ML typing, explicit polymorphism and qualified types

Mark P. Jones

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

Abstract. The ML type system was originally introduced as a means of identifying a class of terms in a simple untyped language, often referred to as core-ML, whose evaluation could be guaranteed not to "go wrong". In subsequent work, the terms of core-ML have also been viewed as a 'convenient shorthand' for programs in typed languages. Notable examples include studies of ML polymorphism and investigations of overloading, motivated by the use of type classes in Haskell.
In this paper, we show how *qualified types*, originally developed to study type class overloading, can be used to explore the relationship between core-ML programs and their translations in an explicitly typed language. Viewing these two distinct applications as instances of a single framework has obvious advantages; many of the results that have been established for one can also be applied to the other.
We concentrate particularly on the issue of *coherence*, establishing sufficient conditions to guarantee that all possible translations of a given core-ML term are equivalent. One of the key features of this work is the use of *conversions*, similar to Mitchell's *retyping functions*, to provide an interpretation of the ordering between type schemes in the target language.

1 Introduction

In his seminal paper on polymorphism in programming languages [14], Milner introduced the ML type system as a means of identifying a large class of programs in an untyped λ-calculus whose execution would not "go wrong". The language that Milner used, with only minor variations, has subsequently become known as core-ML and is widely used in studies of type inference.

A number of authors have shown how particular programming language features can be understood using type inference to guide a translation from core-ML into some suitably extended language. For example, Harper and Mitchell [5] have presented a semantics for ML polymorphism using core-XML, an explicitly typed version of core-ML that includes constructs for type abstraction and application, similar to those in the polymorphic λ-calculus.

Another example is the treatment of overloading proposed by Wadler and Blott [25] in which overloading is represented by introducing so-called dictionary abstractions and applications. This basic idea has been used in all current implementations of Haskell type classes [8] and has been widely studied [1, 9, 10, 12, 17, 18, 21, 23, 24].

This paper is motivated by the observation that both ideas can be dealt with in the same framework, allowing us to exploit the connections between the two systems so that results and ideas originally developed with one application in mind can also be applied to the other. More precisely, we show how qualified types, originally developed to to study type class overloading, can be used to investigate the translation of core-ML programs into a language that includes constructs for dealing with explicit polymorphism.

We begin with a brief review of core-ML in Section 2 and, in Section 3, show how the structure of typing derivations can be used to guide a translation from core-ML into the polymorphic λ-calculus. One particular problem is that a single term can have many different translations, each with potentially distinct semantics.

Our main aim is to give *coherence* conditions that can be used to identify a large class of core-ML terms for which all translations are guaranteed to be semantically equivalent. Section 4 shows how the type systems of core-ML and polymorphic λ-calculus can be extended to support qualified types and then Section 5 formalizes what it means for two translations to be equivalent. The following three sections extend the conventional type inference algorithm for core-ML to calculate *principal translations*. First, Section 6 extends the standard ordering between type schemes to the explicitly typed calculus. The algorithm itself is presented in Section 7 and used to establish the desired coherence results in Section 8.

Section 9 addresses another important issue, that of showing that the semantics of translated terms agrees with the intended semantics of the original programs.

2 A review of core-ML

We begin with a quick review of core-ML. This material is standard (see [4, 14] for example) but is included here to illustrate the notation used in the following sections. It also provides a summary of the most important tools used in the study of ML type inference, including in particular, the ordering on type schemes.

The terms of core-ML are those of simple untyped λ-calculus with the addition of a **let** construct to enable the definition and use of polymorphic terms:

$$E ::= x \mid \lambda x.E \mid E\,F \mid \textbf{let } x = E \textbf{ in } F.$$

The syntax for types distinguishes monotypes from (polymorphic) type schemes:

$$\begin{array}{ll} \tau ::= b \mid t \mid \tau \to \tau & \quad \textit{monotypes} \\ \sigma ::= \forall T.\tau & \quad \textit{type schemes} \end{array}$$

where b denotes a base type, t a type variable and T a finite set of type variables. For convenience, if $\sigma = \forall T.\tau$, then we write $\forall t.\sigma$ as an abbreviation for $\forall(T \cup$

$\{t\}).\tau$. Type schemes that are equivalent up to renaming of bound variables will be considered equal. The set of type variables appearing (free) in an expression X is denoted $TV(X)$ and is defined in the obvious way. In particular, $TV(\forall T.\tau) = TV(\tau) \setminus T$.

The typing rules will be described using type assignments, i.e. (finite) sets 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 maps x to σ. The complete typing rules for core-ML are given in Figure 1. Each judgement of the form $A \vdash E : \sigma$ is

$$(var)\ \frac{(x\!:\!\sigma) \in A}{A \vdash x : \sigma} \qquad (\forall E)\ \frac{A \vdash E : \forall\alpha.\sigma}{A \vdash E : [\tau/\alpha]\sigma}$$

$$(\to\! E)\ \frac{A \vdash E : \tau' \to \tau \quad A \vdash F : \tau'}{A \vdash EF : \tau} \qquad (\forall I)\ \frac{A \vdash E : \sigma \quad \alpha \notin TV(A)}{A \vdash E : \forall\alpha.\sigma}$$

$$(\to\! I)\ \frac{A_x, x\!:\!\tau' \vdash E : \tau}{A \vdash \lambda x.E : \tau' \to \tau} \qquad (let)\ \frac{A \vdash E : \sigma \quad A_x, x\!:\!\sigma \vdash F : \tau}{A \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) : \tau}$$

Fig. 1. Typing rules for core-ML.

called a *typing* and represents the assertion that, if the types of free variables are as specified by the type assignment A, then the expression E can be treated as having type σ.

As a first step to characterizing the set of types that can be assigned to a given term, we introduce an ordering $\leq$ on the set of type schemes.

Definition 1 *Suppose that $\sigma = \forall\alpha_i.\tau$, $\sigma' = \forall\beta_j.\tau'$ and that none of β_j appears free in τ (this last condition can always be satisfied by renaming bound variables in σ'). Then σ is* less general *than σ' (written $\sigma \leq \sigma'$) if there are types ν_i such that $\tau' = [\nu_i/\alpha_i]\tau$.*

It is easy to show that $\leq$ is reflexive, transitive and preserved by substitutions.

There is a standard procedure, based on Milner's algorithm W [14], which, given a term E and a type assignment A, calculates a type scheme σ such that $A \vdash E : \sigma$. (It is also possible for the algorithm to fail, in which case there are no derivable typings of this form). Furthermore, the inferred type scheme σ is *principal* (or *most general*) in the sense that, if $A \vdash E : \sigma'$ for some σ', then $\sigma' \leq \sigma$. In fact, we can strengthen this to show that $\sigma' \leq \sigma \Leftrightarrow A \vdash E : \sigma'$.

3 From core-ML to polymorphic $\boldsymbol{\lambda}$-calculus

As part of an attempt to describe the type structure of Standard ML, Harper and Mitchell [5] introduced an explicitly typed variant of core-ML called core-XML. Their intention was that any implicitly typed term in the former might be regarded as a convenient shorthand for an explicitly typed term in the latter.

Given a semantics for the target language, a translation process can also be used to study the meaning of programs in the source language. This section describes a similar process for translating core-ML into an explicitly typed language. Since our interest is more in the semantics of core-ML than its type structure, we will use the polymorphic λ-calculus, abbreviated here to PΛ, as the target language. The set of types in PΛ is given by:

$$\sigma ::= t \mid b \mid \sigma \to \sigma \mid \forall t.\sigma.$$

Notice that this allows polymorphic values to be used as the arguments and results of PΛ functions. This extra flexibility will be useful in subsequent sections, particularly Section 6.

The types of core-ML correspond to a subset of PΛ types, given by $\sigma' ::= \tau \mid \forall t.\sigma'$. For any σ' in this set, the corresponding type scheme will be written as $\mathit{scheme}(\sigma')$. For example, $\mathit{scheme}(\forall\alpha.\forall\beta.\alpha \to \beta) = \forall\{\alpha,\beta\}.\alpha \to \beta$.

The terms of PΛ are similar to those for core-ML except that λ abstractions and **let** constructs are annotated with types and additional constructs are included for type abstraction ($\lambda t.E$) and application ($E\,\tau$):

$$E ::= x \mid \lambda x\!:\!\tau.E \mid E\,F \mid \mathbf{let}\ x\!:\!\sigma = E\ \mathbf{in}\ F \mid \lambda t.E \mid E\,\tau$$

Strictly speaking, there is no need to include the **let** construct in the definition of PΛ because the type system is sufficiently powerful to allow us to encode any term of the form **let** $x : \sigma = E$ **in** F in the form $(\lambda x : \sigma.F)E$. On the other hand, including **let** makes it easier to maintain a direct correspondence between core-ML terms and their translations.

The typing rules for PΛ are given in Figure 2. Clearly, there is a strong correspondence between these rules and those in Figure 1. This can be captured formally using a function *Erase* mapping PΛ terms to core-ML terms:

$$\begin{aligned}
\mathit{Erase}\ (x) &= x \\
\mathit{Erase}\ (\lambda x\!:\!t.E) &= \lambda x.(\mathit{Erase}\ E) \\
&\ \vdots \\
\mathit{Erase}\ (Et) &= \mathit{Erase}\ E \\
\mathit{Erase}\ (\lambda t.E) &= \mathit{Erase}\ E
\end{aligned}$$

It is straightforward to show that, for any core-ML derivation of a typing $A \vdash E : \sigma$, there is a PΛ term E' such that $A \vdash E : \sigma'$ by a PΛ derivation of the same structure, $\mathit{scheme}(\sigma') = \sigma$ and $\mathit{Erase}\ E' = E$. We will describe the term E' as a

$$
\begin{array}{llll}
(var) & \dfrac{(x\!:\!\sigma) \in A}{A \vdash x : \sigma} & (let) & \dfrac{A \vdash E : \sigma \quad A_x, x\!:\!\sigma \vdash F : \sigma'}{A \vdash (\mathbf{let}\ x\!:\!\sigma = E\ \mathbf{in}\ F) : \sigma'} \\
(\to I) & \dfrac{A_x, x\!:\!\sigma' \vdash E : \sigma}{A \vdash \lambda x\!:\!\sigma'.E : \sigma' \to \sigma} & (\to E) & \dfrac{A \vdash E : \sigma' \to \sigma \quad A \vdash F : \sigma'}{A \vdash EF : \sigma} \\
(\forall I) & \dfrac{A \vdash E : \sigma \quad t \notin TV(A)}{A \vdash \lambda t.E : \forall t.\sigma} & (\forall E) & \dfrac{A \vdash E : \forall t.\sigma}{A \vdash E\tau : [\sigma'/t]\sigma}
\end{array}
$$

Fig. 2. Typing rules for polymorphic λ-calculus, PΛ.

translation of E and use the notation $A \vdash E \rightsquigarrow E' : \sigma$ to refer to a translation of a term in a specific context.

The relationship between the two type systems can be pictured by the following diagram.

$$\begin{tikzcd}
\text{core-ML derivations} \arrow[r] \arrow[d, "\phi"'] & \text{P}\Lambda\ \text{derivations} \arrow[d, "\psi"] \\
\text{core-ML typings} & \text{P}\Lambda\ \text{typings}
\end{tikzcd}$$

The two vertical arrows, ϕ and ψ, are functions mapping derivations to the typings that appears as their conclusion. As a result of the explicit type annotations, the structure of any PΛ derivation of a typing $A \vdash E' : \sigma$ is uniquely determined by the syntactic structure of E' and hence ψ is injective. However, a single core-ML typing can have many distinct derivations, so ϕ is certainly not injective. It follows that a core-ML term may have many distinct translations in a given context.

There are a number of conditions that need to be established to justify the use of translations as a semantics for core-ML terms:

- **Existence of translations**: For every derivable core-ML typing $A \vdash E : \sigma$, there is a translation E' such that $A \vdash E \rightsquigarrow E' : \sigma$. This follows directly from the definition of translations given above. Moreover, there is an effective way of calculating a translation using the structure of the typing derivation.
- **Soundness of translation**: Every translation of a well-typed core-ML term is well-typed in PΛ. This again follows directly from the definition of translations.
- **Agreement with expected semantics**: The semantics of translated terms should agree with the intended semantics of the original core-ML programs.

Of course, the ‘intended’ semantics may vary depending on the language in question. We discuss this further in Section 9.

- **Coherence of translation**: The mapping from terms to translations must be well-defined. More accurately, we must show that any translations E_1 and E_2 of a core-ML term E given by derivations $A \vdash E \leadsto E_1 : \sigma$ and $A \vdash E \leadsto E_2 : \sigma$ are semantically equivalent. Our use of the term *coherence* follows [2]; the meaning of a term should not depend on the way that it is type checked.

Unfortunately, it is relatively easy to find core-ML terms for which the coherence property does not hold. For example, consider the structure of the typing derivation for the term *foo bar* using the type assignment $A = \{\mathit{foo} : \forall t.t \to b,\ \mathit{bar} : \forall t.t\}$ that would be constructed by an implementation of core-ML type inference:

$$\dfrac{\dfrac{\dfrac{(\mathit{foo} : \forall t.t \to b) \in A}{A \vdash \mathit{foo} : \forall t.t \to b}\,(\mathit{var})}{A \vdash \mathit{foo} : t' \to b}\,(\forall E) \qquad \dfrac{\dfrac{(\mathit{bar} : \forall t.t) \in A}{A \vdash \mathit{bar} : \forall t.t}\,(\mathit{var})}{A \vdash \mathit{bar} : t'}\,(\forall E)}{A \vdash \mathit{foo}\ \mathit{bar} : b}\,(\to E)$$

From this derivation we obtain $A \vdash \mathit{foo}\ \mathit{bar} \leadsto (\mathit{foo}\ t')\,(\mathit{bar}\ t') : b$. Clearly, by replacing t' with distinct base types b_1 and b_2, we can obtain distinct translations $(\mathit{foo}\ b_1)\,(\mathit{bar}\ b_1)$ and $(\mathit{foo}\ b_2)\,(\mathit{bar}\ b_2)$ that are not equivalent.

There is a second reason for loss of coherence. In core-ML, we chose to use a set of variables in the syntax for type schemes, capturing the intuition that the ordering of the quantified variables is not significant. However, the ordering of quantified variables in PΛ is significant because it determines the ordering of type parameters and we would not expect two arbitrary terms of the form $\lambda\alpha.\lambda\beta.E$ and $\lambda\beta.\lambda\alpha.E$ to be equivalent.

Perhaps the coherence property is too strong? Is it really necessary to ensure that *all* translations of a term are equivalent? One alternative would be to choose a particular type-inference algorithm for core-ML that attempts to construct a derivation for a given typing and fails only if no such derivation exists. If, on the other hand, the algorithm succeeds then we can treat the PΛ term corresponding to the constructed derivation as a ‘canonical’ translation of the input term. We will not consider this approach any further here; it seems unnatural to base a definition of the semantics of a language on the almost arbitrary choice of a particular version of the type checking algorithm.

With the examples above, it is clear that we cannot establish a coherence property for all core-ML typings and translations. The best we can hope for is to identify a large class of terms for which coherence can be guaranteed. There are two reasons why the type systems of core-ML and PΛ are not suitable for this task:

- The typing rules do not provide enough information about the way that polymorphic types are instantiated within a derivation. For the example above, we might have expected the translation of *foo bar* to be $\lambda t'.(foo\ t')\,(bar\ t')$, capturing the free variable t'. However, the standard definition of ML type inference uses only the type of a term (not its translation) to determine when universal quantification (and hence, type abstraction in the translation) should be used. Since t' does not appear free in the inferred type b, no attempt will be made to quantify over this variable. (This is consistent with the logical interpretation of type schemes where we would expect $\forall t'.b$ to be equivalent to just b.)
- The typing rules do not provide enough of a distinction between quantification and type abstraction. This is particularly noticeable in the example above illustrating the importance of the ordering of type parameters.

In the following sections, we will present modified versions of core-ML and PΛ that include the additional information needed to avoid these problems.

As a reference to related work, we should mention that the coherence problem for the translation from core-ML to core-XML was first noted by Ohori [19] and the significance of free type variables in translated terms was commented on by Harper and Mitchell [5].

4 Translations using qualified types

A *qualified type* is a type of the form $\pi \Rightarrow \sigma$ denoting those instances of type σ which satisfy the *predicate* π. In previous work [9, 10], we have concentrated on the use of qualified types to explore the combination of overloading and polymorphism, with applications to Haskell type classes, extensible records and subtyping. In the following sections, we will show that they can also be used to establish sufficient conditions that guarantee coherence for a large class of core-ML terms.

4.1 Predicate systems

Different systems of predicates can be used to describe different applications. In each case, the properties of predicates can be captured by an entailment relation $\Vdash$ between finite sets of predicates. In fact, for the purposes of this work, we need to extend the definition of entailment to describe not only what is entailed, but also how. The basic idea is to require that a value of type $\pi \Rightarrow \sigma$ can only be used if we are supplied with *evidence* that the predicate π holds. We will assume that the definition of a system of predicates includes a language of evidence expressions e and evidence variables v. The entailment relation is expressed by judgements of the form:

$$v_1 : \pi_1,\ \ldots,\ v_n : \pi_n \Vdash e_1 : \pi'_1,\ \ldots,\ e_m : \pi'_m.$$

indicating that, if the variables v_i are bound to evidence values for the predicates π_i, then the evidence expressions e_j give evidence for the corresponding predicates π'_j. We will often abbreviate such entailments, writing them in the form $v : P \Vdash e : Q$ where v, e and P, Q are the appropriate sequences of evidence variables, evidence expressions and predicates respectively.

We refer to expressions of the form $v : P$ on the right of an entailment as *predicate assignments*. The empty predicate assignment is denoted $\emptyset$. For any predicate system, the definition of entailment is required to satisfy the rules in Figure 3. The variable S in rule (*close*) denotes an arbitrary substitution of types for type

$$
\begin{array}{llll}
(\mathit{id}) & v : P \Vdash v : P & (\mathit{term}) & v : P \Vdash \emptyset \\
(\mathit{fst}) & v : P, w : Q \Vdash v : P & (\mathit{snd}) & v : P, w : Q \Vdash w : Q \\
(\mathit{univ}) & \dfrac{v : P \Vdash e : Q \quad v : P \Vdash e' : R}{v : P \Vdash e : Q, e' : R} & (\mathit{trans}) & \dfrac{v : P \Vdash e : Q \quad v' : Q \Vdash e' : R}{v : P \Vdash [e/v']e' : R} \\
(\mathit{close}) & \dfrac{v : P \Vdash e : Q}{v : SP \Vdash e : SQ} & (\mathit{evars}) & \dfrac{v : P \Vdash e : Q}{EV(e) \subseteq v}
\end{array}
$$

Fig. 3. Predicate entailment with evidence.

variables, while the expression $EV(e)$ in rule (*evars*) denotes the collection of evidence variables appearing in the evidence expression e.

For the purposes of this paper, we will use the set of types τ as predicates with a collection of evidence expressions given by:

$$e ::= v \mid b \mid e \to e$$

together with the entailment relation given by extending the rules in Figure 3 with the axioms:

$$\emptyset \Vdash b : b \qquad v : \tau, w : \tau' \Vdash (v \to w) : (\tau \to \tau').$$

These are just the standard rules for type formation with evidence variables in place of type variables. We have used the same syntax for types and evidence expressions; in a practical implementation, evidence expressions would be used to construct and manipulate the concrete representations of types.

4.2 Extending core-ML and PΛ with predicates

In the following we will work with extensions of core-ML and PΛ that make use of a system of predicates. We refer to these languages as core-MLP and PΛP respectively (corresponding to OML and OP in [10]).

The terms of core-MLP are the same as those of core-ML, but the set of types includes not only types and type schemes, but also qualified types:

$$\begin{array}{ll} \tau ::= b \;\mid\; t \;\mid\; \tau \to \tau & \textit{monotypes} \\ \rho ::= \tau \;\mid\; \pi \Rightarrow \rho & \textit{qualified types} \\ \sigma ::= \forall T.\rho & \textit{type schemes} \end{array}$$

It follows from this grammar that function arguments and results in core-MLP cannot be polymorphic or qualified (in much the same way that the language of types in core-ML makes it impossible to use polymorphic functions as first-class values).

The language of terms in PΛP is essentially the same as those of PΛ except that we omit the type annotations for λ- and **let**-bound variables and replace the constructs for type abstraction and application with equivalent constructs for evidence values:

$$E ::= x \;\mid\; \lambda x.E \;\mid\; E\,F \;\mid\; \textbf{let}\ x = E\ \textbf{in}\ F \;\mid\; \lambda v.E \;\mid\; E\,e.$$

The set of PΛP types is a direct extension of the types in PΛ to include qualified types:

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

Rather than giving separate definitions for the type systems of core-MLP, PΛP, and the translation between them, the rules in Figure 4 define all three using judgements of the form $P \mid A \vdash E \leadsto E' : \sigma$. The first component, P, is a predicate assignment, E is a core-MLP term and E' is the corresponding PΛP translation. Ignoring the translations E', these rules describe the type system of core-MLP. Note the use of the symbols τ, ρ and σ to restrict the application of certain rules to particular kinds of type, according to the grammar for core-MLP types given above. On the other hand, if we disregard the core-MLP term in each judgement and ignore the distinction between different kinds of type expression, then the same rules define the type system of PΛP. Finally, since the rules identify terms in each language with typing derivations of the same structure, it follows that $P \mid A \vdash E \leadsto E' : \sigma$ is derivable if and only if E' is a translation of E in the sense of Section 3.

4.3 From core-ML to core-MLP

The rules in Figure 4 break the direct connection between universal quantification and type abstraction since there are different introduction and elimination rules for each. To understand how core-ML typing can be studied using the type system of core-MLP, we need to be able to map core-ML type schemes $\forall \alpha_i.\tau$ to core-MLP type schemes of the form $\forall \alpha_i.\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \tau$, forcing us to arrange the type parameters in some specific order. If these annotations are included in the types assigned to all free variables in a given term, then the typing rules in Figure 4 will ensure that information about the way that polymorphic types are instantiated will be propagated throughout the typing derivation.

Standard rules: (var)
$$\frac{(x : \sigma) \in A}{P \,|\, A \vdash x \leadsto x : \sigma}$$

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

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

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

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

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

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

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

Fig. 4. Typing and translation rules for core-MLP and PΛP.

As an aside, we mention that there may be some situations where it may be useful to be able to omit some quantified variables from the list of qualifying predicates. For example, we might use the type scheme $\forall t.t \to t$ for an implementation of the identity function that can be used for all types of values, while $\forall t.t \Rightarrow t \to t$ would be more appropriate in a system where different representations are used for different types of values and the implementation of the identity function truly does depend on the value that is assigned to t. We can capture this formally by requiring that any expression E whose type should unconditionally be reflected in the predicate assignment of a typing be replaced by *id* E where *id* is a predefined identifier representing the identity function with polymorphic type $\forall t.t \Rightarrow t \to t$.

5 A definition of equality for PΛP 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 PΛP 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 $P \,|\, A \vdash F : \sigma$). Our task for establishing coherence can then be described formally as:

> Given derivations $P \,|\, A \vdash E \leadsto E_1 : \sigma$ and $P \,|\, A \vdash E \leadsto E_2 : \sigma$, determine sufficient conditions to guarantee that $P \,|\, A \vdash E_1 = E_2 : \sigma$.

One reason for including type information as part of the definition of equality is to avoid making unnecessary constraints on the choice of semantic model. Given a judgement $P \,|\, A \vdash E = F : \sigma$ we require only that E and F have the same meaning (which must be an element of the type denoted by σ) in environments that satisfy P and A.

5.1 Uniqueness of evidence

Another reason for using predicate assignments in the definition is to enable 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 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:

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

This condition is essential if any degree of coherence is to be obtained and is central to establishing the coherence criteria given in Section 8. Defining the equality of evidence expressions and proving uniqueness of evidence is straightforward for the current application and we omit further details here.

5.2 Reduction of PΛP terms

In common with many treatments of typed λ-calculi, we will define the equality relation between terms using a notion of *reduction* between terms. More precisely, we use a judgement of the form $P \,|\, A \vdash E \rhd F : \sigma$ to describe a (typed) reduction from E to F with the implicit side condition that $P \,|\, A \vdash E : \sigma$. There is no need to include $P \,|\, A \vdash F : \sigma$ as a second side condition since it can be shown that this condition is implied by the first. This is a consequence of the *subject reduction theorem* – 'reduction preserves typing' – which is proved using standard techniques as in [7].

The most important rules in the definition of reduction are given in Figure 5, including definitions of β-conversion for both kinds of λ-abstractions and **let** expressions and a rule of η-conversion for evidence abstractions. One unfortunate consequence of our 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

$$\begin{array}{ll}
(\beta) & P\,|\,A \vdash (\lambda x.E)F \;\rhd [F/x]E : \sigma \\
(\beta_e) & P\,|\,A \vdash (\lambda v.E)e \;\rhd [e/v]E : \sigma \\
(\beta\text{-}let) & P\,|\,A \vdash (\mathbf{let}\ x = E\ \mathbf{in}\ F) \;\rhd [E/x]F : \sigma \\
(\eta_e) & \dfrac{v \notin EV(E)}{P\,|\,A \vdash (\lambda v.Ev) \;\rhd E : \sigma}
\end{array}$$

Fig. 5. Rules of computation

from the difficulty of axiomatizing call-by-value equality than from anything implicit in our particular application; for example, Ohori [19] mentions similar problems in his work to describe a simple semantics for ML Polymorphism.

Additional rules are needed to describe the renaming of bound variables (α-reduction) and structural properties (to allow reduction of subterms within a given term). These are standard and will be omitted from the presentation here.

5.3 Equalities between PΛP terms

As we have already mentioned, equalities between PΛP terms will be represented by judgements of the the form $P\,|\,A \vdash E = F : \sigma$ with the implicit side condition that both $P \mid A \vdash E : \sigma$ and $P \mid A \vdash F : \sigma$. Figure 6 gives the definition of the equality between terms as the transitive, symmetric closure of the reduction relation described in the previous section. The first two rules ensure that equality

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

$$\frac{P\,|\,A \vdash E = E' : \sigma \quad P\,|\,A \vdash E' = E'' : \sigma}{P\,|\,A \vdash E = E'' : \sigma}$$

$$\frac{P\,|\,A \vdash E \;\rhd F : \sigma}{P\,|\,A \vdash E = F : \sigma}$$

Fig. 6. Definition of equality between terms

is an equivalence relation. (There is no need to include reflexivity here since this is a direct consequence of the structural rules.) The last rule shows how reductions give rise to equalities. Note that in this case there is no need to establish that both $P \,|\, A \vdash E : \sigma$ and $P \,|\, A \vdash F : \sigma$ since the latter follows from the former by the subject reduction theorem mentioned above.

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:

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

Notice that the context in which this equality is established (given by P, A and σ) is not significant. 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 E and F are equivalent in the sense that $P \,|\, A \vdash E = F : \sigma$ for any choice of P, A and σ for which the implicit side conditions hold.

The above property of **let** expressions may seem a little unfamiliar, so it is worth illustrating how it can be useful in the work described here. Suppose that $g : \forall t. t \to t \to t$ and consider the core-MLP term:

$$\textbf{let}\ f = \lambda x. g\ x\ x\ \textbf{in}\ f\ 1$$

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

$$\textbf{let}\ f = \lambda x{:}Int. g\ Int\ x\ x\ \textbf{in}\ f\ 1$$

However, the type inference algorithm uses $\forall t. t \to t$ as the type for f and leads to a translation of the form:

$$\textbf{let}\ f = \lambda t. \lambda x{:}t. g\ t\ x\ x\ \textbf{in}\ f\ Int\ 1$$

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.

$$\begin{array}{lll} \vdash \textbf{let}\ f = \lambda t.\lambda x{:}t. g\ t\ x\ x\ \textbf{in}\ f\ Int\ 1 & & \\ \quad = \textbf{let}\ f = \lambda t.\lambda x{:}t. g\ t\ x\ x\ \textbf{in}\ [f\ Int/f](f\ 1) & (\text{substitution}) \\ \quad = \textbf{let}\ f = [\lambda t.\lambda x{:}t. g\ t\ x\ x/f](f\ Int)\ \textbf{in}\ (f\ 1) & (\text{by result above}) \\ \quad = \textbf{let}\ f = (\lambda t.\lambda x{:}t. g\ t\ x\ x)\ Int\ \textbf{in}\ (f\ 1) & (\text{substitution}) \\ \quad = \textbf{let}\ f = \lambda x{:}Int. g\ Int\ x\ x\ \textbf{in}\ (f\ 1) & (\beta) \end{array}$$

6 Ordering and conversion functions

The ordering $\leq$ is a central part of the treatment of core-ML type inference described in Section 2. used to express when one type is more general than another. For example, an ordering of the form $(b \to b) \leq (\forall t. t \to t)$ might be used to argue that, for the purposes of type inference, any term of type $b \to b$ can be replaced by a term with type $\forall t. t \to t$. This property does not extend to the explicitly typed language PΛ; if $f : b \to b$, $f' : \forall t.t \to t$ and $x : b$, then $f\ x$ has type b, but $f'\ x$ is not even well-typed! Of course, the correct approach is to replace f with $f'\ b$.

To understand the role of the $\leq$ ordering in the explicitly typed calculus, we will define a collection of terms for each $\sigma' \leq \sigma$, referred to as *conversions* from σ to σ'. Each conversion is a closed PΛP 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:

$$(\lambda x.xb) : (\forall t.t \to t) \to (b \to b).$$

Note that the type of this conversion (as in the general case) cannot be expressed as an core-MLP type scheme since it uses the richer structure of PΛP 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 $\sigma' \leq \sigma$ since CE has type σ' for any term E of type σ. 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 $\mathit{Erase}\ (CE) = \mathit{Erase}\ E$ for all PΛP terms E (or at least, all those occurring as translations of core-MLP terms). Since $\mathit{Erase}\ (CE) = (\mathit{Erase}\ C)\ (\mathit{Erase}\ E)$, the obvious way to ensure that this condition holds is to require that $\mathit{Erase}\ C$ is equivalent to the identity term $\mathit{id} = \lambda x.x$.

In previous work with qualified types, we have used the concept of a *constrained type scheme*, written as a pair $(P \,|\, \sigma)$, that captures both the type scheme σ for a term and the constraints P on the environments in which that typing can be used. In PΛP, the constrained type scheme $(P \,|\, \sigma)$ corresponds to the qualified type $P \Rightarrow \sigma$. The following definition can be used to describe 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 PΛP terms $C : (P \mid \sigma) \to (P' \mid \sigma')$ for which $\mathit{Erase}\ C$ is equivalent to id. In practice it is more convenient to choose a more conservative definition that gives a little more insight into the structure of conversions. The following definition is closely based on the syntactic characterization of the ordering between constrained type schemes given in [10] and based, in turn, on the definition of the ordering on core-ML type schemes presented in Section 2:

Definition 2 *Suppose that $\sigma = (\forall \alpha_i.Q \Rightarrow \tau)$ and $\sigma' = (\forall \beta_j.Q' \Rightarrow \tau')$ and that none of the variables β_j appear free in σ, P or P'. A closed PΛP term C of type $(P \,|\, \sigma) \to (P' \,|\, \sigma')$ such that Erase C is equivalent to id is called a* conversion *from $(P \mid \sigma)$ to $(P' \mid \sigma')$, written $C : (P \mid \sigma) \geq (P' \mid \sigma')$, if there are types τ_i, evidence variables v and w and evidence expressions e and f such that:*

$$v : P', w : Q' \Vdash e : P, f : [\tau_i/\alpha_i]Q, \quad \tau' = [\tau_i/\alpha_i]\tau \quad \textit{and} \quad \vdash C = \lambda x.\lambda v.\lambda w.xef.$$

As stated in Section 2, the ordering relation between core-ML type schemes is reflexive, transitive and preserved by substitutions. The corresponding results for conversions are given by the following proposition:

Proposition 3 *For all appropriate constrained type schemes:*

- $id : (P \mid \sigma) \geq (P \mid \sigma)$.
- *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* $C' \circ C = \lambda x.C'(Cx)$.
- *If* $C : (P \mid \sigma) \geq (P' \mid \sigma')$, *then* $C : S(P \mid \sigma) \geq S(P' \mid \sigma')$ *for any substitution* S *of types for type variables.*

Note that the concept of a *conversion* is closely related to that of Mitchell's *retyping functions*, as used in [16] to obtain minimal typings for a restricted set of terms in a version of the pure polymorphic λ-calculus.

7 Type inference and translation

The development of a type inference algorithm that can be used to calculate a principal type scheme for every core-MLP term was presented in [9]. In this section, we show that this can be extended to allow the calculation of translations, as described by the rules in Figure 7. These rules can be interpreted as an attribute grammar. The type assignment A and core-MLP term E in a judgement of the form $P \mid TA \overset{\mathrm{W}}{\vdash} E \leadsto E' : \tau$ are inherited attributes, while the predicate assignment P, substitution T, translation E' and type τ are synthesized. The notation $Gen(A, \rho)$ used in rule $(let)^{\mathrm{W}}$ represents the generalization of ρ with respect to A and is defined by: $Gen(A, \rho) = \forall(TV(\rho) \setminus TV(A)).\rho$.

The following theorem shows that any typing and translation that is obtained using the type inference algorithm can also be derived using the rules in Figure 4.

Theorem 4 (Soundness) *If* $P \mid TA \overset{\mathrm{W}}{\vdash} E \leadsto E' : \tau$, *then* $P \mid TA \vdash E \leadsto E' : \tau$.

Given that the algorithm described here calculates a principal type scheme for each well-typed core-MLP term, we will refer to the translations that it produces as *principal translations*. The following theorem provides further motivation for this terminology, showing that every translation can be expressed in terms of a principal translation.

Theorem 5 (Completeness) *If* $v : P \mid SA \vdash E \leadsto E' : \sigma$, *then there are* $w : Q$, T, E'' *and* ν *such that* $w : Q \mid TA \overset{\mathrm{W}}{\vdash} E \leadsto E'' : \nu$ *and there is a substitution* R *and a conversion* $C : RGen(TA, Q \Rightarrow \nu) \geq (P \mid \sigma)$ *such that* $S = RT$ *(except perhaps for new variables) and*

$$v : P \mid SA \vdash C(\lambda w.E'')v = E' : \sigma.$$

A detailed presentation of the type inference algorithm described in this section, with full proofs of Theorems 4 and 5, is included in [10].

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

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

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

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

Fig. 7. Type inference algorithm with translation

8 Coherence results

Theorem 5 is important because it shows that any translation of an core-MLP 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. For arbitrary derivations $v : P \,|\, A \vdash E \leadsto E_1' : \sigma$ and $v : P \,|\, A \vdash E \leadsto E_2' : \sigma$, it follows that:

$$v : P \,|\, A \vdash E_1' = C_1(\lambda w.E')v : \sigma \quad \text{and} \quad v : P \,|\, A \vdash E_2' = C_2(\lambda w.E')v : \sigma$$

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

8.1 Equality of conversions and translations

Taking a more slightly more general view, suppose that C_1, C_2 are conversions from $(P' \mid \sigma')$ 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_i only appear in $(Q \Rightarrow \nu)$. Using the definition of conversions, it follows that:

$$\nu = [\tau_j/\alpha_j']\nu' \quad \text{and} \quad v : P, w : Q \Vdash e : P', f : [\tau_j/\alpha_j']Q'$$

for some types τ_j and that $\vdash C_1 = \lambda x.\lambda v.\lambda w.xef$. Similarly for C_2 there are types τ_j' such that:

$$\nu = [\tau_j'/\alpha_j']\nu' \quad \text{and} \quad v : P, w : Q \Vdash e' : P', f' : [\tau_j'/\alpha_j']Q'$$

and $\vdash C_2 = \lambda x.\lambda v.\lambda w.xe'f'$. 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 hence must be equivalent. The same argument cannot be applied to the second equality since the predicates $[\tau_j/\alpha'_j]Q'$ may not be the same as those in $[\tau'_j/\alpha'_j]Q'$ due to differences between the types τ_j and τ'_j. Nevertheless, since $[\tau_j/\alpha'_j]\nu = \nu' = [\tau'_j/\alpha'_j]\nu$, it follows that $\tau_j = \tau'_j$ for all $\alpha'_j \in TV(\nu)$. Notice then that, if $\{\alpha'_j\} \cap TV(Q') \subseteq TV(\nu)$, the two predicate sets $[\tau_j/\alpha'_j]Q'$ and $[\tau'_j/\alpha'_j]Q'$ must be equal and hence $f = f'$ as required. We will give a special name to type schemes with this property:

Definition 6 (Unambiguous type schemes) *A type scheme $\sigma = \forall\alpha_i.Q \Rightarrow \nu$ is* unambiguous *if $\{\alpha_i\} \cap TV(Q) \subseteq TV(\nu)$.*

This definition coincides with that of an unambiguous type scheme in the treatment of type classes in Haskell, motivating our use of the same term here. Using this terminology, the discussion above shows that all conversions from an unambiguous type scheme to an arbitrary constrained type scheme are equivalent:

Proposition 7 *If C_1, $C_2 : (P \mid \sigma) \geq (P' \mid \sigma')$ are conversions and σ is an unambiguous type scheme then $\vdash C_1 = C_2$.*

As an immediate corollary, we obtain:

Theorem 8 *If $v : P \mid A \vdash E \leadsto E'_1 : \sigma$ and $v : P \mid A \vdash E \leadsto E'_2 : \sigma$ 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 system of type classes in [25]. To illustrate how this result can be used to detect incoherence in the translation from core-ML to PΛ, we recast the example from Section 3 in the type system of core-MLP, using the type assignment $A = \{foo : \forall t.t \Rightarrow t \to b,\ bar : \forall t.t \Rightarrow t\}$. The principal type of *foo bar* with respect to A is $\forall t.t \Rightarrow b$, which is ambiguous in the sense described above.

The restriction to unambiguous principal types is sufficient, but not necessary, to guarantee coherence. Thus we may be forced to reject some terms that have a well-defined meaning, despite the fact that they have an ambiguous principal type. For example, if we define $foo = \lambda t.\lambda x.b_0$ for some constant $b_0 : b$, then evaluating the two translations $(foo\ b_1)\,(bar\ b_1)$ and $(foo\ b_2)\,(bar\ b_2)$ will in fact produce the same result, b_0. Of course, this requires additional information about *foo*, not included in the original typing derivation. We conjecture that Theorem 8 might also be used as a necessary condition to establish coherence with respect to the definition of provable equality in Section 5 but we have not attempted to prove this.

8.2 Related work on coherence

Coherence results have been established for a number of different systems including those with subtyping [2, 3], intersection types [20], scaling [22] and type classes [1, 6, 10]. One standard approach used to establish results of this kind is to define a system of reduction rules on typing derivations. The required coherence property can then be established by showing that the reduction rules are strongly normalizing and preserve meaning, and that normal forms of derivations are uniquely determined by their conclusions.

Much of this could have been applied to the type systems described in this paper, but this would not have yielded a proof of coherence. The examples of incoherence show that there are typings that do not yield unique normal forms. 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.

9 Agreement with expected semantics

Consider the core-ML expression **let** $x = E$ **in** F and suppose that E has a polymorphic type. It follows that this term will have a translation of the form **let** $x = \lambda t.E'$ **in** F'. Assuming the semantics of Standard ML [15], these two programs may not have the same meaning. For example, if the evaluation of E diverges then the original term will also diverge but the translation may still converge if x is not used in F because the additional type abstraction for t is sufficient to delay the evaluation of E'. Alternatively, with a lazy evaluation strategy as in Haskell [8], the two terms will have the same denotational semantics, but the extra type parameter may cause a loss of sharing if x appears multiple times in F. These problems occur because the **let** construct, intended from the type theoretic perspective purely as a way of defining polymorphic values, is also used in practice for describing shared computation, sequencing (in a call-by-value language) and creating cyclic structures (in a call-by-name language). When extra parameters are required we must either compromise sharing (as in the translation above) or polymorphism (by using a translation of the form **let** $x{:}\sigma = [\tau/t]E'$ **in** F', restricting the opportunities for polymorphism).

Sharing is also important in languages where computations can cause side-effects. Motivated in particular by the desire to support references and first-class continuations in a language with a polymorphic type system, Leroy [13] has suggested a modification to the core-ML language that provides two distinct forms of the **let** construct, one for polymorphic definitions and one for shared (but monomorphic) definitions. In fact, there is no real need to extend the language since a monomorphic **let** expression can be treated as syntactic sugar for an expression of the form $(\lambda x.E)F$. The essence of Leroy's proposal is a change to the semantics of polymorphic **let** expressions, adding an extra (dummy) parameter in the definition of polymorphic values. This agrees with the semantics of core-ML given by the translations described in this paper.

The same ideas are already used in Haskell [8] as part of the infamous 'monomorphism restriction' (but the relationship between the two does not appear to be well-known). This again provides two syntactically distinct forms of **let** bindings, one of which allows overloading while the other does not, motivated by the desire to avoid a loss of sharing. The rules for distinguishing between the two are rather subtle, depending on the number of function arguments supplied and the presence of an explicit type signature. It may be preferable to make the distinction between the two forms of binding more explicit, making it easier for the programmer to anticipate where additional parameters may be inserted.

10 Conclusions

By viewing the translation from core-ML to PΛ as an application of qualified types we have been able to combine the results of previous work in each of these areas, recognizing related concepts such as the importance of coherence, and the similarity between Leroy's proposals for polymorphism by name and the monomorphism restriction in Haskell. The same unified view has already proved useful for other work, for example, in [11] where techniques from partial evaluation are suggested as an implementation of both polymorphism and overloading. We anticipate that this will also bring several other benefits and useful insights in future work.

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. R. Harper and J.C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15, 2, April 1993.
6. B. Hilken and D. Rhydeheard. Towards a categorical semantics of type classes. In *Theoretical aspects of computer software*. Springer Verlag LNCS 526, 1991.
7. J.R. Hindley and J.P. Seldin. *Introduction to combinators and λ-calculus*. London mathematical society student texts 1. Cambridge University Press, 1986.
8. 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.

9. M.P. Jones. A theory of qualified types. In *European symposium on programming.* Springer Verlag LNCS 582, 1992.
10. M.P. Jones. Qualified types: Theory and Practice. D. Phil. Thesis. Programming Research Group, Oxford University Computing Laboratory. July 1992.
11. M.P. Jones. From polymorphism to monomorphism by partial evaluation. Yale University, Department of Computer Science. Submitted for publication, July 1993.
12. S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *ACM Conference on LISP and functional programming* San Francisco, California, June 1992.
13. X. Leroy. Polymorphism by name for references and continuations. In *20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.
14. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 3, 1978.
15. R. Milner, M. Tofte and R. Harper. *The definition of Standard ML.* The MIT Press, 1990.
16. J.C. Mitchell, Polymorphic type inference and containment. In G. Huet (ed.), *Logical Foundations of Functional Programming*, Addison Wesley, 1990.
17. T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *5th ACM conference on Functional Programming Languages and Computer Architecture*, Cambridge, MA, August 1991. Lecture notes in computer science 523, Springer Verlag.
18. T. Nipkow and C. Prehofer. Type checking type classes. In *20th Annual Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.
19. A. Ohori. A simple semantics for ML polymorphism. In *4th International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London, September 1989. ACM Press.
20. J.C. Reynolds. The coherence of languages with intersection types. In *Theoretical aspects of computer software.* Springer Verlag LNCS 526, 1991.
21. G. Smith. Polymorphic type inference for languages with overloading and subtyping. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York. August 1991.
22. S. Thatte. Type inference and implicit scaling. In *European Symposium on Programming.* Springer Verlag LNCS 432, 1990.
23. S. Thatte. Typechecking with ad hoc polymorphism.Manuscript, Department of mathematics and computer science, Clarkson University, Potsdam, NY. May 1992.
24. D. Volpano and G. Smith. On the complexity of ML typability with overloading. In *5th ACM conference on Functional Programming Languages and Computer Architecture.* Lecture notes in computer science 523. Springer Verlag. 1991.
25. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Principles of Programming Languages*, 1989.

This article was processed using the LaTeX macro package with LLNCS style