A theory of qualified types*

Mark P. Jones†
Programming Research Group
Oxford University Computing Laboratory
8–11 Keble Road
Oxford OX1 3QD

Introduction

In a language with a polymorphic type system, a term of type $\forall t.f(t)$ can be treated (possibly after suitable instantiation) as having any of the types in the set:

$$\{\, f(t) \mid t \text{ is a type} \,\}.$$

It is natural to consider a more restricted form of polymorphism in which the value taken by t may be constrained to a particular subset of types. In this situation, we write $\forall t.\pi(t) \Rightarrow f(t)$, where $\pi(t)$ is a predicate on types, for the type of an object that can be treated (after suitable instantiation) as having any of the types in the set:

$$\{\, f(t) \mid t \text{ is a type such that } \pi(t) \text{ holds} \,\}.$$

A term with a restricted polymorphic type of this kind is often said to be *overloaded*, having different interpretations for different argument types.

This paper presents a general theory of overloading based on the use of *qualified types*, which are types of the form $\pi \Rightarrow \sigma$ denoting those instances of type σ that satisfy the predicate π. The main benefits of using qualified types are:

- A general approach that includes a range of familiar type systems as special cases. Results and tools developed for the general system are immediately applicable to each particular application.
- A precise treatment of the relationship between implicit and explicit overloading. This is particularly useful for describing the implementation of systems supporting qualified types.
- The ability to include local constraints as part of the type of an object. This enables the definition and use of polymorphic overloaded values within a program.

*An earlier version of this paper was presented at the European Symposium on Programming, Rennes, France, February 1992 and appears in the conference proceedings published in Springer Verlag LNCS Volume 582.

†Address from September 1992: Yale University, Department of Computer Science, P.O. Box 2158 Yale Station, New Haven, Connecticut 06520-2158, USA.

Outline of paper

Each of the type systems considered in this paper is parameterised by the choice of a system of predicates on type expressions, whose basic properties are described in Section 1. A number of examples are included to illustrate the use of this framework to describe a range of type systems including Haskell type classes, extensible records and subtyping. Section 2 describes the use of qualified types in the context of polymorphic λ-calculus with explicit typing. This is extended in Section 3 using a general notion of *evidence* to explore the relationship between implicit and explicit overloading. An alternative approach, suitable for use in an implicitly typed language, is introduced in Section 4 using an extension of the ML type system [Mil78] to support qualified types. Although substantially less powerful than polymorphic λ-calculus, we show that the resulting system is suitable for use in a language based on type inference, that allows the type of a term to be determined without explicit type annotations. The development of a suitable type inference algorithm is described in Sections 5 and 6. Finally, Section 7 surveys some areas for further work.

Detailed proofs for many of the results described in this paper may be found in [Jon91b, Jon92]; For reasons of space, they cannot be included here.

1 Predicates

Each of the type systems considered in this paper is parameterised by the choice of a language of *predicates* π whose properties are described by an entailment relation $\Vdash$ between (finite) sets of predicates. Individual predicates may be written using expressions of the form $\pi = p\ \tau_1\ \ldots\ \tau_n$ where p is a predicate symbol corresponding to an n-place relation between types; the predicate π represents the assertion that the types $\tau_1, \ldots, \tau_n$ are in this relation. The definition of $\Vdash$ varies from one application to another. The only properties that we will assume are:

- **monotonicity**. $P \Vdash P'$ whenever $P \supseteq P'$.
- **transitivity**. if $P \Vdash Q$ and $Q \Vdash R$, then $P \Vdash R$.
- **closure property**. if $P \Vdash Q$, then $SP \Vdash SQ$ for any substitution S mapping type variables (and hence type expressions) to type expressions.

If P is a set of predicates and π is a predicate, then we write $P \Vdash \pi$ and P, π as abbreviations for $P \Vdash \{\pi\}$ and $P \cup \{\pi\}$ respectively.

The following subsections illustrate the languages of predicates used in three applications of qualified types. Only the basic ideas are sketched here; further details are given in [Jon91a, Jon92].

1.1 Example: type classes

Introduced in [WB89] and adopted as part of the standard for the programming language Haskell [HPJW92], type classes are particularly useful for describing the implementation of standard polymorphic operators such as computable equality. Much of the original motivation for qualified types came from the study of type classes.

Broadly speaking, a *type class* is a family of types (the *instances* of the class) on which a number of values (the *member functions*) are defined. Each predicate symbol corresponds to a user-defined class and a predicate of the form $C\ \tau$ represents the assertion that τ is an instance of the class named C. The class Eq is a standard example whose instances are those types whose elements can be tested for equality using the operator $(==) :: \forall a.Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$. As a further example, one possible type for a function to test for membership of a value in a list is $\forall a.Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$ where $[a]$ denotes the type of lists of values of type a.

Differences in the basic approach to type classes are reflected in the properties of the $\Vdash$ relation. In a standard Haskell system we have axioms such as $\emptyset \Vdash Eq\ Int$ and $Eq\ a \Vdash Eq\ [a]$. The same framework can also be used to describe the use of Haskell *superclasses*, and to support the extension to classes with multiple parameters.

Type classes are best suited to systems with a type inference algorithm such as that described in Section 6 where the appropriate instances of each overloaded operator can be determined automatically as part of the type inference process.

1.2 Example: extensible records

A record is a set of values labelled by the elements l of a specified set of *labels*. There has been considerable interest in the use of record types to model inheritance in object oriented programming languages and a number of different approaches have been considered. We can construct a system of extensible records, strongly reminiscent of [HP90] using predicates of the form:

$r\ \mathsf{has}\ l\!:\!t$	indicating that a record of type r has a field labelled l of type t.
$r\ \mathsf{lacks}\ l$	indicating that a record of type r does not have a field labelled l.

This also requires an extension of the language of type expressions to allow types of the form $\langle\rangle$ (the empty record, which lacks any fields), $r \setminus l$ (the type of a record obtained by removing a field labelled l from a record of type r) and $\langle r \mid l\!:\!t\rangle$ (the type of a record obtained by extending a record of type r with a new field of type t labelled l). The definition of the entailment relation includes axioms such as $\emptyset \Vdash (\langle\rangle\ \mathsf{lacks}\ l$ and $r\ \mathsf{lacks}\ l \Vdash \langle r \mid l\!:\!t\rangle\ \mathsf{has}\ l\!:\!t$.

The primitive operations of record *restriction*, *extension* and *selection* can then be represented by families of functions (indexed by labels) of type:

$$\begin{array}{lll}
(_ \setminus l) & :: & \forall r.\forall t.(r\ \mathsf{has}\ l\!:\!t) \Rightarrow r \rightarrow r \setminus l \\
(_ \mid l = _) & :: & \forall r.\forall t.(r\ \mathsf{lacks}\ l) \Rightarrow r \rightarrow t \rightarrow \langle r \mid l\!:\!t\rangle \\
(_ .\, l) & :: & \forall r.\forall t.(r\ \mathsf{has}\ l\!:\!t) \Rightarrow r \rightarrow t
\end{array}$$

Details of the relationship between this approach and those of [Rem89, CM90] are given in [HP90].

1.3 Example: subtyping

Languages with subtyping can be described using predicates of the form $\sigma \subseteq \sigma'$, representing the assertion that σ is a subtype of σ'. Many such systems, including those of [Mit84, FM89], allow the use of implicit coercions from one type to another. The extensions required to support this are discussed in Section 7.4.

2 Polymorphic λ-calculus with qualified types

2.1 Basic definitions

In this section, we work with a variant of the polymorphic λ-calculus that includes qualified types using type expressions of the form:

$$\sigma ::= t \mid \sigma \to \sigma \mid \forall t.\sigma \mid \pi \Rightarrow \sigma$$

where t ranges over a given set of type variables. The $\to$ and $\Rightarrow$ symbols are treated as right associative infix binary operators with $\to$ binding more tightly than $\Rightarrow$. Additional type constructors such as those for integers, lists and record types will be used as required. The set of type variables appearing (free) in an expression X is denoted $TV(X)$.

To begin with we use an unmodified form of the (unchecked) terms of polymorphic λ-calculus, given by expressions of the form:

$$M ::= x \mid MN \mid \lambda x\!:\!\sigma.M \mid M\sigma \mid \lambda t.M$$

where x ranges over a given set of term variables. The set of free (term) variables appearing in a term M will be denoted $FV(M)$. Note that we do not provide constructs for the introduction of new overloadings such as **inst** and **over** in [WB89]. If none of the free variables for a given term have qualified (i.e. overloaded) types, then no overloading will be used in the expression.

2.2 Typing rules

A *type assignment* is a (finite) set of *typing statements* of the form $x\!:\!\sigma$ in which no term variable x appears more than once. If A is a type assignment, then we write $dom\ A = \{\, x \mid (x\!:\!\sigma) \in A \,\}$, and if x is a term variable with $x \notin dom\ A$, then we write $A, x\!:\!\sigma$ as an abbreviation for the type assignment $A \cup \{x\!:\!\sigma\}$. The type assignment obtained from A by removing any typing statement for the variable x is denoted A_x. A type assignment A can be interpreted as a function mapping each element of $dom\ A$ to a type scheme. In particular, if $(x\!:\!\sigma) \in A$, then we write $A(x) = \sigma$.

An expression of the form $P \mid A \vdash M : \sigma$ represents the assertion that the term M has type σ when the predicates in P are satisfied and the types of free variables in M are as specified in the type assignment A. The typing rules for this system are given in Figure 1. Most of these are similar to the rules for explicit typing of polymorphic λ-calculus and do not involve the predicate set.

By an abuse of notation, we will also use $P \mid A \vdash M : \sigma$ as a proposition asserting the existence of a derivation of $P \mid A \vdash M : \sigma$.

3 Evidence

Although the system of qualified types described in the previous sections is suitable for reasoning about the types of overloaded terms, it cannot be used to describe their evaluation. For example, the knowledge that *Int* is an instance of the class *Eq* is not sufficient to determine the value of the expression $2 == 3$; we must also be provided with the value of the equality operator that makes *Int* an instance of *Eq*. In general, we can only use a term of type $\pi \Rightarrow \sigma$ if we are also supplied with suitable *evidence* that the predicate π does indeed hold.

Standard rules:	(var)	$$\frac{(x:\sigma) \in A}{P \mid A \vdash x : \sigma}$$
	$(\rightarrow E)$	$$\frac{P \mid A \vdash M : \sigma' \rightarrow \sigma \quad P \mid A \vdash N : \sigma'}{P \mid A \vdash MN : \sigma}$$
	$(\rightarrow I)$	$$\frac{P \mid A, x:\sigma' \vdash M : \sigma}{P \mid A \vdash \lambda x:\sigma'.M : \sigma' \rightarrow \sigma}$$
Qualified types:	$(\Rightarrow E)$	$$\frac{P \mid A \vdash M : \pi \Rightarrow \sigma \quad P \Vdash \pi}{P \mid A \vdash M : \sigma}$$
	$(\Rightarrow I)$	$$\frac{P, \pi \mid A \vdash M : \sigma}{P \mid A \vdash M : \pi \Rightarrow \sigma}$$
Polymorphism:	$(\forall E)$	$$\frac{P \mid A \vdash M : \forall t.\sigma}{P \mid A \vdash M\tau : [\tau/t]\sigma}$$
	$(\forall I)$	$$\frac{P \mid A \vdash M : \sigma \quad t \notin TV(A) \cup TV(P)}{P \mid A \vdash \lambda t.M : \forall t.\sigma}$$

Figure 1: Typing rules for polymorphic λ-calculus with qualified types

This leads us to consider an extension of the term language that makes the role of evidence explicit, using:

- **Evidence expressions**: A language of *evidence expressions* e denoting evidence values, including a set of *evidence variables* v.
- **Evidence construction**: A *predicate assignment* is a set of elements of the form $(v:\pi)$ in which no evidence variable appears more than once. The $\Vdash$ relation is extended to a three place relation $P \Vdash e : \pi$, indicating that it is possible to construct evidence e for the predicate π in any environment binding the variables in the predicate assignment P to appropriate evidence values. Thus predicates play a similar role for evidence expressions as types for simple λ-calculus terms.
- **Evidence abstraction**: A term M of type $\pi \Rightarrow \rho$ is implemented by a term of the form $\lambda v:\pi.M'$ where v is an evidence variable and M' is a term of type ρ corresponding to M using v in each place where evidence for π is needed.
- **Evidence application**: Each use of an overloaded expression N of type $\pi \Rightarrow \rho$ is replaced by a term of the form $N'e$ where N' is a term corresponding to N and e is an evidence expression for π.
- **Evidence reduction**: The standard rules of computation are augmented by a variant of β-reduction for evidence abstraction and application:

$$(\lambda v.M)e \ \rhd_{\beta e} [e/v]M.$$

Most of the typing rules given in Figure 1 can be used with the extended system without modification. The only exceptions are the rules for dealing with qualified types; suitably modified versions of these are given in Figure 2.

$$\frac{P, v\!:\!\pi \mid A \vdash M : \sigma}{P \mid A \vdash \lambda v\!:\!\pi.M : \pi \Rightarrow \sigma} \quad (\Rightarrow I)$$

$$\frac{P \mid A \vdash M : \pi \Rightarrow \sigma \quad P \Vdash e\!:\!\pi}{P \mid A \vdash Me : \sigma} \quad (\Rightarrow E)$$

Figure 2: Modified rules for qualified types with evidence

Notice that extending the term language to make the use of evidence explicit gives unicity of type; each well-typed term has a uniquely determined type. This approach is very similar to the techniques used to make polymorphism explicit in the translation from implicit to explicit typed λ-calculus using abstraction and application over types [Mit90]. As in that situation, there is a simple correspondence between derivations in the two systems, described by means of a function *Erase* mapping explicitly overloaded terms to their implicitly overloaded counterparts:

$$\begin{array}{lcl} Erase(x) & = & x \\ Erase(MN) & = & (Erase(M))(Erase(N)) \\ & \vdots & \\ Erase(\lambda v\!:\!\pi.M) & = & Erase(M) \\ Erase(Me) & = & Erase(M) \end{array}$$

The correspondence between the two systems can now be described by:

Theorem 1 *$P \mid A \vdash M : \sigma$ using the original typing rules if and only if $P' \mid A \vdash M' : \sigma$ by a derivation of the same structure in the extended system such that $P = \{\, \pi \mid (v:\pi) \in P' \,\}$ and $Erase(M') = M$.*

Given a term M in the original system, each corresponding term using explicit overloading is called a *translation* of M and can be used to give a semantics for the term. We write $P' \mid A \vdash M \leadsto M' : \sigma$ to refer to the translation of a term in a specific context. Note that the translation of a given term may not be uniquely defined (with distinct translations corresponding to distinct derivations of $P \mid A \vdash M : \sigma$). This is discussed in more detail in Section 7.1.

The form of evidence required will vary from one application to another. Suitable choices for each of the examples described in Section 1 are as follows:

- **Type classes**: The evidence for a type class predicate of the form $C\ \tau$ is a *dictionary* containing the values of the members of C at the instance τ. For example, in the simplest case, the evidence for a predicate $Eq\ \tau$ might be an equality test function for values of type τ.

- **Extensible records**: The evidence for a predicate of the form $(r$ **lacks** $l)$ is the function:

  $$(\_ \mid l = \_) \quad :: \quad \forall t\,.\,r \to t \to (r \mid l\!:\!t)$$

  The evidence for a predicate of the form $(r$ **has** $l\!:\!t)$ is the pair of functions:

  $$\begin{array}{lcl} (\_ \setminus l) & :: & r \to r \setminus l \\ (\_ . l) & :: & r \to t \end{array}$$

  In practice, a concrete implementation of extensible records is likely to use offsets into a table of values used to store a record as evidence, passing these values to generic functions for updating or selecting from a record where necessary.

- **Subtypes**: The evidence for a predicate $\sigma \subseteq \sigma'$ is a coercion function that maps values of type σ to values of type σ'.

Further details of the use of evidence in these applications is included in [Jon92].

4 An extension of ML using qualified types

Polymorphic λ-calculus is not a suitable language to describe an implicitly typed language in which the need for explicit type annotations is replaced by the existence of a type inference algorithm. In practice, the benefits of type inference are often considered to outweigh the disadvantages of a less powerful type system. The ML type system [Mil78, DM82] is a well-known example in which the price of type inference is the inability to define functions with polymorphic arguments. Nevertheless, it has proved to be very useful in practice and has been adopted by a number of later languages.

4.1 Basic definitions

Following the definition of types and type schemes in ML we consider a structured language of types, with the principal restriction being the inability to support functions with either polymorphic or overloaded arguments:

$$\begin{array}{rcll} \tau & ::= & t \mid \tau \rightarrow \tau & \textit{types} \\ \rho & ::= & P \Rightarrow \tau & \textit{qualified types} \\ \sigma & ::= & \forall T.\rho & \textit{type schemes} \end{array}$$

(P and T range over finite sets of predicates and finite sets of type variables respectively).

It is convenient to introduce some abbreviations for qualified type and type scheme expressions. In particular, if $\rho = (P \Rightarrow \tau)$ and $\sigma = \forall T.\rho$, then we write:

Abbreviation	Qualified type	Abbreviation	Type scheme
τ	$\emptyset \Rightarrow \tau$	ρ	$\forall \emptyset.\rho$
$\pi \Rightarrow \rho$	$P, \pi \Rightarrow \tau$	$\forall t.\sigma$	$\forall (T \cup \{t\}).\rho$
$P' \Rightarrow \rho$	$P \cup P' \Rightarrow \tau$	$\forall T'.\sigma$	$\forall (T \cup T').\rho$

In addition, if $\{\alpha_i\}$ is an indexed set of variables, we write $\forall \alpha_i.\rho$ as an abbreviation for $\forall \{\alpha_i\}.\rho$. As usual, type schemes are regarded as equal if they are equivalent upto renaming of bound variables.

Using this notation, any type scheme can be written in the form $\forall \alpha_i.P \Rightarrow \tau$, representing the set of qualified types $\{\, [\tau_i/\alpha_i]P \Rightarrow [\tau_i/\alpha_i]\tau \mid \tau_i \in \mathit{Type} \,\}$ where $[\tau_i/\alpha_i]$ is the substitution mapping each of the variables α_i to the corresponding type τ_i and *Type* is the set of all simple type expressions (represented by τ in the grammar above).

As in [Mil78, DM82, CDK86], we use a term language based on simple untyped λ-calculus with the addition of a *let* construct to enable the definition and use of polymorphic (and in this case, overloaded) terms.

$$M ::= x \mid MN \mid \lambda x.M \mid \textbf{let } x = M \textbf{ in } N$$

A suitable set of typing rules for this system is given in Figure 3. Note the use of the symbols τ, ρ and σ to restrict the application of certain rules to specific sets of type expressions.

Standard rules:	(var)	$$\frac{(x\!:\!\sigma)\in A}{P \mid A \vdash x : \sigma}$$
	$(\to E)$	$$\frac{P \mid A \vdash M : \tau' \to \tau \quad P \mid A \vdash N : \tau'}{P \mid A \vdash MN : \tau}$$
	$(\to I)$	$$\frac{P \mid A_x, x\!:\!\tau' \vdash M : \tau}{P \mid A \vdash \lambda x.M : \tau' \to \tau}$$
Qualified types:	$(\Rightarrow E)$	$$\frac{P \mid A \vdash M : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid A \vdash M : \rho}$$
	$(\Rightarrow I)$	$$\frac{P, \pi \mid A \vdash M : \rho}{P \mid A \vdash M : \pi \Rightarrow \rho}$$
Polymorphism:	$(\forall E)$	$$\frac{P \mid A \vdash M : \forall t.\sigma}{P \mid A \vdash M : [\tau/t]\sigma}$$
	$(\forall I)$	$$\frac{P \mid A \vdash M : \sigma \quad t \notin TV(A) \cup TV(P)}{P \mid A \vdash M : \forall t.\sigma}$$
Local Definition:	(let)	$$\frac{P \mid A \vdash M : \sigma \quad Q \mid A_x, x\!:\!\sigma \vdash N : \tau}{P \cup Q \mid A \vdash (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau}$$

Figure 3: ML-like typing rules for qualified types

4.2 Constrained type schemes

A typing judgement $P \mid A \vdash M : \sigma$ assigns a type scheme σ to a term M, but also constrains uses of this typing to environments satisfying the predicates in P. This observation motivates the use of *constrained type schemes*, written as pairs of the form $(P \mid \sigma)$ where P is a set of predicates and σ is a type scheme. Following the development of type inference in [DM82], we will define an ordering that can be used to describe when one constrained type scheme is more general than another. As a first step, we introduce the concept of generic instances:

Definition 1 *A qualified type $R \Rightarrow \mu$ is said to be a* generic instance *of the constrained type scheme $(P \mid \forall\alpha_i.Q \Rightarrow \tau)$ if there are types τ_i such that $R \Vdash P \cup [\tau_i/\alpha_i]Q$ and $\mu = [\tau_i/\alpha_i]\tau$.*

The principal motivation for the definition of the ordering $(\leq)$ between type schemes is that a statement of the form $\sigma' \leq \sigma$ should mean that it is possible to use an object of type σ wherever an object of type σ' is required.

Definition 2 *The constrained type scheme $(Q \mid \eta)$ is said to be* more general *than a constrained type scheme $(P \mid \sigma)$, written $(P \mid \sigma) \leq (Q \mid \eta)$, if every generic instance of $(Q \mid \eta)$ is also a generic instance of $(P \mid \sigma)$.*

It is straightforward to show that this defines a preorder on the set of constrained type schemes, such that a qualified type ρ is a generic instance of the type scheme σ if and only if $\rho \leq \sigma$. We will write $(P \mid \sigma) \simeq (Q \mid \eta)$ to indicate when two constrained type schemes are equivalent with respect to $(\leq)$, i.e. when each is more general than the other. The following properties are easily established:

- $\sigma \simeq (\emptyset \mid \sigma)$ for any type scheme σ.
- If ρ is a qualified type and P is a set of predicates, then $(P \mid \rho) \simeq P \Rightarrow \rho$.
- If σ is a type scheme and P is a set of predicates, then $(P \mid \sigma) \leq \sigma$.
- If $\sigma' \leq \sigma$ and $P' \Vdash P$, then $(P' \mid \sigma') \leq (P \mid \sigma)$.
- If none of the variables α_i appear in P, then the constrained type scheme $(P \mid \forall \alpha_i.\rho)$ is equivalent to the type scheme $\forall \alpha_i.P \Rightarrow \rho$. Thus every constrained type scheme can be represented by a simple type scheme using a renaming of bound variables.

The application of a substitution S to a constrained type scheme $(P \mid \sigma)$ is defined by $S(P \mid \sigma) = (SP \mid S\sigma)$. The next proposition describes an important property of the ordering on constrained type schemes.

Proposition 1 *For any substitution S and constrained type schemes $(P \mid \sigma)$ and $(Q \mid \eta)$:*

$$(P \mid \sigma) \leq (Q \mid \eta) \quad \Rightarrow \quad S(P \mid \sigma) \leq S(Q \mid \eta).$$

4.3 Ordering of type assignments

The definition of constrained type schemes and the ordering $(\leq)$ extends naturally to an ordering on (constrained) type assignments.

Definition 3 *If A and A' are type assignments and P, P' are sets of predicates, then we say that $(P \mid A)$ is* more general *than $(P' \mid A')$, written $(P' \mid A') \leq (P \mid A)$, if $dom\ A = dom\ A'$ and $(P' \mid A'(x)) \leq (P \mid A(x))$ for each $x \in dom\ A$.*

The results of the previous section can be used to prove that this ordering on type assignments is reflexive, transitive and preserved by substitutions. In this paper, we will only use the special case where $P = \emptyset$ in which case we write $(P' \mid A') \leq A$. This can be interpreted as indicating that each of the types assigned to a variable in A is more general than the type assigned in A' in any environment that satisfies the predicates in P'.

4.4 Generalisation

Given a derivation $P \mid A \vdash M : \tau$, it is useful to have a notation for the most general type scheme that can be obtained for M from this derivation using the rules $(\Rightarrow I)$ and $(\forall I)$ given in Figure 3.

Definition 4 *The* generalisation *of a qualified type ρ with respect to a type assignment A is written $Gen(A, \rho)$ and defined by:*

$$Gen(A, \rho) \quad = \quad \forall (TV(\rho) \setminus TV(A)).\rho.$$

In other words, if $\{\alpha_i\} = TV(\rho) \setminus TV(A)$, then $Gen(A, \rho) = \forall \alpha_i.\rho$. The following propositions describe the interaction of generalisation with predicate entailment and substitution.

Proposition 2 *Suppose that A is a type assignment, P and P' are sets of predicates and τ is a type. Then $Gen(A, P' \Rightarrow \tau) \leq Gen(A, P \Rightarrow \tau)$ whenever $P' \Vdash P$.*

Proposition 3 *If A is a type assignment, ρ is a qualified type and S is a substitution, then:*

$$Gen(SA, S\rho) \leq S(Gen(A, \rho)).$$

Furthermore, there is a substitution R such that:

$$RA = SA \quad \textit{and} \quad SGen(A, \rho) = Gen(RA, R\rho).$$

5 A syntax-directed approach

The typing rules in Figure 3 provide clear descriptions of the treatment of each of the syntactic constructs of the term and type languages. Unfortunately, they are not suitable for use in a type inference algorithm where it should be possible to determine an appropriate order in which to apply the typing rules by a simple analysis of the syntactic structure of the term whose type is required.

In this section, we introduce an alternative set of typing rules with a single rule for each syntactic construct in the term language. We refer to this as the *syntax-directed* system because it has the following important property: *all typing derivations for a given term M (if there are any) have the same structure, uniquely determined by the syntactic structure of M*. We regard the syntax-directed system as a tool for exploring the type system of Section 4 and we establish a congruence between the two systems so that results about one can be translated into results about the other. The advantages of working with the syntax-directed system are:

- The rules are better suited to use in a type inference algorithm; having found types for each of the subterms of a given term M, there is at most one rule that can be used to obtain a type for the term M itself.
- Only type expressions are involved in the matching process. Type schemes and qualified types can only appear in type assignments.
- There are fewer rules and hence fewer cases to be considered in formal proofs.

A similar approach is described in [CDK86] which gives a deterministic set of typing rules for ML and outlines their equivalence to the rules in [DM82].

5.1 Syntax-directed typing rules

The typing rules for the syntax-directed system are given in Figure 4. Typings in this system are written in form $P \mid A \overset{s}{\vdash} M : \tau$, where τ ranges over the set of type expressions rather than the set of type schemes as in the typing judgements of Section 4. Other than this, the principal differences between the two systems are in the rules $(var)^s$ and $(let)^s$ which use the operations of instantiation and generalisation introduced in Sections 4.2 and 4.4.

5.2 Properties of the syntax-directed system

The following proposition illustrates the parametric polymorphism present in the syntax-directed system; instantiating the free type variables in a derivable typing with arbitrary types produces another derivable typing.

Proposition 4 *If $P \mid A \overset{s}{\vdash} M : \tau$ and S is a substitution, then $SP \mid SA \overset{s}{\vdash} M : S\tau$.*

$$
\begin{array}{lcl}
(var)^s & \dfrac{(x\!:\!\sigma) \in A}{P \mid A \overset{s}{\vdash} x : \tau} & (P \Rightarrow \tau) \leq \sigma \\[2ex]
(\to E)^s & \dfrac{P \mid A \overset{s}{\vdash} M : \tau' \to \tau \quad P \mid A \overset{s}{\vdash} N : \tau'}{P \mid A \overset{s}{\vdash} MN : \tau} & \\[2ex]
(\to I)^s & \dfrac{P \mid A_x, x\!:\!\tau' \overset{s}{\vdash} M : \tau}{P \mid A \overset{s}{\vdash} \lambda x.M : \tau' \to \tau} & \\[2ex]
(let)^s & \dfrac{P \mid A \overset{s}{\vdash} M : \tau \quad P' \mid A_x, x\!:\!\sigma \overset{s}{\vdash} N : \tau'}{P' \mid A \overset{s}{\vdash} (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau'} & \sigma = Gen(A, P \Rightarrow \tau)
\end{array}
$$

Figure 4: Syntax-directed inference system

A similar result is established in [Dam85] where it is shown that for any derivation $A \vdash M : \tau$ in the usual (non-deterministic) ML type system and any substitution S, there is a derivation $SA \vdash M : S\tau$ which can be chosen in such a way that the height of the latter is bounded by the height of the former. This additional condition is needed to ensure the validity of proofs by induction on the size of a derivation. This complication is avoided by the syntax-directed system; the derivations in proposition 4 are guaranteed to have the same structure because the term M is common to both.

There is also a form of polymorphism over the sets of environments in which a particular typing can be used, as described by the following proposition:

Proposition 5 *If $P \mid A \overset{s}{\vdash} M : \tau$ and $Q \Vdash P$, then $Q \mid A \overset{s}{\vdash} M : \tau$.*

Recall that an ordering $\sigma' \leq \sigma$ is intended to mean that, at least for the purposes of type inference, it is possible to use an object of type σ whenever with an object of type σ' is required. In much the same way, given two type assignments such that $A' \leq A$ (so that the type assigned to each variable in A is more general than the corresponding type in A'), then we would expect that any typing that can be derived using A' could also be derived from A. The following proposition establishes a slightly more general form of this result:

Proposition 6 *If $P \mid A' \overset{s}{\vdash} M : \tau$ and $(P \mid A') \leq A$, then $P \mid A \overset{s}{\vdash} M : \tau$.*

The hypothesis $(P \mid A') \leq A$ means that the types assigned to variables in A are more general than those given by A' in any environment that satisfies the predicates in P. For example:

$$(Eq\ Int \mid \{(==)\!:\!Int \to Int \to Bool\}) \quad \leq \quad \{(==)\!:\!\forall a.Eq\ a \Rightarrow a \to a \to Bool\}$$

and hence, by the proposition above, it is possible to replace an integer equality function with a generic equality function of type $\forall a.Eq\ a \Rightarrow a \to a \to Bool$ in any environment that satisfies $Eq\ Int$.

5.3 Relationship with original type system

In order to use the syntax-directed system as a tool for reasoning about the type system described in Section 4, we need to investigate the way in which the existence of a derivation in one system determines the existence of derivations in the other.

Our first result establishes the soundness of the syntax-directed system with respect to the original typing rules, showing that any derivable typing in the former system is also derivable in the latter.

Theorem 2 *If $P \mid A \overset{s}{\vdash} M : \tau$, then $P \mid A \vdash M : \tau$.*

The translation of derivations in the original type system to those of the syntax-directed system is less obvious. For example, if $P \mid A \vdash M : \sigma$, then it will not in general be possible to derive the same typing in the syntax-directed system because σ is a type scheme, not a simple type. However, for any derivation $P' \mid A \overset{s}{\vdash} M : \tau$, theorem 2 guarantees the existence of a derivation $P' \mid A \vdash M : \tau$ and hence $\emptyset \mid A \vdash M : Gen(A, P' \Rightarrow \tau')$ by definition 4. The following theorem shows that it is always possible to find a derivation in this way such that the inferred type scheme $Gen(A, P' \Rightarrow \tau')$ is more general than the constrained type scheme $(P \mid \sigma)$ determined by the original derivation.

Theorem 3 *If $P \mid A \vdash M : \sigma$, then $P' \mid A \overset{s}{\vdash} M : \tau$ for some set of predicates P' and type τ such that $(P \mid \sigma) \leq Gen(A, P' \Rightarrow \tau)$.*

6 Type inference

In this section, we give an algorithm for calculating a typing for a given term, using an extension of Milner's algorithm W [Mil78] to support qualified types. We show that the typings produced by this algorithm are derivable in the syntax-directed system and that they are, in a certain sense, the most general typings possible. Combining this with the results of the previous section, the algorithm can be used to reason about the type system in Section 4.

6.1 Unification

This section describes the unification algorithm which is a central component of the type inference algorithm. A substitution S is called a *unifier* for the type expressions τ and τ' if $S\tau = S\tau'$. The following theorem is due to Robinson [Rob65].

Theorem 4 (Unification algorithm) *There is an algorithm whose input is a pair of type expressions τ and τ' such that either:*

> *the algorithm succeeds with a substitution U as its result and the unifiers of τ and τ' are precisely those substitutions of the form RU for any substitution R. The substitution U is called a* most general unifier *for τ and τ', and is denoted $mgu(\tau, \tau')$.*

or the algorithm fails and there are no unifiers for τ and τ'.

In the following, we write $\tau \overset{U}{\sim} \tau'$ for the assertion that the unification algorithm succeeds by finding a most general unifier U for τ and τ'.

6.2 A type inference algorithm

Following the presentation of [Rem89], we describe the type inference algorithm using the inference rules in Figure 5. These rules use typings of the form $P \mid TA \overset{W}{\vdash} M : \tau$ where P is a set of predicates, T is a substitution, A is a type assignment, M is a term and τ is a simple type expression. The typing rules can be interpreted as an attribute grammar in which A are M

$(var)^{\mathrm{W}}$	$\dfrac{(x:\forall\alpha_i.P\Rightarrow\tau)\in A}{[\beta_i/\alpha_i]P \mid A \overset{\mathrm{W}}{\vdash} x : [\beta_i/\alpha_i]\tau}$		β_i new
$(\to E)^{\mathrm{W}}$	$\dfrac{P \mid TA \overset{\mathrm{W}}{\vdash} M : \tau \quad Q \mid T'TA \overset{\mathrm{W}}{\vdash} N : \tau' \quad T'\tau \overset{U}{\sim} \tau' \to \alpha}{U(T'P\cup Q) \mid UT'TA \overset{\mathrm{W}}{\vdash} MN : U\alpha}$		α new
$(\to I)^{\mathrm{W}}$	$\dfrac{P \mid T(A_x, x:\alpha) \overset{\mathrm{W}}{\vdash} M : \tau}{P \mid TA \overset{\mathrm{W}}{\vdash} \lambda x.M : T\alpha \to \tau}$		α new
$(let)^{\mathrm{W}}$	$\dfrac{P \mid TA \overset{\mathrm{W}}{\vdash} M : \tau \quad P' \mid T'(TA_x, x:\sigma) \overset{\mathrm{W}}{\vdash} N : \tau'}{P' \mid T'TA \overset{\mathrm{W}}{\vdash} (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau'}$		$\sigma = Gen(TA, P \Rightarrow \tau)$

Figure 5: Type inference algorithm W

inherited attributes, while P, T and τ are synthesised. One of the advantages of this choice of notation is that it highlights the relationship between W and the syntax-directed system. This point is illustrated by the following theorem.

Theorem 5 *If $P \mid TA \overset{\mathrm{W}}{\vdash} M : \tau$, then $P \mid TA \overset{s}{\vdash} M : \tau$.*

Combining this with the result of theorem 2 gives the following important corollary.

Corollary 1 (Soundness of W) *If $P \mid TA \overset{\mathrm{W}}{\vdash} M : \tau$, then $P \mid TA \vdash M : \tau$.*

With the exception of $(let)^{\mathrm{W}}$, each of the rules in Figure 5 introduces 'new' variables; i.e. variables that do not appear in the hypotheses of the rule nor in any other distinct branches of the complete derivation. Note that it is always possible to choose type variables in this way because the set of type variables is assumed to be countably infinite. In the presence of new variables, it is convenient to work with a weaker form of equality on substitutions, writing $S \approx R$ to indicate that $St = Rt$ for all but a finite number of new variables t. In most cases, we can treat $S \approx R$ as $S = R$, since the only differences between the substitutions occur at variables that are not used elsewhere in the algorithm.

This notation enables us to give an accurate statement of the following result which shows that the typings obtained by W are, in a precise sense, the most general derivable typings for a given term.

Theorem 6 *Suppose that $P \mid SA \overset{s}{\vdash} M : \tau$. Then $Q \mid TA \overset{\mathrm{W}}{\vdash} M : \nu$ and there is a substitution R such that $S \approx RT$, $\tau = R\nu$ and $P \Vdash RQ$.*

Combining the result of theorem 6 with that of theorem 3 we obtain a similar completeness result for W with respect to the type system of Section 4.

Corollary 2 (Completeness of W) *Suppose that $P \mid SA \vdash M : \sigma$. Then $Q \mid TA \overset{\mathrm{W}}{\vdash} M : \nu$ and there is a substitution R such that $S \approx RT$ and $(P \mid \sigma) \leq RGen(TA, Q \Rightarrow \nu)$.*

6.3 Principal type schemes

A term M is *well-typed* under a type assignment A if $P \mid A \vdash M : \sigma$ for some P and σ. It is natural to try to characterise the set of constrained type schemes $(P \mid \sigma)$ for which such a derivation can be found. This can be described using the concept of a *principal type scheme*:

Definition 5 *A* principal type scheme *for a term M under a type assignment A is a constrained type scheme $(P \mid \sigma)$ such that $P \mid A \vdash M : \sigma$, and $(P' \mid \sigma') \leq (P \mid \sigma)$ whenever $P' \mid A \vdash M : \sigma'$.*

The following result gives a sufficient condition for the existence of principal type schemes, by showing how they can be constructed from typings produced by W.

Corollary 3 *Suppose that M is a term, A is a type assignment and $Q \mid TA \stackrel{W}{\vdash} M : \nu$ for some Q, T and ν. Then $Gen(TA, Q \Rightarrow \nu)$ is a principal type scheme for M under TA.*

Combining this with corollary 2 gives a necessary condition for the existence of principal type schemes: a term is well-typed if and only if it has a principal type scheme. Furthermore, if it exists, a suitable principal type can be calculated using the type inference algorithm W.

Corollary 4 (Principal type theorem) *Let M be a term and A an arbitrary type assignment. The following conditions are equivalent:*

- *M is well-typed under A.*
- *$Q \mid TA \stackrel{W}{\vdash} M : \nu$ for some Q and ν and there is a substitution R such that $RTA = A$.*
- *M has a principal typing under A.*

7 Extensions and topics for further work

7.1 The coherence problem

It is important to point out that the type systems described by the rules in the previous sections are not *coherent* (in the sense of [BCGS89]). In other words, it is possible to construct translations $P \mid A \vdash M \leadsto M_1' : \sigma$ and $P \mid A \vdash M \leadsto M_2' : \sigma$ in which the terms M_1' and M_2' are not equivalent, and hence the semantics of M are not well-defined.

For an example in which the coherence problem arises, consider the term $out\ (in\ x)$ under the predicate assignment $P = \{u : C\ Int, v : C\ Bool\}$ and the type assignment:

$$A = \{x : Int,\ in : \forall a.C\ a \Rightarrow Int \rightarrow a,\ out : \forall a.C\ a \Rightarrow a \rightarrow Int\}$$

for some unary predicate symbol C. Instantiating the quantified type variable in the type of in (and hence also in that of out) with the types Int and $Bool$ leads to distinct derivations $P \mid A \vdash out\ (in\ x) : Int$ in which the corresponding translations, $out\ u\ (in\ u\ x)$ and $out\ v\ (in\ v\ x)$ are clearly not equal.

Note that the principal type scheme of $out\ (in\ x)$ in this example is $\forall a.C\ a \Rightarrow Int$ and that the type variable a (the source of the lack of coherence in the derivations above) appears only in the predicate qualifying the type of the term, not in the type itself. Motivated by the functional programming language Haskell [HPJW92], we say that a type of the form $\forall \alpha_i.P \Rightarrow \tau$ is *unambiguous* if $\{\alpha_i\} \cap TV(P) \subseteq TV(\tau)$. Extending the results of this paper to describe the use of translations in the syntax-directed system and the type inference algorithm, we have established the following coherence result:

Theorem 7 *If $P \mid A \vdash M \leadsto M_1' : \sigma$ and $P \mid A \vdash M \leadsto M_2' : \sigma$ and the principal type scheme of M in A is unambiguous, then the translations M_1' and M_2' are equivalent.*

This generalises an earlier result by Blott [Blo90] for the special case of [WB89]. Full details are included in [Jon92] and we expect to describe this work more fully in a forthcoming paper.

7.2 Eliminating evidence parameters

Using translations as described in Section 3, a term M of type $\forall\alpha_i.P \Rightarrow \tau$ will be implemented by a term of the form $\lambda v_1.\ldots\lambda v_n.M'$, where $P = \{\pi_1, \ldots, \pi_n\}$ and each v_i is an evidence variable for the corresponding predicate π_i. The following subsections outline a number of situations in which it is useful to reduce or even eliminate the use of evidence parameters, either to obtain a more efficient implementation or to avoid unnecessary repeated calculations.

7.2.1 Simplification

The translation of a term whose type is qualified by a set of predicates P requires one evidence abstraction for each element of P. Thus the number of evidence parameters that are required can be reduced by finding a smaller set of predicates Q, equivalent to P in the sense that $P \Vdash Q$ and $Q \Vdash P$ (and hence the type of the new term is equivalent to that of the original term). In this situation, we have a compromise between reducing the number of evidence parameters required and the cost of constructing evidence for P from evidence for Q. The process of simplification can be formalised by allowing the rule:

$$\frac{P \mid A \overset{\text{W}}{\vdash} M : \nu \quad P \Vdash Q \quad Q \Vdash P}{Q \mid A \overset{\text{W}}{\vdash} M : \nu}$$

to be used at any stage during type inference to simplify the inferred predicate set. It is relatively straightforward to show that this rule is sound and that the extended algorithm still calculates principal (but potentially simplified) type schemes.

In general, the task of finding an optimal set of predicates with which to replace P is likely to be intractable. One potentially useful approach would be to determine a minimal subset $Q \subseteq P$ such that $Q \Vdash P$. To see that this is likely to be a good choice, note that:

- $P \Vdash Q$, by monotonicity of $\Vdash$ and hence Q is equivalent to P as required.
- Since $Q \subseteq P$, the number of evidence abstractions required using Q is less than or equal to the number required when using P.
- The construction of evidence for a predicate in P using evidence for Q is trivial for each predicate that is already in Q.

7.2.2 Evidence parameters considered harmful

The principal motivation for including the **let** construct in the term language was to enable the definition and use of polymorphic and overloaded values. In practice, the same construct is also used for a number of other purposes:

- To avoid repeated evaluation of a value that is used at a number of points in an expression.
- To create *cyclic data structures* using recursive bindings [BW89].
- To enable the use of identifiers as abbreviations for the subexpressions of a large expression.

Note however that the addition of evidence parameters to the value defined in a **let** expression may mean that the evaluation of an overloaded term will not behave as intended. For example, if $f : \forall a.C\ a \Rightarrow \mathit{Int} \rightarrow a$, then we have a translation:

$$\textbf{let}\ x = f\ 0\ \textbf{in}\ (x, x) \quad \leadsto \quad \lambda v\,.\textbf{let}\ x = (\lambda v.f\ v\ 0)\ \textbf{in}\ (x\ v, x\ v)$$

and the evaluation of $x\ v$ in the translation is no longer shared. There are a number of potential solutions to this problem. In the example above, one method would be to rewrite the translation as:

$$\lambda v\,.\,\mathbf{let}\ x = (f\ \ v\ \ 0)\ \mathbf{in}\ (x, x).$$

This is the kind of translation which will be obtained using a *monomorphism restriction* such as that proposed for Haskell [HPJW92] which restricts the amount of overloading that can be used in particular syntactic forms of binding. Note that this approach is only suitable when the variable defined in the **let** expression is not required to have a polymorphic type in the scope of that definition.

7.2.3 Constant and locally-constant overloading

Consider the typing of local definitions in the type system of Section 4 using the rule:

$$\frac{P \mid A \vdash M : \sigma \qquad Q \mid A_x, x\!:\!\sigma \vdash N : \tau}{P \cup Q \mid A \vdash (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau}$$

Notice that this allows some of the predicates constraining the typing of M (i.e. those in P) to be retained as a constraint on the environment in the conclusion of the rule rather than being included in the type scheme σ. However, in the corresponding rule $(let)^s$ for the syntax-directed system, all of the predicates constraining the typing of M are included in the inferred type scheme $Gen(A, P \Rightarrow \tau)$:

$$\frac{P \mid A \overset{s}{\vdash} M : \tau \qquad P' \mid A_x, x : Gen(A, P \Rightarrow \tau) \overset{s}{\vdash} N : \tau'}{P' \mid A \overset{s}{\vdash} (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau'}$$

As a consequence, evidence parameters are needed for all of the predicates in P, even if some of the corresponding evidence values are the same for each occurrence of x in N. In particular, this includes *constant* evidence (for predicates with no free type variables) and *locally-constant* evidence (for predicates, each of whose free variables also appears free in A).

From the relationship between the type inference algorithm W and the syntax-directed system, it follows that W has the same behaviour; indeed, this is essential to ensure that W calculates principal types: if $x \notin FV(N)$, then none of the environment constraints described by P need be reflected by the constraints on the complete expression in P'.

However, if $x \in FV(N)$, it is possible to find a set $F \subseteq P$ such that $P' \Vdash F$ and hence the type scheme assigned to x can be replaced by $Gen(A, (P \setminus F) \Rightarrow \tau)$, potentially decreasing the number of evidence parameters required by x. To see this, suppose that $Gen(A, P \Rightarrow \tau) = (\forall \alpha_i . P \Rightarrow \tau)$. A straightforward induction, based on the hypothesis that $x \in FV(N)$, shows that $P' \Vdash [\tau_i/\alpha_i]P$ for some types τ_i. If we now define:

$$FP(A, P) \quad = \quad \{\, (v : \pi) \in P \mid TV(\pi) \subseteq TV(A) \,\},$$

then $F = FP(A, P)$ is the largest subset of P that is guaranteed to be unchanged by the substitution $[\tau_i/\alpha_i]$. These observations suggest that $(let)^s$ could be replaced by the two rules:

- In the case where $x \notin FV(N)$:

  $$\frac{P \mid A \overset{s}{\vdash} M : \tau \qquad P' \mid A \overset{s}{\vdash} N : \tau'}{P' \mid A \overset{s}{\vdash} (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau'}\ (let)_f{}^s$$

  The typing judgement involving M serves only to preserve to property that all subterms of a well-typed term are also well-typed.

- In the case where $x \in FV(N)$:

$$\frac{P \mid A \overset{s}{\vdash} M : \tau \quad P' \mid A_x, x\!:\!Gen(A, P \setminus F \Rightarrow \tau) \overset{s}{\vdash} N : \tau' \quad P' \Vdash F}{P' \mid A \overset{s}{\vdash} (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau'}\ (let)_b{}^s$$

where $F = FP(A, P)$.

While these rules retain the syntax-directed character necessary for use in a type inference algorithm, they are not suitable for typing top-level definitions (such as those in Haskell or ML) which are treated as **let** expressions in which the scope of the defined variable is not fully determined at compile-time.

A more realistic approach would be to use just $(let)_b{}^s$in place of $(let)^s$, with the understanding that type schemes inferred by W are only guaranteed to be principal in the case where $x \in FV(N)$ for all subterms of the form **let** $x = M$ **in** N in the term whose type is being inferred. Justification for this approach is as follows:

- For a top-level declaration of the identifier x, we can take the scope of the declaration to be the set of all terms that might reasonably be evaluated in the scope of such a declaration, which of course includes the term x.

- For **let** expressions in which the scope of the defined variable is known, the local definition in an expression of the form **let** $x = M$ **in** N is redundant, and the expression is semantically equivalent to N. However, expressions of this form are sometimes used in implicitly typed languages to force a less general type than might otherwise be obtained by the type inference mechanism. For example, if $(==)$ is an integer equality function and 0 is an integer constant, then $\lambda x.\mathbf{let}\ y = (x == 0)\ \mathbf{in}\ x$ has principal type scheme $Int \to Int$, whereas the principal type scheme for $\lambda x.x$ is $\forall a.a \to a$. Such ad-hoc 'coding-tricks' become unnecessary if the term language is extended to allow explicit type declarations.

In a practical implementation, it would be useful to arrange for suitable diagnostic messages to be generated whenever an expression of the form **let** $x = M$ **in** N with $x \notin FV(N)$ is encountered; this would serve as a warning to the programmer that the principal type property may be lost (in addition to catching other potential program errors).

7.3 Satisfiability

One of the most important features of the systems of qualified types described in this thesis is the ability to move 'global' constraints on a typing derivation into the type of an object using $(\Rightarrow I)$:

$$\frac{P, \pi \mid A \vdash M : \rho}{P \mid A \vdash M : \pi \Rightarrow \rho}$$

This is essential in many situations where overloading is combined with polymorphism: without the ability to move predicates from the first component of a typing $P \mid A \vdash M : \rho$ into the type of an object we would not be able to apply $(\forall I)$ for any type variables appearing in $TV(P)$, severely limiting the use of polymorphism.

On the other hand, with the formulation of the typing rules used in the previous sections there is no attempt to guarantee that the predicates introduced into the type of an object using $(\Rightarrow I)$ are satisfiable. As we have already mentioned, an object of type $\pi \Rightarrow \rho$ can only be used if we can provide evidence for the predicate π. If no such evidence can be obtained, then any object with this type is useless.

This problem was noted by Volpano and Smith [VS91] for the special case of the system of type classes described in [WB89]. With this in mind, they gave a stronger definition of well-typing that includes testing for satisfiability of an inferred type scheme and showed that this makes the process of determining whether a particular term is well-typed undecidable in an restricted version of the Wadler-Blott system. The framework used in this thesis allows us to separate typability from predicate entailment and to identify the problem as undecidability of the latter. Nevertheless, the difficulty remains.

On the one hand we could simply ignore the problem since it will never be possible to resolve the overloading for an object with an unsatisfiable type scheme and hence any attempt to use it will fail. On the other hand, it would certainly be useful if the type system could be used to identify such objects at the point where they are defined and produce suitable error diagnostics to assist the programmer. One possibility would be to modify the rule for typing **let** expressions with:

$$\frac{P \mid A \vdash M : \sigma \quad Q \mid A_x, x\!:\!\sigma \vdash N : \tau \quad P_0 \ \mathsf{sat}\ \sigma}{P, Q \mid A \vdash (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau}$$

to ensure satisfiability with respect to a fixed set of predicates P_0, where:

$$P_0\ \mathsf{sat}\ (\forall \alpha_i . P \Rightarrow \tau) \quad \Leftrightarrow \quad \exists \nu_i . P_0 \Vdash [\nu_i/\alpha_i]P.$$

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 our use of polymorphism, entailment and ordering:

- If $P\ \mathsf{sat}\ \sigma$, then $SP\ \mathsf{sat}\ S\sigma$ for any substitution S.
- If $P\ \mathsf{sat}\ \sigma$ and $Q \Vdash P$, then $Q\ \mathsf{sat}\ \sigma$.
- If $P\ \mathsf{sat}\ \sigma'$ and $(P \mid \sigma') \leq \sigma$, then $P\ \mathsf{sat}\ \sigma$.

We conjecture that, if we restrict our attention to derivations $P \mid A \vdash M : \sigma$ for which $P_0 \Vdash P$, then the development of a principal type algorithm and coherence conditions described in the previous sections will extend naturally to deal with this extension. Note however that we will require decidability of $P_0\ \mathsf{sat}\ \sigma$ for arbitrary P_0 and σ to ensure decidability of type checking.

Another, more positive, application of satisfiability that does not appear to have been considered elsewhere is to allow the use of more accurate types for particular objects. As an example, consider the function $\lambda r.(r.l,\, r.l)$ using the record selection operator described in Section 1.2 which has principal type scheme:

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

On the other hand, for any given 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*:

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

To see how this might be dealt with more formally, recall the treatment of the ordering between type schemes in Section 4.2. Writing the set of generic instances of a type scheme as:

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

the ordering on type schemes is described by:

$$\sigma \leq \sigma' \quad \Leftrightarrow \quad [\![\sigma]\!] \subseteq [\![\sigma']\!]$$

In a similar way can define the *generic satisfiable instances* of a type scheme with respect to a predicate set P_0 as:

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

and define a satisfiability ordering, again with respect to P_0, by:

$$\sigma \leq^{sat}_{P_0} \sigma' \quad \Leftrightarrow \quad [\![\sigma]\!]^{sat}_{P_0} \subseteq [\![\sigma']\!]^{sat}_{P_0}$$

We can formalise the notion of principal satisfiable type in the same way as in Section 6.3 using the $(\leq^{sat}_{P_0})$ ordering in place of $(\leq)$. For the example above, both of the type schemes given are principal satisfiable type schemes for the term $\lambda r.(r.l,\ r.l)$. The first of these is the type scheme that would be obtained using our type inference algorithm, but it would clearly be preferable if the algorithm could be modified to give the second alternative. Further investigation is needed to discover effective procedures or heuristics for calculating more informative types that can be used to support this extension.

7.4 The use of subsumption

The typing rules in Figure 1 are only suitable for reasoning about systems with explicit coercions. For example, if $\mathit{Int} \subseteq \mathit{Real}$, then we can use an addition function:

$$\mathit{add} :: \forall a . a \subseteq \mathit{Real} \Rightarrow a \rightarrow a \rightarrow \mathit{Real}$$

to add two integers together, obtaining a real number as the result. More sophisticated systems, such as those in [Mit84, FM89], cannot be described without adding a form of the rule of subsumption:

$$\frac{P \mid A \vdash M : \tau' \quad P \Vdash \tau' \subseteq \tau}{P \mid A \vdash M : \tau}$$

Each use of this rule corresponds to an implicit coercion; the addition of two integers to obtain a real result can be described without explicit overloading using a function:

$$\mathit{add} :: \mathit{Real} \rightarrow \mathit{Real} \rightarrow \mathit{Real}$$

with two implicit coercions from Int to Real. As a further example, in the framework of Section 2, the polymorphic identity function $\lambda t.\lambda x : t.x$ can be treated as having type $\forall a.\forall b.a \subseteq b \Rightarrow a \rightarrow b$ and hence acts as a generic coercion function.

No attempt has been made to deal with systems including the rule of subsumption in the development of the type inference algorithm in Section 6, which is therefore only suitable for languages using explicit coercions. The results of [FM89] and [Smi91] are likely to be particularly useful in extending the present system to support the use of implicit coercions.

Acknowledgements

I am very grateful for conversations with members of the Department of Computing Science, Glasgow whose suggestions prompted me to investigate the ideas presented in this paper, and in particular to Phil Wadler for his comments on several earlier versions of this paper. I would also like to thank the referees for ESOP '92 for their suggestions and advice concerning my original submission to the symposium.

The Science and Engineering Research Council of Great Britain provided financial support for this work.

References

[BCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter and A. Scedrov. Inheritance and coercion. Proceedings of the fourth annual symposium on logic in computer science, 1989.

[Blo90] Stephen Blott. An approach to overloading with polymorphism. Ph.D. thesis, Department of computing science, University of Glasgow, 1990 (in preparation).

[BW89] Richard Bird and Philip Wadler. Introduction to functional programming. Prentice Hall International, 1989.

[CDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux and Gilles Kahn. A simple applicative language: Mini-ML. ACM symposium on LISP and functional programming, 1986.

[CM90] Luca Cardelli and John Mitchell. Operations on records. Proceedings of the Fifth International Conference on Mathematical Foundations of Programming Language Semantics. Lecture notes in computer science 442, Springer Verlag, 1990.

[Dam85] Luis Damas. Type assignment in programming languages. PhD thesis, University of Edinburgh, CST-33-85, 1985.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. Proceedings of the 8th annual ACM symposium on Principles of Programming languages, Albuquerque, New Mexico, January 1982.

[FM89] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. Lecture notes in computer science 352, Springer Verlag, 1990.

[HP90] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical report CMU-CS-90-102, Carnegie Mellon University, School of computer science, February 1990.

[HPJW92] Paul Hudak, Simon Peyton Jones and Philip Wadler (editors). Report on the programming language Haskell, version 1.2. ACM SIGPLAN notices, 27, 5, May 1992.

[Jon91a] Mark P. Jones. Towards a theory of qualified types. Technical report PRG-TR-6-91, Programming Research Group, Oxford University Computing Laboratory, April 1991.

[Jon91b] Mark P. Jones. Type inference for qualified types. Technical report PRG-TR-10-91, Programming Research Group, Oxford University Computing Laboratory, June 1991.

[Jon92] Mark P. Jones Qualified types: Theory and practice. D. Phil. thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992.

[Mil78] Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17, 3, 1978.

[Mit84] John C. Mitchell. Coercion and type inference (summary). Proceedings of the 11th annual ACM symposium on Principles of Programming Languages, 1984.

[Mit90] John C. Mitchell A type-inference approach to reduction properties and semantics of polymorphic expressions. Logical Foundations of Functional Programming, Gérard Huet (ed.), Addison Wesley, 1990.

[PJW90] Simon L. Peyton Jones and Philip Wadler A static semantics for haskell (draft). Department of Computing Science, University of Glasgow, August 1990.

[Rem89] Didier Rémy. Typechecking records and variants in a natural extension of ML. Proceedings of the 16th annual ACM symposium on Principles of Programming Languages, Austin, Texas, January 1989.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. Journal of the Association for Computing Machinery, 12, 1965.

[Smi91] Geoffrey Smith. Polymorphic type inference for languages with overloading and subtyping. Ph.D. thesis, Department of Computer Science, Cornell University, August 1991.

[VS91] Dennis Volpano and Geoffrey Smith. On the complexity of ML typability with overloading. Proceedings of the 5th ACM conference on Functional Programming Languages and Computer Architecture. Lecture notes in computer science 523, Springer Verlag, 1991.

[WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. Proceedings of the 16th annual ACM symposium on Principles of Programming Languages, Austin, Texas, January 1989.