Simplifying and Improving Qualified Types

Mark P. Jones
Yale University, Department of Computer Science
P.O. Box 208285, New Haven, CT 06520-8285.
`jones-mark@cs.yale.edu`

Research Report YALEU/DCS/RR-1040, June 1994

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 for the previously troublesome multiple parameter extensions of Haskell type classes. For example, a form of *parametric type classes*, proposed by Chen, Hudak and Odersky, can be viewed as a special case of this work.

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 allow the use of qualified types. In particular, we have shown that 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 also flexible enough to allow some 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 be used to support a form of *parametric type classes* [1].

Outline of this report

We start, in Section 1, 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 2 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 3, we introduce the concept of improvement which is the major contribution of this report, 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 4. The proofs of soundness and completeness properties for a type inference algorithm that uses simplification and improvement are discussed in Section 5, with detailed proofs of new results provided in the appendix. We conclude with a discussion on the use of our new framework in Section 6.

1 A brief overview of qualified types

To describe the contributions of this report 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 detailed presentations of [10, 13, 12] and assume some familiarity with this previous work.

1.1 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

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 [22, 8].
$t \in Num$	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 [22, 8].
$t \in Collect(s)$	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 [11].
$s \subseteq t$	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 [17, 18, 5, 4, 20].
r has $l\!:\!t$	r is a record type containing an field labelled l of type t [7].
r lacks l	r is a record type, not including a field labelled l [7].
$r_1\#r_2$	The record types r_1 and r_2 do not have any fields in common [6].

Figure 1: Examples of individual predicates and their informal intepretation

different syntax from earlier presentations, in the hope that this will make the interpretation of predicate expressions a little more obvious.

1.2 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:

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

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:

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

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. 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'.

1.3 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 corresponding to global constraints

Standard rules:	(var)	$\dfrac{(x\!:\!\sigma) \in A}{P \,\vert\, A \vdash x : \sigma}$
	$(\to E)$	$\dfrac{P \,\vert\, A \vdash E : \tau' \to \tau \quad P \,\vert\, A \vdash F : \tau'}{P \,\vert\, A \vdash EF : \tau}$
	$(\to I)$	$\dfrac{P \,\vert\, A_x, x\!:\!\tau' \vdash E : \tau}{P \,\vert\, A \vdash \lambda x.E : \tau' \to \tau}$
Qualified types:	$(\Rightarrow E)$	$\dfrac{P \,\vert\, A \vdash E : Q \Rightarrow \rho \quad P \Vdash Q}{P \,\vert\, A \vdash E : \rho}$
	$(\Rightarrow I)$	$\dfrac{P, Q \,\vert\, A \vdash E : \rho}{P \,\vert\, A \vdash E : Q \Rightarrow \rho}$
Polymorphism:	$(\forall E)$	$\dfrac{P \,\vert\, A \vdash E : \forall\alpha.\sigma}{P \,\vert\, A \vdash E : [\tau/\alpha]\sigma}$
	$(\forall I)$	$\dfrac{P \,\vert\, A \vdash E : \sigma \quad \alpha \notin TV(A) \cup TV(P)}{P \,\vert\, A \vdash E : \forall\alpha.\sigma}$
Local Definition:	(let)	$\dfrac{P \,\vert\, A \vdash E : \sigma \quad Q \,\vert\, A_x, x\!:\!\sigma \vdash F : \tau}{P, Q \,\vert\, A \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$

Figure 2: Typing rules for OML.

on the environment in which the type scheme can be used. We start by defining the set of *generic instances* of a constrained type scheme, given by:

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

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

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

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 $S(P \mid \sigma) \leq S(P' \mid \sigma')$, whenever $(P \mid \sigma) \leq (P' \mid \sigma')$. 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 pre-

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

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

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

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

Figure 3: Type inference algorithm W.

sentation, as in previous descriptions of qualified types, follows [19], using judgements of the form $Q \mid TA \overset{\mathrm{W}}{\vdash} E : \nu$ 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 $\mathit{Gen}(A, \rho)$ 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:

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

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 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 $P \,|\, A \vdash E : \sigma$.

2 Simplification

In this section, we will show how the principal types calculated by the type inference algorithm described above can often be simplified by using a different, but equivalent, set of predicates in the infered type. This is not a new idea; similar techniques are already used in other theoretical work, and in the implementations of systems like Haskell [8] and Gofer [16]. One of the advantages of the framework used in this report is that it allows us to view simplification independently of other aspects of the type system, revealing opportunities for specific design decisions that are sometimes hidden in other presentations of constrained type inference. Simplification of qualified types, as described in this section, has previously been suggested in [13, 12].

For convenience, we will 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 the type schemes $(\forall t_i.P \Rightarrow \tau)$ and $(\forall t_i.Q \Rightarrow \tau)$ are equivalent with respect to the $\leq$ ordering on type schemes. To see how this works in practice, suppose that the addition operator $+$ is treated as a function of type $\forall t.(t \in Num) \Rightarrow t \to t \to t$, and consider the following examples:

- **Repeated predicates**: Given the term $(\lambda x.\lambda y.\lambda z.x + y + z)$, the type checking algorithm described above introduces two predicates of the form $(t \in Num)$, one for each use of $+$, where t is the type of the parameters x, y, and z, and of the result of the λ-term. In the current setting, repeated predicates are automatically eliminated by our use of predicate sets. More generally, for example, in [14], where predicate sets are replaced by ordered sequences of predicates, simplification can be used to remove any duplicates.

- **Constant predicates**: Treating 1 as an integer constant of type *Int*, the principal type of the expression $(1 + 1)$ produced by the algorithm above is $(Int \in Num) \Rightarrow Int$. Simplification corresponds to discharging the assumption that *Int* is included in the *Num* class, and allows us to treat the expression as having type *Int*. This is an example of the concept of *constant overloading* described in [13].

- **Detecting type errors**: Consider the function $\lambda x.('a' + x)$ which attempts to add its argument value x to a character constant $'a'$. Since characters are not usually regarded as numeric values, we might expect this term to produce a type error. Instead, the standard type inference algorithm assigns a principal type of $(Char \in Num) \Rightarrow Char \to Char$ to this term. Examples like this were presented by Volpano and Smith [21] as criticisms of the original Wadler and Blott proposals for type class overloading [22], and used to motivate the use of slightly different set of typing rules that would reject such terms. In fact, the Haskell type system already has this behaviour. A similar effect can be obtained by arranging for the simplification process to fail when an unsatisfiable predicate set is encountered.

  However, it is important to realize that this is a specific design decision, not an essential part of the system. For example, in a language emphasizing extensibility, we might argue that the predicate $Char \in Num$ should not be treated as an error. Instead, we simply prohibit any use of a function whose context includes this predicate until some time later, when the programmer has supplied definitions to add *Char* to the *Num* class.

  Another reason for avoiding a check for satisfiability in the simplification process is that it can make the type inference process undecidable. This is exactly the problem that Volpano and Smith described for an unrestricted version of the Wadler and Blott proposal. On the other hand, Haskell avoids this difficulty by imposing static constraints on the form of class and instance declarations that can be used in a program.

  We will see in fact that these issues are best dealt with using the concept of improvement described in Section 3.

- **Superclass constraints**: The type class definitions in the Haskell standard prelude specify that *Eq* is a *superclass* of *Num*, i.e. that $Num \subseteq Eq$. This property is guaranteed by the compiler using static checks to ensure that $t \in Eq$ whenever $t \in Num$. Now consider the function:

$$f \quad = \quad (\lambda x.\lambda y.\lambda z.(x + y) == z).$$

Assuming types $\forall t.(t \in Eq) \Rightarrow t \to t \to Bool$ and $\forall t.(t \in Num) \Rightarrow t \to t \to t$ for the equality and addition operators, respectively, the principal type of the function f is:

$$\forall t.(t \in Eq,\ t \in Num) \Rightarrow t \to t \to t \to Bool.$$

The two predicates here correspond to the use of $==$ and $+$ in the definition of f. However, using the superclass relation described above, this can be simplified to just:

$$\forall t.(t \in Num) \Rightarrow t \to t \to t \to Bool.$$

- **Context reduction**: In Haskell, the types in a type class are described by a collection of *instance declarations*. For example, the standard prelude includes a declaration that includes the type $[t]$ of lists of values of type t in the Eq class, whenever $t \in Eq$. In our formal system, this is described by an entailment $(t \in Eq) \Vdash ([t] \in Eq)$. Now suppose that we define a function:

  $$null \quad = \quad (\lambda xs.xs == [\,])$$

  where $[\,]$ is the empty list. The principal type for $null$ is:

  $$\forall t.([t] \in Eq) \Rightarrow [t] \to Bool.$$

  As it stands, this type is already in the simplest form possible; the entailment given above is not sufficient to derive any other predicate set that is equivalent to $([t] \in Eq)$. However, according to the definition of Haskell, the instance declaration described above is the *only* way to define an equality on lists of type $[t]$. We could therefore argue that this predicate set is 'observationaly equivalent' to $(t \in Eq)$, capturing this property by extending the entailment relation with $([t] \in Eq) \Vdash (t \in Eq)$. Combined with the previous entailment, this shows that $([t] \in Eq) \Leftrightarrow (t \in Eq)$ and hence we can reduce the inferred type for $null$ to:

  $$\forall t.(t \in Eq) \Rightarrow [t] \to Bool.$$

  This idea is used in Haskell to justify the use of rules for simplifying inferred types to use only predicates of the form $t \in C$ where t is a type variable. This process is described as *context reduction* in [12].

In all these examples, we have delayed the use of simplification until the very last stage of type inference, after the calculation of the principal types. In fact, there are obvious benefits to allowing simplification to occur at earlier stages in the typing process. We can extend the algorithm in Figure 3 to allow this by adding the rule (*Simp*):

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

Using (*Simp*) makes the type inference algorithm non-deterministic so that 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 1.3, 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 predicates to be equivalent, we will not attempt to formalize what it means to say that one predicate set 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 that they involve. However, we believe that these issues are best dealt with in the design of specific applications of qualified types. In other words, 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 practical implementation, Q will actually be chosen as a simplified version of P in some appropriate manner.

3 Improvement

A second method for improving the accuracy of an inferred principal types, and the most important contribution of this report, 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.

3.1 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:

$$(\_.l) \quad :: \quad \forall r.\forall t.(r \ \mathsf{has}\ l : t) \Rightarrow r \rightarrow t$$

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 1.3, is:

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

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*:

$$\forall r.\forall a.(r \ \mathsf{has}\ l : a) \Rightarrow r \to (a, a).$$

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:

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

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:

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

In this case, we will say that the substitution $[a/b]$ improves the predicate set $\{r \ \mathsf{has}\ l : a,\ r \ \mathsf{has}\ l : b\}$. 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):

$$\frac{Q \mid TA \overset{\scriptscriptstyle W}{\vdash} E : \nu \qquad T' \ \mathit{improves}\ Q}{T'Q \mid T'TA \overset{\scriptscriptstyle W}{\vdash} E : T'\nu}$$

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.

3.2 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:

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

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:

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

[1] Pun intended!

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:

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

3.3 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.

In the past, 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*, was used to describe duality between lattices in [11]. Unfortunately, practical experience with multiple parameter type classes in Gofer [9] 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:

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

The *empty* value here represents an empty collection, while the *insert* and *member* functions might be used to add an element to a collection, or to test whether a particular element is included in a collection. A more realistic class definition might also include operations for removing elements from collections, for applying a function to each element of a collection, etc.

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

[2] Interestingly enough, these problems do not seem to occur for many useful examples involving multiple parameter constructor classes [15].

we can implement the membership test:

$$\begin{array}{l}
\textbf{instance}\ (a \in Eq) \Rightarrow [a] \in Collect(a)\ \textbf{where} \\
\quad \begin{array}{lcl}
empty & = & [\,] \\
insert\ x\ xs & = & (x : xs) \\
member\ x\ [\,] & = & False \\
member\ x\ (y : ys) & = & x == y\ \ ||\ \ member\ x\ ys
\end{array}
\end{array}$$

A more efficient implementation can 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:

$$\textbf{data}\ BSTree\ a \ = \ Empty\ \mid\ Fork\ a\ (BSTree\ a)\ (BSTree\ a)$$

$$\begin{array}{l}
\textbf{instance}\ (a \in Ord) \Rightarrow (BSTree\ a \in Collect(a))\ \textbf{where} \\
\quad \begin{array}{llcl}
empty & & = & Empty \\
 & & \vdots & \\
member\ x\ Empty & & = & False \\
member\ x\ (Fork\ y\ l\ r) & \mid\ x == y & = & True \\
 & \mid\ x \leq y & = & member\ x\ l \\
 & \mid\ \textbf{otherwise} & = & member\ x\ r
\end{array}
\end{array}$$

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 [12, 14].

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 containing different types of elements and to define functions like:

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

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 as presented in [1, 2]. 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 $P_0 \Vdash \{\, \tau \in \mathit{Collect}(\nu),\ \tau \in \mathit{Collect}(\nu') \,\}$, then $\nu = \nu'$. In effect, this means that, for any satisfiable instance of a predicate $c \in \mathit{Collect}(a)$, 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:

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

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 $[\nu] \in \mathit{Collect}(\nu')$ can only be satisfied if $\nu = \nu'$, so we can define:

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

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 $c \in \mathit{Collect}(a)$ 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 *int_or_bool* will be treated as a type error.

4 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* $Q \mid TA \overset{W}{\vdash} E : \nu$, *then* $Q \mid TA \vdash E : \nu$.

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 3.1, either of which could be produced by the new algorithm, are:

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

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.

Fortunately, 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:

  $$[\![P' \,|\, \forall\alpha_i.P \Rightarrow \tau]\!]^{sat}_{P_0} \quad = \quad \{\,[\nu_i/\alpha_i]\tau \mid \nu_i \in \mathit{Type},\ P_0 \Vdash P', [\nu_i/\alpha_i]P\,\}.$$

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

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

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

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

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

- 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

[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 $S\eta_1 \not\leq S\eta_2$.

way to satisfy the predicates involved in σ. We can capture these conditions formally by defining:

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

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:

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

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

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

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.

More generally, we will describe an expression of the form **let** $x = E$ **in** F where $x \notin FV(F)$ as a *redundant let-binding.* Such bindings serve no practical purpose, except:

- To add extra typing constraints to a term, for example, the principal type of $\lambda x.x$ is $\forall t.t \to t$, but the principal type of $\lambda x.\textbf{let}\ z = x+1\ \textbf{in}\ x$ is $\mathit{Int} \to \mathit{Int}$.
- To control the sequencing of effects, for example in a call-by-value language like Standard ML.

In each case, there are more elegant ways to achieve the same effect, without using a redundant binding. However, a more general approach is simply to replace any redundant bindings with corresponding expressions of the form $(\lambda x.F)E$. Since $x \notin FV(F)$, this term is well-typed if, and only if, the original expression is well-typed. With these observations, 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 will 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:

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

$$(P \,|\, \sigma) \leq^{sat}_{P_0} \mathit{Gen}(A, Q \Rightarrow \nu).$$

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 $\eta = \mathit{Gen}(A, Q \Rightarrow \nu)$ 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 $\emptyset \,|\, A \vdash E : \eta$.

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.

5 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

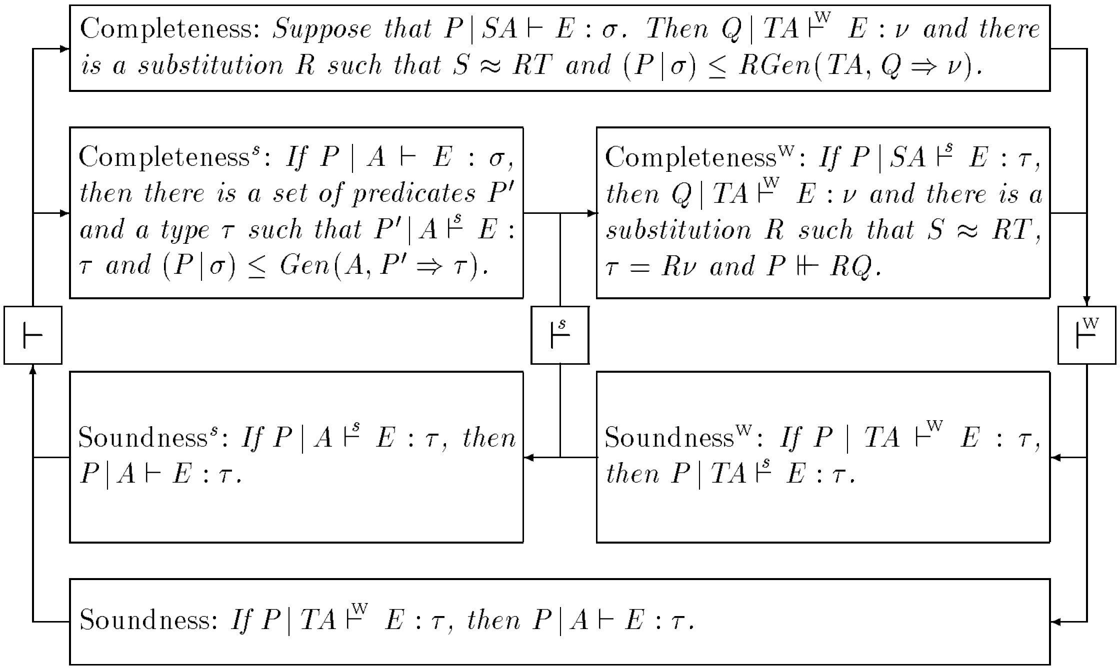

Figure 4: A summary of the original soundness and completeness results.

system of qualified types presented in [13, 12] and 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 main 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 [3], 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^{W} in Figure 4. The most important property of the syntax-directed system is that the 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 $P \,|\, A \vdash^{s} E : \tau$.

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

$$(\to E)^{s} \qquad \frac{P \,|\, A \vdash^{s} E : \tau' \to \tau \quad P \,|\, A \vdash^{s} F : \tau'}{P \,|\, A \vdash^{s} EF : \tau}$$

$$(\to I)^{s} \qquad \frac{P \,|\, A_x, x : \tau' \vdash^{s} E : \tau}{P \,|\, A \vdash^{s} \lambda x.E : \tau' \to \tau}$$

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

Figure 5: Syntax-directed inference system.

Several useful properties of the syntax-directed system were established in [13, 12], including:

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

For the purposes of the work described in this report, the first two properties are important because they are exactly what we need to establish the soundness properties for the 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 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} in [12]. Unfortunately, the corresponding result with $\leq$ replaced by $\leq^{sat}_{P_0}$ does not hold. A simple counterexample can be obtained from the definitions on Page 17, since $(P', P) \mid A \overset{s}{\vdash} f\ z : a$, and $A \leq^{sat}_{P_0} A' = \{f : (\mathit{Int} \to \mathit{Int}),\ z : \mathit{Bool}\}$, but there is no derivable typing for the expression $f\ z$ using the assignments in 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 [20].

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

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

Working towards a completeness result for the type inference algorithm with respect to the syntax-directed type system, suppose that we have a derivation $P \mid A \overset{s}{\vdash} E : \tau$. 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 $P \mid A \overset{s}{\vdash} E : \tau$ such that P_0 sat $\mathit{Gen}(A, P \Rightarrow \tau)$.

- 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 $P \mid A \overset{s}{\vdash} E : \tau$, 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 $Q \mid TA' \overset{W}{\vdash} E : \nu$, there is a substitution R such that:*

$$RT \approx S \quad \textit{and} \quad Gen(A, P \Rightarrow \tau) \leq^{sat}_{P_0} RGen(TA', Q \Rightarrow \nu).$$

The proof of this theorem is a little complex; full details are included in the appendix. 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 $(P \,|\, \sigma)$, 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.

6 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 1.1. 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 2 and 3, 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 2 might be replaced by:

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

  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 3.1. 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 3 can be rewritten to use an improving function:

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

  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 [21], 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, in [20], 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, we might calculate the type of a let-expression with a derivation of the form:

$$\dfrac{\dfrac{\dfrac{P \,|\, TA \overset{\text{W}}{\vdash} E : \nu \quad T' = \mathit{impr}\ P}{T'P \,|\, T'TA \overset{\text{W}}{\vdash} E : T'\nu} \qquad Q = \mathit{simp}\ (T'P)}{Q \,|\, T'TA \overset{\text{W}}{\vdash} E : T'\nu} \qquad P' \,|\, T''(T'TA_x, x : \sigma) \overset{\text{W}}{\vdash} F : \tau'}{P' \,|\, T''T'TA \overset{\text{W}}{\vdash} (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau'}$$

where $\sigma = \mathit{Gen}(T'TA, Q \Rightarrow T'\nu)$. This can be packaged up as a new inference rule for typing let-expressions to replace the original $(\mathit{let})^{\text{W}}$ rule:

$$\dfrac{P \,|\, TA \overset{\text{W}}{\vdash} E : \nu \quad T' = \mathit{impr}\ P \quad Q = \mathit{simp}\ (T'P) \quad P' \,|\, T''(T'TA_x, x : \sigma) \overset{\text{W}}{\vdash} F : \tau'}{P' \,|\, T''T'TA \overset{\text{W}}{\vdash} (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau'}$$

Of course, we probably 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 there current state, the ideas presented in the report 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. Such a system might be used to support parametric type classes as a programmable extension of the language, rather than a built-in part of the type system. One way that this might be achieved for the *Collect* class described in Section 3.3 is to augment the class and instance declarations given there with a declaration of the form:

$$\textbf{improve}\ (a \in \mathit{Collect}(b),\ a \in \mathit{Collect}(c))\ \textbf{using}\ b = c$$

The biggest problems with this approach would be to ensure that the compiler is able to enforce the restrictions on instance definitions that this declaration implies, and to maintain decidability of type inference.

In passing, we mention that this **improve** P **using** S construct can be used to provide similar behaviour to the 'default' mechanism in Haskell [8, Section 4.3.4], for example:

$$\textbf{improve}\ (a \in \mathit{Num},\ a \in \mathit{Integral})\ \textbf{using}\ a = \mathit{Int}$$

However, there are some significant theoretical problems with the current default mechanism causing a loss of coherence in some cases. The same is true for the improving rule given above which is not actually valid in Haskell since the standard prelude provides at least two distinct types, *Int* and *Integer*, both of which can be used as solutions for the predicates $(a \in \mathit{Num},\ a \in \mathit{Integral})$.

We should also note that the use of information about satisfiability must be balanced against the goals of extensibility. For example, consider a module in which the only class and instance declarations are:

$$\begin{array}{l} \textbf{class}\ a \in C\ \textbf{where} \\ \quad \mathit{compare}\ \ ::\ \ a \to a \to \mathit{Bool} \\ \\ \textbf{instance}\ \mathit{Int} \in C\ \textbf{where}\ \ldots \end{array}$$

In this situation, we might be tempted to add an implicit improving rule of the form:

$$\textbf{improve}\ (a \in C)\ \textbf{using}\ a = \mathit{Int}.$$

However, this would not be valid if, in another module in the same program, we tried to extend the class C with an instance:

$$\textbf{instance}\ \mathit{Float} \in C\ \textbf{where}\ \ldots$$

With this example in mind, it is probably better to require **improving** declarations to be written explicitly by the programmer, and to be exported from a module as part of the definition of a class.

The work described here provides simple correctness criteria for simplifying and improving functions, but it does not provide any further insights into the construction of such functions for specific applications. For example, the task of simplifying predicate sets containing subtyping constraints has been studied in some depth by several researchers, including [17, 18, 5, 4, 20]. This report does not subsume the results of those papers. Rather, it aims to provide a general framework, to which they may be applied in the design of type systems combining polymorphism and subtyping.

Acknowledgements

This work 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 report, and for helpful insights into his own work on parametric type classes [2].

The concept of principal satisfiable type schemes was originally introduced in [12, 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!

Appendix: Proofs

This appendix contains detailed proofs for the main new results presented in this report. For convenience, we repeat the statement of each result in a box at the beginning of the corresponding proof.

> **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* $P \Vdash [\tau_i/\alpha_i]Q$.

By induction on the structure of $P \,|\, A \overset{s}{\vdash} E : \tau$:

Case $(var)^s$**:** We have a derivation of the form:

$$\frac{(x : \sigma) \in A \quad (P \Rightarrow \tau) \leq \sigma}{P \,|\, A \overset{s}{\vdash} x : \tau}$$

where $\sigma = A(x) = (\forall \alpha_i . Q \Rightarrow \nu)$. Since $(P \Rightarrow \tau) \leq \sigma$, it follows that $P \Vdash [\tau_i/\alpha_i]Q$, giving an entailment of the required form.

Case $(\to E)^s$**:** We have a derivation of the form:

$$\frac{P \,|\, A \overset{s}{\vdash} E : \tau' \to \tau \quad P \,|\, A \overset{s}{\vdash} F : \tau'}{P \,|\, A \overset{s}{\vdash} EF : \tau}$$

Since $x \in FV(EF) = FV(E) \cup FV(F)$, either $x \in FV(E)$ or $x \in FV(F)$. The result follows by induction on either the first or second hypothesis, respectively.

Case $(\to I)^s$**:** We have a derivation of the form:

$$\frac{P \,|\, A_y, y : \tau' \overset{s}{\vdash} E : \tau}{P \,|\, A \overset{s}{\vdash} \lambda y.E : \tau' \to \tau}$$

Since $x \in FV(\lambda y.E)$, we know that $x \not\equiv y$ and that $x \in FV(E)$. Thus $(x : \sigma) \in (A_y, y : \tau')$ and the result follows by induction.

Case $(let)^s$**:** We have a derivation of the form:

$$\frac{P \,|\, A \overset{s}{\vdash} E : \tau \quad P' \,|\, A_y, y : \sigma \overset{s}{\vdash} F : \tau'}{P' \,|\, A \overset{s}{\vdash} (\textbf{let } y = E \textbf{ in } F) : \tau'}$$

where $\sigma = Gen(A, P \Rightarrow \tau)$. By hypothesis, $x \in FV(\textbf{let } y = E \textbf{ in } F) = FV(E) \cup (FV(F) \setminus \{y\})$ so there are two cases to consider:

- If $x \in FV(E)$, then $P \Vdash [\tau_i/\alpha_i]Q$ by induction on the first hypothesis. Note that $y \in FV(F)$ since we consider only terms with no redundant let-bindings. Writing $\sigma = (\forall \beta_j.P \Rightarrow \tau)$, it follows by induction on the second hypothesis that $P' \Vdash [\nu_j/\beta_j]P$ for some types ν_j. Combining these entailments, we obtain $P' \Vdash [\nu_j/\beta_j]([\tau_i/\alpha_i]Q)$.

  Note that the only variables appearing free in Q, other than those in $\{\alpha_i\}$, also appear free in A. However, by definition of σ, none of β_j appears free in A, and hence the entailment above is equivalent to:

  $$P' \Vdash [[\nu_j/\beta_j]\tau_i/\alpha_i]Q$$

  which has the required form.
- If $x \in FV(F)$ and $x \not\equiv y$, then the result follows directly by induction on the second hypothesis.

This completes the proof. □

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:*

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

For convenience, we will write:

$$\begin{array}{rcccl} \sigma & = & Gen(A, P \Rightarrow \tau) & = & \forall \alpha_i . P \Rightarrow \tau \\ \sigma' & = & Gen(A, Q \Rightarrow \nu) & = & \forall \beta_j . Q \Rightarrow \nu \end{array}$$

By hypothesis, P_0 sat σ, and hence there are types τ_i such that $P_0 \Vdash [\tau_i/\alpha_i]P$. By Proposition 3, $P \Vdash [\nu_j/\beta_j]Q$ for some types, ν_j. Combining these entailments gives:

$$P_0 \Vdash [\tau_i/\alpha_i]([\nu_j/\beta_j]Q).$$

However, The only variables that appear free in Q, other than β_j, also appear free in A, but none of α_i appears free in A, so the entailment above is equivalent to:

$$P_0 \Vdash [[\tau_i/\alpha_i]\nu_j/\beta_j]Q.$$

It follows that P_0 sat σ', as required. $\Box$

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:*

$$RT \approx S \quad \textit{and} \quad Gen(A, P \Rightarrow \tau) \leq^{sat}_{P_0} RGen(TA', Q \Rightarrow \nu).$$

The proof of this theorem is quite complicated and will therefore be presented in three parts:

- In the first part, we show that the theorem can be proved using an auxiliary result.
- In the second part, we prove the auxiliary result by structural induction.
- In the third part, we consider the use of the rules (*Simp*) and (*Imp*) for simplification and improvement, respectively.

First part: For the proof the main theorem, we will concentrate on establishing the following auxiliary result:

If $P \,|\, A \overset{s}{\vdash} E : \tau$, $A \leq^{sat}_{P_0} SA'$, and $P_0 \Vdash S'P$ for some S' with $S'A = A$, then the type inference algorithm will not fail, and for every $Q \,|\, TA' \vdash$ $^{\mathrm{W}}$ $E : \nu$, there is a substitution R such that:

$$RT \approx S, \quad P_0 \Vdash RQ, \quad \text{and} \quad S'\tau = R\nu.$$

To see why this is sufficient, note that:

- P_0 sat $Gen(A, P \Rightarrow \tau)$ guarantees the existence of at least one substitution S' such that $P_0 \Vdash S'P$ and $S'A = A$. This, in turn, guarantees that $Q \,|\, TA \overset{\mathrm{W}}{\vdash} E : \nu$ for some Q, T and ν, independent of S'.
- Given the existence of a derivation for the type inference algorithm, the auxiliary result above provides exactly the hypotheses needed to apply Lemma 6 below, and hence to prove the existence of a substitution R such that:

$$RT \approx S \quad \text{and} \quad Gen(A, P \Rightarrow \tau) \leq^{sat}_{P_0} RGen(TA', Q \Rightarrow \nu).$$

Second part: We prove the auxiliary result by induction on the structure of $P \,|\, A \overset{s}{\vdash} E : \tau$.

Case $(var)^s$**:** We have a derivation of the form:

$$\frac{(x : \sigma) \in A \quad (P \Rightarrow \tau) \leq \sigma}{P \,|\, A \overset{s}{\vdash} x : \tau}$$

where $\sigma = A(x) = (\forall \alpha_i . Q \Rightarrow \nu)$. Let $\sigma' = A'(x) = (\forall \beta_j . Q' \Rightarrow \nu')$ and note that, by hypothesis, $\sigma \leq^{sat}_{P_0} S\sigma'$.

Since $S'A = A$ and $P_0 \Vdash S'P$, it follows that:

$$S'\tau \in [\![Gen(A, P \Rightarrow \tau)]\!]^{sat}_{P_0} \subseteq [\![\sigma]\!]^{sat}_{P_0} \subseteq [\![S\sigma']\!]^{sat}_{P_0}.$$

As an aside, the first inclusion follows from the fact that, if $\rho = (P \Rightarrow \tau) \leq \sigma$, then any variable appearing free in ρ, but not in A, does not appear free in σ, and hence $Gen(A, \rho) \leq \sigma$. The inclusion follows from the fact that $\leq^{sat}_{P_0}$ is weaker than $\leq$ and from the definition of $\leq^{sat}_{P_0}$.

Choosing new variables γ_j, we have $S\sigma' = \forall \gamma_j . S[\gamma_j/\beta_j](Q' \Rightarrow \nu')$. Hence, by $(var)^{\mathrm{W}}$, there is a derivation:

$$[\gamma_j/\beta_j]Q' \,|\, A' \overset{\mathrm{W}}{\vdash} x : [\gamma_j/\beta_j]\nu'.$$

Since $S'\tau \in [\![S\sigma']\!]^{sat}_{P_0}$, there are types ν_j such that:

$$S'\tau = [\nu_j/\gamma_j](S[\gamma_j/\beta_j]\nu') = S[\nu_j/\beta_j]\nu' = S[\nu_j/\gamma_j]([\gamma_j/\beta_j]\nu')$$

and:

$$P_0 \Vdash [\nu_j/\gamma_j](S[\gamma_j/\beta_j]Q') = S[\nu_j/\beta_j]Q' = S[\nu_j/\gamma_j]([\gamma_j/\beta_j]Q').$$

Let $R = S[\nu_j/\gamma_j]$. Then $R \approx S$, $S'\tau = R([\gamma_j/\beta_j]\nu')$, and $P_0 \Vdash R([\gamma_j/\beta_j]Q')$.

Case $(\to E)^s$: We have a derivation of the form:

$$\frac{P \,|\, A \overset{s}{\vdash} E : \tau' \to \tau \quad P \,|\, A \overset{s}{\vdash} F : \tau'}{P \,|\, A \overset{s}{\vdash} EF : \tau}$$

By induction, $Q \mid TA' \overset{W}{\vdash} E : \nu$ and there is a substitution R such that $RT \approx S$, $P_0 \Vdash RQ$, and $S'(\tau' \to \tau) = R\nu$.

In a similar way, since $A \leq^{sat}_{P_0} SA' = R(TA')$, it follows by induction on the second hypothesis that $Q' \mid T'TA \overset{W}{\vdash} F : \nu'$ and there is a substitution R' such that $R'T' \approx R$, $P_0 \Vdash R'Q'$, and $S'\tau' = R'\nu'$.

Pick a new variable α and let $R'' = R'[S'\tau/\alpha]$. Note that:

$$\begin{array}{rcl} R''(T'\nu) & = & R'T'\nu \\ & = & R\nu \\ & = & S'(\tau' \to \tau) \\ & = & S'\tau' \to S'\tau \\ & = & R'\nu' \to S'\tau \\ & = & R''(\nu' \to \alpha) \end{array}$$

and hence R'' is a unifier of $T'\nu$ and $\nu' \to \alpha$. It follows that $T'\nu \overset{U}{\sim} (\nu' \to \alpha)$ for some most general unifier U such that $R'' = U'U$ for some U'.

By $(\to E)^W$, there is a derivation:

$$U(T'Q, Q') \,|\, UT'TA' \overset{W}{\vdash} EF : U\alpha.$$

Note that $S \approx RT \approx R'T'T \approx R''T'T \approx U'UT'T$, $U'(U\alpha) = R''\alpha = S'\tau$ and:

$$P_0 \Vdash (RQ, R'Q') = (R'T'Q, R'Q') = R''(T'Q, Q') = U'(U(T'Q', Q')).$$

Case $(\to I)^s$: We have a derivation of the form:

$$\frac{P \,|\, A_x, x : \tau' \overset{s}{\vdash} E : \tau}{P \,|\, A \overset{s}{\vdash} \lambda x.E : \tau' \to \tau}$$

Let α be a new variable and set $S'' = S[S'\tau'/\alpha]$ so that:

$$(A_x, x:\tau') \leq^{sat}_{P_0} S''(A'_x, x:\alpha).$$

By induction, $Q \mid T(A'_x, x:\alpha) \vdash^{W} E : \nu$ for some Q, T and ν and there is a substitution R such that $S'' \approx RT$, $P_0 \Vdash RQ$ and $S'\tau = R\nu$.

By $(\to I)^{W}$, we have $Q \mid TA' \vdash^{W} \lambda x.E : T\alpha \to \nu$. We already know that $P_0 \Vdash RQ$, and $S \approx S'' \approx RT$. To complete the proof for this case, note that:

$$R(T\alpha \to \nu) = (RT\alpha \to R\nu) = (S''\alpha \to S'\tau) = S'(\tau' \to \tau).$$

Case $(let)^{s}$**:** We have a derivation of the form:

$$\frac{P \mid A \vdash^{s} E : \tau \quad P' \mid A_x, x:\sigma \vdash^{s} F : \tau' \quad \sigma = Gen(A, P \Rightarrow \tau)}{P' \mid A \vdash^{s} (\textbf{let}\ x = E\ \textbf{in}\ F) : \tau'}$$

Since we ignore terms containing redundant let-bindings, we can assume that $x \in FV(F)$. By hypothesis, P_0 sat $Gen(A, P' \Rightarrow \tau')$ using the substitution S', and hence P_0 sat σ by Corollary 4.

By induction, using the original statement of the theorem in the form given in the box at the start of the proof, there is a derivation $Q \mid TA' \vdash^{W} E : \nu$ and a substitution R such that:

$$RT \approx S \quad \text{and} \quad \sigma \leq^{sat}_{P_0} RGen(TA', Q \Rightarrow \nu).$$

This use of induction can be justified by repeating the the argument in the first part above.

Writing $\sigma' = RGen(TA', Q \Rightarrow \nu)$, we know that

$$\sigma \leq^{sat}_{P_0} R\sigma' \quad \text{and} \quad A_x \leq^{sat}_{P_0} SA'_x = R(TA'_x),$$

and hence $(A_x, x:\sigma) \leq^{sat}_{P_0} R(TA'_x, x:\sigma')$. By induction, there is a derivation $Q' \mid T'(TA'_x, x:\sigma') \vdash^{W} F : \nu'$ and there is substitution R' such that $R'T' \approx R$, $P_0 \Vdash R'Q'$, and $S'\tau' = R'\nu'$.

By $(let)^{W}$ there is a derivation:

$$Q' \mid T'TA' \vdash^{W} (\textbf{let}\ x = E\ \textbf{in}\ F) : \nu'$$

and $R'T'T \approx RT \approx S$, $P_0 \Vdash R'Q'$, and $S'\tau' = R'\nu'$.

Third part: The only type inference rules used in the proof of the second part above are $(var)^{\mathrm{W}}$, $(\to E)^{\mathrm{W}}$, $(\to I)^{\mathrm{W}}$ and $(let)^{\mathrm{W}}$. For the extended type inference algorithm we need to allow for the use of the rules (*Simp*) and (*Imp*) for simplification and improvement, respectively, at arbitrary points in the derivation. It is sufficient to show that each of the these rules preserves the conclusions of the auxiliary result in the second part. More precisely, suppose that $Q \mid TA' \overset{\mathrm{W}}{\vdash} E : \nu$ and that there is a substitution R such that:

$$RT \approx S, \quad P_0 \Vdash RQ, \quad \text{and} \quad S'\tau = R\nu.$$

As an aside, note that, for any simplifying function *simp* and any improving function *impr*, neither *simp* Q or *impr* Q can fail since $\lfloor Q \rfloor_{P_0} \neq \emptyset$. (See Section 6.)

There are two cases to consider:

Case (*Simp*): We have a type inference derivation of the form:

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

From $P \Leftrightarrow Q$, it follows that $Q \Vdash P$, and hence $P_0 \Vdash RP$ as required.

Case (*Imp*): We have a type inference derivation of the form:

$$\frac{Q \mid TA \overset{\mathrm{W}}{\vdash} E : \nu \quad T' \ improves\ Q}{T'Q \mid T'TA \overset{\mathrm{W}}{\vdash} E : T'\nu}$$

Since $P_0 \Vdash RQ$ and T' *improves* Q, it follows that $RQ \in \lfloor Q \rfloor_{P_0} = \lfloor T'Q \rfloor_{P_0}$ and hence $RQ = R'T'Q$ for some substitution R'. Let V be the set of new variables involved in T'; the only other variables involved in T' are members of $TV(Q)$. Now consider the substitution R'' using:

$$\begin{array}{rcll} R''\alpha & = & R'\alpha, & \textbf{if } \alpha \in TV(Q) \cup V \\ & = & R\alpha, & \textbf{otherwise} \end{array}$$

We claim that $R''T' \approx R$. To see this, note that:

- If $\alpha \in TV(Q)$, then $R''(T'\alpha) = R'(T'\alpha) = R\alpha$.
- If $\alpha \notin TV(Q) \cup V$, then $R''(T'\alpha) = R''\alpha = R\alpha$.

It follows that $R''T'T \approx RT \approx S$, $P_0 \Vdash RQ = R''(T'Q)$, and $S'\tau = R\nu = R''(T'\nu)$ as required.

This completes the proof. $\square$

Lemma 6 *Suppose that:*

$$\forall S'.(P_0 \Vdash S'P \wedge S'A = A) \Rightarrow \exists R.(RT \approx S \wedge P_0 \Vdash RQ \wedge S'\tau = R\nu).$$

Then there is a substitution R such that $RT \approx S$ and:

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

For convenience, we define:

$$\begin{aligned}\sigma &= Gen(A, P \Rightarrow \tau) = \forall\alpha_i.P \Rightarrow \tau\\ \sigma' &= Gen(TA', Q \Rightarrow \nu) = \forall\beta_j.Q \Rightarrow \nu.\end{aligned}$$

Our goal is to find a substitution R such that $\sigma \leq^{sat}_{P_0} R\sigma'$.

Suppose that $[\tau_i/\alpha_i]\tau \in [\![\sigma]\!]^{sat}_{P_0}$. Hence $P_0 \Vdash [\tau_i/\alpha_i]P$. Furthermore, since none of α_i appears free in A, we have $[\tau_i/\alpha_i]A = A$. By hypothesis, there is a substitution R such that:

$$RT \approx s, \quad P_0 \Vdash RQ, \quad \text{and} \quad [\tau_i/\alpha_i]\tau = R\nu.$$

Pick new variables γ_j, so that $R\sigma' = \forall\gamma_j.R[\gamma_j/\beta_j](Q \Rightarrow \nu)$. Let $\nu_j = R\beta_j$, and note that $[\nu_j/\gamma_j] \cdot R[\gamma_j/\beta_j] = R[R\beta_j/\beta_j] = R$. Thus:

$$P_0 \Vdash [\nu_j/\gamma_j](R[\gamma_j/\beta_j]Q) \quad \text{and} \quad [\tau_i/\alpha_i]\tau = [\nu_j/\gamma_j](R[\gamma_j/\beta_j]\nu).$$

It follows that $[\tau_i/\alpha_i]\tau \in [\![R\sigma']\!]^{sat}_{P_0}$.

Thus far, it appears that the choice of R may depend of the initial choice of types τ_i for the representative element, $[\tau_i/\alpha_i]\tau \in [\![\sigma]\!]^{sat}_{P_0}$. Now suppose that we pick another element, $[\tau'_i/\alpha_i]\tau \in [\![R'\sigma']\!]^{sat}_{P_0}$, where $R'T \approx S \approx RT$. The only free variables in σ' are also free in TA'. But $R(TA') = SA' = R'(TA')$, so R and R' agree on the free variables of σ'. Thus $R\sigma' = R'\sigma'$, and:

$$[\![\sigma]\!]^{sat}_{P_0} \subseteq [\![R\sigma']\!]^{sat}_{P_0}$$

for some fixed substitution R such that $RT \approx S$. $\Box$

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] Kung Chen. *A parametric extension of Haskell's type classes.* PhD thesis, Yale University, Department of Computer Science, 1994 (forthcoming).

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] Mark P. Jones. *Introduction to Gofer 2.20*, September 1991. Available by anonymous ftp from `nebula.cs.yale.edu` in the directory `pub/haskell/gofer` as part of the standard Gofer distribution.

[10] Mark P. Jones. Type inference for qualified types. Technical Report PRG-TR-10-91, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, May 1991. Largely superceded by [12].

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

[12] Mark P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. To be published by Cambridge University Press.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.