On variance, injectivity, and abstraction

Jacques Garrigue
Nagoya University
PR#5985: losing injectivity

module F (S : sig type 'a s end) = struct
    include S
    type _ t = T : 'a -> 'a s t
end
module M = F (struct type 'a s = int end)

let M.T x = M.T 3 in x
- : 'a = <poly> (* type is lost *)

After expanding s, the definition of M.t is actually:

    type _ t = T : 'a -> int t

But here 'a is not marked as existential.
Injectivity

In order to protect about this unsoundness, all variables appearing in type definitions must be bound

– either appear inside the type parameters
– or existentially bound (only in GADTs)

Inside type parameters, these variables must be injective:

– knowing the parameter must be sufficient to determine the type of the variables
Injectivity and variance

In OCaml, injectivity checking relies on variance inference.

The variance of a parameter is either

- **explicit** for abstract and private types, or constrained parameters
- **inferred** from its occurrences otherwise

\[
\begin{align*}
\text{bivariant} & = \emptyset \quad (\rightarrow \text{not injective}) \\
\{\text{pos}\} & = \text{covariant} \quad \text{contravariant} = \{\text{neg}\} \\
\text{invariant} & = \{\text{pos, neg}\}
\end{align*}
\]
Variance of constrained parameters

Since version 1.00, OCaml allows constrained type parameters:

```ocaml
type 'a t = T of 'b constraint 'a = 'b list
```

Rules for checking variance in that case become more complicated:

- constrained parameters’ variance must be explicit

- the variance of type variables inside constrained parameters must be weaker or equal than inside the body of the definition

```ocaml
type +'a t = T of 'b constraint 'a = 'b list (* 'b covariant *)
type 'a r = 'a -> int (* contravariant *)
type +'a t = T of 'b constraint 'a = 'b r (* Fails *)
   (* 'b contravariant in parameters but covariant inside *)
```
Variance subsumption

In OCaml, the variance of a parameter is allowed to be weakened through abstraction.

```ocaml
module M : sig type +'a u end = struct type 'a u = int end
```

This is correct for the type themselves, but the information becomes wrong when using it for type parameters.

```ocaml
module F (X : sig type 'a r end) = struct
  type +'a t = T of 'b constraint 'a = 'b X.r
end
module N = F (struct type 'a r = 'a -> int end)
```

By assuming \( r \) invariant, \( \forall b \) is inferred as invariant from the parameter of \( t \), which subsumes the covariance of the body. But in \( N \), \( \forall b \) becomes contravariant, which is wrong.
Fixing variance

If we want to approximate the variance of types inside parameters, we need to refine the definition.

- traditional variance subsumption defines a \textit{lower bound} on the variance of parameters

- we need to add \textit{upper bound} information, to be sure that parameters cannot have a stronger variance

If we represent the lower bound by the two flags \texttt{may-pos} and \texttt{may-neg}, we can introduce two flags \texttt{pos} and \texttt{neg} to guarantee the presence of occurrences.

By definition \texttt{pos} $\Rightarrow$ \texttt{may-pos} and \texttt{neg} $\Rightarrow$ \texttt{may-neg}.
Further refinements

While adding an upper bound to variance is sufficient for soundness, it doesn’t handle all cases of injectivity.

- We add a special flag \textit{inj} to denote guaranteed \textit{injectivity}.

\[
\text{pos} \lor \text{neg} \Rightarrow \text{inj}
\]

We can set \textit{inj} for all parameters of \textit{concrete} type definitions (by opposition to abbreviations), since they do not vanish.

- By symmetry we also add a flag \textit{inv} to denote \textit{strong invariance}. It is added automatically to parameters of \textit{concrete} definitions which are both \textit{pos} and \textit{neg}.

\[
\text{inv} \Rightarrow \text{pos} \land \text{neg}
\]
Composing variances

To determine the flags corresponding to an occurrence, one has to compose them. Upper and lower bound can be handled separately.

\[
\begin{array}{c|ccc}
\circ & \text{may_pos} & \text{may_neg} \\
\text{may_pos} & \text{may_pos} & \text{may_neg} \\
\text{may_neg} & \text{may_neg} & \text{may_pos} \\
\end{array}
\]

\[
\begin{array}{ccccc}
\circ & \text{inj} & \text{pos} & \text{neg} & \text{inv} \\
\text{inj} & \text{inj} & \text{inj} & \text{inj} & \text{inj} \\
\text{pos} & \text{inj} & \text{pos} & \text{neg} & \text{inv} \\
\text{neg} & \text{inj} & \text{neg} & \text{pos} & \text{inv} \\
\text{inv} & \text{inv} & \text{inv} & \text{inv} & \text{inv} \\
\end{array}
\]

- an occurrence in an inj context gives at most inj

- an inj occurrence in an inv context is sufficient to obtain inv
Composing variances

- \( \text{inj} \circ \text{inv} = \text{inj} \)
  
  Since an injective parameter may be changed through subtyping, it cannot guarantee invariance.

  ```plaintext
  type 'a t = T
  let f x = (x : 'a ref t -> bool t)
  ```

- \( \text{inv} \circ \text{inj} = \text{inv} \)

  Reciprocally, an injective parameter may only be changed through subtyping, so it becomes invariant in an invariant context.

  ```plaintext
  type 'a t = T
  let f x = (x : <m:int> t ref -> < > t ref) (* fails *)
  ```
OCaml 4.01 status

- Full variance inference is done, using 7 flags. The 7\textsuperscript{th} is a special case of \texttt{may\_neg}, needed for principality.

- However, variance annotations are only available for \texttt{may\_pos} and \texttt{may\_neg}. All abstract types excepted predefined ones (and local ones) are assumed non-injective. Some programs will not type anymore.

- For GADT indices, it is suggested to use concrete (injective) types rather than abstract ones.

```ocaml
type zero = Zero
  type 'a succ = Succ
```

Since a GADT index parameter is always invariant, injectivity is enough.
Future improvements ?

(With Jeremy Yallop and Leo White)

- Add **injectivity annotations** for abstract types.
  
  ```
  type #'a s
  (* also #+’a or #-'a *)
  type _ t = T : 'a -> 'a s t
  ```

- Add **new types** for isomorphic abbreviations (*cf. Haskell*).
  
  ```
  module M : sig
  type #’a t
  val f : int -> [‘pos] t
  end =
  struct
  type ’a t = new int
  let f x = (abs x : int :> ’a t)
  end
  ```

  - Similar to **private**, but subtyping works both ways
  - Useful in many situations (efficiency, runtime types, . . .)
  - May delay coercions to the signature
Other problems with abstraction

- One cannot prove the uniqueness of abstract types.
  
  ```ocaml
  type (_,_) eq = Eq : ('a,'a) eq
  module M : sig type t val eq : (t,int) eq end
  = struct type t = int let eq = Eq end
  ```

- One doesn’t know whether an abstract type is contractive.
  
  ```ocaml
  (* Using -rectypes *)
  module Fixpoint (M : sig type 'a t end) =
  struct type fix = fix M.t end
  Error: The type abbreviation fix is cyclic
  ```

- One cannot know whether an abstract type may be float.
  
  ```ocaml
  module M : sig type t type r = {x:t; y:t} end =
  struct type t = float type r = {x:t; y:t} end
  Error: Signature mismatch: ...
  ```
Conclusion

- PR#5985 is now fixed, thanks to improved variance inference

- Introduces some new restrictions on type definitions

- Could be alleviated by further extensions: injectivity annotations and new types

- Abstraction loses too much information?