be generalized.

6 Conclusion

We have looked at a flexible and convenient type system for the specification of object-logics with simple type systems. It offers an extension of ML-style polymorphism and automatic type inference. Where applicable, it makes life considerably easier. The concept of order-sorted polymorphism has applications beyond Isabelle involving type systems for programming languages and computer algebra. The former is sketched in Section 5, the latter can be found in an article by Comon *et al.* [3] where order-sorted polymorphism (although not under this name) is further enriched with subtypes. The resulting system is very expressive but type inference becomes undecidable.

Although a detailed comparison between Isabelle's order-sorted polymorphism and the concept of type classes in Haskell [20] is beyond the scope of this paper, we should point out the main difference between the two systems. Haskell's analogue of sorts are not classes but finite sets of classes (representing their intersection). Hence the sort structure is always a lattice where the infimum of two sorts is their union. This makes perfect sense in the Haskell context but may be too liberal for other applications where the intersection of some sorts might be empty.

Nevertheless it is tempting to follow the Haskell approach because it simplifies the declaration of the sort structure. For example the 4-element lattice at the end of Section 5 can be generated automatically from the atoms e and i as $\{\}$, $\{e\}$, $\{i\}$ and $\{e, i\}$. If the sort structure is small and almost linear, not much is gained by going to the powerset. Further experience with Isabelle's sort mechanism may, however, convince us to change it in the direction of Haskell.

Acknowledgements

This work would have been impossible without Larry Paulson who invented Isabelle, provided constant feed-back, and made valuable comments on draft versions of the paper. Mads Tofte helped me to understand imperative types in Standard ML.

References

[1] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[2] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[3] H. Comon, D. Lugiez, and P. Schnoebelen. A rewrite-based type discipline for a subset of computer algebra. *J. Symbolic Computation*, 11:349–368, 1991.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. Principles of Programming Languages*, pages 207–212, 1982.

[5] 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*, 34:381–392, 1972.

[6] C. Elliott. Higher-order unification with dependent function types. In N. Dershowitz, editor, *Proc. 3rd Int. Conf. Rewriting Techniques and Applications*, pages 121–136. LNCS 355, 1989.

[7] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. 12th ACM Symp. Principles of Programming Languages*, pages 52–66, 1985.

[8] C. A. Gunter, E. L. Gunter, and D. B. MacQueen. An abstract interpretation for ML equality kinds. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 112–130. LNCS 526, 1991.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. 2nd IEEE Symp. Logic in Computer Science*, pages 194–204, 1987.

[10] P. Hudak and P. Wadler. Report on the programming language Haskell. Version 1.0, April 1990.

[11] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[12] M. P. Jones. Type inference for qualified types. Technical Report PRG-TR-10-91, Oxford University Computing Laboratory, 1991.

[13] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 257–321. MIT Press, 1991.

[14] S. Kaes. Parametric overloading in polymorphic programming languages. In *Proc. 2nd European Symposium on Programming*, pages 131–144. LNCS 300, 1988.

[15] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[16] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, 8:383–413, 1989.

[17] R. Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.

[18] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[19] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, Philadelphia, 1987.

[20] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, pages 1–14. LNCS 523, 1991.

[21] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[22] L. C. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge, Computer Laboratory, 1990.

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

[24] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted equational computation. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2*, pages 297–367. Academic Press, 1989.

[25] W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symbolic Computation*, 8:101–140, 1989.

[26] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[27] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 60–76, 1989.

[28] U. Waldmann. Unification in order-sorted signatures. Technical Report 298, Fachbereich Informatik, Universität Dortmund, 1989.