PHANTOM TYPES AND SUBTYPING

Matthew Fluet and Riccardo Pucella
Department of Computer Science
Cornell University
{fluet,riccardo}@cs.cornell.edu

Abstract We investigate a technique from the literature, called the phantom types technique, that uses parametric polymorphism, type constraints, and unification of polymorphic types to model a subtyping hierarchy. Hindley-Milner type systems, such as the one found in ML, can be used to enforce the subtyping relation. We show that this technique can be used to encode any finite subtyping hierarchy (including hierarchies arising from multiple interface inheritance). We then formally demonstrate the suitability of the phantom types technique for capturing subtyping by exhibiting a type-preserving translation from a simple calculus with bounded polymorphism to a calculus embodying the type system of ML.

1. Introduction

It is well known that traditional type systems, such as the one found in Standard ML [10], with parametric polymorphism and type constructors can be used to capture program properties beyond those naturally associated with a Hindley-Milner type system [9]. For concreteness, let us review a simple example, due to Leijen and Meijer [8]. Consider a type of atoms, either booleans or integers, that can be easily represented as an algebraic datatype:

```
datatype atom = I of int | B of bool
```

There are a number of operations that we may perform on such atoms (see Figure 1(a)). When the domain of an operation is restricted to only one kind of atom, as with `conj` and `double`, a run-time check must be made and an error or exception reported if the check fails.

One aim of static type checking is to reduce the number of run-time checks by catching type errors at compile time. Of course, in the example above, the ML type system does not consider `conj (mkI 3, mkB true)` to be ill-typed; evaluating this expression will simply raise a run-time exception.

If we were working in a language with subtyping, we would like to consider integer atoms and boolean atoms as distinct subtypes of the general type of atoms and use these subtypes to refine the types of the operations. Then the type system would report a type error in the expression `double (mkB false)` at compile time. Fortunately, we can write the operations in a way that utilizes the ML type system to do just this. We

```
fun mkI (i:int):atom = I (i)
fun mkB (b:bool):atom = B (b)

fun toString (v:atom):string =
  (case v
    of I (i) => Int.toString (i)
     | B (b) => Bool.toString (b))
fun double (v:atom):atom =
  (case v
    of I (i) => I (i * 2)
     | _ => raise Fail "type mismatch")
fun conj (v1:atom,
          v2:atom):atom =
  (case (v1,v2)
    of (B (b1), B (b2)) => B (b1 andalso b2)
     | _ => raise Fail "type mismatch")
```

(a) Unsafe operations

```
fun mkI (i:int):int atom = I (i)
fun mkB (b:bool):bool atom = B (b)

fun toString (v:'a atom):string =
  (case v
    of I (i) => Int.toString (i)
     | B (b) => Bool.toString (b))
fun double (v:int atom):int atom =
  (case v
    of I (i) => I (i * 2)
     | _ => raise Fail "type mismatch")
fun conj (v1:bool atom,
          v2:bool atom):bool atom =
  (case (v1,v2)
    of (B (b1), B (b2)) => B (b1 andalso b2)
     | _ => raise Fail "type mismatch")
```

(b) Safe operations

Figure 1

change the definition of the datatype to the following:

```
datatype 'a atom = I of int | B of bool
```

and constrain the types of the operations (see Figure 1(b)). We use the superfluous type variable in the datatype definition to encode information about the kind of atom. (Because instantiations of this type variable do not contribute to the run-time representation of atoms, it is called a *phantom type*.) The type *int atom* is used to represent integer atoms and *bool atom* is used to represent boolean atoms. Now, the expression `conj (mkI 3, mkB true)` results in a compile-time type error, because the types *int atom* and *bool atom* do not unify. (Observe that our use of `int` and `bool` as phantom types is arbitrary; we could have used any two types that do not unify to make the integer versus boolean distinction.) On the other hand, both `toString (mkI 3)` and `toString (mkB true)` are well-typed; `toString` can be used on any atom. This is the essence of the technique explored in this paper: using a free type variable to encode subtyping information and using an ML-like type system to enforce the subtyping. This "phantom types" technique, where user-defined restrictions are reflected in the constrained types of values and functions, underlies many interesting uses of type systems [14, 12, 2, 13, 6, 8, 5, 11, 1].

The main contributions of this paper are to exhibit a general encoding of subtyping hierarchies and to give one formalization of the use of the phantom types technique. We present a type-preserving translation from a calculus with subtyping to a calculus with let-bounded polymorphism. The kind of subtyping that can be captured turns out to be an interesting variant of bounded polymorphism [3], with a very restricted subsumption rule.

This paper is structured as follows. In the next section, we describe a simple recipe for deriving an interface enforcing a given subtyping hierarchy. The interface is parameterized by an encoding, via phantom types, of the subtyping hierarchy. In Section 3, we focus on a simple encoding for hierarchies. In Section 4, we extend the recipe to capture a limited form of bounded polymorphism. In Section 5, we formally define the kind of subtyping captured by our encodings by giving a simple calculus with subtyping and showing that our encodings provide a type-preserving translation to a variant of the Damas-Milner calculus, embodying the essence of the ML type system. We conclude with some problems inherent to the approach and a consideration of future work. Due to space considerations, proofs of our results, a more involved discussion of the encodings in Section 3, as well as the full typing rules for the formalization in Section 5 have been deferred to the full paper.

2. From subtyping to polymorphism

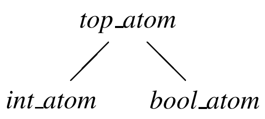

Figure 2

The example in the introduction has the following features: an underlying primitive type of values (the original type *atom*), a set of operations, and "implicit" subtypes that correspond to the sensible domains of the operations. The subtyping hierarchy corresponding to the example is given in Figure 2. The subtyping hierarchy is modeled by assigning a type to every implicit subtype in the hierarchy. For instance, integer atoms with implicit subtype *int_atom* are encoded by the ML type *int atom*. The appropriate use of polymorphic type variables in the type of an operation indicates the maximal type in the domain of the operation. For instance, the operation `toString` has the conceptual type *top_atom* $\rightarrow$ *string* which is encoded by the ML type *'a atom* $\rightarrow$ *string*. The key observation is the use of type unification to enforce the subtyping hierarchy: an *int atom* can be passed to a function expecting an *'a atom*, because these types unify.

We consider the following problem. Given an abstract type τ_p, a subtyping hierarchy, and an implementation of τ_p and its operations, we wish to derive a "safe" ML signature which uses phantom types to encode the subtyping and a "safe" implementation from the "unsafe" implementation. We will call the elements of the subtyping hierarchy *implicit types* and talk about *implicit subtyping* in the hierarchy. All values share the same underlying representation and each operation has a single implementation that acts on this underlying representation. The imposed subtyping captures restrictions that arise because of some external knowledge about the semantics of the operations; intuitively, it captures a "real" subtyping relationship that is not exposed by the abstract type.

We first consider deriving the safe interface. The new interface defines a type $\alpha\ \tau$ corresponding to the abstract type τ_p. The type variable α will be used to encode implicit subtype information. We require an encoding $\langle\sigma\rangle$ of each implicit type σ in the hierarchy; this encoding should yield a type in the underlying ML type system, with the property that $\langle\sigma_1\rangle$ unifies with $\langle\sigma_2\rangle$ if and only if σ_1 is an implicit subtype of σ_2. An obvious issue is that we want to use unification (a symmetric relation) to capture subtyping (an asymmetric relation). The simplest approach is to use two encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ defined over all the implicit types in the hierarchy. A *value* of implicit type σ will be assigned a type $\langle\sigma\rangle_C\ \tau$. We call $\langle\sigma\rangle_C$ the *concrete* subtype encoding of σ, and we assume that it uses only ground types (i.e., no type variables). In order to restrict the domain of an operation to the set of values in any implicit subtype of σ, we use $\langle\sigma\rangle_A$, the *abstract* subtype encoding of σ. In order for the underlying type system to enforce the subtype hierarchy, we require the encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ to be

```
signature ATOM = sig
  type atom
  val int : int -> atom
  val bool : bool -> atom
  val toString : atom -> string
  val double : atom -> atom
  val conj : atom * atom -> atom
end
```

(a) Unsafe signature

```
signature SAFE_ATOM = sig
  type 'a atom
  val int : int -> ⟨int⟩_C atom
  val bool : bool -> ⟨bool⟩_C atom
  val toString : ⟨top⟩_A atom -> string
  val double : ⟨int⟩_A atom -> ⟨int⟩_C atom
  val conj : ⟨bool⟩_A atom * ⟨bool⟩_A atom -> ⟨bool⟩_C atom
end
```

(b) Safe signature

Figure 3

respectful by satisfying the following property:

$$\text{for all } \sigma_1 \text{ and } \sigma_2,\ \langle\sigma_1\rangle_C \text{ matches } \langle\sigma_2\rangle_A \text{ iff } \sigma_1 \leq \sigma_2.$$

For example, the encodings used in the introduction are respectful:

$$\begin{array}{rclrcl} \langle \textit{top\_atom} \rangle_A & = & \textit{'a atom} & \langle \textit{top\_atom} \rangle_C & = & \textit{unit atom} \\ \langle \textit{int\_atom} \rangle_A & = & \textit{int atom} & \langle \textit{int\_atom} \rangle_C & = & \textit{int atom} \\ \langle \textit{bool\_atom} \rangle_A & = & \textit{bool atom} & \langle \textit{bool\_atom} \rangle_C & = & \textit{bool atom} \end{array}$$

The utility of the phantom types technique relies on being able to find respectful encodings for subtyping hierarchies of interest.

To allow for matching, the abstract subtype encoding will introduce free type variables. Since in a Hindley-Milner type system, a type cannot contain free type variables, the abstract encoding will be part of the larger type scheme of some polymorphic function operating on the value of implicit subtypes. This leads to some restrictions on when we should constrain values by concrete or abstract encodings. We will restrict ourselves to using concrete encodings in all covariant type positions, and using abstract encodings in most contravariant type positions. We will return to this issue in Section 5.

Consider again the example from the introduction. Assume we have encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ for the hierarchy and a structure `Atom` implementing the "unsafe" operations, with the signature given in Figure 3(a). Deriving an interface using the recipe above, we get the safe signature given in Figure 3(b).

We must now derive a corresponding "safe" implementation. We need a type $\alpha\ \tau$ isomorphic to τ_p such that the type system considers $\tau_1\ \tau$ and $\tau_2\ \tau$ equivalent iff τ_1 and

```
structure SafeAtom1 :> SAFE_ATOM = struct
  type 'a atom = Atom.atom
  val int = Atom.int
  val bool = Atom.bool
  val toString = Atom.toString
  val double = Atom.double
  val conj = Atom.conj
end
```

(a) Opaque signature

```
structure SafeAtom2 : SAFE_ATOM = struct
  datatype 'a atom = C of Atom.atom
  fun int (i) = C (Atom.int (i))
  fun bool (b) = C (Atom.bool (b))
  fun toString (C v) = Atom.toString (v)
  fun double (C v) = C (Atom.double (v))
  fun conj (C b1, C b2) = C (Atom.conj (b1,b2))
end
```

(b) Datatype declaration

Figure 4

τ_2 are equivalent. (Note that this requirement precludes the use of type abbreviations of the form `type` $\alpha\ \tau$ `=` τ_p, which define constant type functions.) We can then constrain the types of values and operations using $\langle\sigma\rangle_C\ \tau$ and $\langle\sigma\rangle_A\ \tau$. In ML, the easiest way to achieve this is to use an abstract type at the module system level, as shown in Figure 4(a). The use of an opaque signature is critical to get the required behavior in terms of type equivalence. The advantage of this method is that there is no overhead.

In a language without abstract types at the module level, another approach is to wrap the primitive type τ_p using a datatype declaration

$$\texttt{datatype 'a } \tau \texttt{ = C of } \tau_p$$

The type $\alpha\,\tau$ behaves as required, because the datatype declaration defines a generative type operator. However, we must explicitly convert primitive values to and from $\alpha\ \tau$ to witness the isomorphism. This yields the implementation given in Figure 4(b).

We should stress that the "safe" interface must ensure that the type $\alpha\,\tau$ is abstract—either through the use of opaque signature matching, or by hiding the value constructors of the type. Otherwise, it may be possible to create values that do not respect the subtyping invariants enforced by the encodings. Similarly, the use of an abstract subtype encoding in a covariant type position can lead to violations in the subtyping invariants.

We now have a way to derive a safe interface and implementation, by adding type information to a generic, unsafe implementation. In the next section, we show how to construct respectful encodings $\langle\cdot\rangle_C$ and $\langle\cdot\rangle_A$ by taking advantage of the structure of the subtyping hierarchy.

3. Encoding subtyping hierarchies

The framework presented in the previous section relies on having concrete and abstract encodings of the implicit subtypes in the subtyping hierarchy with the property that unification of the results of the encoding respects the subtype relation. In this section, we describe one general construction for such encodings.

We first consider a particular lattice that will be useful in our development. Recall that a lattice is a hierarchy where every set of elements has both a least upper bound and a greatest lower bound. Given a finite set S, we let the *powerset lattice* of S be the lattice of all subsets of S, ordered by inclusion, written $(\wp(S), \subseteq)$. We now exhibit an encoding of powerset lattices.

Let n be the cardinality of S and assume an ordering $s_1, \ldots, s_n$ on the elements of S. We encode subset X of S as an n-tuple type, where the i^{th} entry expresses that $s_i \in X$ or $s_i \notin X$. First, we introduce a datatype definition:

```
datatype 'a z = Z
```

(The name of the datatype constructor is irrelevant, because we will never construct values of this type.) The encoding of an arbitrary subset of S is given by:

$$\begin{aligned}
\langle X\rangle_C &= (t_1, \ldots, t_n) \quad \text{where } t_i = \begin{cases} \texttt{unit} & \text{if } s_i \in X \\ \texttt{unit z} & \text{otherwise} \end{cases} \\
\langle X\rangle_A &= (t_1, \ldots, t_n) \quad \text{where } t_i = \begin{cases} \alpha_i & \text{if } s_i \in X \\ \alpha_i\ \texttt{z} & \text{otherwise} \end{cases}
\end{aligned}$$

Note that $\langle\cdot\rangle_A$ requires every type variabe α_i to be a fresh type variable, unique in its context. This ensures that we do not inadvertently refer to any type variable bound in the context where we are introducing the abstractly encoded type.

As an example, consider the powerset lattice of $\{1, 2, 3, 4\}$, which encodes into a four-tuple. We can verify, for example, that the concrete encoding for $\{2\}$, namely *(unit z, unit, unit z, unit z)*, unifies with the abstract encoding for $\{1, 2\}$, namely *(*α_1*,* α_2*,* α_3 *z,* α_4 *z)*. On the other hand, the concrete encoding of $\{1, 2\}$ does not unify with the abstract encoding of $\{2, 3\}$.

The main reason we introduced powerset lattices is the fact that any finite hierarchy can be embedded in the powerset lattice of a set S. It is a simple matter, given a hierarchy H' embedded in a hierarchy H, to derive an encoding for H' given an encoding for H. Let $inj(\cdot)$ be the injection from H' to H witnessing the embedding and let $\langle\cdot\rangle_{C_H}$ and $\langle\cdot\rangle_{A_H}$ be the encodings for the hierarchy H. Deriving an encoding for H' simply involves defining $\langle\sigma\rangle_{C_{H'}} = \langle inj(\sigma)\rangle_{C_H}$ and $\langle\sigma\rangle_{A_{H'}} = \langle inj(\sigma)\rangle_{A_H}$. It is straightforward to verify that if $\langle\cdot\rangle_{C_H}$ and $\langle\cdot\rangle_{A_H}$ are respectful encodings, so are $\langle\cdot\rangle_{C_{H'}}$ and $\langle\cdot\rangle_{A_{H'}}$. By the result above, this allows us to derive an encoding for an arbitrary finite hierarchy.

We have presented a strategy for obtaining respectful encodings, which is sufficient for the remainder of this paper. However, there are encodings for specific hierarchies that are in general more efficient than their embedding in a powerset lattice, for instance, the encoding for tree hierarchies found in [6]. We discuss such encodings and address the issue of encoding extensibility in the full paper.

4. Towards bounded polymorphism

As mentioned in Section 3, the handling of type variables is somewhat delicate. If we allow common type variables to be used across abstract encodings, then we can capture a form of *bounded polymorphism* as in $F_{<:}$ [3]. Bounded polymorphism *à la* $F_{<:}$ is a typing discipline which extends both parametric polymorphism and subtyping. From parametric polymorphism, it borrows type variables and universal quantification; from subtyping, it allows one to set bounds on quantified type variables. For example, one can guarantee that the argument and return types of a function are the same and a subtype of σ, as in $\forall\alpha \leq \sigma.\alpha \to \alpha$. Similarly, one can guarantee that two arguments have the same type that is a subtype of σ, as in $\forall\alpha \leq \sigma.(\alpha \times \alpha) \to \sigma$. Notice that neither function can be written in a language that supports only subtyping.

Returning to the example from the introduction, consider adding natural numbers as a subtype of integers, so that *nat_atom* is a subtype of *int_atom*. Using bounded polymorphism, we can assign to `double` the reasonable type $\forall\alpha \leq$ *int_atom*$.\alpha \to \alpha$. However, bounded polymorphism has its limitations. One reasonable type for a `plus` operation is $\forall\alpha \leq$ *int_atom*$.\alpha \times \alpha \to \alpha$ where the same kind of atom is required for both arguments. In order to add an integer and a natural number we need a function `toInt` (operationally, an identity function) to coerce the type of the natural number to that of an integer.

We can adapt our "recipe" from Section 2 to types of the form $\forall\beta \leq \sigma_1.(\beta \times \sigma_2) \to \beta$. Let the "safe" interface use types of the form $\alpha\ \tau$. Since β stands for a subtype of σ_1, we let $\phi_\beta = \langle\sigma_1\rangle_A$, the abstract encoding of the bound. We then translate the type as we did in Section 2, but replace occurrences of the type variable β by ϕ_β instead of applying $\langle\cdot\rangle_A$ repeatedly, thereby sharing the type variables introduced by $\langle\sigma_1\rangle_A$. Hence, we get the type $\phi_\beta\ \tau \times \langle\sigma_2\rangle_A\ \tau \to \phi_\beta\ \tau$. In fact, we can further simplify the process by noting that we can "pull out" all the subtyping into bounded polymorphism. If a function expects an argument of any implicit subtype of σ, we can introduce a fresh type variable for that argument and bound it by σ. For example, the type above can be rewritten as: $\forall\beta \le \sigma_1, \gamma \le \sigma_2.(\beta \times \gamma) \to \beta$.

Unfortunately, this technique does not generalize to full $F_{<:}$. For example, we cannot encode bounded polymorphism where the bound on a type variable uses a type variable, such as a function `f` with type $\forall\alpha \le \sigma, \beta \le \alpha.\alpha \times \beta \to \alpha$. Encoding this type as $\phi_\alpha\ \tau \times \phi_\beta\ \tau \to \phi_\alpha\ \tau$ where $\phi_\alpha = \langle\sigma\rangle_A$ and $\phi_\beta = \langle\alpha\rangle_A$ fails, because we have no definition of $\langle\alpha\rangle_A$. Essentially, we need a different encoding of β for each instantiation of α at each application of `f`, something that cannot be accommodated by a single encoding of the type at the definition of `f`.

Likewise, we cannot encode first-class polymorphism, such as a function `g` with type $\forall\alpha \le \sigma_1.\alpha \to (\forall\beta \le \sigma_2.\beta \to \beta)$. Applying the technique yields a type $\phi_\alpha\ \tau \to \phi_\beta\ \tau \to \phi_\beta\ \tau$ where ϕ_α and ϕ_β contain free type variables. A Hindley-Milner style type system requires quantification over these variables in prenex position, which doesn't match the intuition of the original type. In fact, because we are translating into a language with prenex polymorphism, we can only capture bounded polymorphism that is itself in prenex form.

In other words, we cannot account for the general subsuption rule found in$F_{<:}$. Instead, we require all subtyping to occur at type application. This is the real motivation for the simplification above which "pulls out" all subtyping into bounded polymorphism. By introducing type variables for each argument, we move the resolution of the subtyping to the point of type application (when we instantiate the type variables).

These two restrictions impose one final restriction on the kind of subtyping we can encode. Consider a higher-order function `h` with type $\alpha \le (\sigma_1 \to \sigma_2).\alpha \to \sigma_2$. What are the possible encodings of the bound $\sigma_1 \to \sigma_2$ that allow subtyping? Clearly encoding the bound as $\langle\sigma_1\rangle_C\ \tau \to \langle\sigma_2\rangle_C\ \tau$ does not allow any subtyping. Encoding the bound as $\langle\sigma_1\rangle_A\ \tau \to \langle\sigma_2\rangle_A\ \tau$ or $\langle\sigma_1\rangle_A\ \tau \to \langle\sigma_2\rangle_C\ \tau$ leads to an unsound system. (Consider applying the argument function to a value of type $\sigma_0 \ge \sigma_1$, which would type-check in the encoding, because $\langle\sigma_0\rangle_C$ unifies with $\langle\sigma_1\rangle_A$ by the definition of a respectful encoding.) However, we can soundly encode the bound as $\langle\sigma_1\rangle_C\ \tau \to \langle\sigma_2\rangle_A\ \tau$. This corresponds to a subtyping rule on functional types that asserts $\tau_1 \to \tau_2 \le \tau_1 \to \tau_2'$ iff $\tau_2 \le \tau_2'$.

Despite these restrictions, the phantom types technique is still a viable method for encoding subtyping in a language like ML. All of the examples of phantom types found in the literature satisfy these restrictions. In practice, one rarely needs first-class polymorphism or complicated dependencies between the subtypes of function arguments, particularly when implementing a safe interface to existing library functions.

5. A formalization

There are subtle issues regarding the kind of subtyping that can be captured using phantom types. In this section, we clarify the picture by exhibiting a typed calculus with a suitable notion of subtyping that can be faithfully translated into a language such as ML, via a phantom types encoding. The idea is simple: to see if an interface can be implemented using phantom types, first express the interface in this calculus in such a way that the program type-checks. If it is possible to do so, our results show that a translation using phantom types exists. The target of the translation is a calculus embodying the essence of ML, essentially the calculus of Damas and Milner [4], a predicative polymorphic λ-calculus.

Let us first introduce the source calculus, $\lambda^{\mathrm{DM}}_{<:}$, also a variant of the Damas-Milner calculus, but with a very restricted notion of subtyping, and allowing multiple types for constants. We assume a partially ordered set $(T, \leq)$ of basic types. The types and prenex quantified type schemes of $\lambda^{\mathrm{DM}}_{<:}$ are as follows:

$$\begin{array}{lll} \tau & ::= & t \mid \alpha \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \forall \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n . \tau \end{array}$$

(where $t \in T$). Furthermore, we make a syntactic restriction that precludes the use of type variables in the bounds of quantified type variables.

An important aspect of our calculus, at least for our purposes, is the constants that we allow. We distinguish between two types of constants: basic constants and primitive operations. Basic constants, taken from a set C_b, are constants representing values of basic types $t \in T$. We suppose a function $\pi_b : C_b \to T$ assigning a basic type to every basic constant. The primitive operations, taken from a set C_p, are operations acting on constants and returning constants.[1] Rather than giving primitive operations polymorphic types, we assume that the operations can have multiple types, which encode the allowed subtyping. The primitive operation `double` in our example would get the types *int_value* $\to$ *int_value* and *nat_value* $\to$ *nat_value*. We suppose a function π_p assigning to every constant $c \in C_p$ a set of types $\pi_p(c)$, each type a functional type of the form $t \to t'$ (for $t, t' \in T$).

Our expression language is again typical:

$$\begin{array}{lll} e & ::= & c \mid \lambda x : \tau . e \mid e_1\, e_2 \mid x \mid p\,[\tau_1, \ldots, \tau_n] \mid \textbf{let}\ x = p\ \textbf{in}\ e \\ p & ::= & x \mid \Lambda \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n . e \\ v & ::= & c \mid \lambda x : \tau . e \\ E & ::= & [\,] \mid E\, e \mid v\, E \mid E\,[\tau_1, \ldots, \tau_n] \mid \textbf{let}\ x = E\ \textbf{in}\ e \end{array}$$

(where $c \in C_b \cup C_p$). The operational semantics are given using a standard rewriting system. The basic reductions are

$$\begin{array}{rcl} (\lambda x : \tau . e)\, v & \longrightarrow_{<:} & e\{v/x\} \\ (\Lambda \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n . e)\,[\tau_1', \ldots, \tau_n'] & \longrightarrow_{<:} & e\{\tau_1'/\alpha_1, \ldots, \tau_n'/\alpha_n\} \\ \textbf{let}\ x = v\ \textbf{in}\ e & \longrightarrow_{<:} & e\{v/x\} \\ c_1\, c_2 & \longrightarrow_{<:} & c_3 \quad \text{iff } \delta(c_1, c_2) = c_3 \end{array}$$

where $\delta : C_p \times C_b \rightharpoonup C_p$ is a partial function defining the result of applying a primitive operation to a basic constant. This reduction extends to contexts via the rule:

$$E[e_1] \longrightarrow_{<:} E[e_2] \ \text{ iff } \ e_1 \longrightarrow_{<:} e_2$$

[1]For simplicity, we will not deal with higher-order functions here—they would simply complicate the formalism without bringing any new insight. Likewise, allowing primitive operations to act on and return tuples of values is a simple extension of the formalism presented here.

As previously noted, we only allow primitive operations to be monotyped. However, we can easily use the fact that they can take on many types to write polymorphic wrappers. Returning to the `double` example, we can write a polymorphic wrapper $\Lambda\alpha <: \mathit{int\_value}.\lambda x : \alpha.\texttt{double}\ x$ to capture the expected behavior. We will see shortly that this function is well-typed.

The typing rules for $\lambda^{\mathrm{DM}}_{<:}$ are the standard Damas-Milner typing rules, modified to account for subtyping. Subtyping is given by a judgment $\Delta \vdash_{<:} \tau_1 <: \tau_2$, and is derived from the subtyping on the basic types. The interesting rules are:

$$\frac{t_1 \leq t_2}{\Delta \vdash_{<:} t_1 <: t_2} \qquad \frac{\Delta \vdash_{<:} \tau_2 <: \tau_2'}{\Delta \vdash_{<:} \tau_1 \to \tau_2 <: \tau_1 \to \tau_2'}$$

Notice that subtyping at higher types only involves the result type. The typing rules are given by judgments $\Delta; \Gamma \vdash_{<:} e : \tau$ for monotypes and $\Delta; \Gamma \vdash_{<:} p : \sigma$ for type schemes. The rule for primitive operations is interesting:

$$\frac{\forall \tau_1' <: \tau_1, \ldots, \tau_n' <: \tau_n \quad \tau\{\tau_1'/\alpha_1, \ldots, \tau_n'/\alpha_n\} \in \pi_p(c)}{\Delta, \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n; \Gamma \vdash_{<:} c : \tau} \quad \left(\begin{array}{c} c \in C_p, \\ FV(\tau) = \langle \alpha_1, \ldots, \alpha_n \rangle \end{array} \right)$$

The syntactic restriction on type variable bounds ensures that each τ_i has no type variables, so each $\tau_i' <: \tau_i$ is well-defined. The rule captures the notion that any subtyping on a primitive operation through the use of bounded polymorphism is in fact realized by the "many types" interpretation of the operation.

Subtyping occurs at type application:

$$\frac{\Delta; \Gamma \vdash_{<:} p : \forall \alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n.\tau \quad \Delta \vdash_{<:} \tau_1' <: \tau_1 \quad \cdots \quad \Delta \vdash_{<:} \tau_n' <: \tau_n}{\Delta; \Gamma \vdash_{<:} p\ [\tau_1', \ldots, \tau_n'] : \tau\{\tau_1'/\alpha_1, \ldots, \tau_n'/\alpha_n\}}$$

As discussed in the previous section, there is no subsumption in the system: subtyping must be witnessed by type application. Hence, there is a difference between the type $t_1 \to t_2$ (where $t_1, t_2 \in T$) and $\forall \alpha <: t_1.\alpha \to t_2$; namely, the former does not allow any subtyping. The restrictions of Section 4 are formalized by prenex quantification and the syntactic restriction on type variable bounds.

Clearly, type soundness of the above system depends on the definition of δ over the constants. We say that π_p is sound with respect to δ if for all $c_1 \in C_p$ and $c_2 \in C_b$, we have $\vdash_{<:} c_1\ c_2 : \tau$ implies that $\delta(c_1, c_2)$ is defined and $\pi_b(\delta(c_1, c_2)) = \tau$. This definition ensures that any application of a primitive operation c_1 to a basic constant c_2 yields exactly one value $\delta(c_1, c_2)$ at exactly one type $\pi_b(\delta(c_1, c_2)) = \tau$. This leads to the following conditional type soundness result for $\lambda^{\mathrm{DM}}_{<:}$:

Theorem 1 *If π_p is sound with respect to δ, $\vdash_{<:} e : \tau$, and $e \longrightarrow_{<:} e'$, then $\vdash_{<:} e' : \tau$ and either e' is a value or there exists e'' such that $e' \longrightarrow_{<:} e''$.*

Our target calculus, $\lambda^{\mathrm{DM}}_{\mathsf{T}}$, is meant to capture the appropriate aspects of ML that are relevant for the phantom types encoding of subtyping. Essentially, it is the Damas-Milner calculus [4] extended with a single type constructor T. Formally,

$$\begin{array}{rcl}
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \mathsf{T}\ \tau \mid 1 \mid \tau_1 \times \tau_2 \\
\sigma & ::= & \forall \alpha_1, \ldots, \alpha_n.\tau \\
e & ::= & c \mid \lambda x : \tau.e \mid e_1\ e_2 \mid p[\tau_1, \ldots, \tau_n] \mid x \mid \textbf{let}\ x = p\ \textbf{in}\ e \\
v & ::= & c \mid \lambda x : \tau.e \\
p & ::= & x \mid \Lambda \alpha_1 \ldots, \alpha_n.e \\
E & ::= & [\,] \mid E\ e \mid v\ E \mid E\ [\tau_1, \ldots, \tau_n] \mid \textbf{let}\ x = E\ \textbf{in}\ e
\end{array}$$

The operational semantics (via a reduction relation $\longrightarrow_{\mathsf{T}}$) and most typing rules (via a judgment $\Delta;\Gamma \vdash_{\mathsf{T}} e : \tau$) are standard. As before, we assume that we have constants C_b and C_p and a function δ providing semantics for primitive applications. Likewise, we assume that π_b and π_p provide types for constants, with the same restrictions. The typing rule for primitive operations in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$ is similar to the corresponding rule in $\lambda_{<:}^{\mathrm{DM}}$. Given two types τ and τ' in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$, we define their unification $\mathit{unify}(\tau,\tau')$ to be a sequence of bindings $\langle(\alpha_1,\tau_1),(\alpha_2,\tau_2),\ldots\rangle$ in depth-first, left-to-right order of appearance of $\alpha_1,\ldots,\alpha_n$ in τ, or $\emptyset$ if τ' is not a substitution instance of τ. Given a type τ in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$, we define $FV(\tau)$ to be the sequence of free type variables appearing in τ, in depth-first, left-to-right order.

$$\frac{\begin{array}{c}\forall\tau'\in\pi_b(C_b)\text{ with }\mathit{unify}(\tau_1,\tau')=\langle(\alpha_1,\tau_1'),\ldots,(\alpha_n,\tau_n'),\ldots\rangle\\ (\tau_1\to\tau_2)\{\tau_1'/\alpha_1,\ldots,\tau_n'/\alpha_n\}\in\pi_p(c)\end{array}}{\Delta,\alpha_1,\ldots,\alpha_n;\Gamma\vdash_{\mathsf{T}} c:\tau_1\to\tau_2}\quad\left(\begin{array}{c}c\in C_p,\\ FV(\tau_1\to\tau_2)=\\ \langle\alpha_1,\ldots,\alpha_n\rangle\end{array}\right)$$

Again, this rule captures our notion of "subtyping through unification" by ensuring that the operation is defined at every basic type that unifies with its argument type. Our notion of soundness of π_p with respect to δ carries over and we can again establish a conditional type soundness result:

Theorem 2 *If π_p is sound with respect to δ, $\vdash_{\mathsf{T}} e:\tau$, and $e \longrightarrow_{\mathsf{T}} e'$, then $\vdash_{\mathsf{T}} e':\tau$ and either e' is a value or there exists e'' such that $e' \longrightarrow_{\mathsf{T}} e''$.*

Note that the types $\mathsf{T}\ \tau$, 1, and $\tau_1\times\tau_2$ have no corresponding introduction and elimination expressions. We include these types for the exclusive purpose of constructing the phantom types used by the encodings. We could add other types to allow more encodings, but these suffice for the lattice encodings of Section 3.

Thus far, we have a calculus $\lambda_{<:}^{\mathrm{DM}}$ embodying the notion of subtyping that interests us and a calculus $\lambda_{\mathsf{T}}^{\mathrm{DM}}$ capturing the essence of the ML type system. We now establish a translation from the first calculus into the second using phantom types to encode the subtyping, showing that we can indeed capture that particular notion of subtyping in ML. Moreover, we show that the translation preserves the soundness of the types assigned to constants, thereby guaranteeing that if the original system was sound, the system obtained by translation is sound as well.

We first describe how to translate types in $\lambda_{<:}^{\mathrm{DM}}$. Since subtyping is only witnessed at type abstraction, the type translation realizes the subtyping using the phantom types encoding of abstract and concrete subtypes. The translation is parameterized by an environment ρ associating every (free) type variable with a type in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$ representing the abstract encoding of the bound.

$$\begin{array}{rcl}\mathcal{T}[\![\alpha]\!]\rho &=& \rho(\alpha)\\ \mathcal{T}[\![t]\!]\rho &=& \mathsf{T}\ \langle t\rangle_C\\ \mathcal{T}[\![\tau_1\to\tau_2]\!]\rho &=& \mathcal{T}[\![\tau_1]\!]\rho\to\mathcal{T}[\![\tau_2]\!]\rho\\ \mathcal{T}[\![\forall\alpha_1<:\tau_1,\ldots,\alpha_n<:\tau_n.\tau]\!]\rho &=& \\ \multicolumn{3}{r}{\forall\alpha_{11},\ldots,\alpha_{1k_1},\ldots,\alpha_{n1},\ldots,\alpha_{nk_n}.\mathcal{T}[\![\tau]\!]\rho[\alpha_i\mapsto\tau_i^A]}\\ \multicolumn{3}{r}{\text{where }\tau_i^A=\mathcal{A}[\![\tau_i]\!]\text{ and }FV(\tau_i^A)=\langle\alpha_{i1},\ldots,\alpha_{ik_i}\rangle}\end{array}$$

If ρ is empty, we will simple write $\mathcal{T}[\![\tau]\!]$. To compute the abstract and concrete encodings of a type, we define:

$$\begin{array}{rclcrcl}\mathcal{A}[\![t]\!] &=& \mathsf{T}\ \langle t\rangle_A & \qquad & \mathcal{C}[\![t]\!] &=& \mathsf{T}\ \langle t\rangle_C\\ \mathcal{A}[\![\tau_1\to\tau_2]\!] &=& \mathcal{C}[\![\tau_1]\!]\to\mathcal{A}[\![\tau_2]\!] & & \mathcal{C}[\![\tau_1\to\tau_2]\!] &=& \mathcal{C}[\![\tau_1]\!]\to\mathcal{C}[\![\tau_2]\!]\end{array}$$

Note that the syntactic restriction on type variable bounds ensures that $\mathcal{A}$ and $\mathcal{C}$ are always well-defined, as they will never be applied to type variables. Furthermore, observe that the above translation depends on the fact that the type encodings $\langle t\rangle_C$ and $\langle t\rangle_A$ are expressible in the $\lambda_{\mathsf{T}}^{\mathrm{DM}}$ type system using T, 1, and $\times$.

We extend the type transformation $\mathcal{T}$ to type contexts Γ in the obvious way:

$$\mathcal{T}[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!]\rho \quad = \quad x_1 : \mathcal{T}[\![\tau_1]\!]\rho, \ldots, x_n : \mathcal{T}[\![\tau_n]\!]\rho$$

Finally, if we take the basic constants and the primitive operations in $\lambda_{<:}^{\mathrm{DM}}$ and assume that π_p is sound with respect to δ, then the translation can be used to assign types to the constants and operations such that they are sound in the target calculus. We first extend the definition of $\mathcal{T}$ to π_b and π_p in the obvious way:

$$\begin{array}{rcl} \mathcal{T}[\![\pi_b]\!] & = & \pi_b' \quad \text{where } \pi_b'(c) = \mathcal{T}[\![\pi_b(c)]\!] \\ \mathcal{T}[\![\pi_p]\!] & = & \pi_p' \quad \text{where } \pi_p'(c) = \{\mathcal{T}[\![\tau]\!] \mid \tau \in \pi_p(c)\} \end{array}$$

We can further show that the translated types do not allow us to "misuse" the constants in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$:

Theorem 3 *If π_p is sound with respect to δ in $\lambda_{<:}^{\mathrm{DM}}$, then $\mathcal{T}[\![\pi_p]\!]$ is sound with respect to δ in $\lambda_{\mathsf{T}}^{\mathrm{DM}}$.*

We can now define the translation of expressions via a translation of typing derivations, $\mathcal{E}$, taking care to respect the types given by the above type translation. We note that the translation below only works if the concrete encodings being used do not contain free type variables. Again, the translation is parameterized by an environment ρ, as in the type translation.

$$\begin{array}{rcl} \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} x : \tau]\!]\rho & = & x \\ \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} c : \tau]\!]\rho & = & c \\ \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} \lambda x\!:\!\tau'.e : \tau]\!]\rho & = & \lambda x\!:\!\mathcal{T}[\![\tau]\!]\rho.\mathcal{E}[\![e]\!]\rho \\ \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} e_1\ e_2 : \tau]\!]\rho & = & (\mathcal{E}[\![e_1]\!]\rho)\ \mathcal{E}[\![e_2]\!]\rho \\ \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} \textbf{let}\ x = p\ \textbf{in}\ e : \tau]\!]\rho & = & \textbf{let}\ x = \mathcal{E}[\![p]\!]\rho\ \textbf{in}\ \mathcal{E}[\![e]\!]\rho \\ \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} p[\tau_1, \ldots, \tau_n] : \tau]\!]\rho & = & \\ \multicolumn{3}{r}{(\mathcal{E}[\![p]\!]\rho)[\tau_{11}, \cdots, \tau_{1k_1}, \cdots, \tau_{n1}, \cdots, \tau_{nk_n}]} \\ \multicolumn{3}{r}{\text{where } \mathcal{B}[\![p]\!]\Gamma = (\alpha_1, \tau_1^B), \ldots, (\alpha_n, \tau_n^B)\rangle \text{ and } \tau_i^A = \mathcal{A}[\![\tau_i^B]\!]} \\ \multicolumn{3}{r}{\text{and } FV(\tau_i^B) = \langle \alpha_{i1}, \ldots, \alpha_{ik_i}\rangle \text{ and } \tau_i^T = \mathcal{T}[\![\tau_i]\!]\rho} \\ \multicolumn{3}{r}{\text{and } \mathit{unify}(\tau_i^A, \tau_i^T) = \langle(\alpha_{i1}, \tau_{i1}), \ldots, (\alpha_{ik_i}, \tau_{ik_i}), \ldots\rangle} \end{array}$$

$$\begin{array}{rcl} \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} x : \sigma]\!]\rho & = & x \\ \mathcal{E}[\![\Delta;\Gamma \vdash_{<:} \Lambda\alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n.e : \sigma]\!]\rho & = & \\ \multicolumn{3}{r}{\Lambda\alpha_{11}, \ldots, \alpha_{1k_1}, \ldots, \alpha_{n1}, \ldots, \alpha_{nk_n}.\mathcal{E}[\![e]\!]\rho[\alpha_i \mapsto \tau_i^A]} \\ \multicolumn{3}{r}{\text{where } \tau_i^A = \mathcal{A}[\![\tau_i]\!] \text{ and } FV(\tau_i^A) = \langle \alpha_{i1}, \ldots, \alpha_{ik_i}\rangle} \end{array}$$

Again, if ρ is empty, we simply write $\mathcal{E}[\![e]\!]$. The function $\mathcal{B}$ returns the bounds of a type abstraction, using the environment Γ to resolve variables. It is defined as follows:

$$\begin{array}{rcl} \mathcal{B}[\![x]\!]\Gamma & = & \langle(\alpha_1, \tau_1), \ldots, (\alpha_n, \tau_n)\rangle \\ \multicolumn{3}{r}{\text{where } \Gamma(x) = \forall\alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n.\tau} \\ \mathcal{B}[\![\Lambda\alpha_1 <: \tau_1, \ldots, \alpha_n <: \tau_n.e]\!]\Gamma & = & \langle(\alpha_1, \tau_1), \ldots, (\alpha_n, \tau_n)\rangle \end{array}$$

We use $\mathcal{B}$ and *unify* to perform unification "by hand." In most programming languages, type inference performs this automatically.

We can verify that this translation is type-preserving:

Theorem 4 *If* $\vdash_{<:} e : \tau$, *then* $\vdash_{\mathsf{T}} \mathcal{E}[\![\vdash_{<:} e : \tau]\!] : \mathcal{T}[\![\tau]\!]$.

Theorem 4 is interesting in that it shows that the translation, in a sense, captures the right notion of subtyping, particularly when designing an interface. Given a set of constants making up the interface, suppose we can assign types to those constants in $\lambda^{\mathrm{DM}}_{<:}$ in a way that gives the desired subtyping; that is, we can write type correct expressions of the form $\Lambda\alpha <: t.\lambda x : \alpha.c\ x$ with type $\forall \alpha <: t.\alpha \rightarrow \tau$. In other words, the typing π_p is sound with respect to the semantics of δ. By Theorem 1, this means that $\lambda^{\mathrm{DM}}_{<:}$ with these constants is sound and we can safely use these constants in $\lambda^{\mathrm{DM}}_{<:}$. In particular, we can write the program:

$$\begin{array}{l} \textbf{let } f_1 = \Lambda\alpha <: t_{i_1}.\lambda x : \alpha.c_1\ x \textbf{ in} \\ \quad \vdots \\ \textbf{let } f_n = \Lambda\alpha <: t_{i_n}.\lambda x : \alpha.c_n\ x \textbf{ in} \\ e \end{array}$$

By Theorem 4, the translation of the above program executes without run-time errors. Furthermore, by Theorem 3, the phantom types encoding of the types of these constants are sound with respect to δ in $\lambda^{\mathrm{DM}}_{\mathsf{T}}$. Hence, by Theorem 2, $\lambda^{\mathrm{DM}}_{\mathsf{T}}$ with these constants is sound and we can safely use these constants in $\lambda^{\mathrm{DM}}_{\mathsf{T}}$. Therefore, we can replace to the body of the translated program with an arbitrary $\lambda^{\mathrm{DM}}_{\mathsf{T}}$ expression that type-checks in that context and the resulting program will still execute without run-time errors. Essentially, the translation of the let bindings corresponds to a "safe" interface to the primitives; programs that use this interface in a type-safe manner are guaranteed to execute without run-time errors.

6. Conclusion

Essentially, the phantom types technique uses the definition of type equivalence in ML to encode information in a free type variable of a type. Unification can then be used to enforce a particular structure on the information carried by two such types. In this paper, we have focused on encoding subtyping information. We also showed how to extend the techniques we developed to encode a form of prenex bounded polymorphism, with subsumption occuring only at type application. It goes without saying that this approach to encoding subtyping is not without its problems from a practical point of view. As the encodings in this paper show, the types involved can become quite large. Type abbreviations can help simplify the presentation of concrete types, but for abstract encodings, which require type variables, those type variables must appear in the interface.

We also note that the source language of Section 5 provides only a lower bound on the power of phantom types. For example, one can use features of the *specific* encoding used to further constrain operations [13]. One can also capture programming invariants associated with user-defined datatypes [7]. An interesting direction for future work is formalizing these additional applications of the phantom types technique.

Acknowledgments

We have benefitted from discussions with Greg Morrisett and Dave MacQueen. John Reppy pointed out the work of Burton. Stephanie Weirich, Vicky Weissman, and Steve Zdancewic provided helpful comments on an early draft of this paper. Thanks also to the anonymous referees. The second author was partially supported by ONR grant N00014-00-1-03-41.

References

[1] M. Blume. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". In *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[2] F. Burton. Type extension through polymorphism. *ACM Transactions on Programming Languages and Systems*, 12(1):135–138, January 1990.

[3] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[5] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *Workshop on Semantics, Applications, and Implementation of Program Generation*, 2000.

[6] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 114–125. ACM Press, 1999.

[7] S. Kahrs. Red-black trees with types. *Journal of Functional Programming*, 11(3):425–432, 2001.

[8] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL'99)*, pages 109–122, 1999.

[9] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17(3):348–375, 1978.

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

[11] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 276–290. ACM Press, 1999.

[12] D. Rémy. Records and variants as a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–88. ACM Press, 1989.

[13] J. H. Reppy. A safe interface to sockets. Technical memorandum, AT&T Bell Laboratories, 1996.

[14] M. Wand. Complete type inference for simple objects. In *Proceedings of the 2nd Annual IEEE Symposium on Logic in Computer Science*, 1987.