Coherence For Qualified Types

Mark P. Jones
Research Report YALEU/DCS/RR-989
September, 1993

This work was supported by DARPA N00014-91-J-4043

Coherence for qualified types

Mark P. Jones*

All translations are equal, but some translations are more equal than others.

Misquoted, with apologies to George Orwell, from *Translation Farm*, 1945.

Research Report YALEU/DCS/RR-989, September 1993

Abstract

The meaning of programs in a language with implicit overloading can be described by translating them into a second language that makes the use of overloading explicit. A single program may have many distinct translations and it is important to show that any two translations are semantically equivalent to ensure that the meaning of the original program is well-defined. This property is commonly known as *coherence*.

This paper deals with an implicitly typed language that includes support for parametric polymorphism and overloading based on a system of *qualified types*. Typical applications include Haskell type classes, extensible records and subtyping. In the general case, it is possible to find examples for which the coherence property does not hold. Extending the development of a type inference algorithm for this language to include the calculation of translations, we give a simple syntactic condition on the principal type scheme of a term that is sufficient to guarantee coherence for a large class of programs.

One of the most interesting aspects of this work is the use of terms in the target language to provide a semantic interpretation for the ordering relation between types that is used to establish the existence of principal types.

On a practical level, our results explain the importance of *unambiguous* type schemes in Haskell.

Introduction

Consider the task of evaluating an expression of the form $x + y + z$. Depending on the way that it is parsed, this expression might be treated as either $(x+y)+z$ or $x+(y+z)$. Fortunately, it does not matter which of these we choose since the fact that $(+)$ is associative is both necessary and sufficient to guarantee that they are actually equivalent. We are therefore free to choose whichever is more convenient, retaining the same well-defined semantics in either case.

This paper deals with a similar problem that occurs with programs in OML, a simple implicitly typed language with overloading. The meaning of such programs can be described by translating them into OP, an extended language which uses additional constructs to make the use of overloading explicit. However, different typing derivations for a given OML program can lead to distinct translations and, just as in the example above, it is important to show that any two translations have the same meaning. In the terminology of [2], we need to show that 'the meaning of a term does not depend on the way that it was type checked', a property that they refer to as *coherence*.

The type system of OML is an extended form of the ML type system that includes support for qualified types [7]. The central idea is to allow the use of type expressions of the form $\pi \Rightarrow \sigma$ to represent all those instances of σ which satisfy π, a *predicate* on types. Applications of qualified types include Haskell type classes, extensible records and subtyping.

In previous work, we have described how the standard type inference algorithm for ML can be extended to calculate *principal type schemes* for terms in OML. In this paper, we extend these results to show how an arbitrary translation of an OML term can be written in terms of a particular *principal translation* determined by the type inference algorithm. Exploiting this relationship, we give conditions that can be used to guarantee that all of the translations for a given term are equivalent.

The remaining sections of this paper are as follows. Section 1 outlines the use of qualified types and defines the languages OML and OP and the translation between them that is used in this paper. A simple example in Section 2 shows that a single term may have semantically distinct translations and hence that we cannot hope to establish a general coherence result for arbitrary terms. Instead, we must look for conditions which can be used to ensure coherence for as wide a class of programs as possible.

As a first step, we need to specify exactly what it means for two translations to be equivalent. This is dealt with in Section 3 using a syntactic definition of (typed) equality between OP terms.

One of the most important tools in the development of a type inference algorithm is the ordering relation $(\leq)$ between type schemes. Indeed, without a notion of ordering, it would not even be possible to talk about principal or most general type schemes! Motivated by this, Section 4 gives a semantic interpretation for $(\leq)$ using OP terms which we call *conversions*.

Sections 5 and 6 extend the development of type inference

*This paper summarizes work carried out while the author was a member of the Programming Research Group, Oxford, supported by a SERC studentship [8]. Current address: Yale University, Department of Computer Science, P.O. Box 208285, New Haven, Connecticut 06520-8285, USA. Electronic mail jones-mark@cs.yale.edu. Supported in part by a grant from DARPA, contract number N00014-91-J-4043.

for qualified types in [7] to include the calculation of translations. In particular, we show that any translation of an OML term can be written in the form $C(\lambda w.E')v$ where C is a conversion of a particular kind, E' is the principal translation and v, w are fixed collections of variables. Hence the task of establishing the equivalence of two arbitrary translations reduces to showing the equivalence of two terms of the form $C_1(\lambda w.E')v$ and $C_2(\lambda w.E')v$ where C_1 and C_2 are conversions of a particular type. One obvious way to approach this problem is to show that that these conversions are equivalent.

Exploring this possibility in Section 7, we obtain sufficient conditions for the equivalence of a pair of conversions of a particular type and hence for an arbitrary pair of translations. In particular, we show that the meaning of any term with an ***unambiguous*** principal type scheme (a simple syntactic condition) is well-defined, generalizing an earlier result in [1] for the special case of a system of type classes.

Section 8 concludes with a description of related work and some ideas for further research. Further details of the work described in this paper together with full proofs for the results presented here may be found in [8].

1 Basic definitions

This section outlines the principal features of a system of qualified types and of OML in particular. We refer the reader to either [7] or [8] for further details.

1.1 Predicates and evidence

A language of *predicates* on types is an essential component of any system of qualified types. For example, predicates of the form $Eq\ \tau$ that hold precisely when there is an equality operator defined for values of type τ are often used in work with type classes. Predicates are used to identify particular sets of types, so a term with type $\forall t.\pi(t) \Rightarrow f(t)$ can be treated as having any of the types in the set

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

An object with a qualified type can only be used if we provide *evidence* that the predicates involved are satisfied. A simple choice for a predicate of the form $Eq\ \tau$ might be an equality function for values of type τ, but the exact form of evidence used in any particular application does not affect the work described here and we think of it purely as a semantic interpretation of predicates. Evidence values are written using a language of evidence expressions that includes a set of evidence variables v. The set of evidence variables in an expression e is denoted $EV(e)$.

The final component in a system of predicates is an entailment relation $\Vdash$ which may vary from one application to another. An expression of the form $P \Vdash e:\pi$ indicates that we can obtain evidence e for the predicate π from the *predicate assignment* P, a list of pairs of the form $v:\pi'$. For example, $v:Eq\ \tau \Vdash eqList\ v:Eq\ [\tau]$ might indicate that, if v is bound to an equality test for values of type τ, then the expression *eqList* v gives an equality test for lists of such values.

Figure 1 lists the properties which the entailment relation is expected to satisfy. These rules make use of some simple abbreviations, blurring the distinction between sequences

(*id*)	$v:P \Vdash v:P$
(*term*)	$v:P \Vdash \emptyset$
(*fst*)	$v:P, w:Q \Vdash v:P$
(*snd*)	$v:P, w:Q \Vdash w:Q$
(*univ*)	$\dfrac{v:P \Vdash e:Q \qquad v:P \Vdash e':R}{v:P \Vdash e:Q, e':R}$
(*trans*)	$\dfrac{v:P \Vdash e:Q \qquad v':Q \Vdash e':R}{v:P \Vdash [e/v']e':R}$
(*close*)	$\dfrac{v:P \Vdash e:Q}{v:SP \Vdash e:SQ}$
(*evars*)	$\dfrac{v:P \Vdash e:Q}{EV(e) \subseteq v}$

Figure 1: Predicate entailment with evidence.

and individual objects. For example, if $P = \pi_1, \ldots, \pi_n$ is a list of predicates and $v = v_1, \ldots, v_n$ is a list of evidence variables then we write $v:P$ for the predicate assignment $v_1:\pi_1, \ldots, v_n:\pi_n$. The empty sequence is written $\emptyset$ and the concatenation of two sequences P and Q is written P, Q. The letter S in rule (*close*) denotes an arbitrary substitution of types for type variables.

1.2 Terms and types

This paper deals with the relationship between two implicitly typed λ-calculi with support for qualified types. The first of these is an extension of ML and will be referred to as OML – an abbreviation of 'Overloaded ML'. As in [4], the terms of OML are those of simple untyped λ-calculus with the addition of a *let* construct to enable the definition and use of polymorphic, overloaded terms:

E	::=	x	*variable*
	|	EF	*application*
	|	$\lambda x.E$	*abstraction*
	|	let $x = E$ in F	*local definition*

Following the distinction between types and type schemes in ML, the types of terms in OML are given by:

τ	::=	t	*type variables*
	|	$\tau \to \tau$	*function types*
ρ	::=	$P \Rightarrow \tau$	*qualified types*
σ	::=	$\forall T.\rho$	*type schemes*

where t denotes a type variable, P a finite sequence of predicates and T a finite 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 lists, pairs and record types will be used as required. The set of type variables appearing (free) in an expression X is denoted $TV(X)$ and is defined in the obvious way. In particular, $TV(\forall T.\rho) = TV(\rho) \setminus T$.

The second language will be referred to as OP – an abbreviation for 'Overloaded Polymorphic λ-calculus'. The terms

of OP are the same as those for OML with additional constructs for evidence abstraction and application:

$$\begin{array}{rcll} E & ::= & \ldots & \\ & | & Ee & \textit{evidence application} \\ & | & \lambda v.E & \textit{evidence abstraction} \end{array}$$

As before, it is convenient to use some abbreviations for dealing with sequences of evidence abstractions or applications. For example, if $v = v_1, \ldots, v_n$ and $e = e_1, \ldots, e_n$, then we write $\lambda v.E$ and Ee as abbreviations for $\lambda v_1 \ldots \lambda v_n.E$ and $(\ldots(Ee_1)\ldots)e_n$ respectively.

The language of types in OP is given by the grammar:

$$\begin{array}{rcll} \sigma & ::= & t & \textit{type variables} \\ & | & \sigma \to \sigma & \textit{function types} \\ & | & \forall t.\sigma & \textit{polymorphic types} \\ & | & \pi \Rightarrow \sigma & \textit{qualified types} \end{array}$$

An OML qualified type $(\pi_1, \ldots, \pi_n) \Rightarrow \tau$ can be identified with the OP type $\pi_1 \Rightarrow \ldots \Rightarrow \pi_n \Rightarrow \tau$. OP types are considerably more flexible than those of OML since there is no distinction between simple types and type schemes. In particular, OP allows functions with polymorphic and/or overloaded values as their arguments. Strictly speaking, there is no need to include the **let** construct in OP since we can code a polymorphic local definition **let** $x = E$ **in** F as $(\lambda x.F)E$. The most important benefit of including **let** is that it makes it easier to treat OML as a (proper) sublanguage of OP.

1.3 Typing rules

Reasons of space prevent us from including the full typing rules for OML and OP. Instead, we present both systems using a hybrid with judgements $P \mid A \vdash E \rightsquigarrow E' : \sigma$ where E is an OML term and E' is a corresponding OP term referred to as a *translation* of E. The first component P in these judgements is a predicate assignment, while A is a type assignment, i.e. a (finite) set of pairs of the form $x:\sigma$ in which no term variable x appears more than once. Type assignments can be interpreted as finite functions mapping term variables to types. We write $A(x)$ for the type assigned to x by A, A_x for the assignment obtained by removing x from the domain of A, and $A, x:\sigma$ for the assignment which is the same as A except that it also maps x to σ.

The typing rules are given in Figure 2. Note the use of the symbols τ, ρ and σ to restrict the application of certain rules to particular kinds of type expression, given by the grammar for OML types above. With these restrictions in mind, and ignoring the translation component of each judgement, Figure 2 gives the typing rules for OML. On the other hand, if we disregard the OML term in each judgement and ignore the distinction between the three classes of type in OML, we obtain the typing rules for OP. The significance of this hybrid formulation is that it enables us to deal simultaneously with OML and OP terms whose typing derivations have the same structure.

The relationship between OML terms and translations given by the rules in Figure 2 is not functional – there is no unicity of type, and different derivations of the same typing in OML can result in distinct translations. On the other hand, we can always recover the original OML term corresponding to a given translation using the function:

$$\begin{array}{lcl} \textit{Erase}\ (x) & = & x \\ \textit{Erase}\ (EF) & = & (\textit{Erase}\ E)\ (\textit{Erase}\ F) \\ & \vdots & \\ \textit{Erase}\ (Ee) & = & \textit{Erase}\ E \\ \textit{Erase}\ (\lambda v.E) & = & \textit{Erase}\ E \end{array}$$

1.4 Ordering type schemes

Each OML typing includes a predicate set that restricts its use to environments in which the given predicates hold. As such, it is convenient to work with a slightly more general notion of type scheme which also contains constraints on the environments in which it may be used.

Definition 1 *A* constrained type scheme *is an expression of the form $(P \mid \sigma)$ where P is a set of predicates and σ is a type scheme.*

Any type scheme σ may be identified with a constrained type scheme of the form $(\emptyset \mid \sigma)$. Note that there is no need to use constrained type schemes in OP since we can treat $(P \mid \sigma)$ as an abbreviation for the OP type $P \Rightarrow \sigma$

We will write $(P' \mid \sigma') \leq (P \mid \sigma)$ to indicate that $(P \mid \sigma)$ is *more general* than $(P' \mid \sigma')$. A suitable ordering relation, extending the ordering between type schemes in Damas and Milner's treatment of type inference for ML, was suggested in [7] and can be characterized as follows:

Definition 2 *Suppose that $\sigma = \forall\alpha_i.Q \Rightarrow \nu$, $\sigma' = \forall\beta_j.Q' \Rightarrow \nu'$ and that none of the variables β_j appears free in σ, P or P'. Then $(P' \mid \sigma') \leq (P \mid \sigma)$ if and only if there are types τ_i such that $\nu' = [\tau_i/\alpha_i]\nu$ and $P', Q' \Vdash P, [\tau_i/\alpha_i]Q$.*

2 The coherence problem

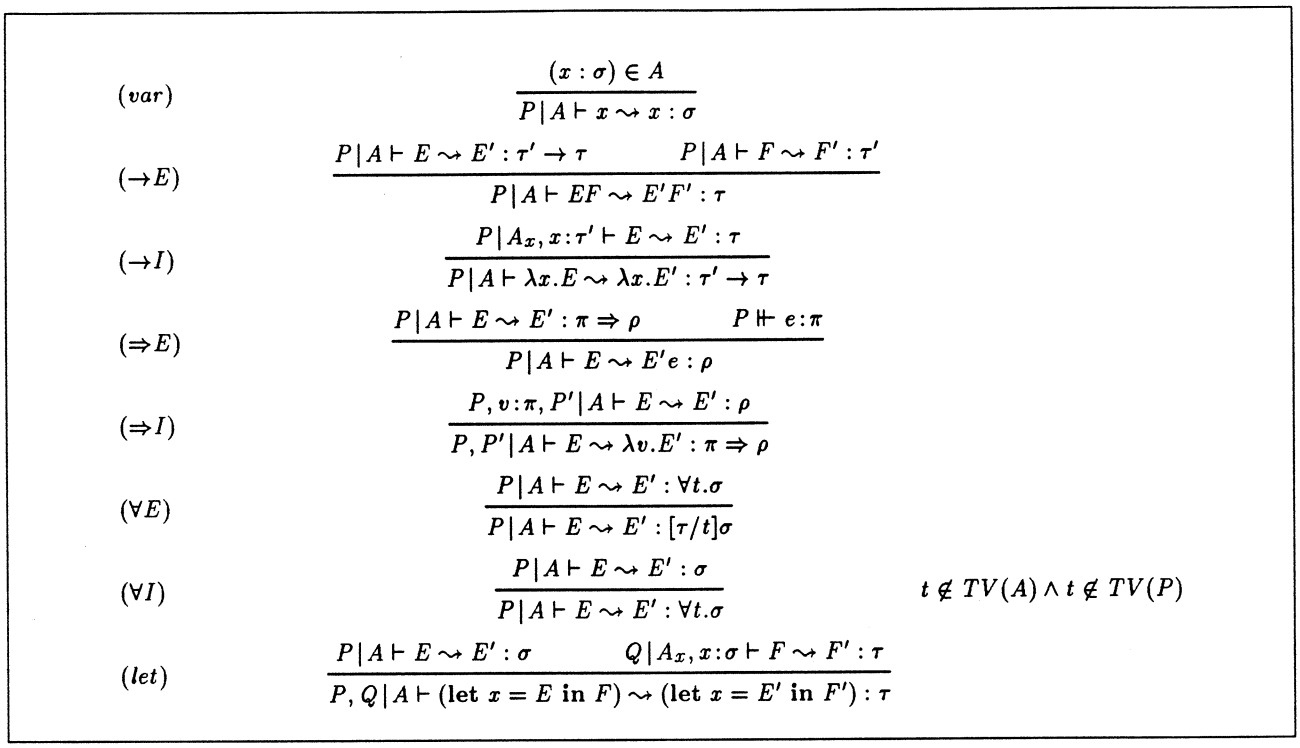
To justify the use of translations as a semantics for OML, we need to show that:

- For each OML term E there is an OP term E' that is a translation of E.
- Any translation of a well-typed OML term is well-typed in OP.
- The mapping from terms to translations must be well-defined. In other words, we must show that any translations E_1 and E_2 of an OML term E given by derivations $P \mid A \vdash E \rightsquigarrow E_1 : \sigma$ and $P \mid A \vdash E \rightsquigarrow E_2 : \sigma$ are, in some precise sense, equivalent.

The first two properties follow immediately from the typing rules given above, but it is relatively simple to show that the third property does not hold in general. For example, Haskell [6] provides two standard functions:

$$\begin{array}{lcl} \textit{read} & : & \forall a.\textit{Text}\ a \Rightarrow \textit{String} \to a \\ \textit{show} & : & \forall a.\textit{Text}\ a \Rightarrow a \to \textit{String} \end{array}$$

for converting between values and their printable representations as strings. Now suppose that the type assumption A containing these functions and that implementations have been provided for both integers and booleans, in a predicate assignment $P = \{u : \textit{Text Int}, v : \textit{Text Bool}\}$. Now



$$(var)\quad \frac{(x:\sigma)\in A}{P\,|\,A\vdash x\rightsquigarrow x:\sigma}$$

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

$$(\rightarrow I)\quad \frac{P\,|\,A_x,x{:}\tau'\vdash E\rightsquigarrow E':\tau}{P\,|\,A\vdash \lambda x.E\rightsquigarrow \lambda x.E':\tau'\rightarrow\tau}$$

$$(\Rightarrow E)\quad \frac{P\,|\,A\vdash E\rightsquigarrow E':\pi\Rightarrow\rho \qquad P\Vdash e{:}\pi}{P\,|\,A\vdash E\rightsquigarrow E'e:\rho}$$

$$(\Rightarrow I)\quad \frac{P,v{:}\pi,P'\,|\,A\vdash E\rightsquigarrow E':\rho}{P,P'\,|\,A\vdash E\rightsquigarrow \lambda v.E':\pi\Rightarrow\rho}$$

$$(\forall E)\quad \frac{P\,|\,A\vdash E\rightsquigarrow E':\forall t.\sigma}{P\,|\,A\vdash E\rightsquigarrow E':[\tau/t]\sigma}$$

$$(\forall I)\quad \frac{P\,|\,A\vdash E\rightsquigarrow E':\sigma}{P\,|\,A\vdash E\rightsquigarrow E':\forall t.\sigma}\qquad t\notin TV(A)\wedge t\notin TV(P)$$

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

Figure 2: Hybrid typing rules for OML and OP

consider the composition of these two functions *read . show* converting a string to some type of values, and then back to a string. Instantiating the quantified type variable in the type of *read* (and hence also that of *show*) determines the type of intermediate values used and leads to the following derivations with translations that are clearly not equivalent:

$$P\,|\,A\vdash (read\,.\,show)\rightsquigarrow(read\ u\,.\,show\ u):String\rightarrow String$$
$$P\,|\,A\vdash (read\,.\,show)\rightsquigarrow(read\ v\,.\,show\ v):String\rightarrow String$$

Clearly, we cannot hope to establish the general coherence result in the third item above; i.e. that all translations of an arbitrary OML term are semantically equivalent. In the rest of this paper we work towards a more modest goal – to identify a collection of OML terms for which the coherence property can be established.

3 Equality of OP terms

Before we can establish sufficient conditions to guarantee coherence, we need to specify formally what it means for two terms (specifically, two translations) to be equivalent. This section gives a syntactic characterization of (typed) equality between OP terms using judgements of the form $P\,|\,A\vdash E=F:\sigma$ (with the implicit side-condition that both $P\,|\,A\vdash E:\sigma$ and $P\,|\,A\vdash F:\sigma$ in OP).

3.1 Uniqueness of evidence

The use of predicate assignments in the definition of equality enables us to capture the 'uniqueness of evidence'; to be precise, we require that any evidence values e and f constructed by entailments $P\Vdash e:Q$ and $P\Vdash f:Q$ are semantically equivalent, in which case we write $P\vdash e=f:Q$. Since we only intend such judgements to be meaningful when both entailments hold, the definition of equality on evidence expressions can be described directly using:

$$P\vdash e=f:Q \quad\Leftrightarrow\quad P\Vdash e:Q \ \wedge\ P\Vdash f:Q.$$

This condition is essential if any degree of coherence is to be obtained. Without it, for example, it would be possible to have semantically distinct versions of an overloaded operator that cannot be distinguished either by name or by type.

3.2 Reduction of OP terms

We will base the definition of equality on OP terms on a notion of (typed) *reductions* using judgements of the form $P\,|\,A\vdash E\rhd F:\sigma$ with the implicit side condition that $P\,|\,A\vdash E:\sigma$ in OP. There is no need to include $P\,|\,A\vdash F:\sigma$ as a second side condition since this follows from the first by the *subject reduction theorem* – 'reduction preserves typing'.

We split the definition of reduction into three parts, the first of which appears in Figure 3. This includes the familiar definitions of β-conversion for evidence and term abstractions and **let** expressions and a rule of η-conversion for evidence abstractions.

An unfortunate consequence of this approach is that the axiom (β) is not sound in models of the λ-calculus with call-by-value semantics and hence our results can only be applied to languages with lazy or call-by-name semantics. This limitation stems more from the difficulty of axiomatizing call-by-value equality than from anything implicit in our particular application and will be discussed in Section 8.

A second collection of rules in Figure 4 is used to describe the renaming of bound variables in λ-abstractions, evidence

$$(\beta) \qquad P \mid A \vdash (\lambda x.E)F \rhd [F/x]E : \sigma$$

$$(\beta_e) \qquad P \mid A \vdash (\lambda v.E)e \rhd [e/v]E : \sigma$$

$$(\beta\text{-}let) \qquad P \mid A \vdash (\textbf{let } x = E \textbf{ in } F) \rhd [E/x]F : \sigma$$

$$(\eta_e) \qquad \frac{v \notin EV(E)}{P \mid A \vdash (\lambda v.Ev) \rhd E : \sigma}$$

Figure 3: Rules of computation

$$(\alpha) \qquad \frac{x \notin FV(\lambda y.E)}{P \mid A \vdash (\lambda y.E) \rhd (\lambda x.[x/y]E) : \sigma}$$

$$(\alpha_e) \qquad \frac{v \notin EV(\lambda w.E)}{P \mid A \vdash (\lambda w.E) \rhd (\lambda v.[v/w]E) : \sigma}$$

$$(\alpha\text{-}let) \qquad \frac{x \notin FV(\lambda y.E)}{P \mid A \vdash (\textbf{let } y = E \textbf{ in } F) \rhd (\textbf{let } x = E \textbf{ in } [x/y]F) : \sigma}$$

Figure 4: Rules for renaming bound variables

$$\frac{(x{:}\sigma) \in A}{P \mid A \vdash x \rhd x : \sigma}$$

$$\frac{P \mid A \vdash E \rhd E' : \sigma' \to \sigma \qquad P \mid A \vdash F \rhd F' : \sigma'}{P \mid A \vdash EF \rhd E'F' : \sigma}$$

$$\frac{P \mid A_x, x{:}\sigma' \vdash E \rhd E' : \sigma}{P \mid A \vdash \lambda x.E \rhd \lambda x.E' : \sigma' \to \sigma}$$

$$\frac{P \mid A \vdash E \rhd E' : \pi \Rightarrow \sigma \qquad P \vdash e = e' : \pi}{P \mid A \vdash Ee = E'e' : \sigma}$$

$$\frac{P, v{:}\pi, P' \mid A \vdash E \rhd E' : \sigma}{P, P' \mid A \vdash \lambda v.E \rhd \lambda v.E' : \pi \Rightarrow \sigma}$$

$$\frac{P \mid A \vdash E \rhd E' : \forall t.\sigma}{P \mid A \vdash E \rhd E' : [\tau/t]\sigma}$$

$$\frac{P \mid A \vdash E \rhd E' : \sigma \qquad t \notin TV(A) \cup TV(P)}{P \mid A \vdash E \rhd E' : \forall t.\sigma}$$

$$\frac{P \mid A \vdash E \rhd E' : \sigma \qquad P \mid A_x, x{:}\sigma \vdash F \rhd F' : \tau}{P \mid A \vdash (\textbf{let } x = E \textbf{ in } F) \rhd (\textbf{let } x = E' \textbf{ in } F') : \tau}$$

Figure 5: Structural laws for reductions between terms.

abstractions and **let** expressions. Any such renaming is permitted so long as we avoid clashes with free variables.

The final group of *structural* rules in Figure 5 is closely modeled on the typing rules for OP and can be used to describe the reduction of subterms within a given term.

3.3 Equalities between terms

The rules in Figure 6 define the equality relation for terms in OP as the transitive, symmetric closure of the reduction relation described above. The first two rules ensure that

$$\frac{P \mid A \vdash E = F : \sigma}{P \mid A \vdash F = E : \sigma}$$

$$\frac{P \mid A \vdash E = E' : \sigma \qquad P \mid A \vdash E' = E'' : \sigma}{P \mid A \vdash E = E'' : \sigma}$$

$$\frac{P \mid A \vdash E \rhd F : \sigma}{P \mid A \vdash E = F : \sigma}$$

Figure 6: Definition of equality between terms.

equality is an equivalence relation. There is no need to include reflexivity here since this is a direct consequence of the structural rules in Figure 5. The last rule shows how reductions give rise to equalities.

In practice, many of the rules used in the definition of equality above will be used implicitly in the proof of equalities between terms. The following example uses all three of the rules in Figure 6 (as well as subject reduction to justify the fact that the intermediate steps are well-typed):

$$\begin{array}{lll} P \mid A \vdash \textbf{let } x = E \textbf{ in } [F/x]F' & & \\ \quad = \quad [E/x]([F/x]F') & & (\beta\text{-}let) \\ \quad = \quad [[E/x]F/x]F' & & \\ \quad = \quad \textbf{let } x = [E/x]F \textbf{ in } F' : \sigma & & (\beta\text{-}let) \end{array}$$

The context in which this equality is established (given by P, A and σ) does not play a part in the calculation. Examples like this are quite common and we will often avoid mentioning the context altogether in such situations, writing $\vdash E = F$ to indicate that $P \mid A \vdash E = F : \sigma$ for any choice of P, A and σ for which the required side conditions hold.

The above property of **let** expressions may seem unfamiliar, and it is worth illustrating why it is useful in our work. Suppose that $\emptyset \Vdash e : Eq\ Int$ and that $(==)$ denotes an equality function of type $\forall a.Eq\ a \Rightarrow a \to a \to Bool$. Now consider the OML term:

$$\textbf{let } f = (\lambda x.\lambda y.x == y) \textbf{ in } f\ 2\ 3.$$

Since the function f is only ever applied to integer values, it is sufficient to treat f as having type $Int \to Int \to Bool$, with translation:

$$\textbf{let } f = (\lambda x.\lambda y.(==)\ e\ x\ y) \textbf{ in } f\ 2\ 3$$

However, the type inference algorithm calculates the type of f as $\forall a.Eq\ a \Rightarrow a \to a \to Bool$ and results in a translation of the form:

$$\textbf{let } f = (\lambda v.\lambda x.\lambda y.(==)\ v\ x\ y) \textbf{ in } f\ e\ 2\ 3.$$

The following calculation shows that these translations are equal and hence that it is possible to eliminate the evidence abstraction used in the second case. The second step is justified by the result above.

$$\begin{array}{ll} \vdash \textbf{let } f = (\lambda v.\lambda x.\lambda y.(==)\ v\ x\ y) \textbf{ in } f\ e\ 2\ 3 & \\ \quad = \quad \textbf{let } f = (\lambda v.\lambda x.\lambda y.(==)\ v\ x\ y) & \\ \qquad \textbf{in } [f\ e/f](f\ 2\ 3) & \\ \quad = \quad \textbf{let } f = [\lambda v.\lambda x.\lambda y.(==)\ v\ x\ y/f](f\ e) \textbf{ in } f\ 2\ 3 & \\ \quad = \quad \textbf{let } f = (\lambda v.\lambda x.\lambda y.(==)\ v\ x\ y)\ e \textbf{ in } f\ 2\ 3 & \\ \quad = \quad \textbf{let } f = (\lambda x.\lambda y.(==)\ e\ x\ y) \textbf{ in } f\ 2\ 3 & \end{array}$$

As in the last step here, many equalities between terms can be obtained by replacing one subterm with an equivalent term. These steps are justified by the structural rules in Figure 5 and are often used implicitly in proofs.

4 Conversions

One of the most important tools in the treatment of type inference is the ordering relation $\leq$ used to describe when one (constrained) type scheme is more general than another. For example, assuming that $\emptyset \Vdash e : Eq\ Int$, the ordering:

$$(\forall a.Eq\ a \Rightarrow a \to a \to Bool) \geq (Int \to Int \to Bool)$$

might be used to justify replacing an integer equality function, say $primEqInt :: Int \to Int \to Bool$ with a generic equality function with the more general type $\forall a.Eq\ a \Rightarrow a \to a \to Bool$ as in the previous section. This breaks down in OP due to the presence of evidence abstraction and application: simply replacing *primEqInt* with $(==)$ in *primEqInt* 2 3 does not even give a well-typed expression! The correct approach is to replace *primEqInt* by $(==)\ e$.

More generally, we will deal with examples like this using OP terms as an interpretation of the ordering between type schemes. For each $\sigma \geq \sigma'$ we identify a particular collection of terms that we call *conversions* from σ to σ'. Each such conversion is a closed OP term $C : \sigma \to \sigma'$ and hence any term of type σ can be treated as having type σ' by applying the conversion C to it. One possible conversion for the example above is:

$$(\lambda x.xe) : (\forall a.Eq\ a \Rightarrow a \to a \to Bool) \to (Int \to Int \to Bool).$$

Note that the type of this conversion (as in the general case) cannot be expressed as an OML type scheme since it uses the richer structure of OP types.

For the purposes of type inference it would be sufficient to take any term C of type $\sigma \to \sigma'$ as a conversion for $\sigma' \leq \sigma$ but this is clearly inadequate if we are also concerned with the semantics of the terms involved; we can only replace E with CE if we can guarantee that these terms are equivalent, except perhaps in their use of evidence abstraction and application. More formally, we need to ensure that $\vdash Erase\ (CE) = Erase\ E$ for all OP terms E (or at least, all those occurring as translations of OML terms). Since $Erase\ (CE) = (Erase\ C)\ (Erase\ E)$, the obvious way to

ensure that this condition holds is to require that *Erase* C is equivalent to the identity term $id = \lambda x.x$.

These ideas extend to conversions between arbitrary constrained type schemes. It is tempting to define the set of conversions from $(P' \mid \sigma')$ to $(P \mid \sigma)$ as the set of all closed OP terms $C : (P \mid \sigma) \to (P' \mid \sigma')$ for which *Erase* C is equivalent to id. In practice it is more convenient to choose a more conservative definition that gives more information about the structure of conversions:

Definition 3 *Suppose* $\sigma = (\forall\alpha_i.Q \Rightarrow \tau)$, $\sigma' = (\forall\beta_j.Q' \Rightarrow \tau')$ *and none of* β_j *appear free in* σ, P *or* P'. *A* conversion C *from* $(P \mid \sigma)$ *to* $(P' \mid \sigma')$, *written* $C : (P \mid \sigma) \geq (P' \mid \sigma')$, *is a closed OP term of type* $(P \mid \sigma) \to (P' \mid \sigma')$ *such that:*

- *Erase* $C = id$,
- $v : P', w : Q' \Vdash e : P, f : [\tau_i/\alpha_i]Q$,
- $\tau' = [\tau_i/\alpha_i]\tau$,
- *and* $\vdash C = \lambda x.\lambda v.\lambda w.xef$

for some types τ_i, *evidence variables* v *and evidence* e.

It is straightforward to verify that the term $\lambda x.\lambda v.\lambda w.xef$ mentioned in this definition is a conversion from $(P \mid \sigma)$ to $(P' \mid \sigma')$ and it follows that any equivalent OP term with the same type will also be a conversion of the same kind. On the other hand, we cannot assume that all such conversions will be equivalent to this particular term since there may be more than one choice for the types τ_i and hence for the evidence expressions f in the definition above.

It is immediate from the definitions above that $(P \mid \sigma) \geq (P' \mid \sigma')$ if and only if there is a conversion $C : (P \mid \sigma) \geq (P' \mid \sigma')$ (this may require renaming the bound variables of σ' to apply the definition of conversions). As a result, all of the properties of the $(\leq)$ ordering described in [7] can be extended to analogous results for conversions. For example, the following proposition shows that reflexivity of $(\leq)$ corresponds to the identity conversion while transitivity of $(\leq)$ corresponds to composition of conversions.

Proposition 1 *For any* $(P \mid \sigma)$ *there is a conversion* $id : (P \mid \sigma) \geq (P \mid \sigma)$. *Furthermore, if* $C : (P \mid \sigma) \geq (P' \mid \sigma')$ *and* $C' : (P' \mid \sigma') \geq (P'' \mid \sigma'')$, *then* $(C' \circ C) : (P \mid \sigma) \geq (P'' \mid \sigma'')$ *where* $(C' \circ C) \equiv \lambda x.C'(Cx)$.

From a categorical perspective, this proposition can be used to show that there is a category whose objects are type schemes and whose arrows are (equivalence classes of) conversions. The only additional properties needed to justify this are that the composition of equivalence classes is well-defined and associative, both of which are easily verified.

The ordering relation $(\leq)$ is preserved by substitutions and the corresponding result for conversions is:

Proposition 2 *If* $C : (P \mid \sigma) \geq (P' \mid \sigma')$ *and* S *is a substitution of types for type variables, then* $C : S(P \mid \sigma) \geq S(P' \mid \sigma')$.

The ordering between type schemes extends to an ordering between (constrained) type assignments, writing $(P \mid A) \geq (P' \mid A')$ to indicate that $(P \mid A(x)) \geq (P' \mid A'(x))$ for each $x \in dom\ A = dom\ A'$. It is useful to extend the definition of conversions to orderings between type assignments. For the purposes of this work, it is sufficient to consider only the case of orderings of the form $A \geq A'$ and $A \geq (P \mid A')$, the first of which is just a special case of the second with $P = \emptyset$.

One simple approach would be to define a conversion for an ordering $A \geq (P \mid A')$ as a function that gives a conversion from $A(x)$ to $(P \mid A'(x))$ for each $x \in dom\ A$. However, whereas we might use a conversion $C : \sigma \geq (P \mid \sigma')$ to treat a term of type σ as having type σ', we will typically use a conversion between type assignments to simultaneously replace each occurrence of a variables mentioned in the type assignment with an appropriate new term. From this perspective it seems more sensible to think of a conversion between type assignments as a term substitution.

Furthermore, the translations of a term are calculated with respect to a particular predicate assignment (the first component in a derivation $v : P \mid A \vdash E \rightsquigarrow E' : \sigma$) and may involve the evidence variables in the domain of that assignment. It is therefore necessary to specify these variables explicitly as part of the type of the conversion.

Definition 4 *A* conversion C *from a type assignment* A *to a constrained type assignment* $(v : P \mid A')$ *with the same domain, written* $C : A \geq (v : P \mid A')$, *is a substitution such that:*

- $dom\ C \subseteq dom\ A = dom\ A'$. *In particular, if* $x \notin dom\ A$, *then* $Cx \equiv x$.
- $(\lambda x.\lambda v.Cx) : A(x) \geq (P \mid A'(x))$ *for each* $x \in dom\ A$.

Note that the expression Cx in this definition denotes an application of a (meta-language) substitution to a particular variable; C is not an OP term.

Continuing with the previous example and assuming that $\emptyset \Vdash e : Eq\ Int$, one possible conversion for the type assignment ordering

$$\begin{array}{c} \{(==) : Int \to Int \to Bool\} \\ \leq \\ \{(==) : \forall a.Eq\ a \Rightarrow a \to a \to Bool\} \end{array}$$

would be the substitution that maps $(==)$ to $(==)\ e$ and fixes every other variable. To see how this might be used, consider an OP term in which the $(==)$ has been treated as having type $Int \to Int \to Bool$. If we replace this with a generic equality function with the more general type, then we need to include the evidence e for $Eq\ Int$ with every use of $(==)$. This is precisely the effect obtained by applying the conversion to the original term.

5 Syntax-directed translation

The next two sections follow the development of [7] to describe the relationship between an arbitrary translation of an OML term and a particular translation determined by the type inference algorithm. For reasons of space, we will only sketch the details here and refer the reader to [8] for further explanation and motivation.

The rules in Figure 2 are not well-suited to use in a type inference algorithm since it is not always clear which (if any) should be used to obtain an optimal (i.e. principal) typing for a given term. Our solution is to work with the set of typing rules in Figure 7 in which the structure of a derivation $P \mid A \overset{S}{\vdash} E \rightsquigarrow E' : \tau$ is uniquely determined by the syntactic structure of the OML term E and to show that this *syntax-directed* approach is equivalent to the original type system.

$$(var)^s \qquad \frac{(x:(\forall\alpha_i.Q \Rightarrow \nu)) \in A \qquad P \Vdash e:[\tau_i/\alpha_i]Q}{P \mid A \vdash^s x \rightsquigarrow xe : [\tau_i/\alpha_i]\nu}$$

$$(\rightarrow E)^s \qquad \frac{P \mid A \vdash^s E \rightsquigarrow E' : \tau' \rightarrow \tau \qquad P \mid A \vdash^s F \rightsquigarrow F' : \tau'}{P \mid A \vdash^s EF \rightsquigarrow E'F' : \tau}$$

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

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

Figure 7: Syntax-directed typing rules with translation

Note that the OP translation for a given OML term need not be uniquely determined since there may be distinct choices for the evidence values e introduced in $(var)^s$. This, of course, is the source of the incoherence in the translation semantics of OML.

A simple proof by induction establishes the soundness of the syntax-directed rules with respect to those in Figure 2.

Theorem 1 *If $P \mid A \vdash^s E \rightsquigarrow E' : \tau$, then $P \mid A \vdash E \rightsquigarrow E' : \tau$.*

The reverse process, to establish a form of completeness property by showing that every translation and typing obtained using the general rules in Figure 2 can, in some sense, be described by a syntax-directed derivation is less obvious. For example, if $P \mid A \vdash E \rightsquigarrow F : \sigma$, 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 $v' : P' \mid A \vdash^s E \rightsquigarrow F' : \tau$, Theorem 1 shows that $v' : P' \mid A \vdash E \rightsquigarrow F' : \tau$, and the most general typing that can be obtained from this using $(\Rightarrow I)$ and $(\forall I)$ is $\emptyset \mid A \vdash E \rightsquigarrow \lambda v'.F' : Gen(A, P' \Rightarrow \tau)$ where:

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

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 $(P \mid \sigma)$ in the original derivation and that the translations are related by the corresponding conversion.

Theorem 2 *If $v : P \mid A \vdash E \rightsquigarrow E' : \sigma$, then there is a predicate assignment $v' : P'$, a type τ' and a term E'' such that $v' : P' \mid A \vdash^s E \rightsquigarrow E'' : \tau'$ and $v : P \mid A \vdash C(\lambda v'.E'')v = E' : \sigma$ for some conversion $C : Gen(A, P' \Rightarrow \tau') \geq (P \mid \sigma)$.*

The proof (by structural induction) is quite complicated and makes use of several results which are of interest in their own right. In particular, if we assume that $v : P \mid A \vdash^s E \rightsquigarrow E' : \tau$, then:

- $SP \mid SA \vdash^s E \rightsquigarrow E' : S\tau$ for any substitution S.
- If $Q \Vdash e : P$, then $Q \mid A \vdash^s E \rightsquigarrow [e/v]E' : \tau$.
- If $C : A' \geq (v : P \mid A)$, then $v : P \mid A' \vdash^s E \rightsquigarrow E'' : \tau$ and $v : P \mid A' \vdash CE' = E'' : \tau$.

Each of these extends earlier results described in [7].

6 Type inference and translation

Figure 8 gives the rules necessary to extend the type inference algorithm from [7] to include the calculation of a translation. These rules can be interpreted as an attribute grammar in which the type assignment A and OML term E in a judgement of the form $P \mid TA \vdash^W E \rightsquigarrow E' : \tau$ are inherited attributes, while the predicate assignment P, substitution T, OP translation E' and type τ are synthesized.

Any typing and translation that is obtained using the type inference algorithm can be derived in the syntax-directed system:

Theorem 3 *If $P \mid TA \vdash^W E \rightsquigarrow E' : \tau$, then $P \mid TA \vdash^s E \rightsquigarrow E' : \tau$.*

Combining this with Theorem 1 we obtain:

Corollary 1 *If $P \mid TA \vdash^W E \rightsquigarrow E' : \tau$, then $P \mid TA \vdash E \rightsquigarrow E' : \tau$.*

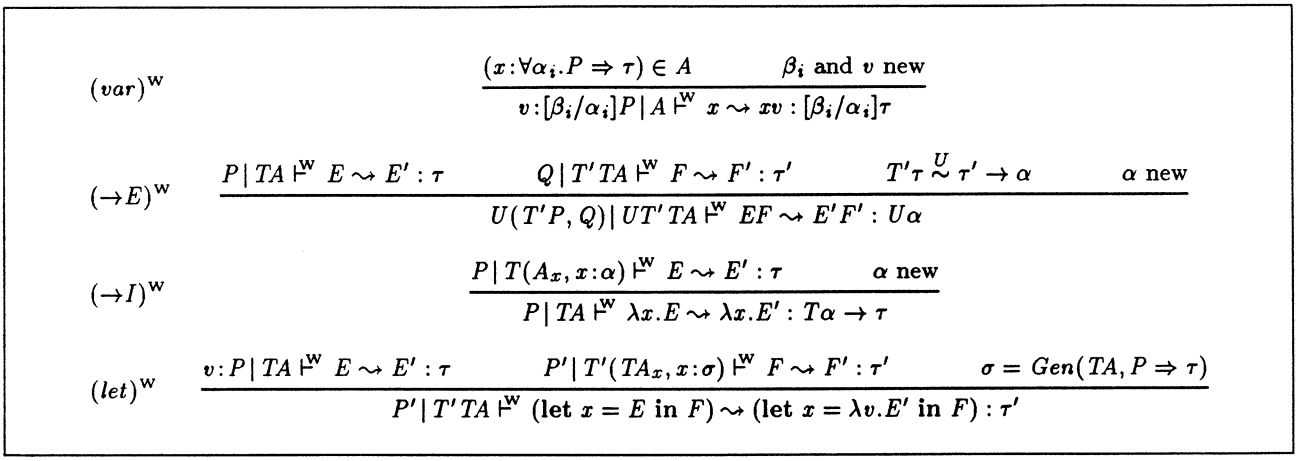
This is important because it shows that the 'translation' E' of an OML term E produced by the algorithm above is a valid translation of E and, in particular, that it is a well-typed OP term. We will refer to the translations produced by this algorithm as *principal translations*. The following theorem provides strong motivation for this terminology, showing that every translation obtained using the syntax-directed system can be expressed in terms of a principal translation.

Theorem 4 *If $v:P \mid SA \vdash^s E \rightsquigarrow E' : \tau$, then $w:Q \mid TA \vdash^W E \rightsquigarrow E'' : \nu$ and there is a substitution R such that $\tau = R\nu$, $v:P \Vdash e:RQ$, $v:P \mid SA \vdash E' = [e/w]E'' : \tau$ and $S \approx RT$.*

The notation $S \approx RT$ used here means that $S\alpha = RT\alpha$ for all but a finite number of new type variables α.

Theorem 4 can now be used to describe the relationship between arbitrary translations of an OML term and a principal translation:

Corollary 2 *If $v:P \mid SA \vdash E \rightsquigarrow E' : \sigma$, then $w:Q \mid TA \vdash^W E \rightsquigarrow E'' : \nu$ for some $w:Q$, T, E'' and σ and there is a substitution R and a conversion $C : RGen(TA, Q \Rightarrow \nu) \geq (P \mid \sigma)$ such that $S \approx RT$ and $v:P \mid SA \vdash C(\lambda w.E'')v = E' : \sigma$.*



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

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

$$(\rightarrow I)^{\mathrm{W}} \quad \frac{P \mid T(A_x, x{:}\alpha) \overset{\mathrm{W}}{\vdash} E \rightsquigarrow E' : \tau \qquad \alpha \text{ new}}{P \mid TA \overset{\mathrm{W}}{\vdash} \lambda x.E \rightsquigarrow \lambda x.E' : T\alpha \rightarrow \tau}$$

$$(let)^{\mathrm{W}} \quad \frac{v{:}P \mid TA \overset{\mathrm{W}}{\vdash} E \rightsquigarrow E' : \tau \qquad P' \mid T'(TA_x, x{:}\sigma) \overset{\mathrm{W}}{\vdash} F \rightsquigarrow F' : \tau' \qquad \sigma = Gen(TA, P \Rightarrow \tau)}{P' \mid T'TA \overset{\mathrm{W}}{\vdash} (\textbf{let } x = E \textbf{ in } F) \rightsquigarrow (\textbf{let } x = \lambda v.E' \textbf{ in } F) : \tau'}$$

Figure 8: Type inference algorithm with translation

7 Coherence results

Corollary 2 is important because it shows that any translation of an OML term E in a particular context can be written in the form $C(\lambda w.E')v$ where E' is a principal translation and C is the corresponding conversion. Applied to two arbitrary derivations $v : P \mid A \vdash E \rightsquigarrow E'_1 : \sigma$ and $v{:}P \mid A \vdash E \rightsquigarrow E'_2 : \sigma$, it follows that:

$$v{:}P \mid A \vdash E'_1 = C_1(\lambda w.E')v : \sigma \quad \text{and}$$
$$v{:}P \mid A \vdash E'_2 = C_2(\lambda w.E')v : \sigma$$

where C_1 and C_2 are conversions from the principal type scheme to $(P \mid \sigma)$. One obvious way to ensure that these translations are equal is to show that $C_1 = C_2$.

7.1 Equality of conversions

Taking a slightly more general view, suppose that C_1, C_2 are conversions from σ to $(P' \mid \sigma')$. Without loss of generality, we can assume that $\sigma = (\forall\alpha_i.Q \Rightarrow \nu)$ and $\sigma' = (\forall\alpha'_j.Q' \Rightarrow \nu')$ where the variables α'_j only appear in $(Q' \Rightarrow \nu')$. Using the definition of conversions, it follows that there are types τ_i such that $\nu' = [\tau_i/\alpha_i]\nu$, $C_1 = \lambda x.\lambda v'.\lambda w'.xef$ and

$$v' : P', w' : Q' \Vdash e : P, f : [\tau_i/\alpha_i]Q.$$

Similarly for C_2 there are types τ'_i such that $\nu' = [\tau'_i/\alpha_i]\nu$, $C_2 = \lambda x.\lambda v'.\lambda w'.xe'f'$ and

$$v' : P', w' : Q' \Vdash e' : P, f' : [\tau'_i/\alpha_i]Q.$$

Clearly, it is sufficient to show $e = e'$ and $f = f'$ to prove that the these two conversions are equivalent. The first equality is an immediate consequence of the uniqueness of evidence; both e and e' are evidence for the predicates P under the evidence assignment $v' : P', w' : Q'$ and so must be equal. The same argument cannot applied to the second equality since the predicates $[\tau_i/\alpha_i]Q$ may not be the same as those in $[\tau'_i/\alpha_i]Q$ due to differences between τ_i and τ'_i. Nevertheless, $[\tau_i/\alpha_i]\nu = \nu' = [\tau'_i/\alpha_i]\nu$, and so $\tau_i = \tau'_i$ for all $\alpha_i \in TV(\nu)$. Hence if $\{\alpha_i\} \cap TV(Q) \subseteq TV(\nu)$, then the two predicate sets $[\tau_i/\alpha_i]Q$ and $[\tau'_i/\alpha_i]Q$ will be equal and $f = f'$ as required. We will give a special name to type schemes with this property:

Definition 5 *A type scheme $\sigma = \forall\alpha_i.Q \Rightarrow \nu$ is* unambiguous *if $\{\alpha_i\} \cap TV(Q) \subseteq TV(\nu)$.*

The same concept of unambiguous type schemes is used in Haskell, motivating our use of the term here. The discussion above shows that all conversions from an unambiguous type scheme are equivalent:

Proposition 3 *If C_1, $C_2 : \sigma \geq (P' \mid \sigma')$ are conversions and σ is an unambiguous type scheme then $C_1 = C_2$.*

7.2 Equality of translations

As an immediate corollary, it follows that, if the principal type scheme for a term E is unambiguous, then any two translations of E must be equivalent:

Theorem 5 *If $v : P \mid A \vdash E \rightsquigarrow E'_1 : \sigma$ and $v : P \mid A \vdash E \rightsquigarrow E'_2 : \sigma$ and the principal type scheme of E in A is unambiguous, then $v{:}P \mid A \vdash E'_1 = E'_2 : \sigma$.*

This generalizes an earlier result by Blott [1] for the special case of the type system in [13].

Theorem 5 is easy to work with in concrete implementations. The first step in type-checking an OML program is to use the type inference algorithm to calculate its principal type and translation. If the program does not have a principal type, then it cannot be well-typed and will be rejected. If the principal type is not unambiguous, then we cannot guarantee a well-defined semantics and the program must again be rejected. For example, the principal type scheme of the term *out* (*in* x) in the example in Section 2 is $\forall a.C\ a \Rightarrow Int$ which is ambiguous and hence the program will not be accepted.

Note that Theorem 5 gives a condition that is sufficient, but not necessary, to guarantee coherence. For example, any attempt to compare the empty list with itself in Haskell by evaluating `[] == []` leads to an error since this term has an ambiguous principal type `Eq [a] => Bool`, even though it should evaluate to `True` for any choice of the type variable `a`. On the other hand, this cannot be established using the definition of equality in Section 3 and we might conjecture that the restriction to terms with unambiguous principal types is both necessary and sufficient to guarantee coherence with respect to that formulation of provable equality.

8 Related work

A number of researchers have investigated the coherence properties of particular type systems using a process of normalization of typing derivations. Examples of this include systems with explicit subtyping [2, 3], a form of implicit subtyping called *scaling* [12] and an earlier treatment of type classes [1]. The basic idea in each case is to give a collection of reduction rules and prove that they are confluent, that they preserve meaning and that any reduction sequence terminates (and hence, that the rules are strongly normalizing). The confluence property guarantees the existence of a unique normal form and the fact that meaning is preserved by reduction is then sufficient to guarantee coherence.

In the work described in this paper, the rules for reductions between terms in Section 3.2 correspond to reductions between derivations and the formulation of the syntax-directed system can be thought of as a means of identifying the 'normal forms' of a derivation. From this perspective, Theorem 2 can be interpreted as a proof that the reduction process terminates and that it preserves meaning. However, having shown that the coherence property does not hold in the general case (Section 2) we do not guarantee the existence of unique normal forms or confluence.

The most important and novel feature of our work is the use of conversions to give a semantic interpretation to to ordering between constrained type schemes. In effect, a conversion acts as a record of the way in which one derivation is reduced to another. Some of this information is lost because we do not distinguish between conversions that are provably equal but, as we have seen, we retain sufficient detail to establish useful conditions that guarantee coherence.

Our use of conversions is closely related to Mitchell's use of *retyping functions* in [9] to give minimal typings for a restricted set of terms in a version of the pure polymorphic λ-calculus. The flexibility of the language of types in the systems considered by Mitchell (essentially the same as those in OP but without qualified types) is largely responsible for the difficulty of extending this to a larger collection of terms. These problems have been avoided here by working with the OML type system which is based on a more restricted collection of type schemes.

One of the biggest limitations of our work is caused by the decision to include β-reduction in the definition of equality (Section 3.2). As an immediate consequence, the results in this paper cannot be applied to languages with call-by-value semantics. The same problem occurs in other work, including the coherence proof in [1]. One possibility would be to rework these results using an axiomatization of equality for call-by-value semantics such as that given by Riecke [11], but it would clearly be preferable to find a single formulation that can be used for both cases.

Another promising approach would be to use ideas from category theory as in [10] for a language with intersection types and subtyping, and in [5] for a system of type classes. One of the main attractions of the categorical approach from the theoretical standpoint is the increased generality resulting from a higher level of abstraction. The main benefit for practical work is likely to be the 'variable-free' approach which avoids some of the messy technical details involving free and bound variables. As mentioned in Section 4, our treatment of conversions has a strong categorical flavour and we would hope to be able to extend the techniques developed here to provide a more general treatment of coherence for qualified types.

References

[1] S.M. Blott. An approach to overloading with polymorphism. Ph.D. thesis, Department of computing science, University of Glasgow, July 1991 (draft version).

[2] V. Breazu-Tannen, T. Coquand, C.A. Gunter and A. Scedrov. Inheritance and coercion. In *IEEE Symposium on Logic in Computer Science*, 1989.

[3] P.-L. Curien and G. Ghelli. Coherence of subsumption. In *Fifteenth Colloquium on Trees in Algebra and Programming*. Springer Verlag LNCS 431, 1990.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *8th Annual ACM Symposium on Principles of Programming languages*, 1982.

[5] B. Hilken and D. Rhydeheard. Towards a categorical semantics of type classes. In *Theoretical aspects of computer software*. Springer Verlag LNCS 526, 1991.

[6] P. Hudak, S.L. Peyton Jones and P. Wadler (eds.). Report on the programming language Haskell, version 1.2. *ACM SIGPLAN notices*, 27, 5, May 1992.

[7] M.P. Jones. A theory of qualified types. In *European symposium on programming*. Springer Verlag LNCS 582, 1992.

[8] M.P. Jones. Qualified types: Theory and Practice. D. Phil. Thesis. Programming Research Group, Oxford University Computing Laboratory. July 1992.

[9] J.C. Mitchell, Polymorphic type inference and containment. In G. Huet (ed.), *Logical Foundations of Functional Programming*, Addison Wesley, 1990.

[10] J.C. Reynolds. The coherence of languages with intersection types. In *Theoretical aspects of computer software*. Springer Verlag LNCS 526, 1991.

[11] J.G. Riecke. A complete and decidable proof system for call-by-value equalities (preliminary report). In *17th International Colloquium on Automata, Languages and Programming*. Springer Verlag LNCS 443, 1990.

[12] S. Thatte. Type inference and implicit scaling. In *European Symposium on Programming*. Springer Verlag LNCS 432, 1990.

[13] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Principles of Programming Languages*, 1989.