Simplifying and Improving Qualified Types

Mark P. Jones
Department of Computer Science, University of Nottingham
University Park, Nottingham NG7 2RD.
mpj@cs.nott.ac.uk

Abstract

Qualified types provide a general framework for constrained type systems, with applications including type class overloading, subtyping and record calculi. This paper presents an extended version of the type inference algorithm used in previous work, that can take account of the satisfiability of constraints to obtain more accurate principal types. The new algorithm is obtained by adding two new rules, one for *simplification* and one for *improvement* of constraint sets. In particular, it permits a better treatment of the previously troublesome multiple parameter extensions of Haskell type classes, generalizing the system of *parametric type classes* proposed by Chen, Hudak and Odersky.

1 Introduction

Qualified types provide a general framework for constrained type systems; typical applications include type class overloading, subtyping and record calculi. In previous work, we have shown how the standard treatment of ML style polymorphism can be adapted to support qualified types. In particular, any well-typed program has a *principal type* that can be calculated by an extended version of Milner's type inference algorithm. This is useful both for describing the set of types that can be assigned to a term and for detecting possible semantic ambiguities.

Unfortunately, while technically correct, the principal types produced by our algorithm do not take account of the satisfiability of constraints. As a result, they are sometimes more complicated and less accurate than we might hope. Even if the additional complexity were not an issue, this can sometimes cause perfectly reasonable programs to be rejected when the principal type suggests, wrongly, that the term does not have a well-defined semantics. In other cases, the inferred types are too liberal, including unsatisfiable constraints and delaying the detection of type errors.

This paper shows how these problems can be avoided by using a notion of *principal satisfiable types* and extending the type inference algorithm with two new rules, one dealing with *simplification*, the other with *improvement*. Our approach is flexible enough to allow variations between different applications of qualified types, offering better principal types without compromising the decidability of type inference. In addition to other applications, the new algorithm permits a better treatment for the previously troublesome multiple parameter extensions of Haskell type classes. In particular, we show how improvement can support a form of *parametric type classes* [1].

We start, in Section 2, with an overview of the system of qualified types used in earlier work, including examples of predicate systems, a description of the type system and an outline of the type inference algorithm. Section 3 describes the use of simplification, allowing the constraint set included in the type of an expression to be replaced by an equivalent, but simpler, set of constraints. In Section 4, we introduce the concept of improvement which is the major contribution of this paper, using information about satisfiability of constraints to refine the inferred types. This requires a modification to our treatment of type inference, shifting attention to satisfiable typings and a satisfiability ordering between type schemes in Section 5. The proofs of soundness and completeness properties for a type inference algorithm that uses simplification and improvement are discussed in Section 6; detailed proofs of new results are provided elsewhere [15]. We conclude with a discussion on the use of our new framework in Section 7.

2 A brief overview of qualified types

To describe the contributions of this paper we need to begin with a brief description of the framework of *qualified types* on which it builds. We make no attempt to repeat the full details of earlier presentations [11, 10].

2.1 Predicates

The key idea motivating the use of qualified types is the ability to include *predicates* in the type of a term, capturing restrictions on the ways that it can be used. The properties of predicates themselves are described using an entailment relation, denoted by the symbol $\Vdash$. If P and Q are finite sets of predicates, then the assertion that $P \Vdash Q$ means that the predicates in Q hold, whenever the predicates in P are satisfied. The only assumptions that we make about the predicate entailment relation are that it is transitive, closed under substitutions, and such that $P \Vdash Q$ whenever Q is a subset of P.

Simple examples of single predicates that are useful in practical applications are illustrated in Figure 1. In some cases, we have taken the liberty of using a slightly different syntax

Predicate	Interpretation
$t \in Eq$	Values of type t can be tested for equality using the == operator. This usually includes all types, except those with functional components [21, 7].
$t \in Num$	t is a numeric type, for example, the type of integers, or floating point numbers, and elements of type t can be manipulated using standard arithmetic operators, for example, + for addition and $*$ for multiplication [21, 7].
$t \in Collect(s)$	Values of type t can be used to represent collections of values of type s [1].
$t\ Dual\ s$	Values of types t and s represent the elements of dual lattices [9].
$s \subseteq t$	s is a subtype of t; in practice, this usually means that values of type s can be treated as values of type t by applying a suitable coercion [16, 17, 4, 3, 19].
r has $l{:}t$	r is a record type containing an field labelled l of type t [6].
r lacks l	r is a record type, not including a field labelled l [6].
$r_1 \# r_2$	The record types r_1 and r_2 do not have any fields in common [5].

Figure 1: Examples of individual predicates and their informal interpretation

from earlier presentations, in the hope that this will make the interpretation of some predicates a little more obvious.

2.2 OML—Core-ML with overloading

Working towards an extension of core-ML that supports qualified types, we adopt a structured language of types specified by the grammar:

$$\begin{array}{lcll} \tau & ::= & t & \textit{type variables} \\ & | & \tau \to \tau & \textit{function types} \\ & | & \ldots & \textit{other constructed types} \\ \rho & ::= & P \Rightarrow \tau & \textit{qualified types} \\ \sigma & ::= & \forall T.\rho & \textit{type schemes} \end{array}$$

Here, t ranges over a given set of type variables and P and T range over finite sets of predicates and finite sets of type variables respectively. The set of type variables appearing (free) in an expression X is denoted $TV(X)$ and is defined in the obvious way.

For programs, we use the term language of core-ML:

$$\begin{array}{lcll} E & ::= & x & \textit{variable} \\ & | & EF & \textit{application} \\ & | & \lambda x.E & \textit{abstraction} \\ & | & \mathbf{let}\ x = E\ \mathbf{in}\ F & \textit{local definition} \end{array}$$

For the purposes of this work, we are only interested in terms that can be assigned a type using the rules in Figure 2. These rules use judgements of the form $P \,|\, A \vdash E : \sigma$ where P is a set of predicates and A is a type assignment, i.e. a mapping from term variables to types. Much of the notation used here is standard, as indeed are most of the rules. For example, the notation A_x refers to the type assignment obtained from A by deleting the type assigned to x, if any. Only $(\Rightarrow I)$ and $(\Rightarrow E)$, moving global constraints in to, or out of, the type of an object, and the $(\forall I)$ rule for polymorphic generalization, actually involve the predicate set P.

We refer, collectively, to the type, term and typing rules given above as OML, a mnemonic for 'Overloaded ML'.

2.3 Type inference for OML

An important property of OML is the existence of an algorithm for calculating *principal typings* for a given term. More precisely, there is an effective algorithm, taking a term E and a type assignment A as its input, for calculating the most general type that can be assigned to E, given the assumptions in A.

To describe what it means for one type to be more general than another, we define an ordering between *constrained type schemes*, i.e. pairs of the form $(P \,|\, \sigma)$ where σ is a type scheme and P is a set of predicates. We start by defining the set of *generic instances* of a constrained type scheme:

$$\begin{array}{c} [\![P' \,|\, \forall \alpha_i.P \Rightarrow \tau]\!] \\ = \\ \{\, Q \Rightarrow [\nu_i/\alpha_i]\tau \;|\; \nu_i \in \mathit{Type},\; Q \Vdash P', [\nu_i/\alpha_i]P \,\}. \end{array}$$

Using this definition, the required ordering between constrained type schemes is specified by law:

$$(P \,|\, \sigma) \leq (P' \,|\, \sigma') \;\Leftrightarrow\; [\![P \,|\, \sigma]\!] \subseteq [\![P \,|\, \sigma']\!].$$

In other words, $\sigma \leq \sigma'$ if and only if every generic instance of σ is a generic instance of σ'. It follows immediately from the form of the definition that the ordering is reflexive and transitive. Furthermore, it is reasonably easy to show that the ordering is preserved by substitution, i.e. that $S(P \,|\, \sigma) \leq S(P' \,|\, \sigma')$, whenever $(P \,|\, \sigma) \leq (P' \,|\, \sigma')$. This is important because it indicates that the ordering on type schemes is compatible with our notion of polymorphism, allowing free type variables to be freely instantiated with arbitrary types.

The type inference algorithm itself is described by the rules in Figure 3. This presentation, as in previous descriptions of qualified types, follows Rémy [18], using judgements of the form $Q \mid TA \vdash^{\mathrm{W}} E : \nu$ where A and E are the type assignment and expression provided as inputs to the algorithm, and Q, T and ν are a predicate set, substitution and type, respectively, produced as its results. The notation $Gen(A, \rho)$ used in the rule $(let)^{\mathrm{W}}$ indicates the *generalization* of ρ with respect to A, defined as $\forall a_i.\rho$ where $\{a_i\}$ is the set of type variables $TV(\rho) \setminus TV(A)$. As demonstrated in previous work, the results of the algorithm can be used to construct a principal type scheme, η, such that:

$$P \,|\, A \vdash E : \sigma \Longleftrightarrow (P \,|\, \sigma) \leq \eta.$$

Assuming, as is often the case for top-level definitions, that A does not include any free type variables, then the principal type is just: $\eta = Gen(A, Q \Rightarrow \nu) = (\forall a_i.Q \Rightarrow \nu)$, where $\{a_i\}$ is the set of type variables appearing free in $(Q \Rightarrow \nu)$.

Note that it is also possible for the type inference algorithm to fail, either because E contains a free variable that is not bound in A, or because the calculation of a most general

$$
(var)\quad \frac{(x\!:\!\sigma) \in A}{P\,|\,A \vdash x : \sigma}
\qquad
(\Rightarrow E)\quad \frac{P\,|\,A \vdash E : Q \Rightarrow \rho \quad P \Vdash Q}{P\,|\,A \vdash E : \rho}
$$

$$
(\to E)\quad \frac{P\,|\,A \vdash E : \tau' \to \tau \quad P\,|\,A \vdash F : \tau'}{P\,|\,A \vdash EF : \tau}
\qquad
(\Rightarrow I)\quad \frac{P, Q\,|\,A \vdash E : \rho}{P\,|\,A \vdash E : Q \Rightarrow \rho}
$$

$$
(\to I)\quad \frac{P\,|\,A_x, x\!:\!\tau' \vdash E : \tau}{P\,|\,A \vdash \lambda x.E : \tau' \to \tau}
\qquad
(\forall E)\quad \frac{P\,|\,A \vdash E : \forall\alpha.\sigma}{P\,|\,A \vdash E : [\tau/\alpha]\sigma}
$$

$$
(let)\quad \frac{P\,|\,A \vdash E : \sigma \quad Q\,|\,A_x, x\!:\!\sigma \vdash F : \tau}{P, Q\,|\,A \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}
\qquad
(\forall I)\quad \frac{P\,|\,A \vdash E : \sigma \quad \alpha \notin TV(A) \cup TV(P)}{P\,|\,A \vdash E : \forall\alpha.\sigma}
$$

Figure 2: Typing rules for OML.

unifier, described by the notation $\tau \overset{U}{\sim} \tau'$, fails as a result of a mismatch between the expected and actual type of a function argument. In this case, the completeness property of the type inference algorithm guarantees that there are, in fact, no derivable typings of the form $P\,|\,A \vdash E : \sigma$.

3 Simplification

In this section, we will show how inferred types can be simplified by replacing one predicate set with another, equivalent set of constraints. This is not a new idea; similar techniques are already used in other theoretical work, and in the implementations of systems like Haskell [7] and Gofer [14]. One of the advantages of the framework used in this paper is that it allows us to view simplification independently of other aspects of the type system, revealing opportunities for specific design decisions that are hidden in other presentations of constrained type inference.

For convenience, we write $P \Leftrightarrow Q$ to indicate the equivalence of predicate sets P and Q, i.e. that $P \Vdash Q$ and $Q \Vdash P$. It is a straightforward exercise to show that, if $P \Vdash Q$, then:

$$[\![\forall t_i.P \Rightarrow \tau]\!] \subseteq [\![\forall t_i.Q \Rightarrow \tau]\!],$$

and hence that, if $P \Leftrightarrow Q$, then $(\forall t_i.P \Rightarrow \tau)$ and $(\forall t_i.Q \Rightarrow \tau)$ are equivalent with respect to the $\leq$ ordering on type schemes. Applications of simplification in Haskell include:

- Constant predicates. The *Int* type is an instance of the *Eq* class and hence $\{\, Int \in Eq \,\}$ is equivalent to the empty predicate set, $\{\}$.
- Superclass hierarchies. The *Eq* class is defined as a superclass of *Ord*; static checks are used to ensure that $Ord \subseteq Eq$ and hence a predicate set of the form $\{\, \tau \in Eq,\ \tau \in Ord \,\}$ can be simplified to just $\{\, \tau \in Ord \,\}$.
- Context reduction. The definition of equality on lists relies on the existence of a definition of equality on the individual elements of the list. This corresponds to an equivalence: $\{\, t \in Eq \,\} \Leftrightarrow \{\, [t] \in Eq \,\}$. Rules like this can be used to ensure that all constraints in inferred types are of the form $t \in C$ (t is a type variable), as required by Haskell.

Simplification is also useful in other applications of qualified types.

Simplification may be used at any stage during the typing process. We can extend the algorithm in Figure 3 to allow this by adding the rule (*Simp*):

$$\frac{Q\,|\,TA \overset{\mathrm{W}}{\vdash} E : \nu \quad P \Leftrightarrow Q}{P\,|\,TA \overset{\mathrm{W}}{\vdash} E : \nu}$$

This gives a non-deterministic type inference algorithm: it is possible to obtain distinct principal types that are not equal up to renaming of bound variables. Fortunately, since these types are equivalent under the ordering $\leq$ introduced in Section 2.3, the addition of (*Simp*) still yields a sound algorithm that calculates principal type schemes for OML programs. By adopting a non-deterministic algorithm, we have the flexibility to allow designers of applications of qualified types to refine the algorithm, choosing to use simplification only under certain circumstances or at specific points during type checking.

We should also comment that, although we have specified what it means for two predicate sets to be equivalent, we will not attempt to formalize what it means to say that one is simpler than another. There are some obvious measures of complexity that could be used, for example, the number of predicates or the size of the type expressions involved. However, we believe that these issues are best dealt with in the design of specific applications. More directly, while we use a general and symmetric notion of simplification that allows any equivalent predicate set Q to be used in place of P, we would expect that, in a real implementation, Q will actually be chosen as a simplified version of P in some appropriate manner.

4 Improvement

A second method for inferring more accurate principal types, and the most important contribution of this paper, is based on the concept of *improvement*. Although some special cases of this idea have been used in other systems, we are not aware of any previous work that has either identified the notion of improvement as an independent concept, or developed these ideas in the general framework described below.

4.1 Improving records

The central idea is to use information about the satisfiability of predicate sets to simplify inferred types. As a first example, consider a language with a system of records, using a function:

$$(\_l) \quad :: \quad \forall r.\forall t.(r\ \mathsf{has}\ l\!:\!t) \Rightarrow r \to t$$

$$(var)^{\mathrm{W}} \quad \frac{(x : \forall \alpha_i . P \Rightarrow \tau) \in A \quad \beta_i \text{ new}}{[\beta_i/\alpha_i]P \,|\, A \vdash^{\mathrm{W}} x : [\beta_i/\alpha_i]\tau}$$

$$(\rightarrow E)^{\mathrm{W}} \quad \frac{P \,|\, TA \vdash^{\mathrm{W}} E : \tau \quad Q \,|\, T'TA \vdash^{\mathrm{W}} F : \tau' \quad T'\tau \overset{U}{\sim} \tau' \rightarrow \alpha \quad \alpha \text{ new}}{U(T'P, Q) \,|\, UT'TA \vdash^{\mathrm{W}} EF : U\alpha}$$

$$(\rightarrow I)^{\mathrm{W}} \quad \frac{P \,|\, T(A_x, x : \alpha) \vdash^{\mathrm{W}} E : \tau \quad \alpha \text{ new}}{P \,|\, TA \vdash^{\mathrm{W}} \lambda x.E : T\alpha \rightarrow \tau}$$

$$(let)^{\mathrm{W}} \quad \frac{P \,|\, TA \vdash^{\mathrm{W}} E : \tau \quad P' \,|\, T'(TA_x, x : \sigma) \vdash^{\mathrm{W}} F : \tau' \quad \sigma = Gen(TA, P \Rightarrow \tau)}{P' \,|\, T'TA \vdash^{\mathrm{W}} (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau'}$$

Figure 3: Type inference algorithm W.

to describe the selection of a field l of type t from a record of type r. Following conventional notation, we treat the expression $e.l$ as a sugared version of $(\_.l)\ e$. Now consider the function $f = \lambda r.(r.l,\ r.l)$ whose principal type, according to the algorithm in Section 2.3, is:

$$\forall r.\forall a.\forall b.(r \textsf{ has } l : a,\ r \textsf{ has } l : b) \Rightarrow r \rightarrow (a, b).$$

However, for any particular record type r, the types assigned to the variables a and b must be identical since they both correspond to the same field in r. It would therefore seem quite reasonable to treat f as having a *principal satisfiable type scheme*:

$$\forall r.\forall a.(r \textsf{ has } l : a) \Rightarrow r \rightarrow (a, a).$$

To capture the essence of this example in a more general setting, we introduce the following notation for describing the *satisfiable instances* of a given predicate set P with respect to a predicate set P_0:

$$\lfloor P \rfloor_{P_0} \quad = \quad \{\, SP \mid S \in Subst,\ P_0 \Vdash SP \,\}.$$

The predicate set P_0 used here is arbitrary, although we will often use $P_0 = \emptyset$ and we will always assume that $TV(P_0) = \emptyset$. In practice, the choice of P_0 plays a relatively small part in the following and we will often omit the subscript, writing just $\lfloor P \rfloor$ to avoid unnecessary distraction.

It is easy to show that $\lfloor SP \rfloor \subseteq \lfloor P \rfloor$, for any substitution S, and any predicate set P. The reverse inclusion, $\lfloor P \rfloor \subseteq \lfloor SP \rfloor$, does not always hold, but is more interesting because it tells us that we can apply the substitution S to P without changing its satisfiable instances. In particular, taking S as the substitution $[a/b]$, the argument about the predicates for record types given above is captured by the equality:

$$\lfloor r \textsf{ has } l : a \rfloor = \lfloor r \textsf{ has } l : a,\ r \textsf{ has } l : b \rfloor.$$

In this case, we will say that the substitution $[a/b]$ improves the predicate set $\{r \textsf{ has } l : a,\ r \textsf{ has } l : b\}$. More generally, we write S *improves* P if $\lfloor P \rfloor = \lfloor SP \rfloor$ and the only variables involved in S that do not appear in P are 'new' variables, similar to those introduced by the type inference algorithm in Figure 3.

To make use of improvement during type inference, we extend the type inference algorithm with the rule (*Imp*):

$$\frac{Q \,|\, TA \vdash^{\mathrm{W}} E : \nu \quad T' \textit{ improves } Q}{T'Q \,|\, T'TA \vdash^{\mathrm{W}} E : T'\nu}$$

To get the most benefit from this, we would obviously prefer to use substitutions T' that give, in some sense, the best possible improvement. However, while this is *desirable*, we do not make it a *requirement* of the work described here. This is important because the definition of general predicate systems does not guarantee the existence of 'optimal' improvements, or of computable algorithms for calculating them. Instead, we provide a general framework that allows us to compromise between decidability and improvement. In the simplest case, we can use the identity substitution id as an improving substitution, which satisfies id *improves* P for all predicate sets P. Of course, this just gives the same results as the previous version of the type inference algorithm, without improvement[1].

In practice, the typing algorithm for a language based on the ideas presented here might be parameterized by the choice of an *improving* function, $impr$, such that $(impr\ P)$ *improves* P for any predicate set P. The argument above shows that there is always at least one possible choice for an improving function, namely $impr(P) = id$. We will see that it is also possible to arrange for an improving function to fail, thereby causing the type inference algorithm to fail, if it is applied to an unsatisfiable predicate set, i.e. a predicate set P such that $\lfloor P \rfloor_{P_0} = \emptyset$. This can be used to implement a type checker that produces only satisfiable typing judgements, and fails if and only if there are no satisfiable typings. Of course, this behaviour would not be appropriate for a system in which tests of the form $\lfloor P \rfloor_{P_0} = \emptyset$ are not decidable. Once again, our framework allows the language designer to control these aspects of the type inference algorithm by choosing a suitable improving function.

4.2 Improving subtyping

To see why it may be necessary to introduce new variables in an improving substitution, consider a system of subtyping using predicates of the form $\tau \subseteq \tau'$ to indicate that τ is a subtype of τ' and with an entailment relation that is fully determined by the following rules:

$$\frac{}{P \Vdash \tau \subseteq \tau} \qquad \frac{P \Vdash \tau \subseteq \nu \quad P \Vdash \nu \subseteq \mu}{P \Vdash \tau \subseteq \mu}$$

$$\frac{P \supseteq Q}{P \Vdash Q} \qquad \frac{P \Vdash \tau' \subseteq \tau \quad P \Vdash \nu \subseteq \nu'}{P \Vdash (\tau \rightarrow \nu) \subseteq (\tau' \rightarrow \nu')}$$

[1] Pun intended!

Using an implicit coercion, the function $g = \lambda f.\lambda x.1 + f\ x$ has principal type:

$$\forall a.\forall b.(a \subseteq (b \to Int)) \Rightarrow a \to b \to Int.$$

Taking $P_0 = \{Int \subseteq Float\}$, i.e. assuming that the only primitive coercion is from the type *Int* of integers to the type *Float* of floating point numbers, we can use an improving substitution $[(c \to d)/a]$ since:

$$\lfloor a \subseteq (b \to Int) \rfloor = \lfloor (c \to d) \subseteq (b \to Int) \rfloor .$$

In fact, we can obtain a further improvement by noticing that this requires $d \subseteq Int$, which is only possible if $d = Int$. Hence the 'improved' type for g becomes:

$$\forall b.\forall c.((c \to Int) \subseteq (b \to Int)) \Rightarrow (c \to Int) \to b \to Int.$$

Now, as is often the case, improvement exposes new opportunities for simplification, and we can further refine the type of g to:

$$\forall b.\forall c.(b \subseteq c) \Rightarrow (c \to Int) \to b \to Int.$$

Note that improvement can be considered as a generalization of the use of Mitchell's MATCH algorithm [16, 17].

4.3 Improving type classes

In this section, we will show how improvement can be used to support the use of type classes, concentrating in particular on the proposals by Chen, Hudak and Odersky [1] for parametric type classes.

Several researchers, including this author, have experimented with systems of multiple parameter type classes. Thinking of standard type classes as sets of types, the simplest interpretation of a multiple parameter class is as a set of tuples of types, corresponding to a relation on types. For example, a two parameter class, *Dual*, can be used to describe duality between lattices [9]. Unfortunately, practical experience with multiple parameter type classes in Gofer [8] suggests that the standard mechanisms for defining classes and instances in Haskell are often too weak to define useful relations between types[2].

To illustrate the kind of problems that can occur, suppose that we use predicates of the form $c \in Collect(a)$ to indicate that values of type c can be used to represent collections of values of type a. A simple class for operations on collections can be defined as follows:

$$\begin{array}{l} \textbf{class}\ c \in Collect(a)\ \textbf{where} \\ \quad \begin{array}{lll} empty & :: & c \\ insert & :: & a \to c \to c \\ member & :: & a \to c \to Bool \end{array} \end{array}$$

The *empty* value represents an empty collection, while the *insert* and *member* functions might be used to add an element, or to test whether a particular element is included in a collection.

There are a number of ways to implement collections. One of the simplest ways is to use a list, assuming that the values it holds can be tested for equality so that we can implement the membership test:

$$\textbf{instance}\ (a \in Eq) \Rightarrow [a] \in Collect(a)\ \textbf{where} \quad \cdots$$

[2] Interestingly enough, these problems do not seem to occur for many useful examples involving multiple parameter constructor classes [13].

A more efficient implementation might be obtained using binary search trees if we assume that there is an ordering on the elements held in a collection, captured by the type class *Ord* in the following definition:

$$\begin{array}{lcl} \textbf{data}\ BST\ a & = & Empty \\ & | & Fork\ a\ (BST\ a)\ (BST\ a) \end{array}$$

$$\textbf{instance}\ (a \in Ord) \Rightarrow (BST\ a \in Collect(a))\ \textbf{where} \quad \cdots$$

On the surface, these definitions seem quite reasonable, but we soon run into difficulty if we try to use them. One of the first problems is that the type of the *empty* value is $\forall a.\forall c.(c \in Collect(a)) \Rightarrow c$, which is *ambiguous* in the sense that a type variable a appears on the left of the $\Rightarrow$ symbol, but is not mentioned on the right. As a result, there is no general way to determine the intended value of type a from the context in which *empty* is used. In general, it is not possible to use any term with an ambiguous principal type if we hope to provide a well-defined semantics for the language [10, 12].

Another problem is that the types assigned to *member* and *insert* are more general than we might expect. For example, it is possible to define collections that contain different types of elements and to define functions like:

$$\begin{array}{lcl} intOrBool & :: & \forall c.(c \in Collect(Int),\ c \in Collect(Bool)) \\ & & \quad \Rightarrow c \to Bool \\ intOrBool\ c & = & member\ 1\ c\ ||\ member\ True\ c \end{array}$$

It is certainly possible that such examples might be useful in some applications. However, in many cases, we would prefer to restrict collections to hold values of a single type only, and to treat function definitions like this as type errors.

The problems described above can be avoided by using parametric type classes [1]. This allows us to use the same class and instance declarations as above, but to extend the compiler with additional static checks to ensure that, if $P_0 \Vdash \{\, \tau \in Collect(\nu), \tau \in Collect(\nu') \,\}$, then $\nu = \nu'$. In effect, this means that, for any satisfiable instance of a predicate $c \in Collect(a)$, the choice of type a is uniquely determined by the choice of c. Note that this can also be captured by an improving function *impr* such that:

$$\frac{\nu \overset{U}{\sim} \nu'}{impr\ \{\tau \in Collect(\nu),\ \tau \in Collect(\nu')\} = U}$$

If no unifier exists, then the predicate set is unsatisfiable, and the improving function may fail. Other useful rules for improvement can be derived from this general rule. For example, according to the instance declarations given above, a predicate of the form $[\nu] \in Collect(\nu')$ can only be satisfied if $\nu = \nu'$, so we can define:

$$\frac{\nu \overset{U}{\sim} \nu'}{impr\ \{[\nu] \in Collect(\nu')\} = U}$$

Parametric type classes are a good way of avoiding the problems with ordinary multiple parameter type classes that were sketched above. Since the choice of a in $c \in Collect(a)$ is uniquely determined by the value of c, there is no need to treat *empty* as having an ambiguous type. In addition, the restrictions on instances of parametric type classes are exactly the conditions that we need to ensure that the definition of *intOrBool* will be treated as a type error.

Improvement is particularly important for Haskell since it permits, for the first time, a useful treatment of multiple parameter type classes that generalizes the use of parametric type classes. At the same time, it is fully compatible with the Haskell type system, including context reduction, and with the use of constructor classes.

5 Taking account of satisfiability

Since the types obtained by improvement in the examples above are obviously instances of the original principal types, it is no surprise to find that our extended type inference algorithm is *sound*, i.e. that the typings it produces are derivable in the original typing rules given in Figure 2.

Theorem 1 *If $Q \,|\, TA \stackrel{W}{\vdash} E : \nu$, then $Q \,|\, TA \vdash E : \nu$.*

On the other hand, the new algorithm is certainly not *complete*. Indeed, it is not even well-defined with respect to the natural equivalence on type schemes induced by the $\leq$ ordering. For example, the two type schemes for the function f in Section 4.1, either of which could be produced by the new algorithm, are:

$$\begin{array}{rcl} \sigma_1 & = & \forall r.\forall a.(r \;\mathsf{has}\; l\!:\!a) \Rightarrow r \to (a, a), \\ \sigma_2 & = & \forall r.\forall a.\forall b.(r \;\mathsf{has}\; l\!:\!a,\; r \;\mathsf{has}\; l\!:\!b) \Rightarrow r \to (a, b). \end{array}$$

It is easy to show that $\sigma_1 \leq \sigma_2$, but these types are not equivalent because $\sigma_2 \not\leq \sigma_1$. Hence the new type inference algorithm may give a result type σ_1 that is not principal.

Even worse, there are terms that can be typed using the original rules in Figure 2, but for which the type inference algorithm may *fail* to produce a typing. For example, consider the expression:

$$\lambda r.\mathbf{let}\ (x, y) = f\ r\ \mathbf{in}\ (x + 1,\ \mathit{not}\ y).$$

Using the type scheme σ_2 for f, we can instantiate the type variables a and b to Int and Bool, respectively, to obtain a typing for this term. However, using the type scheme σ_1, the type inference algorithm fails because the types Int and Bool do not match.

The key to solving this problem is to notice that, although we can obtain a typing for this term using type scheme σ_2, the corresponding predicate set, $\{r \;\mathsf{has}\; l\!:\!\mathit{Int},\; r \;\mathsf{has}\; l\!:\!\mathit{Bool}\}$, is not satisfiable. We will see that the problems described above can be avoided by:

- Proving completeness of the algorithm with respect to a weaker ordering that identifies type schemes with the same set of satisfiable instances: Just as the original ordering on type schemes was defined in terms of the set of generic instances of a type scheme, we will define the new ordering in terms of the *generic satisfiable instances* of a type scheme with respect to a predicate set P_0. The set of generic satisfiable instances of a type scheme is defined by:

  $$\begin{array}{c} [\![P' \,|\, \forall\alpha_i.P \Rightarrow \tau]\!]^{sat}_{P_0} \\ = \\ \{\, [\nu_i/\alpha_i]\tau \;|\; \nu_i \in \mathit{Type},\; P_0 \Vdash P', [\nu_i/\alpha_i]P \,\}. \end{array}$$

  The satisfiability ordering, again with respect to P_0, can now be defined using:

  $$(P \,|\, \sigma) \leq^{sat}_{P_0} (P' \,|\, \sigma') \;\Leftrightarrow\; [\![P \,|\, \sigma]\!]^{sat}_{P_0} \subseteq [\![P' \,|\, \sigma']\!]^{sat}_{P_0}.$$

  Clearly, $\leq^{sat}$ is both reflexive and transitive. It is also quite easy to show that it is weaker than $\leq$, i.e. that:

  $$(P \,|\, \sigma) \leq (P' \,|\, \sigma') \;\Rightarrow\; (P \,|\, \sigma) \leq^{sat}_{P_0} (P' \,|\, \sigma').$$

  On the other hand, unlike $\leq$, the satisfiability ordering is not preserved by substitution[3].

[3] As a counter example, consider a predicate $a \in C$ for which the only satisfiable instance is $\mathit{Int} \in C$. Now consider the qualified types $\eta_i = (a_i \in C) \Rightarrow a_i$, for $i = 1, 2$, and let $S = [\mathit{Int}/a_1]$. Then $[\![\eta_1]\!]^{sat} = \emptyset = [\![\eta_2]\!]^{sat}$, and hence $\eta_1 \leq^{sat} \eta_2$, but $[\![S\eta_1]\!]^{sat} = \{\mathit{Int}\}$ while $[\![S\eta_2]\!]^{sat} = [\![\eta_2]\!]^{sat} = \emptyset$, so $S\eta_1 \not\leq S\eta_2$.

- Restricting our attention to satisfiable typings: A typing of the form $P \,|\, A \vdash E : \sigma$ is of no practical use if the predicates in P do not hold or if there is no way to satisfy the predicates involved in σ. We can capture these conditions formally by defining:

  $$P_0 \;\mathsf{sat}\; (P' \,|\, \sigma) \;\Leftrightarrow\; [\![P' \,|\, \sigma]\!]^{sat}_{P_0} \neq \emptyset.$$

  The following properties of this relationship between predicate sets and type schemes are easily established and show that this notion of satisfiability is well-behaved with respect to polymorphism (i.e. instantiating free variables), entailment and ordering:

  - If $P_0 \;\mathsf{sat}\; (P \,|\, \sigma)$, then $SP_0 \;\mathsf{sat}\; S(P \,|\, \sigma)$ for any substitution S.
  - If $P_0 \;\mathsf{sat}\; (P \,|\, \sigma)$ and $Q_0 \Vdash P_0$, then $Q_0 \;\mathsf{sat}\; (P \,|\, \sigma)$.
  - If $P_0 \;\mathsf{sat}\; (P' \,|\, \sigma')$ and $(P' \,|\, \sigma') \leq^{sat}_{P_0} (P \,|\, \sigma)$, then $P_0 \;\mathsf{sat}\; (P \,|\, \sigma)$.

There is one further problem that occurs when an unused let-bound variable is assigned an unsatisfiable type scheme. As an illustration, consider the following type assignment and predicate set:

$$\begin{array}{rcl} A & = & \{\, f\!:\!\forall\alpha.\alpha \in C \Rightarrow \alpha \to \alpha,\; z\!:\!\forall\alpha.\alpha \in D \Rightarrow \alpha \,\} \\ P & = & \{\, a \in C,\; a \in D \,\} \end{array}$$

where C and D are disjoint singletons, say $C = \{\, \mathit{Int} \,\}$ and $D = \{\, \mathit{Bool}, \}$, and hence:

$$[\mathit{Int}/\alpha]\ \mathit{improves}\ \{\alpha \in C\} \text{ and } [\mathit{Bool}/\alpha]\ \mathit{improves}\ \{\alpha \in D\}.$$

Using the original typing rules for OML, we can construct a derivation of the form:

$$\dfrac{\dfrac{\dfrac{(P', P) \,|\, A \vdash f : a \to a \quad (P', P) \,|\, A \vdash z : a}{(P', P) \,|\, A \vdash f\ z : a}}{P' \,|\, A \vdash f\ z : \sigma} \quad P' \,|\, A, x\!:\!\sigma \vdash F : \tau}{P' \,|\, A \vdash \mathbf{let}\ x = f\ z\ \mathbf{in}\ F : \tau}$$

where $\sigma = (\forall a.(a \in C,\ a \in D) \Rightarrow a)$. Clearly σ is not satisfiable, but, if the bound variable x does not appear free in F, then there is no reason for these unsatisfiable constraints to be reflected by the predicates in P'. However, in the type inference algorithm, using improvement immediately after the introduction of the variables f and z would produce typings of the form:

$$Q \,|\, A \vdash f : \mathit{Int} \to \mathit{Int} \quad \text{and} \quad Q \,|\, A \vdash z : \mathit{Bool}$$

and the algorithm will fail to infer a type for the expression $f\,z$. The problem here is that, since $x \notin FV(F)$, the use of generalization in the typing rule for let-expressions allows us to hide, and then discard unsatisfiable constraints.

We will describe an expression of the form $\mathbf{let}\ x = E\ \mathbf{in}\ F$ where $x \notin FV(F)$ as a *redundant let-binding*. Such bindings serve no practical purpose because they can be replaced by the corresponding expression $(\lambda x.F)E$ without a change in either semantics or typeability. With this observation, it is reasonable to restrict our attention to terms with no redundant let-bindings. Obviously, if this property holds for a given term E, then it also holds for all subterms of E, a fact that can be used implicitly in proofs by induction.

Given the definitions above, we can now state the main completeness result for a type inference algorithm that supports both simplification and improvement rules:

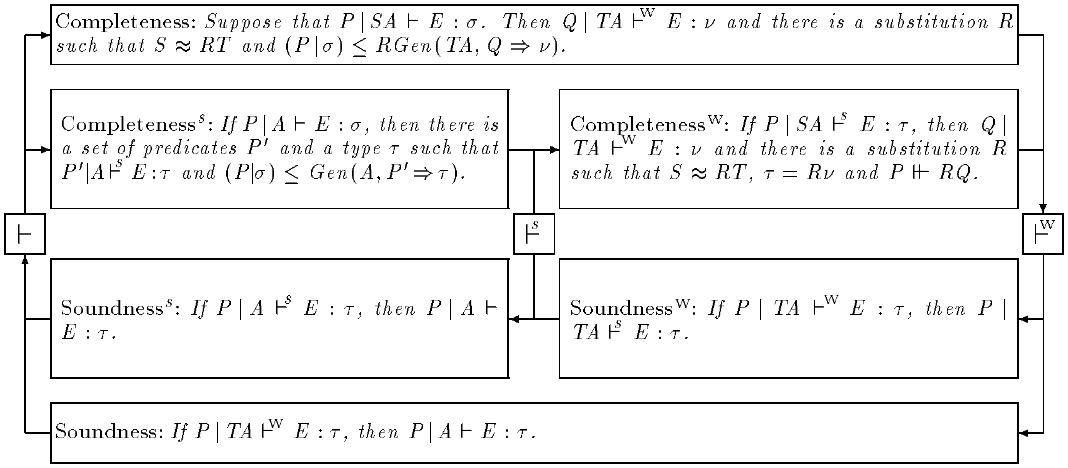

Figure 4: A summary of the original soundness and completeness results.

Theorem 2 *Suppose that* $P \mid A \vdash E : \sigma$ *and* P_0 sat $(P \mid \sigma)$ *where* E *is a term with no redundant let-bindings and* $TV(A) = \emptyset$. *Then the type inference algorithm for* E *in* A *will not fail, and, for any* Q *and* ν *such that* $Q \mid A \vdash^{W} E : \nu$, *we have:*

$$(P \mid \sigma) \leq_{P_0}^{sat} Gen(A, Q \Rightarrow \nu).$$

This result indicates that, if a term E has any satisfiable typings for a set A of typing assumptions, then there is a principal type $\eta = Gen(A, Q \Rightarrow \nu)$ which is more general than every satisfiable typing for E in A. In fact, the principal type is itself a satisfiable typing for E in A:

- Satisfiability follows directly from the fact that it is an upper bound of a non-empty set of satisfiable constrained type schemes.
- To see that η is a typing for E in A, we can use the soundness result above (Theorem 1) to show that $Q \mid A \vdash E : \nu$, and then use $(\Rightarrow I)$ and $(\forall I)$ to obtain a derivation $\emptyset \mid A \vdash E : \eta$.

It is also possible to state a more general version of the completeness theorem without the requirement that $TV(A) = \emptyset$. However, in practice, this special case is usually of most interest, corresponding to the process of calculating the type for a top-level definition in a Haskell or ML program.

6 Proof of soundness and completeness

The main purpose of this section is to prove the soundness and completeness results stated in the previous section as Theorem 1 and Theorem 2, respectively. This is a complicated task that requires careful management and structuring. Fortunately, we can reduce the amount of work involved by building on the results of previous work. The diagram in Figure 4 summarizes the main results for the original system of qualified types [11, 10], describing the relationship between typing judgements in three different systems:

- The original typing rules for OML (Figure 2), described by judgements using the $\vdash$ symbol.
- The type inference algorithm (Figure 3), described by judgements using the $\vdash^{W}$ symbol.
- A collection of 'syntax-directed' typing rules described by judgements using the $\vdash^{s}$ symbol. This system provides a convenient stepping stone between the original typing rules and the type inference algorithm, and will be described in more detail below.

Following convention, the results in the top portion of Figure 4 are described as completeness properties, while those in the lower portion are referred to as soundness properties. In each case, the result linking the $\vdash$ and $\vdash^{W}$ systems can be obtained from the corresponding properties involving $\vdash^{s}$.

The original typing rules for OML are not suitable for type inference: there are many different ways that the rules can be applied to a given term, but it is not clear which, if any, will lead to a principal type. In earlier work, following standard techniques [2], we avoided these problems by defining a syntax-directed system and proving its equivalence with the original type system using the theorems labelled Soundness^{s} and Completeness^{s} in Figure 4. The most important property of the syntax-directed system is that the structure of every typing derivation is uniquely determined by the syntactic structure of the term involved.

The typing rules for the syntax-directed system are given in Figure 5 using judgements of the form $P \mid A \vdash^{s} E : \tau$.

Several useful properties of the syntax-directed system have been established [11, 10], including:

- If $P|A \vdash^{s} E : \tau$ and S is a substitution, then $SP \mid SA \vdash^{s} E : S\tau$.
- If $P \mid A \vdash^{s} E : \tau$ and $Q \Vdash P$, then $Q \mid A \vdash^{s} E : \tau$.
- If $P \mid A' \vdash^{s} E : \tau$ and $A' \leq A$, then $P \mid A \vdash^{s} E : \tau$.

For the purposes of the work described in this paper, the first two properties are important because they are exactly what we need to establish soundness of rules (*Imp*) and (*Simp*), respectively, in the proof of Theorem 1 (The remaining cases are the same as in the original proof of Soundness^{W}).

The expression $A \leq A'$ in the third property indicates that the two type assignments A and A' have the same domain and that $A(x) \leq A'(x)$ for each variable x bound in A. In the following, we will use the obvious counterpart to describe when one type assignment A' is more general than another A

$$(var)^s \quad \frac{(x{:}\sigma) \in A \quad (P \Rightarrow \tau) \leq \sigma}{P \,|\, A \overset{s}{\vdash} x : \tau} \qquad (\rightarrow E)^s \quad \frac{P \,|\, A \overset{s}{\vdash} E : \tau' \rightarrow \tau \quad P \,|\, A \overset{s}{\vdash} F : \tau'}{P \,|\, A \overset{s}{\vdash} EF : \tau}$$

$$(\rightarrow I)^s \quad \frac{P \,|\, A_x, x{:}\tau' \overset{s}{\vdash} E : \tau}{P \,|\, A \overset{s}{\vdash} \lambda x.E : \tau' \rightarrow \tau} \qquad (let)^s \quad \frac{P \,|\, A \overset{s}{\vdash} E : \tau \quad P' \,|\, A_x, x{:}\sigma \overset{s}{\vdash} F : \tau' \quad \sigma = Gen(A, P \Rightarrow \tau)}{P' \,|\, A \overset{s}{\vdash} (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau'}$$

Figure 5: Syntax-directed inference system.

with respect to the satisfiability ordering, written $A \leq^{sat}_{P_0} A'$. The third property plays an important role in the proof of Completeness^{W}, i.e. for the original system of qualified types [10]. Unfortunately, the corresponding result with $\leq$ replaced by $\leq^{sat}_{P_0}$ does not hold. A simple counterexample can be obtained from the example in Section 5, since $(P', P) \,|\, A \overset{s}{\vdash} f\, z : a$, and $A \leq^{sat}_{P_0} A' = \{f{:}(Int \rightarrow Int),\, z{:}Bool\}$, but there is no derivable typing for the expression $f\, z$ using the assignment A'. We avoid this problem in the completeness theorem below by including an assumption of the form $A \leq^{sat}_{P_0} SA'$ as an extra hypothesis. A similar technique is used in Smith's thesis [19].

In the remaining part of this section, we will describe a replacement for the Completeness^{W} result in Figure 4 that allows the use of the rules (*Simp*) and (*Imp*) at arbitrary points during type inference. Combined with Completeness^{s}, we will show how this can be used to establish Theorem 2, an analogue of the Completeness result at the top of Figure 4.

Our first task is to justify the informal comments about terms with no redundant let-bindings in Section 5. Using the syntax-directed typing rules, the following theorem shows that, the class constraints for a variable x appearing free in an expression E will be reflected by the constraints P on the use of E itself:

Proposition 3 *Suppose that* $P \,|\, A \overset{s}{\vdash} E : \tau$, $x \in FV(E)$, $A(x) = (\forall \alpha_i.Q \Rightarrow \nu)$, *and that* E *has no redundant let-bindings. Then there are types* τ_i *such that* $P \Vdash [\tau_i/\alpha_i]Q$.

In particular, if we have a satisfiable syntax-directed typing for a term E with respect to some set of assumptions A, and if the variable x appears free in E, then the type assigned to x in A must also be satisfiable:

Corollary 4 *Suppose that* $P \,|\, A_x, x{:}Gen(A, Q \Rightarrow \nu) \overset{s}{\vdash} E : \tau$, $x \in FV(E)$, P_0 sat $Gen(A, P \Rightarrow \tau)$, *and that* E *has no redundant let-bindings. Then:*

$$P_0 \ \mathsf{sat}\ Gen(A, Q \Rightarrow \nu).$$

Working towards a completeness result for the type inference algorithm with respect to the syntax-directed type system, suppose that we have a derivation $P \,|\, A \overset{s}{\vdash} E : \tau$. Our goal is to prove that:

- The type inference algorithm will not fail to find a type for E in A. Since the algorithm may fail if E does not have any satisfiable typings, it will be necessary to restrict our completeness result to satisfiable syntax-directed derivations, i.e. to derivations $P \,|\, A \overset{s}{\vdash} E : \tau$ such that P_0 sat $Gen(A, P \Rightarrow \tau)$.

- If the type inference algorithm produces a typing $Q \,|\, TA \overset{W}{\vdash} E : \nu$, then the corresponding type scheme $Gen(TA, Q \Rightarrow \nu)$ is more general than the type assigned to E in the syntax-directed system, i.e. we want to show that: $Gen(A, P \Rightarrow \tau) \leq^{sat}_{P_0} Gen(TA, Q \Rightarrow \nu)$. In fact, to carry out the required proof, it is necessary to generalize the hypotheses a little, allowing the use of distinct type assignments, A in the original syntax-directed typing, and A' in the type inference algorithm, related by $A \leq^{sat}_{P_0} SA'$ for some substitution S.

Motivated in part by these comments, we use the following theorem to express the completeness of the type inference algorithm with respect to the syntax-directed rules, and the satisfiability ordering, $\leq^{sat}_{P_0}$:

Theorem 5 *Suppose that* $P \,|\, A \overset{s}{\vdash} E : \tau$, P_0 sat $Gen(A, P \Rightarrow \tau)$, $A \leq^{sat}_{P_0} SA'$, *and that* E *does not contain any redundant let-bindings. Then the type inference algorithm will not fail, and for every* $Q \,|\, TA' \overset{W}{\vdash} E : \nu$, *there is a substitution* R *such that* $RT \approx S$ *and:*

$$Gen(A, P \Rightarrow \tau) \leq^{sat}_{P_0} RGen(TA', Q \Rightarrow \nu).$$

The proof of this theorem is a little complex; full details are given elsewhere [15]. However, with this result in hand, the proof of Theorem 2 is straightforward. Suppose that $P \,|\, A \vdash E : \sigma$ and P_0 sat $(P \,|\, \sigma)$ where E is a term with no redundant let-bindings and $TV(A) = \emptyset$. By Completeness^{s}, we know that $P' \,|\, A \overset{s}{\vdash} E : \tau'$ for some P' and τ' such that:

$$(P \,|\, \sigma) \leq Gen(A, P' \Rightarrow \tau').$$

From the properties of $\leq^{sat}_{P_0}$, it follows that $(P \mid \sigma) \leq^{sat}_{P_0} Gen(A, P' \Rightarrow \tau')$, and hence, since P_0 sat $(P \mid \sigma)$, that P_0 sat $Gen(A, P' \Rightarrow \tau')$. Since $A \leq^{sat}_{P_0} A$, we can use Theorem 5 to show that the type inference algorithm will not fail and that, for each $Q \,|\, TA' \overset{W}{\vdash} E : \nu$, there is a substitution R such that $Gen(A, P' \Rightarrow \tau') \leq^{sat}_{P_0} RGen(TA, Q \Rightarrow \nu)$. Since $TV(A) = \emptyset$, we know that $Gen(TA, Q \Rightarrow \nu)$ has no free variables and that $TA = A$. Combining the two $\leq^{sat}_{P_0}$ orderings above, we obtain $(P \,|\, \sigma) \leq^{sat}_{P_0} Gen(A, Q \Rightarrow \nu)$ as required.

7 Discussion

The ideas described in this paper provide a general and modular framework for the design of constrained type systems, taking advantage of information about satisfiability of constraints to infer more accurate and informative principal types.

The design of specific applications of our framework starts with the choice of a system of predicates and an entailment relation, as described in Section 2.1. Without any further work, the original type inference algorithm presented in Figure 3 can be used to calculate principal typings for the corresponding system of qualified types.

Extending the algorithm with rules for simplification and improvement leads to a non-deterministic type inference algorithm. This allows us to choose how the rules will be combined in particular ways to provide a deterministic algorithm for use in practical implementations. For a more algorithmic flavour, we would normally expect the implementation of simplification and improvement to be described by (deterministic) functions, rather than the more general '$\Leftrightarrow$' and '*improves*' relations used by the presentations in Sections 3 and 4, respectively:

- A simplifying function, *simp*, mapping predicate sets P to appropriate 'simplified' versions, *simp* P, might can be used to implement simplification. The only condition that a simplifying function must satisfy is that $P \Leftrightarrow (simp\ P)$, for all predicate sets P. In this setting, the inference rule (*Simp*) introduced in Section 3 might be replaced by:

$$\frac{Q \mid TA \vdash^{\mathrm{W}} E : \nu \quad P = simp\ Q}{P \mid TA \vdash^{\mathrm{W}} E : \nu}$$

  For any predicate system, the identity function specified by $simp\ P = P$ can be used as a simplifying function. However, more interesting, and more useful functions can be used in specific applications.

- In a similar way, an improving function, *impr*, mapping sets of predicates to suitable improving substitutions, can be used to implement improvement, as described in Section 4.1. The correctness of an improving function can be specified by the requirement that $(impr\ P)$ *improves* P, for all predicate sets P, and the (*Imp*) inference rule introduced in Section 4 can be rewritten to use an improving function:

$$\frac{Q \mid TA \vdash^{\mathrm{W}} E : \nu \quad T' = impr\ Q}{T'Q \mid T'TA \vdash^{\mathrm{W}} E : T'\nu}$$

  The trivial improving function, $impr\ P = id$ can be used with any system of predicates, but it is often possible to find more useful definitions.

Note that it is possible to arrange for simplifying or improving functions to fail if they are applied to an unsatisfiable predicate set, causing the type inference algorithm to fail as a result. However, while this may be useful in some applications, it is not required; in the general case, testing for satisfiability of a predicate set is undecidable and we would not be able to guarantee termination of the type inference algorithm (see the work of Volpano and Smith [20], for example). For similar reasons, while we expect the results of simplifying and improving functions to satisfy certain correctness conditions, we do not insist that they are 'optimal'; in the general case, there may not be any effective way to find such optimal solutions.

Our approach is to leave the task of finding suitable simplifying and improving functions to the designer of specific applications of qualified types. In this way, designers retain control over the balance between making good use of simplification and improvement, and ensuring that type inference remains tractable. This is in contrast to earlier work, for example, by Smith [19], where full tests for satisfiability of predicate sets are needed and strong restrictions on the definition of predicate entailments are needed to guarantee a decidable type inference process.

One of the most obvious places to use simplification and improvement is immediately before generalizing the type of a let-bound variable. For example, using the rules above, we can derive the following rule for calculating the type of a let-expression:

$$\frac{\begin{array}{ll} P \mid TA \vdash^{\mathrm{W}} E : \nu & T' = impr\ P \\ Q = simp\ (T'P) & \sigma = Gen(T'TA, Q \Rightarrow T'\nu) \\ \multicolumn{2}{c}{P' \mid T''(T'TA_x, x : \sigma) \vdash^{\mathrm{W}} F : \tau'} \end{array}}{P' \mid T''T'TA \vdash^{\mathrm{W}} (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau'}$$

Of course, we could have started out with this rule at the very beginning. However, our approach seems much more attractive and modular since it allows us to view the typing of let-expressions and the treatment of simplification and improvement as independent concerns and to combine them in ways that are not captured by the rule above.

In their current state, the ideas presented in this paper are of most use to language designers, not to programmers. For example, the design of a type inference algorithm for a language with parametric type classes can be based on the framework and algorithms presented here. It would also be interesting to explore more ambitious language designs that provide the programmer with the ability to define and extend simplifying and improving functions. Some first steps in this direction have been made [15]. In the meantime, the work described here provides simple correctness criteria for simplifying and improving functions which can be useful in the construction of such functions for specific applications of qualified types. For example, the task of simplifying predicate sets containing subtyping constraints has been studied in some depth by several researchers [16, 17, 4, 3, 19]. This paper does not subsume the results of their work. Rather, it provides a general framework to which their results may be applied in the design of type systems combining polymorphism and subtyping.

Acknowledgements

Much of this work was carried out while the author was a member of the Department of Computer Science, Yale University, and was supported in part by a grant from ARPA, contract number N00014-91-J-4043. It is a pleasure to thank Kung Chen for useful discussions about the ideas presented in this paper, and for helpful insights into his work on parametric type classes.

The concept of principal satisfiable type schemes was originally introduced in [10, Chapter 6]. At that time, we conjectured that every term with a satisfiable typing could be assigned a principal satisfiable typing, a fact which we have, finally, had the opportunity to prove!

References

[1] K. Chen, P. Hudak, and M. Odersky. Parametric type classes (extended abstract). In *ACM conference on LISP and Functional Programming*, San Francisco, CA, June 1992.

[2] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML.

In *ACM symposium on LISP and Functional Programming, Cambridge, Massachusetts*, August 1986.

[3] Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proceedings of TAPSOFT 89*, New York, 1989. Springer-Verlag. Lecture Notes in Computer Science, 352.

[4] Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical computer science*, 73, 1990.

[5] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Conference record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991.

[6] R.W. Harper and B.C. Pierce. Extensible records without subsumption. Technical report CMU-CS-90-102, Carnegie Mellon University, School of computer science, February 1990.

[7] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[8] Mark P. Jones. *Introduction to Gofer 2.20*, September 1991. Available by anonymous ftp as part of the standard Gofer distribution.

[9] Mark P. Jones. Computing with lattices: An application of type classes. *Journal of Functional Programming*, 2(4), October 1992.

[10] Mark P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.

[11] Mark P. Jones. A theory of qualified types. In *ESOP '92: European Symposium on Programming, Rennes, France*, New York, February 1992. Springer-Verlag. Lecture Notes in Computer Science, 582.

[12] Mark P. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, New Haven, Connecticut, USA, September 1993.

[13] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, New York, June 1993. ACM Press.

[14] Mark P. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994.

[15] Mark P. Jones. Simplifying and improving qualified types. Research Report YALEU/DCS/RR-1040, Yale University, New Haven, Connecticut, USA, June 1994.

[16] J.C. Mitchell. Coercion and type inference (summary). In *Conference record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City, Utah, January 1984.

[17] J.C. Mitchell. Type inference with simple subtypes. *Journal of functional programming*, 1(3):245–286, July 1991.

[18] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Conference record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, TX, January 1989.

[19] Geoffrey Seward Smith. *Polymorphic type inference for languages with overloading and subtyping*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, August 1991.

[20] D. Volpano and G. Smith. On the complexity of ML typability with overloading. In *5th ACM conference on Functional Programming Languages and Computer Architecture*, New York, 1991. Springer-Verlag. Lecture Notes in Computer Science, 523.

[21] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Jan 1989.