Refinement Types for ML

Tim Freeman
tsf@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Frank Pfenning
fp@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We describe a refinement of ML's type system allowing the specification of recursively defined subtypes of user-defined datatypes. The resulting system of *refinement types* preserves desirable properties of ML such as decidability of type inference, while at the same time allowing more errors to be detected at compile-time. The type system combines abstract interpretation with ideas from the intersection type discipline, but remains closely tied to ML in that refinement types are given only to programs which are already well-typed in ML.

1 Introduction

Standard ML [MTH90] is a practical programming language with higher-order functions, polymorphic types, and a well-developed module system. It is a statically typed language, which allows the compiler to detect many kinds of errors at compile time, thus leading to more reliable programs. Type inference is decidable and every well-typed expression has a principal type, which means that the programmer is free to omit type declarations (almost) anywhere in a program.

In this paper we summarize the design of a system of subtypes for ML which preserves the desirable properties listed above, while at the same time providing for specification and inference of significantly more precise type information. We call the resulting types *refinement types*, as they can be thought of as refining user-defined data types of ML. In particular, we do not extend the language of programs for ML (only the language of types) and, furthermore, we provide refined type information only for programs which are already well-typed in ML. In this preliminary report we only deal with an extension of Mini-ML [CDDK86], but we believe that the ideas described here can be further extended to the full Standard ML language.

To appear in ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.

To see the opportunity to improve ML's type system, consider the following function which returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons(hd,nil)) = last
  | lastcons (cons(hd,tl)) = lastcons tl
```

We know that this function will be undefined when called on an empty list, so we would like to obtain a type error at compile-time when `lastcons` is called with an argument of `nil`. Using refinement types this can be achieved, thus preventing runtime errors which could be caught at compile-time. Similarly, we would like to be able to write code such as

```
case lastcons y of
     cons(x,nil) => print x
```

without getting a compiler warning. However, the ML type system does not distinguish singleton lists from lists in general, so when compiling this case statement ML compilers will issue a warning because it does not handle all possible forms of lists. Here, refinement types allow us to eliminate unreachable cases.

Attempting to take such refined type information into account at compile time can very quickly lead to undecidable problems. The key idea which allows us to circumvent undecidability is that subtype distinctions (such as singleton lists as a subtype of arbitrary lists) must be made explicitly by the programmer in the form of recursive type declarations. In the example above, we can declare the refinement type of singleton lists as

```
datatype α list = nil | cons of α * α list
rectype α singleton = cons (α, nil)
```

This `rectype` declaration instructs the type checker to distinguish singleton lists from other lists. The datatype constructor names `cons` and `nil` in the right-hand side of the `rectype` declaration stand for subtypes which one can think of as subsets. At any type α the type expression `nil` can be interpreted as the set $\{\texttt{nil}\}$ and `cons(X,Y)` stands for $\{l \mid l = \texttt{cons}(x, y) \wedge x \in X \wedge y \in Y\}$.

We can think of the refinement type inference algorithm as performing abstract interpretation over a programmer-specified finite lattice of refinements of each ML type. Finiteness is important, since it is necessary for the decidability of refinement type inference in our system. Since type inference can only know that a list has one element if its tail is `nil`, it also needs to create a type for the empty list; suppose the name of this type is `?nil`. With this notation our abstract interpretation works over the following lattice:

$$\begin{tikzcd}
 & \alpha\ \texttt{list} \arrow[d, dash] & \\
 & \alpha\ \texttt{singleton} \vee \alpha\ \texttt{?nil} \arrow[dl, dash] \arrow[dr, dash] & \\
\alpha\ \texttt{singleton} \arrow[dr, dash] & & \alpha\ \texttt{?nil} \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

To perform the abstract interpretation, the type system needs to know the behavior of datatype constructors on these abstract domains. This can be expressed through refinement types given to each constructor. For example, `cons` applied to anything of type α and `nil` will return a singleton list:

$$\texttt{cons} : (\alpha * \alpha\ \texttt{?nil}) \to \alpha\ \texttt{singleton}$$

The constructor `cons` also has other types, such as:

$$\begin{aligned}
\texttt{cons} &: (\alpha * \alpha\ \texttt{singleton}) \to \alpha\ \texttt{list} \\
\texttt{cons} &: (\alpha * \alpha\ \texttt{list}) \qquad\;\;\; \to \alpha\ \texttt{list}
\end{aligned}$$

In our refinement type system, we express the principal type for `cons` by using the intersection operator "$\wedge$" to combine all these types, resulting in:

$$\begin{aligned}
\texttt{cons} : & (\alpha * \alpha\ \texttt{?nil}) && \to \alpha\ \texttt{singleton} \wedge \\
& (\alpha * \alpha\ \texttt{singleton}) && \to \alpha\ \texttt{list} \qquad \wedge \\
& (\alpha * \alpha\ \texttt{list}) && \to \alpha\ \texttt{list}
\end{aligned}$$

This type for `cons` can be generated automatically from the `rectype` declaration for `singleton` above.

We borrow the operator $\wedge$ from the intersection type discipline [RDR88], though we use it in a very restricted way here (we can only intersect types which are refinements of the same ML type). Type inference for intersection types, however, is in general undecidable. Thus, for them to be useful in a programming language, some explicit type annotations are required, as, for example, in Forsythe [Rey88]. Here we avoid explicit typing, while still retaining decidability, by allowing only the intersection of types which are subtypes of a common ML type. It is also this restriction which makes the combination of polymorphism and intersection types simple and direct (see [Pie89] for an investigation in a more general context).

As we will see later, we also require a form of union types so we can assign more accurate types to `case` statements. The inspiration for this and for the subtyping rules for function types (sketched in Section 4) are due to Pierce [Pie90].

In many examples, subtypes which could be specified explicitly as refinement types are implicit in current ML programs. For example, when we consider bitstrings to represent natural numbers, we have in mind a "standard form" of representation without leading zeros, and we would like to guarantee that functions such as addition and multiplication return standard forms when given standard forms. As another example, consider the representation of terms in a λ-calculus. Some natural manipulations of these terms will only work on terms in head normal form—a property which can easily be described via a refinement type.

Thus, using refinement types, the programmer is encouraged to make explicit the distinctions which currently must remain implicit or informal in code comments (such as standard form or head normal form in the examples above). Moreover, type errors can be described in a meaningful way if the type checker deals with the same quantities that the programmer understands. Thus, refinement types can increase the benefits of compile-time type checking and inference as already present in ML, without forcing the programmer to take advantage of them (any legal ML program continues to be legal if no refinements are specified). Finally, the fact that we tightly control the lattices making up the abstract domains makes type inference more practical and efficient.

Two examples of subtypes which cannot be specified as refinement types in our system are lists without repeated elements (to efficiently represent sets), and closed terms in a λ-calculus. Intuitively, this is because these sets cannot be described by regular expressions. In fact, our `rectype` declarations (with the proper restrictions, see Section 3) have a close connection to regular expressions since our declarations specify so-called regular tree sets for which many well-understood algorithms exist [GS84]. Regular tree sets have also shown themselves to be useful in the context of typed logic programming [Mis84, YFS91].

One interesting aspect of our proposal is that it merges two views which are traditionally considered as opposites: should recursive types be generative (as the ML `datatype` construct), or should they be non-generative (as, for example, in Quest [Car89] in functional programming or in typed HiLog [YFS91] in logic programming). Our conclusion is that generative types should be the principal notion, but that non-generative recursively defined *subtypes* can make a type system significantly more powerful and useful.

The remainder of the paper is organized as follows. In Section 2 we introduce the syntax of our language, following the presentation of Mini-ML [CDDK86]. In Section 3 we show how recursive refinement type declarations can be used to generate finite lattices defining the domain of abstract interpretation for type inference. These lattices of values induce subtype relationships on the function types, as we describe in Section 4. Polymorphism in ML requires a similar polymorphism for refinement types which we discuss in Section 5. We then present the type inference algorithm in form of some inference rules in Section 6. Finally, in Section 7 we discuss some aspects of implementation and future work.

2 The Language

In this section we define our extension to the language of types as present in ML and give some examples. The next three sections will then discuss the refinement type system and the problems of type checking and inference in more detail.

While we hope to eventually be able to extend all of Standard ML by `rectype` declarations, we confine ourselves in this paper to the functional portion of Standard ML. Moreover, for the purpose of this presentation we ignore product types and multi-arity type constructors. They can be added to the language in a straightforward way and, moreover, are not crucial in this context since functions with multiple arguments can be curried. In order to keep the examples in this paper intuitive and close to ML, we generally follow ML syntax. In order to give a manageable description of type inference in Section 6 we restrict ourselves there to a much simpler expression language which can be thought of as the result of "desugaring" the ML syntax used in the examples.

2.1 Rectype Declarations

The grammar for `rectype` declarations is:

```
rectype      ::= rectype rectypedecl
rectypedecl  ::= < mltyvar > reftyname = recursivety
                 < and rectypedecl >
recursivety  ::= (recursivety | recursivety) |
                 mlty → recursivety |
                 constructor recursivetyseq | mltyvar |
                 < mltyvar > reftyname
```

In this grammar (and other ones appearing in this paper), optional items are enclosed in "<>", and the different productions for each nonterminal are separated by "|". Note that the first occurrence of "`|`" in this grammar is part of the object language. The suffix *seq* added to a syntactic class name means either a nonempty, parenthesized list of elements of that syntactic class, or empty.

The syntactic classes used by the above grammar that are not defined there are

constructor	Constructors, such as `cons`.
mlty	ML types, such as α `list` or `int`.
mltyname	Datatype constructors, such as `list` or `bool`.
mltyvar	ML type variables, such as α.

Each rectype declaration must be consistent with the ML datatype it refines. For example, with the usual definition of lists, we could not accept `rectype` α `bad = nil(nil)` because `nil` does not take arguments.

To limit interactions of refinements types with polymorphism, we also require that when defining a recursive type, any uses of it within its own definition must have the same type variable argument that it has on the left hand side of the declaration.

2.2 Refinement Types

General refinement types can be built up from the usual ML types and from recursive types by the ML type constructor "→", and the new operations of intersection "∧" and union "∨". Intuitively, an expression has type $\sigma \wedge \tau$ if it has both type σ and type τ. Similarly, an expression has type $\sigma \vee \tau$ if it has type σ or type τ, though we may not be able to predict at compile time which one (such union types arise, for example, from the different branches of an `if` expression). Refinement types will be inferred and printed by type inference, or can be used by the programmer to annotate expressions, and, in an extension beyond the scope of this paper, they can appear in signatures.

The grammar for refinement types is:

$$\begin{aligned} \textit{refty} ::= {} & \textit{refty} \ \wedge \ \textit{refty} \mid \textit{refty} \ \vee \ \textit{refty} \mid \\ & \textit{refty} \rightarrow \textit{refty} \mid \ \bot \ \mid \\ & < \textit{refty} > \ \textit{mltyname} \mid \\ & < \textit{refty} > \ \textit{reftyname} \mid \\ & \textit{reftyvar} \ :: \ \textit{mltyvar} \end{aligned}$$

The syntactic classes used in this grammar that are not defined there are

reftyvar	Refinement type variables, written as $r\alpha$, $r\beta$, *etc.*
reftyname	Refinements of datatypes, like `singleton`.

Every refinement type variable is bounded by an ML type variable and thus ranges only over the refinements of an ML type. This is necessary to prevent undesirable interactions between polymorphism and subtypes (see Section 5 for further discussion). In contexts where the bound is obvious, we omit it.

Refinement type names are either declared explicitly (via `rectype`) or implicitly (as `?nil` was in the example in the introduction). See Section 3 for a discussion.

2.3 An Example

As a simple example consider the representation of natural numbers in binary, as in the ordinary ML datatype declaration

```
datatype bitstr =
   e | z of bitstr | o of bitstr
```

Here the constructor `e` makes an empty bitstring, `z` appends a zero as the least significant digit, and `o` appends a one as the least significant digit.

When we write functions to manipulate bitstrings, we would like to guarantee at compile time that a bitstring does not have a zero in the most significant place. We call this "standard form" (`std`). The declaration of this refinement type requires that we also introduce the type of positive natural numbers in standard form (`stdpos`):

```
rectype std = e | stdpos
and stdpos = o(e) | z(stdpos) | o(stdpos)
```

For example, the bitstring `z(e)` represents zero, but is not in standard form. The bitstring `z(o(e))` represents two, and is in standard form.

Using this `rectype` declaration, our type checking algorithm can check that

```
fun add e m = m
  | add n e = n
  | add (z n) (z m) = z(add n m)
  | add (o n) (z m) = o(add n m)
  | add (z n) (o m) = o(add n m)
  | add (o n) (o m) = z(add (add (o e) n) m)
```

maps standard form bitstrings to standard form bitstrings. More generally, it can infer that `add` has this somewhat unwieldy type:

```
?e     → ?e     → ?e     ∧
?e     → stdpos → stdpos ∧
?e     → std    → std    ∧
?e     → bitstr → bitstr ∧
stdpos → ?e     → stdpos ∧
stdpos → stdpos → stdpos ∧
stdpos → std    → stdpos ∧
stdpos → bitstr → bitstr ∧
std    → ?e     → std    ∧
std    → stdpos → stdpos ∧
std    → std    → std    ∧
std    → bitstr → bitstr ∧
bitstr → ?e     → bitstr ∧
bitstr → stdpos → bitstr ∧
bitstr → std    → bitstr ∧
bitstr → bitstr → bitstr
```

In this type we use `?e` to represent the type containing just the empty bitstring. This type has one conjunct for each nonempty refinement type we can assign to the arguments of `add`.

3 From Rectype Declarations to Datatype Lattices

A datatype declaration in ML introduces datatype constructors and declares their type. For the purpose of this exposition, we also assume that it implicitly defines a new constant `CASE_`*datatype* which can be used to simultaneously discriminate and destruct elements of the datatype (see example below). A recursive type declaration for a given datatype introduces at least one refinement type name, but many other refinement types can be formed by intersection, union, function type formation, *etc.*

Many of these refinement types will be equivalent. For example, $\sigma \wedge \sigma$ is always equivalent to σ, and, in the example above, `?e` $\vee$ `stdpos` is the same as `std`. A type checking or inference algorithm needs to understand these equivalences, and we will introduce the necessary structure in two steps. In this section we show how a `rectype` declaration induces a lattice of subtypes of a given ML datatype with the operations of $\wedge$ and $\vee$, understood as meet and join. In the following section we show how this information can be lifted to refinement types including the function type constructor $\rightarrow$.

Our `rectype` declarations are essentially regular tree grammars and they almost define regular tree sets as discussed in *Tree Automata* by Gécseg and Steinby [GS84]. The only change is that we have functions in our trees, but since we require our `rectype` declarations to have an ML type on the left-hand side of any $\rightarrow$, this extension turns out to be benign. We do not know of any useful examples which would be ruled out by this restriction. Algorithms for dealing with such declarations in *Subtyping Recursive Types* by Amadio and Cardelli [AC90] do not appear to apply directly to our situation.

Let us return to the declaration of bitstrings in standard form discussed above.

```
datatype bitstr =
   e | z of bitstr | o of bitstr
rectype std = e | stdpos
and stdpos = o(e) | z(stdpos) | o(stdpos)
```

Because we need to assign a refinement type to each constructor, we need to consider a lattice with more types than just `std` and `stdpos`. For instance, this `rectype` declaration requires the constructor `o` to map `stdpos`'s to `stdpos`'s, and `e`'s to `stdpos`'s. We can only express this as a refinement type if we create a refinement type containing just `e`, which we shall call `?e`. After creating this new refinement type, we can give this type for `o`:

$$\texttt{o} : \texttt{?e} \rightarrow \texttt{stdpos} \wedge \texttt{stdpos} \rightarrow \texttt{stdpos}$$

Thus the refinement types of `bitstr` are `bitstr`, `std`, `stdpos`, and `?e`. Using straightforward generalizations of the algorithms for manipulating regular tree grammars, we can infer that these four refinement types (plus $\perp$) are closed under intersection and union, and they form this lattice:

$$\begin{tikzcd}
 & \texttt{bitstr} \arrow[d, dash] & \\
 & \texttt{std} \arrow[dl, dash] \arrow[dr, dash] & \\
\texttt{?e} \arrow[dr, dash] & & \texttt{stdpos} \arrow[dl, dash] \\
 & \perp &
\end{tikzcd}$$

In general, closure under intersection and union may add many new elements to the lattice—a fact which, in an implementation, must be addressed through compact representation methods such as those described in "Graph-Based Algorithms for Boolean Function Manipulation" by Bryant [Bry86].

The types for the constructors in this example are calculated as

$$\begin{aligned}
&\texttt{e} : \texttt{?e}\\
&\texttt{o} : \texttt{?e} \rightarrow \texttt{stdpos} \wedge \texttt{stdpos} \rightarrow \texttt{stdpos}\\
&\texttt{z} : \texttt{stdpos} \rightarrow \texttt{stdpos}
\end{aligned}$$

Note that, even though `e` also has type `std`, we do not need to write `e : ?e` $\wedge$ `std`, since `?e` $\wedge$ `std` is equivalent to `?e`.

The case statements for elements of the datatype `bitstr` will look like

```
case E of
     e => E1
   | o(m) => E2
   | z(m) => E3
```

which we will treat as the following function call:

```
CASE_bitstr E
   (fn () => E1)
   (fn (m) => E2)
   (fn (m) => E3)
```

The algorithm which analyzes recursive type declarations assigns the type appearing in Figure 1 to `CASE_bitstr`. For an explanation of the type quantifiers in this figure, see Section 5.

4 From Datatype Lattices to Function Types

The datatype lattice is a representation of the subtype relationship and the behavior of intersection and union of refinements of an ML datatype. Next we need to consider function types. More specifically we will deal with how the subtype relationships, intersection, and union behave on the more general class of refinement types including "$\rightarrow$".

The basic principle underlying most subtype systems allowing higher-order functions is that of "contravariance": $\sigma_1 \rightarrow \tau_1 \leq \sigma_2 \rightarrow \tau_2$ if $\tau_1 \leq \tau_2$ and $\sigma_2 \leq \sigma_1$. If we think of $\sigma_1 \rightarrow \tau_1$ as a set of functions and $\leq$ as subset, we can see why: a function accepting σ_1 as type-correct input can certainly be given any element from a subtype of σ_1, and since it then produces a value in τ_1 this value will also be in τ_2. We say the type constructor "$\rightarrow$" is contravariant in its first argument and covariant in its second argument.

Defined datatype constructors may also be covariant or contravariant in their arguments, and our subtyping algorithm keeps track of this information in order to determine, for example, that `stdpos list` is a subtype of `std list`. In the rare case that the constructors for a datatype are neither all covariant nor all contravariant no useful subtyping information can be calculated by our algorithm. An example of this is the declaration

```
datatype α mixed = C1 of α  | C2 of α → bool
```

For defining subtype relations arising out of these basic observations and for the presentation of the type

$$\begin{array}{l}
\texttt{CASE\_bitstr} : \forall\alpha.\forall r\alpha_1 :: \alpha.\ \forall r\alpha_2 :: \alpha.\ \forall r\alpha_3 :: \alpha. \\
\texttt{?e} \quad\ \ \rightarrow(\texttt{unit}\rightarrow r\alpha_1)\rightarrow(\texttt{bitstr} \qquad\quad\ \rightarrow r\alpha_2)\rightarrow(\texttt{bitstr}\rightarrow r\alpha_3)\rightarrow r\alpha_1 \qquad\qquad\qquad \wedge \\
\texttt{stdpos}\rightarrow(\texttt{unit}\rightarrow r\alpha_1)\rightarrow((\texttt{?e} \vee \texttt{stdpos})\rightarrow r\alpha_2)\rightarrow(\texttt{stdpos}\rightarrow r\alpha_3)\rightarrow(r\alpha_2 \vee r\alpha_3) \qquad\quad \wedge \\
\texttt{std} \quad\ \rightarrow(\texttt{unit}\rightarrow r\alpha_1)\rightarrow((\texttt{?e} \vee \texttt{stdpos})\rightarrow r\alpha_2)\rightarrow(\texttt{stdpos}\rightarrow r\alpha_3)\rightarrow(r\alpha_1 \vee r\alpha_2 \vee r\alpha_3) \wedge \\
\texttt{bitstr}\rightarrow(\texttt{unit}\rightarrow r\alpha_1)\rightarrow(\texttt{bitstr} \qquad\quad\ \rightarrow r\alpha_2)\rightarrow(\texttt{bitstr}\rightarrow r\alpha_3)\rightarrow(r\alpha_1 \vee r\alpha_2 \vee r\alpha_3)
\end{array}$$

Figure 1: Type for `CASE_bitstr`.

inference algorithm in the next section, it is convenient to convert types to a normal form. We can do this by rewriting the type according to the following rewrite rules for any refinement types ρ, σ, and τ:

$$\begin{array}{c}
\rho \wedge (\sigma \vee \tau) \Rightarrow (\rho \wedge \sigma) \vee (\rho \wedge \tau) \\
(\rho \vee \sigma)\rightarrow\tau \Rightarrow (\rho\rightarrow\tau) \wedge (\sigma\rightarrow\tau)
\end{array}$$

Thinking of function types as sets of functions provides some insights about why these are valid transformations. Also, for any refinements of data types ρ, σ, and τ such that $\rho = \sigma \vee \tau$, we rewrite ρ to $\sigma \vee \tau$. After we apply these rewrites, the refinement types will fit the grammar

$$\begin{array}{lcl}
\mathit{unf} & ::= & \mathit{inf} \mid \mathit{unf} \vee \mathit{unf} \\
\mathit{inf} & ::= & < \mathit{unf} > \ \mathit{reftyname} \mid \mathit{inf} \wedge \mathit{inf} \mid \\
& & \mathit{inf}\rightarrow\mathit{unf} \mid \mathit{reftyvar} :: \mathit{mltyvar}
\end{array}$$

where *unf* stands for *union normal form* and *inf* for *intersection normal form.*

We now define the subtype ordering $\sigma \leq \tau$ for *unf* refinement types σ and τ where σ and τ are refinements of the same ML type. We have two cases, either their common ML type is a datatype or it is a function type.

If the bounding ML type is a datatype, the subtype relationship is determined by the partial order of the lattice.

If the bounding ML type is a function type, the *unf* refinement types have the form of a union of *inf* refinement types σ_i and σ'_j, and we have the rule

$$\begin{array}{l}
\sigma_1 \vee \sigma_2 \vee \ldots \vee \sigma_n \leq \sigma'_1 \vee \sigma'_2 \vee \ldots \vee \sigma'_m \\
\text{if for each } \sigma_i \text{ there is a } \sigma'_j \text{ such that } \sigma_i \leq \sigma'_j.
\end{array}$$

which leaves us with the problem of comparing *inf* refinements of functional types.

Given *inf* refinements for a function and its argument, we can compute a refinement type for the value of the function application: if the function has type $\sigma = (\rho_1\rightarrow\tau_1) \wedge (\rho_2\rightarrow\tau_2) \wedge \ldots \wedge (\rho_n\rightarrow\tau_n)$ and the argument has type ρ, then the type of their application (written apptype(σ, ρ)) is

$$\bigwedge_{\{i \mid \rho \leq \rho_i\}} \tau_i,$$

where $\bigwedge$ stands for intersection of a set of types.

We can use this to solve the subtype problem for *inf* refinements of functional types. Suppose we are trying to solve the problem $\sigma \leq \sigma'$, where

$$\sigma = (\rho_1\rightarrow\tau_1) \wedge (\rho_2\rightarrow\tau_2) \wedge \ldots \wedge (\rho_n\rightarrow\tau_n),$$

and

$$\sigma' = (\rho'_1\rightarrow\tau'_1) \wedge (\rho'_2\rightarrow\tau'_2) \wedge \ldots \wedge (\rho'_m\rightarrow\tau'_m).$$

In this case we define $\sigma \leq \sigma'$ to mean, for all ρ in $\{\rho_1, \rho_2, \ldots, \rho_n, \rho'_1, \rho'_2, \ldots, \rho'_m\}$,

$$\text{apptype}(\sigma, \rho) \leq \text{apptype}(\sigma', \rho).$$

The correctness of this definition in general is implied by the theorem stated in Section 6: if an expression of type σ evaluates to a value v, then v will also have type σ. On the other hand, it is quite possible that more subtype relationships hold than can be established with the rules above, which means that the types inferred for higher-order functions may not be as accurate as possible. This is another case where decidability must be balanced with the desire for accuracy in type checking.

5 Polymorphism

The interaction between polymorphism and subtypes is potentially problematic. The main mechanism considered so far in the literature is *bounded quantification* [CCHO89, CW85], where the domain of a type variable is restricted to range over subtypes of a given bound. In this paper we continue the separation of the ML types and refinement types and obtain a restricted form of bounded quantification. We define refinement type schemes by the following grammar:

$$\begin{array}{lcl}
\mathit{reftyscheme} & ::= & \mathit{inf} \mid \\
& & \forall\alpha \ . \ \mathit{reftyscheme} \mid \\
& & \forall r\alpha :: \alpha \ . \ \mathit{reftyscheme}
\end{array}$$

The first case in this grammar refers to types in intersection normal form, which we have already discussed.

The second case is quantification over an ML type variable, which is very similar to quantification over ML type variables as used in ML. This can be regarded as an infinite intersection; for instance, the identity function

```
val id = fn x => x
```

has the ML type

$$\forall\alpha \,.\, \alpha\to\alpha$$

which we can loosely regard as the intersection of

$$\alpha\to\alpha$$

for all ML types α, although in practice we do not represent ML types this way.

The third case quantifies over a refinement type variable, and we also regard this as an intersection. However, once we instantiate the ML type variable with an ML type, there are only finitely many refinements of that ML type, so there are only finitely many types in the intersection. When we instantiate a refinement type, we perform this expansion. For example, the refinement type for the identity function `id` is

$$\forall\alpha.\ \forall r\alpha :: \alpha.\ r\alpha\to r\alpha$$

If we instantiate α to `bitstr` and instantiate the refinement type quantifier, we get the refinement type

$$\begin{array}{ccc} \texttt{bitstr}\to\texttt{bitstr} & \wedge \\ \texttt{stdpos}\to\texttt{stdpos} & \wedge \\ \texttt{std}\ \to\ \texttt{std} & \wedge \\ \texttt{?e}\ \to\ \texttt{?e} & \wedge \\ \bot\ \to\ \bot & \end{array}$$

With this notion of instantiation, the refinements of ML type variables are exactly the refinement type variables. This notion is already implicit in the earlier examples: α `singleton` is a refinement of α `list`, but neither `bool singleton` nor `std list` are refinements of α `list`. A relaxation of this notion could quickly lead to undecidable type inference problems, as in the Milner-Mycroft calculus [KTU89, Myc84]. On the other hand, this restriction entails some loss of accuracy in determining refinement type information in some cases. Refinement type schemes are considered during type inference when analyzing a `let` expression and when instantiating the type of a polymorphic variable or constant.

6 The Type Inference Algorithm

This section will present a type inference algorithm for refinement types as a deductive system. Just as the deductive system for ML types leads to type inference by unification, our deductive system, too, can be given an operational interpretation which first performs an ML type inference pass and then a refinement type inference pass using abstract interpretation. Space unfortunately does not permit a more detailed discussion of the inference rules or their operational reading.

We infer refinement types for the program by inference system in Figure 2. The characters are used as follows in the inference rules:

e, e'	are expressions,
x, f	are ML variables,
C	is a refinement type in *inf*,
D	is a refinement type in *unf*,
L	is an ML type,
Γ	is an environment mapping variables to refinement type schemes, and
S	is a refinement type scheme.

The grammar for the language fragment used in this section is

$$\begin{array}{rl} \mathit{exp} ::= & \mathit{variable} \mid \mathit{exp}\ \mathit{exp} \mid \\ & \lambda\ \mathit{variable}.\ \mathit{exp} \mid \\ & \mathit{exp}\ :\ \mathit{refty} \mid \\ & \texttt{let}\ \mathit{variable} = \mathit{exp}\ \texttt{in}\ \mathit{exp} \mid \\ & \texttt{fix}\ \mathit{variable}.\ \mathit{exp} \end{array}$$

The notation $\Gamma \vdash e : D :: L$ means that in the type environment Γ, the expression e has refinement type D which is a refinement of the ML type L.

If we take this inference system and eliminate all of the refinement types, leaving just the expressions and the ML types, we get a conventional inference system for Mini-ML.

The rules in Figure 2 use the auxiliary judgment LOOP to compute successive approximations to the refinement type of recursive functions until a fixpoint is reached. This is guaranteed to terminate because there are only finitely many refinement types below a given ML type (which our algorithm computes first). The expression "close($\Gamma, C :: L$)" generalizes over the free type variables in C and L which are not free in Γ and returns the resulting refinement type scheme.

Now we shall state a theorem that the typing rules stated above are sound. This is sometimes paraphrased as *well-typed programs cannot go wrong*, that is, if an expression has refinement type σ, and evaluation of that expression terminates, then the value of the expression will also have the type σ. The operational semantics is very close to the one given for Mini-ML [CDDK86] and we omit it here.

Theorem: For all valid type environments Γ and expressions e, if e evaluates to v and $\Gamma \vdash e : D :: L$ then $\Gamma \vdash v : D' :: L$ for some $D' \leq D$.

INST: $\Gamma \vdash x : C :: L \qquad$ if $x : S$ is in Γ and $C :: L$ is an instance of S.

$$\text{APPL: } \frac{\Gamma \vdash e : \bigvee_i C_i :: L_1 \to L_2 \qquad \Gamma \vdash e' : \bigvee_j C'_j :: L_1}{\Gamma \vdash e\, e' : \bigvee_{i,j} \text{apptype}(C_i, C'_j) :: L_2}$$

$$\text{ABS: } \frac{\Gamma, x : C_i :: L_1 \vdash e : D_i :: L_2 \qquad \text{for each } C_i \text{ that is a refinement of } L_1}{\Gamma \vdash \lambda x.\ e : \bigwedge_i (C_i \to D_i) :: L_1 \to L_2}$$

$$\text{LET: } \frac{\Gamma \vdash e_1 : \bigvee_i C_i :: L_1 \qquad (\Gamma, x : (\text{close } (\Gamma, C_i :: L_1)) \vdash e_2 : D_i :: L_2) \text{ for each } C_i}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \bigvee_i D_i :: L_2}$$

$$\text{RESTRICT: } \frac{\Gamma \vdash e : D :: L \qquad D \leq D'}{\Gamma \vdash (e{:}D') : D' :: L}$$

$$\text{FIX1: } \frac{}{\Gamma \vdash \text{LOOP}(f, \lambda y.\ e, \bot, L)}$$

$$\text{FIX2: } \frac{\Gamma \vdash \text{LOOP}(f, \lambda y.\ e, C_1, L) \qquad \Gamma, f : C_1 :: L \vdash \lambda y.\ e : C_2 :: L}{\Gamma \vdash \text{LOOP}(f, \lambda y.\ e, C_2, L)}$$

$$\text{FIX3: } \frac{\Gamma \vdash \text{LOOP}(f, \lambda y.\ e, C, L) \qquad \Gamma, f : C :: L \vdash \lambda y.\ e : C :: L}{\Gamma \vdash \texttt{fix } f.\ \lambda y.\ e : C :: L}$$

Figure 2: Rules for type inference system

We omit the proof, which proceeds by induction on the structure of the definition of the "evaluates to" relation.

7 Future Work

We currently have a naive prototype implementation of the type inference algorithm as shown above. This prototype takes the lattice for each datatype, the types for the constructors, and the types for the case statements as inputs. The main implementation problem appears to be to deal efficiently with refinements of types of higher-order functions, as the number of such refinement types can become large very quickly. For example, since there are five refinements of the ML type `stdpos` used in the examples earlier, there are 5^5 functions mapping refinement types of `stdpos` to refinement types of `stdpos`. A naive representation of the refinement types of `stdpos`$\to$`stdpos` would list all of these functions. Compact representations of refinement types, for example through an appropriate generalization of Binary Decision Diagrams [Bry86, BCM+90] to deal with function types, seem promising. Since finding a type error in a program with refinement types will require looking at representations of refinement types, we will have to find a reasonably concise way to print these types.

The more refinements we consider of a given datatype, the slower type checking will be. This problem is alleviated when more distinct datatype declarations are made even if the datatypes present would be sufficient to encode the information we need to represent. In ML, this technique is good programming style in any case, as it enhances program readability and allows more type errors to be detected at compile-time. We also need to consider embedded refinement type declarations (as in `let rectype ... in ... end`) which naturally extends ML datatype declarations and also limits the visibility of refinements, thus cutting down on the size of the refinement type lattice.

The refinement types proposed here address only a subset of Standard ML [MTH90]. We need to carefully examine the interaction of refinement types with other features of the ML type system, such as imperative type variables and equality types, since we would like to extend our proposal to encompass all of Standard ML. Although Standard ML does not provide primitives for manipulating the current continuation, some dialects of ML do, so we would like to be able to deal with `callcc` also. Despite some potential problems which may lead to a loss of accuracy of refinement type information across modules, refinement types open the possibility of communicating some information about functions between modules without violating the privacy of the modules. This appears to be much more difficult, if not impossible, in approaches using general set constraints as, for example, in [HJ90], or abstract interpretation which is not tied to the type system.

We also would like to explore the possibility of refining predefined types, such as `int`, which are not given as `datatype` declarations. There are no conceptual difficulties as long as the appropriate subtype structure forms a lattice. For example, we could distinguish the positive integers, the negative integers, and zero by giving appropriate types to constants appearing in the program and to the arithmetic operators. We would have to devise some notation for doing this other than `rectype` declarations because we do not have constructors for the integers.

In some cases the refinement type information can be used for program optimization during compilation. We would like to explore this possibility further, though our primary motivation remains static detection of program errors which currently elude the ML type checker.

8 Acknowledgements

Thanks to Benjamin Pierce, Dave MacQueen, and John Reynolds for discussions about refinement types; Benjamin Pierce, Nevin Heintze, and the anonymous referees for proofreading this paper; and to John Reynolds for his TeX macros to draw diagrams.

This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under Contract N00039-85-C-0163, ARPA Order No. 5167.

References

[AC90] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. Research Report 62, Digital Systems Research Center, Palo Alto, California, August 1990.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society Press.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Car89] Luca Cardelli. Typeful programming. Research Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1989.

[CCHO89] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Functional Programming Languages and Computer Architecture.* ACM, 1989.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming.* ACM Press, 1986.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[HJ90] Nevin Heintze and Joxan Jaffar. A decision procedure for a class of set constraints. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, Philadelphia.* IEEE, June 1990.

[KTU89] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type-checking in the presence of polymorphic recursion. To appear in TOPLAS, October 1989.

[Mis84] Prateek Mishra. Towards a theory of types in Prolog. In *International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[Myc84] Alan Mycroft. *Polymorphic Type Schemes and Recursive Definitions*, pages 217–228. International Symposium on Programming. Springer-Verlag, New York, 1984. LNCS 167.

[Pie89] Benjamin Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical Report

CMU-CS-89-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1989.

[Pie90] Benjamin C. Pierce. Preliminary investigation of a calculus with intersection and union types. Unpublished manuscript, June 1990.

[RDR88] Simone Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[YFS91] Eyal Yardeni, Thom Fruehwirth, and Ehud Shapiro. Polymorphically typed logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1991. To appear.