QML: Explicit First-Class Polymorphism for ML

Claudio V. Russo
Microsoft Research Cambridge, UK
crusso@microsoft.com

Dimitrios Vytiniotis
Microsoft Research Cambridge, UK
dimitris@microsoft.com

Abstract

Recent years have seen a revival of interest in extending ML's predicative type inference system with impredicative quantification in the style of System F, for which type inference is undecidable. This paper suggests a modest extension of ML with System F types: the heart of the idea is to extend the language of types with unary universal and existential quantifiers. The introduction and elimination of a quantified type is never inferred but indicated *explicitly* by the programmer by supplying the quantified type itself. Quantified types co-exist with ordinary ML schemes, which are in turn *implicitly* introduced and eliminated at `let`-bindings and use sites, respectively. The resulting language, QML, does not impose any restriction on instantiating quantified variables with quantified types; neither `let`- nor λ-bound variables ever require a type annotation, even if the variable's inferred scheme or type involves quantified types. This proposal, albeit more verbose in terms of annotations than others, is simple to specify, implement, understand, and formalize.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—abstract data types, polymorphism

General Terms Languages, Theory

Keywords impredicativity, polymorphism, type inference

1. Introduction

Recent years have seen a revival of interest in extending ML's predicative type system (3), with impredicative quantification in the style of System F. Since type inference for Curry-style System F is known to be undecidable (25), many proposals have relied on various type annotation schemes, restrictions on the typing rules to make type inference possible, or even generalization of System F's type structure (16; 4; 19; 15; 23; 10; 9; 24; 11; 12; 18; 21).

Some of the recent approaches (23; 9; 24; 11) rely on sophisticated specifications and implementations. This paper suggests a simple approach to extending ML with System F universal *and* existential types – albeit more verbose in terms of type annotations when compared to recent work.

The heart of the idea is not fundamentally new, but appears interspersed in the older literature. Like O'Toole and Gifford (16), we take ordinary ML and extend the language of "monotypes" – over which type variables range – with quantified types. Like Garrigue and Rémy (4) we require explicit introduction of quantified types. However, elimination of quantified types is never inferred but also indicated explicitly by supplying the quantified type itself.

The idea of explicit introduction and elimination of polymorphism appears in previous work (6; 12; 20), typically mediated by introducing and eliminating new datatypes, to reduce type inference to simple structural unification. We refine this elegant idea by introducing (anonymous) quantified types that must still be explicitly introduced and eliminated. Hence we can dispense with auxiliary datatype definitions but our structural unification has to take quantifiers into account, using standard techniques (13; 17).

Finally, *as in ML*, polymorphic schemes (which are *not* the same as quantified types) are introduced implicitly via `let`-bound definitions; elimination of polymorphic schemes is always inferred.

Specifically, our contributions are:

- We present an ML-like language, QML, that combines implicit `let`-bound polymorphism with explicit, System F-style, universal and existential polymorphism (Sections 3, 3.1). QML supports partial type annotations that contain schematic type variables (Section 3.2). We show that typeability is robust under η-expansions and applications of polymorphic combinators (Section 3.3). We give a direct, call-by-value reduction semantics that preserves types (Section 3.4) and show that QML is as expressive as System F with existentials (Section 3.5).

- By design, we interpret schematic variables in annotations flexibly as locally scoped, logical metavariables (as in OCaml) rather than as implicitly scoped, rigid type parameters (as in Standard ML). The former interpretation is important to proving subject reduction (as originally observed by (4)) and key to capturing the expressivity of System F. Schematic variables also let programmers elide some type information from annotations.

- We give purely syntactic derived forms for introductions / eliminations of multiple quantifiers (*quantifier prefixes*) that collapse several annotations into one. We also provide sugar for other type annotations typically found in modern functional programming languages (Section 4).

- QML enjoys type inference by a standard reduction to first-order unification under a mixed prefix of quantifiers (Section 5).

- We discuss other features and design choices, and place our proposal in the design space of type inference in the presence of impredicative polymorphism (Sections 6,7,8).

This paper is accompanied by a Coq formalization of most of the results and a prototype Caml light implementation, available from `http://research.microsoft.com/~crusso/qml`. The Coq formalization is based on existing proof infrastructure (1), extending a sample formalization of Pure ML.

ML'09, August 30, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-509-3/09/08. . . $5.00

2. Programming with explicit quantifiers

Central to our proposal is the distinction between polymorphic schemes and (potentially quantified) types.

Schemes	ς	::=	$\Pi(\overline{\alpha})(\tau)$
Types	τ, μ	::=	$\forall\alpha.\tau \mid \exists\alpha.\tau \mid \tau \to \tau \mid \alpha \mid \tau \times \tau \mid \tau\ \mathit{list}$
			$\mid \mathit{int} \mid \mathit{bool} \mid \ldots$

Schemes bind a list of quantified variables $\overline{\alpha}$ in their body τ. Types are ordinary (potentially quantified) System F types.

Expressions that are `let`-bound, get assigned (as in ML) schemes, which can be instantiated freely in the body of the `let`-definition. For example:

```
let id x = x;; % inferred Π(α)(α → α)
(id 3, id false);;
```

By contrast, we have to explicitly introduce and eliminate any other form of quantifiers by providing the quantified type itself. For example, the first-class polymorphic identity is defined as:

```
let pid = {∀α.α → α} id;;
```

The variable `pid` receives scheme $\Pi(\epsilon)(\forall\alpha.\alpha \to \alpha)$ (ϵ means that there are no polymorphic variables in the scheme). To eliminate `pid`'s $\forall$ type, we must explicitly provide the quantified type:

```
let test = (pid {∀α.α → α}) 3 ;;
```

Notice that the programmer did not give the instantiation (as she would give when programming in System F) but rather the polymorphic type itself – a point that we return to in Section 7.

To a first approximation[1] the introduction and elimination of universally quantified types is governed by these rules:

$$\frac{\Gamma \vdash e : \tau \quad \beta \# \mathit{ftv}(\Gamma, e)}{\Gamma \vdash \{\forall\beta.\tau\} e : \forall\beta.\tau}\ \forall\text{I} \qquad \frac{\Gamma \vdash e : \forall\beta.\tau}{\Gamma \vdash e\ \{\forall\beta.\tau\} : [\beta \mapsto \mu]\tau}\ \forall\text{E}$$

Rule $\forall$I introduces a specified universal quantifier and $\forall$E eliminates a specified universal quantifier by instantiating it with some unspecified type.

A function parameter receives a type, not a scheme, so it is impossible to use the parameter at two different types in the body of the function, as the following example demonstrates.

```
let poly f = (f 1, f true);; % rejected
```

Here, `f` has to be used at two different types, which is not possible in both ML and QML. Rejecting this program is good: we do not want to be left "guessing" the polymorphism of function parameters that are used at two or more types as this results in losing the principal types property (see, for instance, the related discussion in (23)); similar non-determinism is the source of undecidability of type inference for System F (25; 7).

On the other hand, if we explicitly eliminate a parameter, we may as well use it at two different types:

```
let poly f = ((f {∀α.α → α}) 1,
              (f {∀α.α → α}) true);;
```

To avoid code duplication, we may `let`-bind the eliminated parameter, as follows:

```
let poly f = let y = f {∀α.α → α}
             in (y 1, y true);; % accepted
```

[1] In this section, we only consider closed type annotations – full QML allows partial type annotations with schematic type variables (Section 3).

In the previous examples, `poly` gets type

$$(\forall\alpha.\alpha \to \alpha) \to \mathit{int} \times \mathit{bool}$$

Finally, our system supports derived forms that have the effect of automatically eliminating polymorphic parameters that are annotated *at their introduction sites* (Section 4). But this is syntactic sugar, not built-in to the type system or algorithm.

Our system never infers introductions of quantifiers, as the following examples show:

```
app : Π(α β)((α→ β) → α→ β)
revapp : Π(α β)(α → (α→ β) → β)

% rejected
let test0 = app poly (fun x → x);;
% passes
let test1 = app poly ({∀α.α → α} fun x → x);;
% passes
let test2 = revapp ({∀α.α → α} fun x → x) poly;;
```

In the above code, `test0` fails, because **fun** `x → x` does not have a first-class quantified type. On the other hand, `test1` succeeds because we have explicitly introduced the quantified type for **fun** `x → x`. Similarly for `test2`.

Via the addition of type annotations, the programmer may choose between several incomparable System F types for an expression.

```
single : Π(α)(α → (α list))

% inferred Π()((∀α.α → α) list)
let ids = single ({∀α.α → α} fun x → x);;
% inferred Π(α)((α → α) list)
let ids0 = single (fun x → x);;
```

Interestingly, `ids0` receives the same type as it would in ML – our system is a conservative extension of the ML type system.

Finally, because quantified types are never implicitly instantiated, we may pass expressions with quantified types around freely, and instantiate the scheme variables of polymorphic combinators (such as `map` and `app`) with quantified types, as shown below:

```
let idss = single ids;; % ok
let test1 = map head (single ids);; % ok
let test2 = app (map head) (single ids);; % ok
```

2.1 Existential types

The story is similar for existential types. Existential types must be explicitly introduced; they are explicitly eliminated using a *scoped* `open` syntax. As usual, existential witnesses are treated as fresh parameters that must not escape the scope of the `open`. For example, consider this existential type for a counter object (a triple of a value, increment and query function):

$$\exists\alpha.\ \alpha\times(\alpha \to \alpha)\times(\alpha \to \mathtt{int})$$

We can introduce and eliminate this existential type as follows:

```
% introduce an existential package ...
let f = {∃α.α×(α → α)×(α → int)}
          (0, (fun x → x + 1) , (fun x → x));;

% ... and eliminate it
open {∃α. α×(α → α)×(α → int)} g = f in
  (snd (snd g)) (fst g);;
```

Terms

$$\begin{array}{llll}
e, u & ::= & x \mid \lambda x \,.\, e \mid e_1\ e_2 & \\
 & \mid & \texttt{let}\ x = u\ \texttt{in}\ e & \\
 & \mid & \{\overline{\alpha}\ \forall\beta.\tau\}\ e & \forall\text{-introduction} \\
 & \mid & e\ \{\overline{\alpha}\ \forall\beta.\tau\} & \forall\text{-elimination} \\
 & \mid & \{\overline{\alpha}\ \exists\beta.\tau\}\ e & \exists\text{-introduction} \\
 & \mid & \texttt{open}\ \{\overline{\alpha}\ \exists\beta.\tau\}\ x = u\ \texttt{in}\ e & \exists\text{-elimination}
\end{array}$$

$$\begin{array}{llll}
\text{Schemes} & \varsigma & ::= & \Pi(\overline{\alpha})(\tau) \\
\text{Types} & \tau, \mu & ::= & \forall\alpha.\tau \mid \exists\alpha.\tau \mid \tau \to \tau \mid \alpha \mid \ldots
\end{array}$$

$$\begin{array}{llll}
\text{Contexts} & \Gamma & ::= & \epsilon \mid \Gamma, (x{:}\varsigma)
\end{array}$$

$$\begin{array}{lrcl}
\text{Abbreviations} & \overline{\alpha} & \equiv & \alpha_1 \cdots \alpha_n (n \geq 0) \\
 & \epsilon & \equiv & \text{empty sequence} \\
 & \Pi(\epsilon)(\tau) & \equiv & \tau \\
 & \{\forall\beta.\tau\} & \equiv & \{\mathit{ftv}(\forall\beta.\tau)\ \forall\beta.\tau\} \\
 & \{\exists\beta.\tau\} & \equiv & \{\mathit{ftv}(\exists\beta.\tau)\ \exists\beta.\tau\}
\end{array}$$

Church-style System F

$$\begin{array}{lll}
M, N ::= & x \mid \lambda x{:}\tau.M \mid M\ N \\
\mid & \Lambda\alpha.M \mid M\ [\tau] \\
\mid & \texttt{pack}(\tau, M)\ \texttt{as}\ \exists\alpha.\mu \mid \texttt{open}\ (\alpha, x) = M\ \texttt{in}\ N
\end{array}$$

Figure 1: Syntax.

$$\frac{(x{:}\Pi(\overline{\alpha})(\tau)) \in \Gamma}{\Gamma \vdash x : [\overline{\alpha \mapsto \mu}]\tau}\ \textsc{Var} \qquad \frac{\Gamma, (x{:}\tau) \vdash e : \mu}{\Gamma \vdash \lambda x \,.\, e : \tau \to \mu}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \mu \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \mu}\ \textsc{App} \qquad \frac{\Gamma \vdash u : \tau \quad \overline{\alpha}\#\mathit{ftv}(\Gamma, u) \quad \Gamma, (x{:}\Pi(\overline{\alpha})(\tau)) \vdash e : \mu}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ e : \mu}\ \textsc{Let}$$

$$\frac{\Gamma \vdash e : [\overline{\alpha \mapsto \mu}]\tau \quad \beta\#\overline{\alpha}, \mathit{ftv}(\Gamma, \overline{\mu}, e)}{\Gamma \vdash \{\overline{\alpha}\ \forall\beta.\tau\}\ e : [\overline{\alpha \mapsto \mu}]\forall\beta.\tau}\ \textsc{AllI}$$

$$\frac{\Gamma \vdash e : [\overline{\alpha \mapsto \mu}]\forall\beta.\tau}{\Gamma \vdash e\ \{\overline{\alpha}\ \forall\beta.\tau\} : [\overline{\alpha \mapsto \mu}, \beta \mapsto \mu]\tau}\ \textsc{AllE}$$

$$\frac{\Gamma \vdash e : [\overline{\alpha \mapsto \mu}, \beta \mapsto \mu]\tau}{\Gamma \vdash \{\overline{\alpha}\ \exists\beta.\tau\}\ e : [\overline{\alpha \mapsto \mu}]\exists\beta.\tau}\ \textsc{SomeI}$$

$$\frac{\Gamma \vdash u : [\overline{\alpha \mapsto \mu}]\exists\beta.\tau \quad \beta\#\overline{\alpha}, \mathit{ftv}(\Gamma, \overline{\mu}, \mu, e) \quad \Gamma, (x{:}[\overline{\alpha \mapsto \mu}]\tau) \vdash e : \mu}{\Gamma \vdash \texttt{open}\ \{\overline{\alpha}\ \exists\beta.\tau\}\ x = u\ \texttt{in}\ e : \mu}\ \textsc{SomeE}$$

Figure 2: Typing relation $\Gamma \vdash e : \tau$.

Values of existential types are first-class (they may be passed around and stored in data structures) and existential types may be used to instantiate type parameters. Unlike in System F, the witness for an existential is never specified, but inferred. We defer further and more substantial examples with existentials to Section 4.

3. The language

Figure 1 defines the syntax of QML. As we have seen, types are just System F types. A type scheme is a type prefixed with an n-ary, (possibly empty) universal quantifier. We write Π and $\forall$ to distinguish universal quantification in schemes and types. Type variables α, β range over *all* types, including quantified types (but not schemes).

Terms include the familiar ML syntactic forms: variables, abstractions, applications and `let` expressions. Notice that abstractions do not carry annotations for their arguments as argument types (even quantified) will be inferred.

Terms include introduction and elimination forms for universal and existential quantifiers. Compared to the examples in the introduction, these forms also include a list of variables $\overline{\alpha}$, which are bound in the quantified type annotation ($\forall\beta.\tau$) or ($\exists\beta.\tau$). For both introduction and elimination terms, the schematic variables in $\overline{\alpha}$ have flexible interpretations as placeholders for types. In general, each typing rule will introduce or eliminate some (inferred) instance $[\overline{\alpha \mapsto \mu}]\forall\beta.\tau$ or $[\overline{\alpha \mapsto \mu}]\exists\beta.\tau$ [2]. The $\overline{\alpha}$ are borrowed from and serve the same rôle as Garrigue and Rémy's (4) $\exists$-bound type variables in their syntax of type annotations. They provide the same advantages: stability under substitution - useful for subject reduction - and implicit reference to otherwise inexpressible types.

Finally, although it is tempting to bind the type parameter β in e (in $\forall$-introductions and $\exists$-eliminations) - in order to allow easy reference to β from inner annotations - it is, in fact, crucial to the definition of our reduction relation and the proof of type preservation that type variables are never bound in terms.

[2] The OCaml programmer might recognize this as the flexible interpretation of type variables appearing in type constraints.

3.1 Typing

Figure 2 defines the declarative typing relation of QML. Rules VAR, ABS, APP and LET are completely standard. Note that we place no restriction on type instantiation nor on the type inferred for λ-bound variables. A minor deviation is the side condition that $\overline{\alpha}\#\mathit{ftv}(u)$, which prevents us from generalizing over the (syntactically) free type variables in a `let`-bound term. The side-condition is vacuous in core ML (since terms have no type annotations) and imposed just to match our formalization, which, for better or worse, allows annotations to contain free variables apart from $\overline{\alpha}$ and β. Such variables are essentially treated as parameters; preventing their generalization ensures we will never have to substitute for them during reduction. (We originally chose to allow free variables simply to make it easier to extend QML with base types represented as free type variables in some initial context.) Since we have a Coq formalization of QML, we allow ourselves some informality in eliding side-conditions on term variables entering the context.

Rule ALLI introduces some instantiation $[\overline{\alpha \mapsto \mu}]$ of the universal type $\forall\beta.\tau$, provided we can show e has type τ modulo the same instantiation for a suitably fresh parameter β.

Rule ALLE eliminates some instantiation $[\overline{\alpha \mapsto \mu}]$ of the universal type $\forall\beta.\tau$, provided we can show e has type $\forall\beta.\tau$ modulo the same instantiation. The actual instantiation of β is chosen non-deterministically, just as in rule VAR.

Rule SOMEI introduces some instantiation $[\overline{\alpha \mapsto \mu}]$ of existential type $\exists\beta.\tau$, provided we can show e has type τ modulo the same instantiation extended with a witnessing type μ for β. The witness for β, i.e. the representation for the abstract type, is chosen non-deterministically, just as in rule ALLE.

Rule SOMEE eliminates some instantiation $[\overline{\alpha \mapsto \mu}]$ of existential type $\exists\beta.\tau$, provided we can show, for a suitably fresh witness β, that the client continuation e has type μ, in the context extended with a binding of x of type τ (modulo the same instantiation). As usual for existential elimination, the type of the continuation must not depend on β, enforced by the side-condition $\beta \# \mu$.

Note that Rules ALLI and SOMEE, like rule LET, employ a side condition to prevent the accidental capture of any free type variables in the body or continuation e. Taken together, these conditions mean the reduction semantics need never substitute types in terms. Indeed, if the reduction semantics did require type substitution then we would be faced with the awkward problem of choosing which types to substitute - the type arguments to variables, universal elimination and existential introduction are implicit and thus not readily available during reduction.

Each quantifier rule is applied modulo some instantiation $[\overline{\alpha \mapsto \mu}]$ of the schematic variables $\overline{\alpha}$. Our side-conditions ensure that $\overline{\mu}$ (and in rule SOMEE, μ) cannot depend on the quantified variable β, so that any equational constraints on $\overline{\mu}$ may be solved using first-order unification. The only higher-order constraints placed on types, notably the dependency of τ on β, are explicitly resolved by the syntax (module the choice of types $\overline{\mu}$ that, by construction, cannot depend on β).

Unlike other proposals, none of our rules have infinitary premises stipulating principal types for subterms. Indeed, if we ignore the rules for explicitly introducing and eliminating universal and existential polymorphism, the typing relation is identical to the ML typing relation, though types are richer. Let $\vdash^{\flat}$ be the ML typing relation, consisting only of rules VAR, ABS, APP, and LET in Figure 2, and where types are allowed to be quantifier-free only. Then it is an easy observation that:
Lemma 3.1. *If* $\Gamma \vdash^{\flat} e : \tau$ *then* $\Gamma \vdash e : \tau$.

One may wonder about the converse – is every program typeable with the above rules (potentially using impredicative instantiation), typeable in ML (perhaps with a different type)? We claim the answer is yes, but the proof is more involved and left to future work.

3.2 The need for partial type annotations

The schematic variables supported by our annotations are not required to prove type soundness or the correctness of type inference.

However, the simpler system with closed type annotations (setting $\overline{\alpha} \equiv \epsilon$ throughout) is *less* expressive than System F. For instance, it is not possible to define the classic encodings of existential types using universal types. In particular, the standard encoding of existential introduction uses an inner type application to eliminate a universal type that mentions a type parameter of an outer type abstraction. Without explicit type binding, we cannot reference this parameter explicitly and in the absence of schematic variables, we cannot reference it implicitly either.

As a concrete example, consider the encoding of the System F term $\mathtt{pack}(\mathtt{int}, 1)\ \mathtt{as}\ \exists\beta.\beta$,

$$\Lambda\gamma.\lambda f{:}\forall\beta.\beta \to \gamma.f[\mathtt{int}]1$$

of encoded existential type $\forall\gamma.(\forall\beta.\beta \to \gamma) \to \gamma$. Notice that the declared type of f - the client of the existential - mentions γ. In the QML translation, we have to specify the universal type of f at its elimination. Although it cannot refer to the intended γ (since it is not in scope), it can refer to it implicitly via a schematic variable α.

$$\{\ \forall\gamma.(\forall\beta.\beta \to \gamma) \to \gamma\}\ \lambda f.(f\ \{\alpha\ \forall\beta.\beta \to \alpha\})\ 1$$

Without the schematic variable, we would not be able to encode the original System F term.

Note that a different approach to partial type annotations would be to treat free variables as "rigid", and bind them in terms as well. This approach is taken in Standard ML and the Glasgow Haskell Compiler. Although feasible, it would complicate our reduction semantics and the subject reduction proof (for example, GHC does *not* have a direct reduction semantics, but rather a semantics-via-elaboration to Church-style System F; SML(14) erases type annotations before untyped evaluation).

Fortunately, QML's approach to partial type annotations is not new and has been proposed and adopted several times (4; 10; 21).

3.3 Robustness

Because all introductions and eliminations of polymorphism are explicit, QML enjoys robustness of typeability under various program transformations. First, η-expansions preserve typing:
Lemma 3.2. *If* $\Gamma \vdash e : \tau_1 \to \tau_2$ *then* $\Gamma \vdash \lambda x.e\ x : \tau_1 \to \tau_2$.

Second, typing is preserved under applications of polymorphic combinators.
Lemma 3.3. *If* $\Gamma \vdash e_1\ e_2 : \mu$ *then also* $\Gamma \vdash \mathtt{app}\ e_1\ e_2 : \mu$ *and* $\Gamma \vdash \mathtt{revapp}\ e_2\ e_1 : \mu$.

Both properties are straightforward checks using the typing rules. Of course, there is a catch: there are fewer System F applications that may be typed without an annotation on the argument at the first place. For instance:

```
run : Π(α)((∀γ.γ→ α)→ α)
e : Π(γ)(γ → int)

let test0 = run e;; % rejected
let test1 = run ({∀γ.γ→ α} e);; % ok
```

Finally, typeability is preserved under β-reductions (Section 3.4), and `let`-expansions as a consequence of the existence of principal schemes (Section 5).

3.4 Reduction semantics

We give a small-step operational semantics as a reduction relation on closed terms (Figure 3). Values are either λ-abstractions, which suspend reduction, or quantifier introductions, which do not:

$$\mathtt{Values}\quad v, w \quad ::= \quad \lambda x\,.\,e \mid \{\overline{\alpha}\ \forall\alpha.\tau\}\ v \mid \{\overline{\alpha}\ \exists\alpha.\tau\}\ v$$

This choice ensures compatibility with type-erasure semantics, but is not essential. Since we choose to reduce under quantifier introductions, the body of a polymorphic or existential value must also be a value.

The reduction rules for applications and `let` are standard.

Contextual rules R-ALLI and R-SOMEI reduce the body of polymorphic or existential values, preserving the outer type annotation. Rule R-ALLE-β, where a polymorphic value is eliminated, just exposes the underlying value. Unlike the corresponding β reduction rule of System F, this rule does not substitute for α - there is no need, since α is only bound in τ, not v.

Contextual rule R-SOMEE reduces the existential term unless it is already a value. Rule R-SOMEE-β, where an existential value is eliminated, just substitutes the underlying value for x in the suspended continuation e. Unlike the corresponding β reduction rule of System F, this rule does not substitute for α – again, α is only bound in τ, not e.

Since we reduce under quantifier introductions and the β-rules for quantifiers never relate the annotation on the value with that of the elimination construct, it is easy to see that QML enjoys an obvious type erasure semantics.

The actual type soundness result is a consequence of the following standard lemmas. The proofs are easy and omitted (but see the Coq formalization for details).

$$\frac{e \longrightarrow e'}{e\ u \longrightarrow e'\ u}\ \text{R-APP-L} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}\ \text{R-APP-R} \qquad \frac{}{(\lambda x\,.\,e)v \longrightarrow [x \mapsto v]e}\ \text{R-APP-}\beta$$

$$\frac{u \longrightarrow u'}{\texttt{let}\ x = u\ \texttt{in}\ e \longrightarrow \texttt{let}\ x = u'\ \texttt{in}\ e}\ \text{R-LET} \qquad \frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow [x \mapsto v]e}\ \text{R-LET-}\beta$$

$$\frac{e \longrightarrow e'}{\{\overline{\alpha}\ \forall\alpha.\tau\}\ e \longrightarrow \{\overline{\alpha}\ \forall\alpha.\tau\}\ e'}\ \text{R-ALLI} \qquad \frac{e \longrightarrow e'}{e\ \{\overline{\alpha}\ \forall\beta.\tau\} \longrightarrow e'\ \{\overline{\alpha}\ \forall\beta.\tau\}}\ \text{R-ALLE} \qquad \frac{}{(\{\overline{\alpha}\ \forall\alpha.\tau\}\ v)\ \{\overline{\beta}\ \forall\beta.\mu\} \longrightarrow v}\ \text{R-ALLE-}\beta$$

$$\frac{e \longrightarrow e'}{\{\overline{\alpha}\ \exists\alpha.\tau\}\ e \longrightarrow \{\overline{\alpha}\ \exists\alpha.\tau\}\ e'}\ \text{R-SOMEI} \qquad \frac{u \longrightarrow u'}{\texttt{open}\ \{\overline{\alpha}\ \exists\beta.\tau\}\ x = u\ \texttt{in}\ e \longrightarrow \texttt{open}\ \{\overline{\alpha}\ \exists\beta.\tau\}\ x = u'\ \texttt{in}\ e}\ \text{R-SOMEE}$$

$$\frac{}{\texttt{open}\ \{\overline{\alpha}\ \exists\alpha.\tau\}\ x = (\{\overline{\beta}\ \exists\beta.\mu\}\ v)\ \texttt{in}\ e \longrightarrow [x \mapsto v]e}\ \text{R-SOMEE-}\beta$$

Figure 3: Reduction relation $e \longrightarrow u$.

Lemma 3.4 ((Parallel) Type Substitution)**.** *If* $\Gamma_1, \Gamma_2 \vdash e : \tau$ *and* $\overline{\alpha}\#\mathit{ftv}(\Gamma_1)$ *then* $\Gamma_1, [\overline{\alpha \mapsto \mu}]\Gamma_2 \vdash [\overline{\alpha \mapsto \mu}]e : [\overline{\alpha \mapsto \mu}]\tau$.

Notice that, because we allow type annotations with free variables (in addition to schematic ones), our substitution lemma needs to substitute in the term as well as the type. Consider for example a term of the form $\{\overline{\alpha}\ \forall\beta.\tau\}\ e$. There may be free variables in $\forall\beta.\tau$ other than $\overline{\alpha}$. If we were to require that $\overline{\alpha}$ is *all* the free variables in $\forall\beta.\tau$, then we would not have to substitute inside the term. Allowing free variables does no harm – but is not essential.

Lemma 3.5 (Weakening)**.** *If* $\Gamma_1, \Gamma_2 \vdash e : \tau$ *implies* $\Gamma_1, \Gamma, \Gamma_2 \vdash e : \tau$ *(provided* $\Gamma_1, \Gamma, \Gamma_2$ *is a well-formed context).*

To state a generalization and a term substitution property, we define the auxiliary scheme typing judgement, below.

Definition 3.1 (Scheme Typing)**.** For scheme $\varsigma \equiv \Pi(\overline{\alpha})(\tau)$, define the relation $\Gamma \vdash e : \varsigma$, read e *has scheme* ς in Γ, if, and only if, forall $\overline{\mu}$, $\Gamma \vdash e : [\overline{\alpha \mapsto \mu}]\tau$.

Definition 3.2 (Generalization)**.** $\Gamma \vdash e : \tau$ and $\overline{\alpha}\#\mathit{ftv}(\Gamma)$ implies $\Gamma \vdash e : \Pi(\overline{\alpha})(\tau)$.

Lemma 3.6 (Term Substitution)**.** *If* $\Gamma_1, (x : \varsigma), \Gamma_2 \vdash e : \tau$ *and* $\Gamma_1 \vdash u : \varsigma$ *then* $\Gamma_1, \Gamma_2 \vdash [x \mapsto u]e : \tau$.

Finally, type safety is given via progress and preservation.

Lemma 3.7 (Preservation)**.** *If* $\Gamma \vdash e : \tau$ *then* $e \longrightarrow e'$ *implies* $\Gamma \vdash e' : \tau$.

Lemma 3.8 (Progress)**.** *If* $\vdash e : \tau$ *then, for some value* v*,* $e = v$ *or for some term* e'*,* $e \longrightarrow e'$.

3.5 Expressiveness

We can substantiate our claim that QML is as expressive as System F by giving a type-preserving translation from Church-style System F typing derivations to QML terms. Figure 4 gives the standard System F typing relation, *augmented* with our translation to output a translated QML term. The translation erases type declarations from System F typed λ-abstractions, type arguments from type applications, and type witnesses from `pack` expressions. At the same time, it uses the conclusion of the System F typing rules to produce the required type annotations in the translated term. Notice the implicit closing over the free variables of the quantified System F types using the abbreviations $\{\forall\alpha.\tau\}$ which stands for $\{\mathit{ftv}(\forall\alpha.\tau)\ \forall\alpha.\tau\}$, and similarly for existential types – see the abbreviation forms in Figure 1. Since Λ- and `unpack`-bound type variables can never be explicitly bound in the translated term, our translation must, instead, close over any free variables occurring in the quantified types. One can now prove (in Coq):

Contexts

$$A \quad ::= \quad \epsilon \mid A, (x{:}\tau)$$

Translation of contexts

$$\epsilon^\circ \quad = \quad \epsilon$$
$$(A, (x : \tau))^\circ \quad = \quad A^\circ, (x : \Pi(\epsilon)(\tau))$$

$$\frac{A, (x{:}\tau) \vdash M : \mu \rightsquigarrow e}{A \vdash \lambda x : \tau\,.\,M : \tau \rightarrow \mu \rightsquigarrow \lambda x\,.\,e}\ \text{TR-ABS}$$

$$\frac{(x{:}\tau) \in A}{A \vdash x : \tau \rightsquigarrow x}\ \text{TR-VAR} \qquad \frac{A \vdash M_1 : \tau \rightarrow \mu \rightsquigarrow e_1 \quad A \vdash M_2 : \tau \rightsquigarrow e_2}{A \vdash M_1\ M_2 : \mu \rightsquigarrow e_1\ e_2}\ \text{TR-APP}$$

$$\frac{A \vdash M : \tau \rightsquigarrow e_1 \quad \alpha\#\mathit{ftv}(A)}{A \vdash \Lambda\alpha\,.\,M : \forall\alpha.\tau \rightsquigarrow \{\ \forall\alpha.\tau\}\ e}\ \text{TR-TABS}$$

$$\frac{A \vdash M : \forall\alpha.\tau \rightsquigarrow e}{A \vdash M\ [\mu] : [\alpha \mapsto \mu]\tau \rightsquigarrow e\ \{\ \forall\alpha.\tau\}}\ \text{TR-TAPP}$$

$$\frac{A \vdash M : [\alpha \mapsto \mu]\tau \rightsquigarrow e}{A \vdash \texttt{pack}(\mu, M)\ \texttt{as}\ \exists\alpha.\tau : \exists\alpha.\tau \quad \rightsquigarrow \{\ \exists\alpha.\tau\}\ e}\ \text{TR-PACK}$$

$$\frac{A \vdash M : \exists\alpha.\tau \rightsquigarrow e \quad \alpha\#\mathit{ftv}(A, \mu) \quad A, (x{:}\tau) \vdash N : \mu \rightsquigarrow u}{A \vdash \texttt{open}\ (\alpha, x) = M\ \texttt{in}\ N : \mu \rightsquigarrow \texttt{open}\ \{\ \exists\alpha.\tau\}\ x = e\ \texttt{in}\ u}\ \text{TR-OPEN}$$

Figure 4: A type-preserving translation, $A \vdash M : \tau \rightsquigarrow e$, from System F to QML.

Theorem 3.9 (Translation preserves types)**.** *If* $A \vdash M : \tau \rightsquigarrow e$ *then* $A^\circ \vdash e : \tau$*, where* A° *is the trivial embedding of System* F *contexts in* QML *contexts.*

As an aside, observe that the translated QML term may have more types than the original System F term because of the dropping of annotations on λ-abstractions. However, QML programs enjoy principal type schemes (see Proposition 5.2) and hence the principal scheme of the translated term will be more general than its source System F type. For example, the System F term $\lambda x{:}(\forall\alpha.\alpha).x$ of ground type $(\forall\alpha.\alpha) \rightarrow (\forall\alpha.\alpha)$ translates to QML

```
% nested introductions
let polyapp = {∀α.∀β.(α→ β)→ α→ β}
                {∀β.(α→ β)→ α→ β}
                   app;;
let abstype = {∀α.∃β.(α→ β) ×(β→ α)}
                {∃β.(α→ β) ×(β→ α)}
                   (id,id);;
let module = {∀α.∃β.∀γ.(α→ β) ×((α→ γ)→ β→ γ)}
               {∃β.∀γ.(α→ β) ×((α→ γ)→ β→ γ)}
                 {∀γ.(α→ β) ×((α→ γ)→ β→ γ)}
                   (id, app);;
% nested eliminations
(polyapp {∀α.∀β.(α→ β)→ α→ β})
           {∀β.(α→ β)→ α→ β}
;;
let x = abstype {∀α.∃β.(α→ β) ×(β→ α)} in
 open {∃β.(α→ β) ×(β→ α)} x = x in
  (snd x) (fst x 1);;

let x = module {∀α.∃β.∀γ.(α→ β)×(α→ γ)→ β→ γ} in
 open {∃β.∀γ.(α→ β) ×(α→ γ)→ β→ γ} x = x in
  let x = x {∀γ.(α→ β) ×(α→ γ)→ β→ γ} in
   (snd x) (fun a → (a,a)) (fst x 1);;
```

Figure 5: Longhand nested quantifier introduction and elimination.

Prefixed Types			
π	$::=$	(τ)	empty prefix
	$\mid$	$\forall\alpha\pi$	universal prefix
	$\mid$	$\exists\alpha\pi$	existential prefix
Derived Terms			
e, u	$::=$	$\ldots$	
	$\mid$	$\{\overline{\alpha}\ \pi\}\ e$	π-introduction
	$\mid$	$\texttt{open}\ \{\overline{\alpha}\ \pi\}\ x = u\ \texttt{in}\ e$	scoped π-elimination
	$\mid$	$e\ \{\overline{\alpha}\ \pi\}$	open π-elimination

Figure 6: Syntactic sugar.

term $\lambda x.x$ with more general scheme $\Pi(\beta)(\beta \to \beta)$, which subsumes its source typing.

More clever translations, that exploit the implicit polymorphism available in the target language, are possible but we have not investigated any; nor have we attempted to prove that our translation preserves reduction behaviour, but this should be easy provided the System F reduction relation also reduces under Λ and pack.

4. Derived Forms

An obvious critique of QML is that manipulating several quantifiers at once is tedious and error prone as it requires the provision of several closely related type annotations.

Consider, for example, the following types with nested quantifiers,

```
∀α.∀β.(α→ β)→ α→ β
∀α.∃β.(α→ β)×(β→ α)
∀α.∃β.∀γ.(α→ β)×((α→ γ)→ β→ γ)
```

corresponding to the type of a first-class polymorphic version of app; a polymorphic abstract datatype with a pair of injection and projection functions; and a polymorphic module declaring an abstract type with a pair of an injection function and generic fold operation - a module for one element containers, if you like. Introducing

Translation of prefixed types

$$\begin{aligned}
[\![(\tau)]\!] &= \tau\\
[\![\forall\alpha\pi]\!] &= \forall\alpha.[\![\pi]\!]\\
[\![\exists\alpha\pi]\!] &= \exists\alpha.[\![\pi]\!]
\end{aligned}$$

Expansion of prefix introduction

$$\begin{aligned}
[\![\{\overline{\alpha}\ (\tau)\}\ e]\!] &= e\\
[\![\{\overline{\alpha}\ \forall\beta\pi\}\ e]\!] &= \{\overline{\alpha}\ \forall\beta.[\![\pi]\!]\}\ [\![\{\overline{\alpha}\beta\ \pi\}\ e]\!]\\
[\![\{\overline{\alpha}\ \exists\beta\pi\}\ e]\!] &= \{\overline{\alpha}\ \exists\beta.[\![\pi]\!]\}\ [\![\{\overline{\alpha}\beta\ \pi\}\ e]\!]
\end{aligned}$$

Expansion of scoped prefix elimination

$$\begin{aligned}
&[\![\texttt{open}\ \{\overline{\alpha}\ (\tau)\}\ x = u\ \texttt{in}\ e]\!] = (\lambda x.\, e)\ u\\
&[\![\texttt{open}\ \{\overline{\alpha}\ \forall\beta\pi\}\ x = u\ \texttt{in}\ e]\!] =\\
&\qquad [\![\texttt{open}\ \{\overline{\alpha}\beta\ \pi\}\ x = u\ \{\overline{\alpha}\ \forall\beta.[\![\pi]\!]\}\ \texttt{in}\ e]\!]\\
&[\![\texttt{open}\ \{\overline{\alpha}\ \exists\beta\pi\}\ x = u\ \texttt{in}\ e]\!] =\\
&\qquad \texttt{open}\ \{\overline{\alpha}\ \exists\beta.[\![\pi]\!]\}\ x = u\ \texttt{in}\ [\![\texttt{open}\ \{\overline{\alpha}\beta\ \pi\}\ x = x\ \texttt{in}\ e]\!]
\end{aligned}$$

Expansion of open prefix elimination

$$[\![e\ \{\overline{\alpha}\ \pi\}]\!] = [\![\texttt{open}\ \{\overline{\alpha}\ \pi\}\ x = e\ \texttt{in}\ x]\!]$$

Figure 7: Expansion of syntactic sugar.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash_i e : (\tau)}\ \textsc{intro-base}$$

$$\frac{\Gamma \vdash_i e : \pi \quad \alpha \# \mathit{ftv}(\Gamma, e)}{\Gamma \vdash_i e : \forall\alpha\pi}\ \textsc{intro-all}$$

$$\frac{\Gamma \vdash_i e : [\alpha \mapsto \mu]\pi}{\Gamma \vdash_i e : \exists\alpha\pi}\ \textsc{intro-some}$$

Figure 8: Prefix introduction $\Gamma \vdash_i e : \pi$.

$$\frac{\Gamma, (x{:}\mu) \vdash e : \tau}{\Gamma\ ;\ x{:}(\mu) \vdash_e e : \tau}\ \textsc{elim-base}$$

$$\frac{\Gamma\ ;\ x{:}[\alpha \mapsto \mu]\pi \vdash_e e : \tau}{\Gamma\ ;\ x{:}\forall\alpha\pi \vdash_e e : \tau}\ \textsc{elim-all}$$

$$\frac{\Gamma\ ;\ x{:}\pi \vdash_e e : \tau \quad \alpha \# \mathit{ftv}(\Gamma, \tau, e)}{\Gamma\ ;\ x{:}\exists\alpha\pi \vdash_e e : \tau}\ \textsc{elim-some}$$

Figure 9: Prefix elimination $\Gamma\ ;\ x{:}\pi \vdash_e e : \tau$.

and eliminating values of these types, as in Figure 5, is extremely tedious, since we have to individually manipulate each quantifier. The mixed quantifier eliminations of abstype and module are particularly appalling since we have to alternate between standalone $\forall$-elimination and scoped $\exists$-elimination, for which we have allowed ourselves the luxury of using let.

Our first reaction was to move to a richer system with mixed prefix quantifiers in types and big bang constructs to introduce and eliminate a mixed prefix of quantifiers all at once. We eventually discarded this idea, for reasons we discuss in Section 7.1.

Instead of extending the language with mixed prefix quantified *types* we can obtain the same convenience, without the drawbacks, by adding some syntactic sugar for dealing with nested quantifiers – our implementation expands this sugar during the parsing phase. Writing open $\{\ \forall\alpha.\tau\}\ x = e$ in $u \equiv (\lambda x.u)(e\ \{\ \forall\alpha.\tau\})$ as symmetric shorthand for scoped $\forall$ elimination, reveals more structure:

```
open {∀α.∃β.(α→ β)×(β→ α)} x = abstype
in open {∃β.(α→ β)×(β→ α)} x = x
   in (snd x) (fst x 1);;

open {∀α.∃β.∀γ.(α→ β)×(α→ γ)→ β→ γ} x = module
in open {∃β.∀γ.(α→ β)×(α→ γ)→ β→ γ} x = x
   in open  {∀γ.(α→ β)×(α→ γ)→ β→ γ} x = x
      in (snd x) (fun a→ (a,a)) (fst x 1);;
```

Each elimination strips a quantifier from x but leaves the remainder of the annotation unchanged. The same observation can be made for nested introductions (c.f. Figure 5).

Figure 6 defines the syntactic sugar. We first introduce a new syntactic category of *prefixed types*, ranged over by π. A prefixed type π is just a sequence of universal or existential quantifications ending in a type (τ) (quantified variables bind to the right in π). Terms are extended with generalized π-introduction and π-elimination constructs, by replacing the unary quantifiers of the base syntax with πs. The π-introduction, $\{\overline{\alpha}\ \pi\}\ e$, is used to introduce both universal types and existential types, inferring witness for those existential types along the way. The scoped π-elimination construct, open $\{\overline{\alpha}\ \pi\}\ x = u$ in e, is used to both instantiate universal types and open existential types in the scope of a continuation e. An implementation can, of course, remove the unary primitives from the concrete (but not abstract) syntax.

Figure 7 defines the expansion of our sugar by induction on π. $[\![\pi]\!]$ translates a prefixed type into an ordinary type with nested quantifiers. $[\![\{\overline{\alpha}\ \pi\}\ e]\!]$ expands a prefix introduction into nested, unary quantifier introductions. The $[\![\text{open } \{\overline{\alpha}\ \pi\}\ x = u \text{ in } e]\!]$ expansion is more interesting. For an empty prefix, we just apply a λ-abstraction of the body e to u. For an existential prefix, we simply inductively unfurl the open. For a universal prefix, we unfurl the open with a modified binding that first eliminates the universal in the expansion of the body. Note that each expansion strips the outermost quantified variable and adds it as a schematic variable of the inductive expansion.

The expansion $[\![e\ \{\overline{\alpha}\ \pi\}]\!]$ of open π-elimination - the prefix generalization of $e\ \{\alpha\ \forall\beta.\tau\}$ - is trivially defined in terms of scoped π-elimination with an identity continuation. Note that this form will fail to type check if π contains a proper[3] existential quantifier since the hypothetical type will escape in the type of x. However, the syntax remains useful shorthand for eliminating several universal quantifiers in one go.

Now, the code in Figure 10 is much more concise (c.f. Figure 5).

Fortunately, once defined, the programmer can forget about the underlying translation. Using the straightforward, and pleasingly symmetric, prefix introduction and elimination judgements defined in Figures 8 and 9, we can derive the following generalized introduction and elimination rules:

Lemma 4.1 (Derived typing rules)**.**

$$\frac{\Gamma \vdash_i e : [\alpha \mapsto \mu]\pi}{\Gamma \vdash [\![\{\overline{\alpha}\ \pi\}\ e]\!] : [\![[\alpha \mapsto \mu]\pi]\!]}\ \text{QUANTI}$$

$$\frac{\Gamma \vdash u : [\![[\overline{\alpha \mapsto \tau}]\pi]\!] \quad \Gamma\,;\, x{:}[\overline{\alpha \mapsto \tau}]\pi \vdash_e e : \mu}{\Gamma \vdash [\![\texttt{open}\ \{\overline{\alpha}\ \pi\}\ x = u\ \texttt{in}\ e]\!] : \mu}\ \text{QUANTE}$$

[3] by proper, we mean a quantified variable with at least one occurrence.

```
% prefix introductions
let polyapp = {∀α∀β ((α→ β)→ α→ β)} app;;

let abstype = {∀α∃β ((α→ β) ×(β→ α))} (id,id);;

let module = {∀α∃β∀γ ((α→ β) ×((α→ γ)→ β→ γ))}
              (id,app);;
% prefix eliminations
polyapp {∀α∀β ((α→ β)→ α→ β)};;

open {∀α∃β ((α→ β) ×(β→ α))} x = abstype
in (snd x) (fst x 1);;

open {∀α∃β∀γ ((α→ β) ×(α→ γ)→ β→ γ)} x = module
in (snd x) (fun a → (a,a)) (fst x 1);;
```

Figure 10: Shorthand prefix introduction and elimination.

Expansion of polymorphic, scoped prefix elimination

$[\![\texttt{open}\ \{\overline{\alpha}\ (\tau)\}\ x = u\ \texttt{in}\ e]\!] \quad = \quad \texttt{let}\ x = u\ \texttt{in}\ e$

$[\![\texttt{open}\ \{\overline{\alpha}\ \forall\beta\pi\}\ x = u\ \texttt{in}\ e]\!] \quad =$

$\qquad [\![\texttt{open}\ \{\overline{\alpha}\beta\ \pi\}\ x = u\ \{\overline{\alpha}\ \forall\beta.[\![\pi]\!]\}\ \texttt{in}\ e]\!]$

$[\![\texttt{open}\ \{\overline{\alpha}\ \exists\beta\pi\}\ x = u\ \texttt{in}\ e]\!] \quad =$

$\qquad \texttt{open}\ \{\overline{\alpha}\ \exists\beta.[\![\pi]\!]\}\ x = u\ \texttt{in}\ [\![\texttt{open}\ \{\overline{\alpha}\beta\ \pi\}\ x = x\ \texttt{in}\ e]\!]$

$$\frac{\overline{\alpha}\#\mathit{ftv}(\Gamma), \overline{\beta} \qquad \Gamma, (x{:}\ \Pi(\overline{\alpha})(\mu)\ ) \vdash e : \tau}{\Gamma\,;\, x{:}(\mu) \vdash_e^{\overline{\beta}} e : \tau}\ \text{POLY-ELIM-BASE}$$

$$\frac{\Gamma\,;\, x{:}[\alpha \mapsto \mu]\pi \vdash_e^{\overline{\beta}} e : \tau}{\Gamma\,;\, x{:}\forall\alpha\pi \vdash_e^{\overline{\beta}} e : \tau}\ \text{POLY-ELIM-ALL}$$

$$\frac{(\Gamma, \_ : [\![\pi]\!])\ ;\, x{:}\pi \vdash_e^{\overline{\beta}} e : \tau \quad \alpha\#\ \mathit{ftv}(\Gamma, \tau, e)}{\Gamma\,;\, x{:}\exists\alpha\pi \vdash_e^{\overline{\beta}} e : \tau}\ \text{POLY-ELIM-SOME}$$

Figure 11: Polymorphic prefix elimination $\Gamma\,;\, x{:}\pi \vdash_e^{\overline{\beta}} e : \tau$.

So far so good, but there is a subtle difference between the code in Figures 5 and 10. With our current definition, open $\{\overline{\alpha}\ \pi\}\ x = u$ in e, introduces x in e under a λ-abstraction, which precludes polymorphic uses of x even when π is just a sequence of universal quantifiers. This is unfortunate. For example, it means that

$$\lambda id.\texttt{open}\ \{\ \forall\alpha(\alpha \to \alpha)\}\ f = id\ \texttt{in}\ (f\ 1, f\ \mathit{true}) \qquad (1)$$

will fail to type check - since it tries to use f polymorphically - even though this very similar term is type correct:

$$\lambda id.\texttt{let}\ f = id\ \{\ \forall\alpha(\alpha \to \alpha)\}\ \texttt{in}\ (f\ 1, f\ \mathit{true}) \qquad (2)$$

The fix, of course, is to use a different expansion that ends in a `let`. The net effect of this alternative translation, captured by the highlighted changes in Figure 11, is that the innermost universal quantifiers of a prefix will be eliminated in the definition of that `let`, and thus become generalized in a type *scheme* for x. Any universal quantifiers bound to the left of an existential in π will enter the context (through the existential open) and thus receive a fixed instantiation and *not* be generalized, as required for type soundness. Now, our previously untypeable term in (1) just expands to the typeable term in (2).

For this variant of open, we can derive the following rule (though we have not yet done so with Coq):

$$\frac{\Gamma \vdash u : [\![[\overline{\alpha \mapsto \tau}]\pi]\!] \quad \Gamma\, ; x{:}[\overline{\alpha \mapsto \tau}]\pi \vdash_e^{ftv(u)} e : \mu}{\Gamma \vdash [\![\texttt{open}\ \{\overline{\alpha}\ \pi\}\ x = u\ \texttt{in}\ e]\!] : \mu}\ \text{POLY-QUANTE}$$

This rule relies on a modified prefix elimination judgement also displayed in Figure 11. Its rules differ from the simpler ones in Figure 9 by introducing let-polymorphism before checking the continuation e, and by throwing any eliminated existential type into the context (for a fresh variable _) before descending into the prefix. The latter step tracks the effect of unary open in the expansion and correctly prevents subsequent generalization over any free variables appearing within the current translation of π, including the existential variable α (if it occurs in π), but excluding variables appearing within future, inner instantiations. Finally, the variables $\overline{\beta}$ (indexing $\vdash^{\overline{\beta}}$) are used to prevent accidental generalization over any free type variables in the let-bound term u; they could be removed by insisting on closed type annotations.

Returning to our example, let us consider a more refined client of module that uses the fold operation polymorphically:

```
% longhand, inner let x generalizes over γ.
let x = module {∀α.∃β.∀γ.(α→ β) ×(α→ γ)→ β→ γ}
in open {∃β.∀γ.(α→ β) ×(α→ γ)→ β→ γ} x = x in
   let x = x {∀γ.(α→ β) ×(α→ γ)→ β→ γ} in
   let p = (snd x) (fun a → (a , a)) (fst x 1) in
           % x used with γ= int ×int
   let i = (snd x) (fun a →a + a) (fst x 1) in
           % x used with γ= int
     (p,i);;
% polymorphic prefix elimination works just as well
open {∀α∃β∀γ ((α→ β) ×(α→ γ)→ β→ γ)} x = module
in let p = (snd x) (fun a → (a , a)) (fst x 1) in
   let i = (snd x) (fun a →a + a) (fst x 1) in
     (p,i);;
```

Now the lower shorthand will typecheck just like upper longhand (using Figure 11's definitions).

4.1 Type Constraints

Sometimes, it is handy to have type constraints in the language. OCaml style type constraints are easily captured by introducing and eliminating a vacuous quantifier:

$$[\![e{:}\overline{\alpha}(\tau)]\!] = (\{\overline{\alpha}\ \forall\beta.\tau\}\ e)\ \{\overline{\alpha}\ \forall\beta.\tau\} \tag{3}$$

where $\beta\#\overline{\alpha}, \mathit{ftv}(\tau)$ is some fresh variable, together with abbreviated notation $e{:}\tau \equiv e{:}\overline{\mathit{ftv}(\tau)}(\tau)$.

We conjecture, but have not formally shown the following, invertible, derived rule.

$$\frac{\Gamma \vdash e : [\overline{\alpha \mapsto \mu}]\tau}{\Gamma \vdash [\![e{:}\overline{\alpha}(\tau)]\!] : \tau[\overline{\alpha \mapsto \mu}]}\ \text{CONSTRAINT}$$

Indeed, with this sugar in hand, it makes sense to refine the translation (and derived rules) in Figure 7 so that:

$$\begin{array}{lcl} [\![\{\overline{\alpha}\ (\tau)\}\ e]\!] & = & [\![e{:}\overline{\alpha}(\tau)]\!] \\ [\![\texttt{open}\ \{\overline{\alpha}\ (\tau)\}\ x = u\ \texttt{in}\ e]\!] & = & (\lambda x\,.\,e)\ [\![u{:}\overline{\alpha}(\tau)]\!] \end{array}$$

or, respectively, in Figure 11:

$$[\![\texttt{open}\ \{\overline{\alpha}\ (\tau)\}\ x = u\ \texttt{in}\ e]\!] \quad = \quad \texttt{let}\ x = [\![u{:}\overline{\alpha}(\tau)]\!]\ \texttt{in}\ e$$

This would ensure that empty prefixes still impose a proper constraint rather than having no effect.

```
type sig (α,ρ,β) =
   ( (α →ρ) ×% init a
     (ρ →int →α) ×% sub r i
     (ρ →int →α→ ρ) ×% update r i a
     ((α →β→ β) →β→ ρ→ β) % fold f b r
   );;
let base = {∃ρ∀β (sig(α,ρ,β))}
    ((fun a →a), % init
     (fun r →fun i →r), % sub
     (fun r →fun i →fun a →a), % update
     (fun f →fun b →fun r →f r b) % fold
    );;
let step = fun x: ∃ρ∀β(sig(α,ρ,β)) →
  let (xinit,xsub,xupdate,xfold) = x in
  let init = fun a →((xinit a), (xinit a)) in
  let sub = fun r →fun i →
   if (i mod 2) = 0
   then (xsub (fst r) (i / 2))
   else (xsub (snd r) (i / 2)) in
  let update = fun r →fun i →fun a →
   if (i mod 2) = 0
   then ((xupdate (fst r) (i /2) a),(snd r))
   else ((fst r),(xupdate (snd r) (i / 2) a)) in
  let fold = fun f →fun b →fun r →
        xfold f (xfold f b (fst r)) (snd r)
  in {∃ρ∀β (sig(α,ρ,β))} (init,sub,update,fold);;

let rec mkMonoArray n =
 if(n=0) then base else step(mkMonoArray(n - 1));;
let mkPolyArray = fun n →
 {∀α(∃ρ.∀β.sig(α,ρ,β))} (mkMonoArray n);;
```

Figure 12: Inductively defined, fixed-sized functional arrays.

4.2 Function Argument Type Annotations

Programmers often document their function arguments with types. We can add support for this, but also go further and exploit the annotation to allow implicit elimination of prefixed types:

$$[\![\lambda x : \overline{\alpha}\ \pi.e]\!] = \lambda x.\texttt{open}\ \{\overline{\alpha}\ \pi\}\ x = x\ \texttt{in}\ e$$

with abbreviated notation $\lambda x : \pi.e \equiv \lambda x : \overline{\mathit{ftv}(\pi)}\ \pi.e$.

Now, for example, the programmer can write:

$$\lambda id{:}\forall\alpha(\alpha \to \alpha).(id\ 1, id\ true)$$

and use the parameter id polymorphically, saving on the insertion of let as in example (2) above.

4.3 Example: Dynamic, Fixed-size Functional Arrays

Figure 4.3 contains a larger example of programming in QML, using some of our derived forms. It is adapted from an example of programming with first-class modules in an extension of Standard ML (22). With impredicative existentials, it is possible to make the witness of an abstract type depend on the result of some computation. A simple example of such a type is the type of fixed-size arrays of size n, where n is a value that is computed at *runtime*. For simplicity, we implement functional arrays of size 2^n, for arbitrary $n \geq 0$. An array module will have type $\forall\alpha.\exists\rho.\forall\beta.\texttt{sig}(\alpha,\rho,\beta)$, for any array element type α, some array type ρ, and all types β, where β is the parameter of a generic fold operation. By construction, ρ will represent arrays containing 2^n entries of type α for some n. The (ordinary) type abbreviation $\texttt{sig}(\alpha,\rho,\beta)$ specifies a tuple of array operations (we could also have used a record). The first component, function `init a`, returns an array that has its entries initialised to the value of `a`. The second component, function `sub a i`, returns the value of the `i mod` 2^n-th entry of the array `a`. The third component, function `update r i a`, returns an array that is equivalent to the array `r`, except for the `i mod` 2^n-th entry that is updated with the value of `a`. The fourth component, function `fold f b r`, folds the function `f`$:\alpha\rightarrow\beta\rightarrow\beta$ over each element in the array `r`, accumulating intermediate results in the β argument. Note that β is quantified to the right of ρ so we will be able to use a particular array type's `fold` operation polymorphically. To omit array bound checks, we interpret each index `i` modulo 2^n. The value `base` implements arrays of size $2^0 = 1$. An array is represented by its sole entry with trivial `init`, `sub`, `update` and `fold` functions. The function `step x` maps a value, `x`, implementing arrays of size 2^n, for some abstract type ρ, to a tuple implementing arrays of size 2^{n+1}. It represents an array of size 2^{n+1} as a product of ρ arrays. Entries with even (odd) indices are stored in the first (second) component of the pair. The function `init e` returns a pair of arrays of size 2^n. The functions `sub r i` and `update r i a` use the parity of i to determine which sub-array to subscript or update. The function `mkMonoArray n` uses recursion on `n` to construct an existential value implementing arrays of size 2^n, keeping the element type fixed at α. Notice that the witness of ρ returned by `mkPolyArray n` is a balanced, nested product of depth `n`: the shape of this type really does depend on the *runtime* value of `n`.

5. Type inference

The correctness of type inference relies on two important lemmas. For soundness, it is necessary that the typing relation is preserved under type substitution (Lemma 3.4). For completeness, it is important that typing is preserved when more general schemes are used in the context. To capture this, we introduce the scheme instance relation $\Pi(\overline{\alpha_1})(\tau_1) \preceq \Pi(\overline{\alpha_2})(\tau_2)$ iff for all $\overline{\beta}\#\mathit{ftv}(\Pi(\overline{\alpha_1})(\tau_1))$ it is the case that there exist $\overline{\mu}$ so that $[\overline{\alpha_1 \mapsto \mu}]\tau_1 = [\overline{\alpha_2 \mapsto \beta}]\tau_2$. We generalize this notation to contexts: $\Gamma_1 \preceq \Gamma_2$ iff, $\mathit{dom}(\Gamma_1) = \mathit{dom}(\Gamma_2)$ and for all $(x{:}\varsigma_2) \in \Gamma_2$ it is the case that $(x{:}\varsigma_1) \in \Gamma_1$ and $\varsigma_1 \preceq \varsigma_2$. Now we can state the strengthening property:
Lemma 5.1 (Strengthening)**.** *If* $\Gamma_2 \vdash e : \tau$ *and* $\Gamma_1 \preceq \Gamma_2$ *then* $\Gamma_1 \vdash e : \tau$.

The actual implementation of type inference relies on unification of terms with quantifiers, and is a folklore subcase of (first-order) mixed-prefix unification (13), itself a decidable case of unification with most general unifiers. There exist several reference implementations of similar algorithms (17; 4).

Showing completeness of type inference and principal type schemes is routine using completeness of the unification algorithm and Lemmas 3.4,5.1.
Proposition 5.2 (Principal schemes)**.** *If* $\Gamma \vdash e : \tau$ *then there exists a* μ *such that* $\Gamma \vdash e : \mu$ *and for every* $\tau', \overline{\alpha}$, *such that* $\Gamma \vdash e : \tau'$ *and* $\overline{\alpha}\#\mathit{ftv}(\Gamma, e)$, *it is the case that* $\Pi(\overline{\beta})(\mu) \preceq \Pi(\overline{\alpha})(\tau')$ *where* $\overline{\beta} = \mathit{ftv}(\mu) - \mathit{ftv}(\Gamma, e)$.

A consequence of the existence of principal schemes is the `let`-expansion property.
Proposition 5.3. *If* $\Gamma \vdash [x \mapsto u]e : \tau$ *and* $\Gamma \vdash u : \mu$ *then* $\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ e : \tau$.

6. Scaling up

We sketch here the addition of several features found in modern functional programming languages.

6.1 References

Most variants of ML include imperative features such as polymorphic references and arrays. Adding these to QML requires the usual care to avoid unsoundness. The standard trick is to adopt the *value restriction*. The idea is modify rule LET to only do scheme generalisation for definitions that are values. But this would be too restrictive for QML, rejecting, for example:

$$\lambda id.\texttt{let}\ f = id\ \{\ \forall\alpha(\alpha\rightarrow\alpha)\}\ \texttt{in}\ (f\ 1, f\ \mathit{true}) \qquad (4)$$

However, the aim of the value restriction is to avoid unsound generalisation of definitions that expand or may expand the store (such as `ref` allocations and function applications). Values do not expand the store but there is a larger class of expressions that are *nonexpansive* too (14). Since our QML semantics is call-by-value and reduces under introductions, the body of a nonexpansive introduction must also be nonexpansive; eliminating a universal quantifier from a nonexpansive expression does not trigger further computation and is obviously nonexpansive; eliminating an existential quantifier from a nonexpansive expression is nonexpansive provided the continuation of the `open` is. Finally, though not usually included, if both the definition and continuation of a `let` are nonexpansive, so is the entire `let`.

This leads us to the following characterisation of nonexpansive terms, range over by n, as subsets of full terms:

Nonexpansive Terms

$$\begin{array}{lcll}
n & ::= & x \mid \lambda x\,.\,e \mid \texttt{let}\ x = n_1\ \texttt{in}\ n_2 & \\
 & \mid & \{\overline{\alpha}\ \forall\beta.\tau\}\ n & \forall\text{-introduction} \\
 & \mid & n\ \{\overline{\alpha}\ \forall\beta.\tau\} & \forall\text{-elimination} \\
 & \mid & \{\overline{\alpha}\ \exists\beta.\tau\}\ n & \exists\text{-introduction} \\
 & \mid & \texttt{open}\ \{\overline{\alpha}\ \exists\beta.\tau\}\ x = n_1\ \texttt{in}\ n_2 & \exists\text{-elimination}
\end{array}$$

Including nonexpansive `let` expressions allows us to show:
Proposition 6.1 (Nonexpansive derived forms)**.** *If expressions* e *and* u *are non expansive then so are the translations of derived forms* $\{\overline{\alpha}\ \pi\}\ e$, $\texttt{open}\ \{\overline{\alpha}\ \pi\}\ x = u\ \texttt{in}\ e$, *and* $e\ \{\overline{\alpha}\ \pi\}$.

However, even with this more liberal definition, we lose some polymorphism. For example, in pure QML, the church encodings of inductive datatypes will admit `let`-polymorphic encodings of their values. But since these values are represented as (expansive) *applications*, in impure QML, these value will never be polymorphic.

6.2 Data constructors and pattern matching

Adding ordinary algebraic datatypes to QML is straightforward. However, our π-elimination construct is reminiscent of nested pattern matching and it would be nice to find a clean integration of pattern matching with quantifier elimination, perhaps by extending the elegant presentation of Krishnaswami (8).

To avoid inferring types with quantifiers, Rémy (20), Jones (6) and Odersky and Läufer (12; 15) instead tie quantifier introduction and elimination to the introduction and elimination of *named* types. This includes introducing and eliminating polymorphism for the *arguments* of data constructors: if a data constructor expects a polymorphic argument, the quantifier of the argument is introduced implicitly. When pattern matching, a variable that is bound to a polymorphic type can freely be instantiated in the scope of the pattern (at two or more types in (12; 15)).

By comparison, the naïve extension of QML with ordinary datatypes would support impredicative constructors but still require separate steps to (i) introduce quantifiers for a polymorphic constructor argument, and (ii) apply the constructor. Symmetrically – in the case of elimination – a pattern match on the constructor would typically continue with an inner elimination of its argument's quantifiers.

As a convenience, we might like to introduce some more syntactic sugar to enhance QML datatype declarations. The idea is to define each data constructor with a prefixed type that, unlike an ordinary constructor definition, declares automatic introduction and elimination forms for that constructor's argument (a separate, back-door mechanism could suppress auto introductions and eliminations).

For example, consider a datatype declaration of the form:

$$\texttt{type}\ \overline{\alpha}\ t = \overline{C\ \texttt{of}\ \pi} \qquad (\text{where } ftv(\pi_i) \subseteq \overline{\alpha})$$

Each term constructor C_i is declared with the closed scheme:

$$\varsigma_i = \Pi(\overline{\alpha})(\llbracket \pi_i \rrbracket \to \overline{\alpha}\ t)$$

But, guided by its declared prefix, every constructor application $C_i\ e$ is syntactically desugared as:

$$\llbracket C_i\ e \rrbracket = C_i\ (\{\overline{\alpha}\ \pi_i\}\ e) \tag{5}$$

and each case on an expression of type $\overline{\alpha}\ t$ is expanded as follows:

$$\llbracket \texttt{case}\ e\ \texttt{of}\ \overline{C\ x\ \to\ e} \rrbracket \quad = \\ \texttt{case}\ e\ \texttt{of}\ \overline{C\ x\ \to\ \texttt{open}\ \{\overline{\alpha}\ \pi\}\ x = x\ \texttt{in}\ e}$$

Thus we recover the convenience of (12; 20; 6; 15) yet keep constructors first-class (since they receive ordinary types or schemes).

7. Discussion

We now discuss interesting points in the design space surrounding the explicit introduction and elimination of polymorphism.

7.1 Mixed Prefix Quantified Types

Our first response to the syntactic horrors of Figure 5 was to move to a more general system with intrinsic, mixed-prefix, quantified types ($\tau ::= \ldots \mid \pi$) and big bang constructs to introduce and eliminate a mixed prefix all at once, very similar to our derived rules in Lemma 4.1. Unfortunately, the problem with treating mixed prefix types atomically is that it becomes impossible to write η-expansions of some types.

Consider this System F term that does an η-expansion on a nested type with mixed quantifiers.

```
F_eta: (∀α.∃β.α → β) → (∀α.∃β.α → β)
F_eta = λp: ∀α.∃β.α → β.
            Λα. open (β,x) = p [α] in
              pack (β,x) as ∃β.α→ β.
```

F_eta has to Λ-abstract α *before* open-ing p $[\alpha]$ and introducing the existential, so the elimination of p's quantifiers happens *between* the introduction of the result quantifiers.

We can write this in QML without any difficulty.

```
let eta = fun p →
  {∀α (∃β.α → β)} open {∀α∃β.(α→ β)} x = p in
                       {∃β.α → β} x ;;
```

However, if π-types are proper atomic types, there would be no way to separate the introduction of each quantifier. In such a system, our best attempts at writing F_eta always wound up splitting the atomic mixed prefix quantifier $(\forall\alpha\exists\beta.(\alpha\to\beta))$ into nested, single quantifiers, $\forall\alpha.(\exists\beta.(\alpha\to\beta))$, which is close to, but not exactly, what we wanted.

```
% illegal, fails to type check as required
let eta_illegal = fun p →
    open {∀α∃β.α→ β} x = p in
       {∀α∃β.α→ β} x
% type checks, but with wrong type
let eta_wrong = fun p →
    {∀α (∃β. α→ β)} open {∀α∃β.α→ β} x = p in
                          x {∃β.α→ β}
% returns ∀α.(∃β.α → β) not ∀α∃β.α → β, as reqd.
```

QML, like System F, lets us manipulate quantifiers individually when necessary. However, modifying the big bang rules to support this, although possible, also has an effect on the reduction semantics, which now has to be extended to allow partial reduction of mixed prefixed values in an ad hoc manner.

7.2 System F-style introductions and eliminations

An alternative point arises from the following question: if all introductions and eliminations of polymorphism are explicit, why not use the syntax of System F itself? This variation may be appealing for dependently typed languages like Coq (2), where programmers are already used to explicitly-typed programming.

For instance, instead of the introduction form $\{\overline{\alpha}\ \forall\beta.\tau\}\ e$ one could imagine a form $\Lambda\beta.e$ where β is now bound in e. Similarly, instead of the elimination form $e\ \{\overline{\alpha}\ \forall\beta.\tau\}$ one could imagine the System F form $e\ [\mu]$. Note that we assume ML-style implicit polymorphism remains available in this alternative.

However, things are not that simple. The first observation is that these forms of annotations are not enough – we would also have to annotate some λ-bound arguments. For example, the QML program:

```
let foo f = ( f {∀α.α → α} ) 3
```

would have to be written:

```
let foo (f : ∀α.α → α) = f [int] 3
```

Hence, in cases like this one, the variant would actually require more annotations. When the function parameter is specialized several times, then the QML function exploiting the derived form for annotated function parameters (Section 4.2), which can be implicitly eliminated at several instantiations, is clearly more concise.

Second, we would still have to provide some support for type applications where the applied type is only partially known. Given:

```
let foo x (y : ∀α.α → α) = y [?] x;;
```

it is not clear what the [?] type instantiation should be and hence our system would have to support both rigid *and* flexible type variables in type applications.

Third, using type abstraction ($\Lambda\alpha.e$) to introduce polymorphism is not that straightforward, because the presence of untyped term abstractions means the type of the *body* of a type abstraction is, in general, not known. Consider the term

```
let foo = Λα. fun x (y : α) → x
```

it is not clear what the intended type of foo is : should it be $\Pi(\beta)(\forall\alpha.\beta \to \alpha \to \beta)$ or $\forall\alpha.\alpha \to \alpha \to \alpha$? Worryingly, if we choose the first scheme (which looks like it might be more general) the program test foo would fail to type check – where test has type $(\forall\alpha.\alpha \to \alpha \to \alpha) \to int$. On the other hand, if choose the more appropriate second type, other programs, that would have passed with the first choice, now fail to type check. Unfortunately, there is no type scheme for foo that subsumes both these choices.

7.3 Merging the quantifiers

One of the key points in our approach has been the distinction between explicitly and implicitly introduced quantifiers. It is this structural distinction that enables complete and decidable type inference with a simple specification, based on unification of types with quantifiers.

By contrast, one may wonder what goes wrong if we attempt to *merge* the two forms of quantifiers, and only have schemes of the form $\Pi(\overline{\alpha})(\tau)$. Consider the program:

```
head : Π(α)(α list → α)
ids : (Π(α)(α → α)) list
choose : Π(α)(α → α→ α)

let foo1 = cons (head ids) ids;;
let foo2 = head ids 3;;
let foo3 = choose (head ids);;
```

In the application `head ids` at the definition of `foo`, should we be instantiating the quantified variable of $\Pi(\alpha)(\alpha \to \alpha)$ or not? Presumably not, but in the second case we should instantiate it to `int`. In the third case it is not even clear – in fact `foo3` does not have a principal System F type.

Since it is now not clear whether we should instantiate expressions with polymorphic types, one can force "structural" unification only by ad hoc conditions in the typing rules. For example, Leijen's HMF (10) will always attempt to instantiate (e.g. `foo3` is typeable in HMF, with an ML type) unless there is some other argument (e.g. the second `ids` in `foo1`) that forces a polymorphic type.

Another possibility would be to always be explicit about generalization (and also generalize implicitly at `let`-nodes), but allow implicit instantiations *only for variables* that are either `let`-bound or λ-bound with polymorphic types. Hence, we would have explicit instantiations for all or at least *some* non variable expressions – in the latter case, to discover which ones the programmer would have to laboriously follow the typing rules. For example, an explicit instantiation of the expression (`head ids`) would be necessary in order to type `foo2` above. Finally, if the rules were to differentiate the treatment of instantiation for variables and some other expressions, it would be harder to give a reduction semantics. E.g. if we had

```
(fun (x : Π(α)(α → α)) → x 3) (head ids)
```

and we did not instantiate `head ids` by default, after one reduction step the program would be untypeable! In QML we have:

```
head : Π(α)(α list → α)
ids : (∀α.α → α) list
choose : Π(α)(α → α→ α)

let foo1 = cons (head ids) ids;;
let foo2 = ((head ids) {∀α.α→ α}) 3;;
% foo3 gets type (∀α.α→ α) → (∀α.α→ α)
let foo3 = choose (head ids);;
% foo4 gets type Π(α)( (α → α) → (α → α) )
let foo4 = choose ((head ids) {∀α.α→ α});;
```

In particular, in `foo2` and `foo4` we explicitly eliminate the polymorphic type of `head ids`, whereas in `foo1` and `foo3` no instantiation of `head ids` takes place.

We believe that the distinction between System F (explicit) polymorphism and ML (implicit) polymorphism is significantly cleaner than a set of restrictions on the typing rules.

8. Related work

Related research has explored several directions.

8.1 Explicit vs implicit polymorphism

The idea of aiding inference by distinguishing between explicitly introduceable / eliminable quantifiers and ML `let`-introduced quantifiers originates in the work of O'Toole and Gifford in the FX programming language (16). In that work, implicit and explicit instantiations are with quantifier-free types and unification is only between quantifier-free types. Function arguments must be quantifier-free, unless an explicit type annotation is provided.

Garrigue and Rémy (4) go further: in their system, as in ours, polymorphism is introduced explicitly. However, the elimination of explicit quantifiers is done by a construct that *does not* specify the type to be eliminated (improving on the idea of fully explicit eliminations). For completeness, the eliminated type must still be "known" at the point of elimination. To ensure this, the type system tracks the types of expressions (using labelled types and label polymorphism) to determine whether their polymorphism has been explicitly introduced somewhere in the program. The advantage is more compact type annotations but types now involve labels and label polymorphism.

Garrigue and Rémy also propose a label-free system of explicit type instantiations as an inferior variant of their semi-explicit elimination system ((4), Section 3.5). In that system, explicit elimination of polymorphism is done by providing the type of the expression to be eliminated, like in our proposal. They discard this idea for two reasons: (i) it is more verbose than semi-explicit eliminations and (ii) even `let`-bound quantifiers have to be explicitly eliminated. The reason for (ii) is that their proposal does not distinguish between `let`-introduced quantifiers and explicitly introduced quantifiers. In our system, however, `let`-bindings and explicit introductions result in *different* forms of quantifiers. Like ML, QML's `let`-introduced polymorphism is instantiated implicitly. We hence classify our proposal in the design space *between* the original Garrigue-Rémy proposal and their fully explicit variant.

Jones (6) describes an approach where universal and existential polymorphism is "boxed" under data constructors. For each required polymorphic type, the programmer declares a datatype whose term constructor accepts a quantified type. Introduction of a named quantified type is achieved by constructor application, and elimination by pattern matching on that constructor; conveniently, neither requires a type annotation beyond the shared datatype declaration. Constructors can introduce/eliminate several universal followed by several existential quantifiers, much as our derived prefix elimination forms do, but without arbitrary quantifier alternation, and without the final, implicit generalization step taken by our derived elimination construct. Note that such constructors are not first-class as they have types that contain quantifiers. Jones' constructors introduce and eliminate *all* quantifiers at once and thus, we believe, evince the same lack of eta-expansion functions for certain datatypes that encode mixed-prefix quantified types (such as $\forall\alpha\exists\beta.(\alpha\to\beta)$), which we observed in Section 7.1. In Jones' setting, this hinders the definition of natural isomorphisms between some structurally equivalent, yet distinct datatypes, obstructing separate development without some prior agreement on datatype definitions. For example, when distinct datatypes `t1` and `t2` encode the same mixed-prefix type $\forall\alpha\exists\beta.(\alpha\to\beta)$ there is, we believe, still no coercion function from `t1` to `t2` in Jones' system.

8.2 Implicit generalizations and instantiations, following the typing rules

Odersky and Läufer (15) take a different path: if all instantiations are with quantifier-free types, the shape of types can guide the type checker with respect to the placement of generalizations and instantiations. In their system, implicit instantiation is with quantifier-free types, and implicit generalization takes place also for arguments in applications, where the function type requires the argument to be polymorphic. There exist several variations on the same theme (18; 21), with some of them exploiting the "flow" of polymorphic information in the program. The Boxy Types proposal (23) is a somewhat awkward attempt to lift the quantifier-free instantiation restriction that is also based on the shape of types to determine implicit instantiations and generalizations – in the presence of polymorphic instantiations, however, the type system specification of Boxy Types is rather complicated.

The HMF system of Leijen (10) is along the same lines. All polymorphic expressions are implicitly instantiated, even with polymorphic types in a carefully crafted specification that uses (i) side-conditions on expressions receiving most general types, (ii) weights on types to ensure that a function is never instantiated with a polymorphic type unless one of its arguments requires that, and (iii) applications of functions to multiple arguments. For convenience, on top of this specification, HMF introduces (iv) "rigid" type annotations for explicit introductions of polymorphism, and employs heuristics that eliminate the need for many type annotations. Our proposal has a relatively small number of features, namely distinguishing two forms of quantifiers and multiple-quantifier eliminations (which is actually just syntactic sugar), but may sometimes require more annotations. Additionally, HMF's *algorithmic* implementation is quite intuitive (as is ours), but the fact that HMF does not require two distinct universal quantifiers (as we do) may be simpler for programmers.

The examples of Section 2 superset the examples in the HMF paper, except for `test0` below:

```
% rejected
let test0 = app poly (fun x → x);;
```

In HMF, `test0` would be accepted. In our system `test0` is rejected and requires an explicit type introduction around (**fun** x → x). Such implicit introductions of polymorphism, are, we believe, the most common situations where HMF would not require an annotation, but our proposal would. Perhaps QML could be improved by propagating expected polymorphism.

8.3 Implicit generalizations and instantiations

The ML^F (9) language of Le Botlan and Rémy and variants (FPH (24), HML (11)) attempt to infer all instantiations and generalizations provided that polymorphic function arguments be annotated (FPH and HML) or that polymorphic function arguments that are *used* at two different types are annotated (ML^F). Hence, they require very few type annotations. We see three potential reasons why a more explicit approach could be more practical: First, there is a merit of having explicit generalization points from a programming perspective as this may achieve better control of effects (for instance, when type lambdas are treated as unevaluated thunks). Second, with the exception of the box-free variant of FPH, these type systems are more sophisticated, involving bounded polymorphism. Third, all these systems are implemented using the non-trivial MLF unification algorithm. From the second author's experience with integrating ML^F-style unification in GHC (which also includes several other features) we consider this task to be not entirely straightforward.

9. Future work

In future work, we would like to extend our Coq formalization to ML with references and pattern matching, and to formally prove the correctness of type inference, instead of deferring to the literature. This will provide a useful stress-test for Coq's metatheory libraries and the techniques we have been using.

Another interesting research direction is to extend the idea of distinguishing between implicit type schemes and explicitly quantified types to higher-order polymorphism, in the style of F_ω. F_ω challenges the completeness and decidability of type inference by giving rise to higher-order unification constraints (19).

Acknowledgements Thanks to the ML'09 anonymous reviewers for their valuable suggestions on related work, Philip Wadler for reminding us of O'Toole and Gifford's work (16), and Simon Peyton Jones for helpful feedback.

References

[1] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *POPL'08*, pages 3–15, ACM, 2008.

[2] The Coq proof assistant. `http://coq.inria.fr`.

[3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL'82*, pages 207–12, ACM, 1982.

[4] J. Garrigue and D. Rémy. Semi-explicit first-class polymorphism for ML. *Journal of Information and Computation*, 155:134–169, 1999.

[5] J. R. Hindley. The principal type-scheme of an object in combinatory logic. (146):29–60, 1969.

[6] M. P. Jones. First-class polymorphism with type inference. In *POPL'97*, pages 483–496, ACM, 1997.

[7] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order lambda calculus. In *ACM Symposium on Lisp and Functional Programming*, pages 196–207. ACM, Orlando, Florida, June 1994.

[8] N. R. Krishnaswami. Focusing on pattern matching. In *POPL '09*, pages 366–378, ACM, 2009.

[9] D. Le Botlan and D. Rémy. MLF: raising ML to the power of System F. In *ICFP'03*, pages 27–38, ACM, 2003.

[10] D. Leijen. HMF: simple type inference for first-class polymorphism. In *ICFP'08*. ACM, 2008.

[11] D. Leijen. Flexible types: robust type inference for first-class polymorphism. In *POPL '09*, pages 66–77, ACM, 2009.

[12] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, 1994.

[13] D. Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.

[14] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[15] M. Odersky and K. Läufer. Putting type annotations to work. In *POPL'96*, pages 54–67, ACM, 1996.

[16] J. O'Toole and D. Gifford. Type reconstruction with first-class polymorphic values. In *PLDI'89*, pages 207–217, 1989. Published as SIGPLAN Notices, volume 27, number 7.

[17] L. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[18] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.

[19] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88*, pages 153–163, ACM, 1988.

[20] D. Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In M. Hagiya and J. C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software*, number 789 in LNCS, pages 321–346, Springer, 1994.

[21] D. Rémy. Simple, partial type inference for System F, based on type containment. In *ICFP'05*, pages 130–143, ACM, 2005.

[22] C. V. Russo. First-class Structures for Standard ML. *Nordic Journal Of Computing*, 7:348–374, January 2000.

[23] D. Vytiniotis, S. Weirich, and S. Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. In *ICFP'06*, ACM, 2006.

[24] D. Vytiniotis, S. Weirich, and S. Peyton Jones. FPH: first-class polymorphism for Haskell. In *ICFP'08*, ACM, 2008.

[25] J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98:111–156, 1999.