First-Class Subtypes

Jeremy Yallop
University of Cambridge
jeremy.yallop@cl.cam.ac.uk

Stephen Dolan
University of Cambridge
stephen.dolan@cl.cam.ac.uk

1 Introduction

One appealing feature of ML-family languages is the ability to define fundamental data structures — pairs, lists, streams, and so on — in user code. For example, although lazy computations are supported as a built-in construct in OCaml, it is also possible to implement them as a library.

Lazy in variants

The following data type can serve as a basis for lazy computations:

```plaintext
type 'a lazy_cell = | Thunk of (unit -> 'a)
                  | Value of 'a
                  | Exn of exn
```

The constructors of a `lazy_cell` value represent the three possible states of a lazy computation: it may be an unevaluated thunk, a fully-evaluated value, or a computation whose evaluation terminated with an exception. Since the state of a lazy computation may change over time, lazy values are represented as mutable references that hold `lazy_cell` values:

```plaintext
type 'a lzy = 'a lazy_cell ref
```

Finally, there are two operations: `delay` creates a thunk from a function, while `force` either forces a thunk and caches the result, or returns the value or exception cached by a previous call.

```plaintext
let delay f = ref (Thunk f)
let force r = match !r with
  | Thunk f -> (match f () with
    | v -> r := Value v; v
    | exception e -> r := Exn e; raise e)
  | Value v -> v
  | Exn e -> raise e
```

Lazy invariants

The characterising feature of lazy computations is that each computation is run only once, although the result may be read many times. The `delay` and `force` functions maintain this invariant. Concealing the representation type of `lzy` behind a module interface ensures that other parts of the program cannot violate it:

```plaintext
module Lazy :
sig
type 'a t
val delay : (unit -> 'a) -> 'a t
val force : 'a t -> 'a
end

module Lazy :
struct
type 'a t = 'a lzy
let delay = ... (* as above *)
let force = ... (* as above *)
end
```

Submitted to:
ML 2017

© J. Yallop & S. Dolan
This work is licensed under the Creative Commons Attribution License.
Lazy invariance  This simple implementation has the same behaviour as the built-in lazy type. However, there is one notable difference: unlike the built-in type, our Lazy.t is not covariant.

The lack of covariance has two significant consequences for programmers. First, computations constructed using the built-in type can be coerced, while computations constructed using our Lazy.t cannot. Here is a coercion from a built-in lazy computation that returns an object with one method \( m \) into a computation that returns an object with no methods:

```ocaml
define let o = object method m = () end;;
val o : < m : unit > = <obj>
define # (Lazy.o :> < > Lazy.t);;;
define - : < > Lazy.t = <lazy>
```

An attempt to similarly coerce our Lazy.t fails:

```ocaml
define # (Lazy.delay (fun () → o) :> < > Lazy.t);;;
Characters 0-40:
Lazy.delay (fun () → o) :> < > Lazy.t);;;

Error: Type < m : unit > Lazy.t is not a subtype of < > Lazy.t

The second object type has no method \( m \)
```

Second, let-bound computations constructed using the built-in lazy receive polymorphic types, following the relaxed value restriction, which generalizes type variables that appear only in covariant positions \(^1\):

```ocaml
define # let f = Lazy.from_fun (fun () → []);;;
val f : 'a list Lazy.t = <lazy>
```

In contrast, types built using our Lazy.t are ungeneralized, as the weak type variable \( '_a \) (distinguished by a leading underscore) indicates:

```ocaml
define # Lazy.delay (fun () → []);;;
- : '_a list Lazy.t = <abstr>
```

The interface to Lazy.t only exposes read operations, and so it would be safe for the type to be treated as covariant in its parameter. However, the assignment of variance considers only the use of the parameter in the definition of Lazy.t, not the broader module interface. Since the type parameter is passed to the the invariant type ref of mutable cells, the type Lazy.t is also considered invariant.

These shortcomings in the Lazy interface can be overcome with more flexible treatment of subtyping and variance\(^1\). In particular, making subtypes first-class make it possible to tie together the type definition and the functions exposed in the interface in the consideration of variance assignment, and so make Lazy.t covariant.

First-class subtypes can be defined using a binary type constructor:

```ocaml
type (-'a, +'b) sub
```

that has a single constructor:

\(^1\) An alternative solution is to switch to a higher-order representation of lazy computations, where \( 'a \) Lazy.t is defined as unit \( → 'a \). We leave the details as an exercise for the reader.
val refl : ('a, 'a) sub

and an operation that turns a sub value into a function:

val (>:) : 'a → ('a, 'b) sub → 'b

These three elements, considered in more detail in Section 2, are sufficient to define a covariant variant of Lazy. Section 2 also adds an addition constructor, lift that, together with the three elements above, suffices as a basis to define a range of useful subtyping operations.

Here is a covariant definition of lzy using sub:

type +_ lzy = L : ('a, 'b) sub * 'a lazy_cell ref → 'b t

That is, a value of type 'a lzy is a pair of an 'a lazy_cell ref and a value of type ('a, 'b) sub that supports coercions from 'a to 'b.

Now delay constructs both a computation in the initial Thunk state and a sub value:

let delay f = L (refl, ref (Thunk f))

Finally, the definition of force is modified to apply the coercion in the two places where the value of a computation is returned:

let force (L (sub, r)) =
  match !r with
  | Thunk f → (match f () with
      | v → r := Value v; (v >: sub)
      | exception e → r := Exn e; raise e)
  | Value v → (v >: sub)
  | Exn e → raise e

The interface of Lazy can now be updated to reflect the fact that lzy is covariant, by adding a + to the type parameter of Lazy.t:

module Lazy :
  sig
    type +'a t
    val delay : (unit → 'a) → 'a t
    val force : 'a t → 'a
  end

module Lazy =
  struct
    type 'a t = 'a lzy
    let delay = ...
    let force = ...
  end

With these modifications, the behaviour of Lazy is closer to the behaviour of the built-in lazy. For example, let-bound values built by Lazy.delay can now receive polymorphic types:

# Lazy.delay (fun () → []);
- : 'a list Lazy.t = <abstr>

2 First-class subtypes defined

2.1 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 [14]:

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. And dually, properties of a variant type \( v \) should also hold for a narrowing of \( v \): any property that holds for all constructors also holds for a subset of constructors.
2.2 Subtypes \textit{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 \cite{17}, 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\footnote{In total languages terms are proofs, but OCaml is not total.} of that type. Liskov and Wing’s proposition that $S$ is a subtype of $T$ corresponds to the following (poly)type:

$$\forall \phi. \phi(T) \rightarrow \phi(S)$$

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.

2.3 Contexts and variance

Figure 1 defines OCaml signatures, \texttt{POS} and \texttt{NEG}, of positive and negative type contexts. The \texttt{-} indicates that the parameter can only appear in negative (contravariant) positions in instantiations of the signature. The \texttt{Id} module and \texttt{Compose} functors represent the identity context and the composition of two contexts. Each composition of variance in the argument contexts requires a separate \texttt{Compose} (but see \S 4 for a generalization).

2.4 Encoding subtypes

Figure 2 defines an interface to subtype witnesses. A value of type $(s, t)$ \texttt{sub} serves as evidence that $s$ is a subtype of $t$. There are two ways to construct such evidence. First, \texttt{refl} represents the fact that every type is a subtype of itself. Second, \texttt{lift} represents the fact that subtyping lifts through covariant contexts, which are passed as implicit arguments \cite{19}. The single destructor, \texttt{>:}, which mimics OCaml’s built-in coercion operator $\Rightarrow$, supports converting a value of type $s$ to a supertype $t$. 

\begin{verbatim}
module type POS = sig type +'a t end
module type NEG = sig type -'a t end

module Id = struct type 'a t = 'a end
module Compose_+_=(F:NEG)(G:POS) = struct type 'a t = 'a F.t G.t end
module Compose_++=(F:POS)(G:POS) = struct type 'a t = 'a F.t G.t end

Figure 1: Positive and negative contexts

type ('-a, +'b) sub
val refl : ('a, 'a) sub
val lift: {P:POS} \rightarrow ('a,'b) sub \rightarrow ('a P.t,'b P.t) sub
val (>:) : 'a \rightarrow ('a, 'b) sub \rightarrow 'b

Figure 2: First-class subtypes: minimal interface
\end{verbatim}
type ('a, +'b) sub = {N:NEG} → ('b N.t → 'a N.t)

let refl {N:NEG} x = x

let lift {P:POS} s {Q:NEG} = s {Compose_-(P)(Q)}

let (>:) (type b) x f =
  let module M = struct type '-'a t = 'a → b end in
  f {M} id 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} s {Q:POS} x = s {Compose_+(P)(Q)} x

let (>:) x f = f {Id} x

Figure 4: First-class subtypes via positive contexts

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:

val trans : ('a,'b) sub → ('b,'c) sub → ('a,'c) 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 {M} y

Similarly, a function that lifts subtyping witnesses through negative contexts

val lift_- : {N:NEG} →('a, 'b) sub → ('b N.t, 'a N.t) sub

may also be defined be supplying a suitable implementation of POS to lift:

let lift_- (type a b) {N:NEG} (x : (a,b) sub) : (b N.t,a N.t) sub =
  let module M = struct type +'b t = (b N.t, a N.t) sub end in
  refl >: lift {M} x

Using the variance of sub, refl can be used to define a witness for any subtyping fact that holds in the typing environment. For example, in OCaml the object type < m:int >, with one method m, 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 => (<m:int>, < >) 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 < >.

The generality of the interface in Figure 2 places constraints on the implementation. Most notably, since lift can transport subtyping evidence through any positive context, coercion must pass values through unexamined. For example, lift might be used to build a value of type (s list, t list) sub from a value of type (s, t) sub:

let l : (s list, t list) sub = lift {List} s_sub_t

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

3.1 First-class subtypes via contexts

Figure [3] gives an implementation of Figure [2] 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 \(\text{lift}\) and \(\text{refl}\) both correspond to the identity function. Figure [4] gives a similar but simpler implementation, based on positive contexts. Apart from the variance annotations, these definitions mirror the standard Leibniz encoding of type equality [20].

3.2 First-class subtypes as an inductive type

Consider an ordinary inductive type, say the Peano natural numbers:

```ocaml
type nat = Zero | Suc nat
```

We can write its constructors in the form of a module signature as follows:

```ocaml
module type NAT = sig
  type t
  val zero : t
  val suc : t → t
end
```

What it means for the type \(\text{nat}\) to be inductive is that it is an initial algebra for this signature: first, it implements the signature by providing \(\text{Zero}\) and \(\text{Suc}\), and secondly, for any other implementation \(M\), we have a function mapping \(\text{nat}\) to \(M.t\) that maps \(\text{Zero}\) to \(M.\text{zero}\) and \(\text{Suc}\) to \(M.\text{suc}\):  

```ocaml
let rec primrec = function
| Zero  → M.zero
| Suc n → M.suc (primrec n)
```

In defining the type \(\text{nat}\), we made use of OCaml’s built-in support for inductive types. Lacking this, we could have used the initial algebra definition directly, and defined the type \(\text{nat}\) as follows:

```ocaml
type nat = \{M : NAT\} → M.t
```

This is the Church encoding of the natural numbers [7], in which a natural number is anything that can produce a \(M.t\) from \(M.\text{zero} : M.t\) and \(M.\text{suc} : M.t \to M.t\). Here and elsewhere we’re using the modular implicit extension to OCaml [19] — not for implicit instantiation of arguments, but because modular implicit supports higher-kinded quantification with propagation of variance information. Other approaches to higher-kinded polymorphism could perhaps be used instead [21].

This approach to inductive types also makes sense for GADTs, such as the equality GADT defined below [3, 6, 18]:

```ocaml
type ('a, 'b) eq = Refl : ('a, 'a) eq
```

This uses OCaml’s GADT syntax, but we can do without it by building a Church encoding of equality [2], using the same technique as before:

---

3 Edward Kmett has used this approach in the magpie library [13], as we discovered after writing this note.
module type SUB =
  sig
    type ('a, 'b) t
    val refl : ('a, 'a) t
  end

type ('a, 'b) sub = {S:SUB} → ('a,'b) S.t

let refl {S:SUB} = S.refl

let lift {P:POS} (f : ('a, 'b) sub) =
  let module L =
    struct
      type ('a,'b) t = ('a P.t,'b P.t) sub
      let refl = refl
    end in
    f {L}
  in

let (>:) x (f : ('a, 'b) sub) =
  let module L =
    struct
      type ('a, 'b) t = 'a → 'b
      let refl = fun x → x
    end in
    f {L} x

Figure 5: First-class subtypes, an initial approach

module type EQ = sig
  type ('a, 'b) t
  val refl : ('a, 'a) t
end

type ('a, 'b) eq = {E : EQ} → ('a, 'b) E.t

We can use this encoding to implement the standard operations on the equality GADT
by providing a suitable implementation of the EQ interface. For instance, we can implement
\texttt{cast : ('a, 'b) eq} \rightarrow \texttt{'a} \rightarrow \texttt{'b} by supplying an implementation of EQ using function types:

let cast (f : ('a, b) eq) x =
  let module L =
    struct
      type ('a, 'b) t = 'a → 'b
      let refl = fun x → x
    end in
    f {L} x

If we modify the signature EQ by adding co- and contra-variance markers to the parameters
of the type t, then we get the Church encoding of first-class subtypes, as shown in Figure 5,
from which we can implement the \texttt{refl}, \texttt{lift} and \texttt{(:>)} functions.

3.3 Converting between encodings

Despite the different starting points, the three implementations are interdefinable. In fact,
given any two implementations A and B of the subtyping interface, a subtyping witness of type
('a,'b) A.t can be converted to a witness of type ('a,'b) B.t.

The \texttt{SUB} module type contains the four elements of (Figure 2) (t, refl, lift, >:):
module type SUB =
  sig
    include DIAG
    val lift : {P:POS} → ('a, 'b) t → ('a P.t, 'b P.t) t
    val (>:) : 'a → ('a, 'b) sub → 'b
  end

The function conv takes two implementations of SUB, A and B, and converts a value in A.t to a value of B.t:

val conv : {A:SUB} → {B: SUB} → ('a, 'b) A.t → ('a, 'b) B.t

As often, implementing conv is a matter of finding a suitable implementation of POS to pass to lift:

let conv (type a b) {A:SUB} {B:SUB} (x : (a,b) A.t) =
  let module M =
    struct type 'a t = (a, 'a) B.t end
  in
    A.(>:) B.refl (A.lift {M} x)

3.4 An extension to GADT syntax

This encoding of first-class subtyping suggests an extension to OCaml’s GADT syntax, allowing us to write the ('a, 'b) sub type as:

type (~'a, +'b) sub = Refl : ('a, 'a) sub

Matching on a value of type ('a,'b) sub would then yield a subtyping constraint 'a ≤ 'b, rather than an equality constraint 'a = 'b.

However, this is only one of several possible interpretations of adding variance markers to GADT definitions. For further discussion of alternative interpretations, see Scherer and Rémy [15].

4 First-class subtypes: further examples

4.1 Arrays and rows

Here is a function that prints arrays by calling the name 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 → unit

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

List.iter print_array [a; b]

Using first-class subtypes it is possible to combine iterations over arrays whose element types belong to the same subtyping hierarchy.

type +'a arr = Arr : 'x array * ('x, 'a) sub → 'a arr
let aiter f (Arr (a,sub)) = Array.iter (fun s → f (s >: sub)) a
List.iter
  (aiter (fun o → print o#name)) [Arr (a,refl); Arr (b,refl)]
4.2 Decomposing mutable types

Several 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/contra-variant pair \([5]\), e.g. writing \(\text{type } (+'r,-'w) \text{ ref} \) rather than \(\text{type } 'a \text{ ref} \). 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.

4.3 Selective abstraction

A third 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:

\[
\text{module } M:\n\begin{align*}
\text{sig} & \text{ type } t \\
\text{val } & t\_\text{sub}\_\text{int} : (t,\text{int}) \text{ sub} \\
& (*\ldots*)
\end{align*}
\]

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 \([11]\) 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 \([15]\)), and zero cost-coercions \([4]\), where coercions in both directions are available, but actual type equality is not exposed.

4.4 Bounded quantification

Dual to abstraction, combining first-class subtypes with OCaml’s first-class polymorphism encodes bounded quantification. For example, the type \(\forall \alpha \leq t. \alpha \rightarrow t\) might be written as follows:

\[
\text{type } t = \{ f: 'a. \ ('a, t) \text{ sub} \rightarrow 'a \rightarrow t \}
\]

4.5 Proofs of variance

Finally, first-class subtypes can express proofs of variance. For example, the covariance of list can be represented by a value of the following type:

\[
('a, 'b) \text{ sub} \rightarrow ('a \text{ list}, 'b \text{ list}) \text{ sub}
\]

Abstracting over proofs of variance, we might build a \(\text{Compose}\) functor that generalizes \(\text{Compose}_{-}\) and \(\text{Compose}_{++}\) (Figure \([1]\)), reflecting OCaml’s subtle rules about the composition of variance \([12]\) as first-class objects.

4.6 Unsoundness in Java

Amin and Tate \([1]\) present an encoding of first-class subtypes in Java, by using Java’s bounded quantification to define a type \(\text{Constrain}<A, B>\) for \(A \leq B\), and then defining the equivalent of \((U, T) \text{ sub} \equiv \exists X \geq T. \text{Constrain}<U, X>\). They use this encoding to demonstrate a soundness bug, in which Java accepts null as a subtyping witness.
5 Discussion

The encodings given here are useful for exploratory work, for demonstrating soundness, and for showcasing OCaml’s expressivity. However, direct language support would make first-class subtypes more usable. Scherer and Rémy [15] discuss design issues and related work (e.g. [9] [16]).

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.

Our encodings share another issue with similar encodings of GADTs [20]: they lack inversion principles. Given (‘a, ‘b) sub, our encodings can be used to derive (‘a list, ‘b list) sub from the covariance of the list type constructor. However, going the other direction, from (‘a list, ‘b list) sub to (‘a, ‘b) sub, is equally valid but not expressible with our encodings.

With language support for subtype witnesses, coercions would still be explicit, but constraints in scope could be implicitly lifted through contexts, and inversion principles could be applied. 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 (Figure 2) might be defined as follows:

\[
\text{type } (\sim\text{a}, +\text{b}) \text{ sub } = \text{Sub : (\text{a}, \text{a}) sub}
\]

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 [8], might prove even more fruitful.

Acknowledgements We thank Francois Pottier, Leo White, and the ML 2017 reviewers for helpful comments.

References


