First-class subtypes

Jeremy Yallop  Stephen Dolan
University of Cambridge
{j Jeremy.yallop, stephen.dolan}@cl.cam.ac.uk

1. Introduction

One purpose of the ML module system is to hide equalities between types exposed in an interface and types used in the implementation [11]. Generalized algebraic data types (GADTs) [14], a more recent addition, make these type equalities first class. GADTs attach first class equalities to data; equalities may be hidden by polymorphism or modular abstraction, and later revealed by scrutinising the data.

While GADTs are frequently useful, type equality is sometimes too strong a property. For example, here is a function that prints arrays by calling the same method of each element:

```
let print_array = Array.iter (fun o -> print o#name)
```

To call `name`, `print_array` does not need to know the full element type: it needs only to know that there is a method `name` returning `string`. OCaml gives `print_array` a row type, indicating that the element type may have other methods:

```
val print_array : <name: string; ..> array
```

But rows are sometimes too inflexible. Given two arrays `a, b`, of different element types, unification will fail:

```
List.iter print_array [a; b]
```

This note introduces an interface based on a type constructor `sub` with a coercion operator `s >: t` that supports passing information around a program, making it possible to combine iterations over arrays whose element types belong to the same subtyping hierarchy.

```
type 'a arr = Arr : 'x array * ('x, 'a) sub
let print_array = Array.iter (fun s -> f (s >: sub)) a
List.iter (f (fun o -> print o#name)) [Arr (a,refl); Arr (b,refl)]
```

2. First-class subtypes defined

**Subtypes ala Liskov & Wing** The first ingredient in a representation of subtyping proofs is a definition of subtyping. Here is Liskov and Wing’s characterization [12]:

Let \( \phi(x) \) be a property provable about objects `x` of type `T`.

Then \( \phi(y) \) should be true for objects `y` of type `S` where `S` is a subtype of `T`.

For instance, properties of a record type `r` should also hold for a widening of `r`, since the extra fields can be ignored.

**Subtypes ala Curry & Howard** The Curry-Howard correspondence turns Liskov and Wing’s characterization of subtyping into an executable program.

With a propositions-as-types perspective [16], a property provable about objects of type `T` is represented as a type \( \phi(T) \) involving `T`, and a proof of that property is a value\(^2\) of that type. Liskov and Wing’s proposition that \( S \) is a subtype of `T` corresponds to the following (poly)type:

\[
\forall \phi. \phi(T) \to \phi(S)
\]

Two points deserve note. First, the characterization involves properties of all objects of a particular type, not properties of individual objects, which would need dependent types. Second, a “property about objects” is a context that consumes an object; \( \phi \) therefore ranges over negative contexts.

**Contexts and variance** Fig. 1 defines OCaml signatures, \( \text{POS} \) and \( \text{NEG} \), of positive and negative type contexts. The \( \_ \) indicates that the parameter can only appear in negative (contravariant) positions in instantiations of the signature. The `Id` module and `Compose` functors represent the identity context and the composition of two contexts. Each composition of variance in the argument contexts requires a separate `Compose` (but see §3 for a generalization).

**Encoding subtypes** Fig. 2 defines an interface to subtype witnesses. A value of type `(s, t)` `sub` serves as evidence that `s` is a subtype of `t`. There are two ways to construct such evidence. First, \( \text{refl} \) represents the fact that every type is a subtype of itself. Second, `lift` represents the fact that subtyping lifts through covariant contexts, which are passed as implicit arguments [18]. The single destructor, `>:`, which mimics OCaml’s built-in coercion operator `>`, supports converting a value of type `s` to a supertype `t`.

This small interface suffices as a basis for many useful subtyping-related functions. For example, the transitivity of subtyping is represented by a function of the following type:

```
let trans : (s : t) sub -> (t : s) sub
```

and may be defined as follows:

```
let trans (type a b) (x : (a,b) sub) y =
let module M = struct type '<c t = (a,'c) sub end
in x >: lift (90) y
```

and other operations, such as a function to lift through negative contexts, can be defined similarly (§A.1).

Using the variance of `s`, `refl` can be used to define a witness for any subtyping fact in the environment. For example, in OCaml the object type `<m:int >`, with one method `a`, is a subtype of the type `< >` of objects with no methods. This fact can be turned into a sub value by coercing `refl`, either by lowering the contravariant parameter:

```
(refl : (< >, < >) sub : (<m:int> , < >) sub)
```

or by raising the covariant parameter:

```
(refl : (<m:int>, <m:int>) sub : (< >, < >) sub)
```

The resulting value can be passed freely through abstraction boundaries that conceal the types involved, eventually being used to coerce a value of type `<m:int>` to its supertype `< >`.  

---

\(^2\)In total languages terms are proofs, but OCaml is not total.
type (-'a, +'b) sub = (N:NEG) → ('b N.t → 'a N.t)

let refl (N:NED) : x → x
let lift (P:POS) : s (Q:NED) x = s (Compose_+ (P) (Q))
let () : (type b) x → x
let module M = struct type '-'a t = 'a → b end in
  f () x

Figure 3. First-class subtypes via negative contexts

| type (-'a, +'b) sub = (P:POS) → ('a P.t → 'b P.t) |
|------|--------------------------------------------------|
| let refl (P:POS) : x → x |
| let lift (P:POS) : (Q:POS) x → s (Compose_+ (P) (Q)) x |
| let () : x → f (16) x |

but applying 1 cannot involve list traversal, since the subtyping interface says nothing about list structure. A polymorphic interface thus ensures an efficient implementation.

Three implementations of subtyping

Several implementations of Fig. 2 are possible. Fig. 3 gives an implementation based on negative contexts that directly follows Liskov & Wing’s definition. A value of type (s, t) sub is a proof that t can be replaced with s in any negative context; operationally it must be the identity, as discussed above, and so the two constructors lift and refl both correspond to the identity function. Fig. 4 gives a similar but simpler implementation, based on positive contexts. Fig. 5 takes an alternative view, based on initiality: (s, t) sub is the smallest contra/co-variant binary type constructor equipped with an inhabitant refl : (s, t) sub, from which there is a mapping into any other such type constructor with refl. Despite the different starting points, the three implementations are interdeﬁnable (§A.2).

All three implementations use the modular implicits extension to OCaml — not for implicit instantiation of arguments, but because modular implicits support higher-kindred quantification with propagation of variance information. Other approaches to higher-kindred polymorphism could perhaps be used instead [20].

From subtyping to equality

The variance annotations (+, -) in Figs. 1–5, constrain the instantiation of each type constructor. Without these annotations each representation of subtyping collapses to a representation of equality for which additional properties such as symmetry become derivable. For example, stripping the variance annotations turns Fig. 1 into the standard Leibniz encoding [6, 2, 5, 17, 19], and Fig. 5 into a Church encoding of the equality GADT [1].

3. First-class subtypes: further examples

Some of OCaml’s built-in type constructors, such as ref and array, are invariant, and so interact poorly with subtyping. The situation may be improved by decomposing each invariant type parameter into a co/contravariant pair [4], e.g. writing type (+r, -w) ref rather than type ‘a ref. With this decomposition, abstraction and first-class subtypes may be combined to selectively expose capabilities to different parts of a program: a function may be given the ability to write to a reference, write at a particular subtype, or not write at all (and similarly for read). Other types of capability (e.g. for file descriptors) can be written similarly.

A second class of examples arises from selective abstraction, where an abstract type comes with a proof of a property about that type. For example, here is a module that exports a type t along with a proof that t is a subtype of int:

module M : sig type (s, t) sub sub t end with type (s, t) sub

let 1 : (s list, t list) sub = lift (List) s_sub_t

but applying 1 cannot involve list traversal, since the subtyping interface says nothing about list structure. A polymorphic interface thus ensures an efficient implementation.

Three implementations of subtyping

Several implementations of Fig. 2 are possible. Fig. 3 gives an implementation based on negative contexts that directly follows Liskov & Wing’s definition. A value of type (s, t) sub is a proof that t can be replaced with s in any negative context; operationally it must be the identity, as discussed above, and so the two constructors lift and refl both correspond to the identity function. Fig. 4 gives a similar but simpler implementation, based on positive contexts. Fig. 5 takes an alternative view, based on initiality: (s, t) sub is the smallest contra/co-variant binary type constructor equipped with an inhabitant refl : (s, t) sub, from which there is a mapping into any other such type constructor with refl. Despite the different starting points, the three implementations are interdeﬁnable (§A.2).

All three implementations use the modular implicits extension to OCaml — not for implicit instantiation of arguments, but because modular implicits support higher-kindred quantification with propagation of variance information. Other approaches to higher-kindred polymorphism could perhaps be used instead [20].

From subtyping to equality

The variance annotations (+, -) in Figs. 1–5, constrain the instantiation of each type constructor. Without these annotations each representation of subtyping collapses to a representation of equality for which additional properties such as symmetry become derivable. For example, stripping the variance annotations turns Fig. 1 into the standard Leibniz encoding [6, 2, 5, 17, 19], and Fig. 5 into a Church encoding of the equality GADT [1].

3. First-class subtypes: further examples

Some of OCaml’s built-in type constructors, such as ref and array, are invariant, and so interact poorly with subtyping. The situation may be improved by decomposing each invariant type parameter into a co/contravariant pair [4], e.g. writing type (+r, -w) ref rather than type ‘a ref. With this decomposition, abstraction and first-class subtypes may be combined to selectively expose capabilities to different parts of a program: a function may be given the ability to write to a reference, write at a particular subtype, or not write at all (and similarly for read). Other types of capability (e.g. for file descriptors) can be written similarly.

A second class of examples arises from selective abstraction, where an abstract type comes with a proof of a property about that type. For example, here is a module that exports a type t along with a proof that t is a subtype of int:

module M : sig type t val t_sub_int (t, int) sub end

Outside the module, values of type t can be coerced to int, but not vice versa. This approach supports a style similar to refinement types, in which abstraction boundaries distinguish values of a type for which some additional predicate has been shown to hold. OCaml’s private types [9] provide direct language support for this feature, but first-class subtypes allow more flexibility: for example, they allow some of the methods of an object type to be hidden from the exposed interface, and also support the dual of private types (called invisible types [13]), and zero cost-coercions [3], where coercions in both directions are available, but actual type equality is not exposed.

Dual to abstraction, combining first-class subtypes with OCaml’s first-class polymorphism encodes bounded quantification. For example, the type ∀α ≤ t. α → t might be written like this: type t = (f : (s, t) sub → 'a → t ).

Finally, first-class subtypes can express proofs of variance. For example, the covariance of list can be represented by a value of type (s, t) sub → (list list) sub. Abstracting over proofs of variance, we might build a Compose functor that generalizes Compose_ and Compose_+ (Fig. 1).

4. Limitations and further work

The encodings given here are useful for exploratory work, for demonstrating OCaml’s expressivity. However, direct language support would make first-class subtypes more usable. Scherer and Rémy [13] discuss design issues and related work (e.g. [8, 15]).

The encodings suffer from some awkwardness, since contexts must be applied explicitly, unlike the equalities revealed by pattern matching with GADTs, which the type checker applies implicitly. With language support for subtype witnesses, coercions would still be explicit, but constraints in scope could be implicitly lifted through contexts.

First-class subtypes might be added to OCaml by extending the interpretation of the existing notation for GADTs, so that indexes with variance annotations carry subtyping constraints rather than equality constraints. For example, the sub type (Fig. 2) might be defined as follows:

| type (-'a, +b) sub = Sub : (s, t) sub |

so that when a value of type (a, b) sub matches a pattern sub the type checker knows that a is a subtype of b.

Extending an ML dialect centred around subtyping, such as MLsub [7], might prove even more fruitful.

Woodward Kmett has used this approach in the magpie library [10], as we discovered after writing this note.
A. Additional definitions

A.1 Lifting through negative contexts

The minimal interface (Fig. 2) supports the definition of several additional functions. The \texttt{lift} function in the interface lifts subtyping witnesses through positive contexts. The elements of the interface can be used to construct a companion function, \texttt{lift}, that lifts subtyping witnesses through negative contexts:

\begin{verbatim}
val lift_ : (N:NEG) \rightarrow (a, b) \texttt{sub} \rightarrow ('b N.t, 'a N.t) \texttt{sub}
\end{verbatim}

As with \texttt{trans}, implementing \texttt{lift_} is a matter of finding a suitable implementation of \texttt{PSG} to pass to \texttt{lift}:

\begin{verbatim}
let lift_ (type a b) (N:NEG) (x: (a,b) \texttt{sub}) : (b N.t, a N.t) \texttt{sub} =
  refl : lift (N) x
\end{verbatim}

A.2 Converting between encodings

The minimal interface (Fig. 2) is sufficiently rich that, given two implementations of the interface $\alpha$ and $\beta$, any subtyping witness of type $(\alpha,\beta) \texttt{A}.t$ can be converted to a witness of type $(\alpha,\beta) \texttt{B}.t$. The \texttt{SUB} module type contains the four elements of (Fig. 2) $(t, \texttt{refl}, \texttt{lift}, \triangleright)$:

\begin{verbatim}
module type SUB =
  sig
    include \texttt{DIAG}
    val lift : (P:POS) \rightarrow (a, 'b) \texttt{t} \rightarrow ('a P.t, 'b P.t) t
    val \triangleright : 'a \rightarrow ('a, 'b) \texttt{sub} \rightarrow 'b
  end
\end{verbatim}

The function \texttt{conv} takes two implementations of \texttt{SUB}, $\alpha$ and $\beta$, and converts a value in $\texttt{A}.t$ to a value of $\texttt{B}.t$:

\begin{verbatim}
val conv : (\alpha:SUB) \rightarrow (\beta:SUB) \rightarrow ('a, 'b) \texttt{A}.t \rightarrow ('a, 'b) \texttt{B}.t
\end{verbatim}

As often, implementing \texttt{conv} is a matter of finding a suitable implementation of \texttt{PSG} to pass to \texttt{lift}:

\begin{verbatim}
let conv (type a b) (\alpha:SUB) (\beta:SUB) (x : (a,b) \texttt{A}.t) =
  let module M = struct type 'a t = (a, 'a) \texttt{B}.t end in
  \alpha.(\triangleright) B.\texttt{refl}(M.\texttt{lift}(M) x)
\end{verbatim}

References