On variance, injectivity, and abstraction

Jacques Garrigue

1 Introduction

OCaml's type system is rich. Its algebraic datatypes support many features that are not included in most other systems, such as variance (inferred or explicit), constrained type parameters (useful for polymorphic classes), and GADTs including existential type variables.

Taken together, this is already an interesting mix, but combining them with the abstraction mechanism coming from the module system creates even more interactions, as known information about a type changes across abstraction barriers.

In april of this year, an unsoundness was discovered in the type system when combining existential GADTs and abstraction (cf. bug report by Jeremy Yallop at `http://caml.inria.fr/mantis/view.php?id=5985`.) This problem was not unique to GADTs, and actually could be reproduced with features present in OCaml since its inception. We explain the problem, how it was solved through a refinement of the variance information, how this impacts programming, and how the language could be extended for more flexibility.

2 Variance

Variance was introduced in OCaml in version 3.01. Before that, subtyping only applied to structural types. With variance inference and variance annotations on abstract types, it became possible to use subtyping on parameters of datatypes and abstract types. Later, with the introduction of the value restriction, this variance information was also used for improving polymorphism.

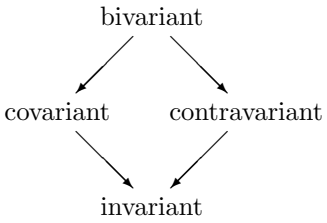
Superficially, inferring variance is a straightforward task. One just visits type definitions, tracking occurrences of type variables in positive and negative positions. Positive and negative positions are switch whenever one goes through a contravariant type constructor, such as the left hand-side of a functional arrow. If a type parameter does not appear in a type it is irrelevant, and the type is bivariant (subtyping can be used in both directions). If it only has positive occurrences it is covariant (*e.g.* the parameter of `list`), only negative occurrences makes it contravariant (*e.g.* the domain of a function type), both positive and negative makes it invariant (*e.g.* arrays or reference cells).

You can already see here that all is not that simple: when computing variance in a type definition, we also use the variance computed for other type definitions. In case of recursive type definitions, one needs to compute a fixed point.

Abstraction introduces a first difficulty: the variance of parameters in an abstract type is not necessarily the same as their variance in its concrete definition:

```
module M : sig type 'a t  type +'a u end =
  struct
    type 'a t = T of 'a
    type 'a u = int
  end
```

Here `M.t` is invariant, but its definition was covariant; `M.u` is covariant, but `'a` was bivariant in its definition. However, intuitively it is fine do decrease variance, *i.e.* go down in the following lattice:



Things become a bit more complicated with constrained type parameters.

```
type +'r t = T of 'key * 'data list
  constraint 'r = < key: 'key; data: 'data>
```

Here `'r` does not appear in the type itself, but is used to write the type parameters in a consolidated and informative way. Since `'r` does not appear inside the type, we cannot infer its variant, and it must be given explicitly (or be assumed invariant). How do we verify that this variance is correct? The idea is to infer the variance of type variables twice, the first time we compute their occurrences in type parameters, and the second time inside the body of the definition. In the above example, we start from `'r` which is assumed covariant, and from it we deduce that `'key` and `'data` are both covariant. Then we look at the body of the definition and see that both `'key` and `'data` have only positive occurrences, so it is fine to see them as covariant.

While this definition is fine by itself, one can quickly see that it is not compatible with the variance weakening we defined for abstraction.

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

```
end
module M = F (struct type 'a r = 'a -> int end)
```

In the body of `F`, `r` is invariant, so even if we assume `t` to be covariant in `'a`, `'b` is inferred invariant from parameters, which is sufficient since `'b` has only positive occurrences in the body of the definition. However, in `M`, `r` becomes contravariant, so that one can actually use contravariant subtyping on `'b`, breaking our assumption on constrained parameters. One can then define the following (invalid) function, where we call a non-existing method.

```
let f (x : < >) =
  let M.T y = (M.T x :> (<m : int> -> int) M.t)
  in y#m
```

What went wrong here? Our assumption on constrained parameters did not take into account that the variance information we have for constructors may change through instantiation.

3 Injectivity

Actually, the same problem also occurs without using subtyping.

```
module F (X : sig type 'a r end) = struct
  type 'a t = T of 'b
    constraint 'a = 'b X.r
end
module M = F (struct type 'a r = unit end)
let f x = let M.T y = M.T x in y
val f : 'a -> 'b
```

This is exactly the same cause: `r` is assumed to be invariant, but it turns out to be bivariant. And since it is just defined by an alias, it is not necessary to use subtyping anymore.

In a type definition, we need all type variables that occur in the body to satisfy *injectivity*: they should be defined in a unique way from the type parameters.

While this is just special case of the variance problem we described above (and was actually delegated to variance inference), this has a particular impact on GADTs. Here is the GADT version of the above problem:

```
module F (X : sig type 'a r end) = struct
  type _ t = T : 'a -> 'a X.r t
end
module R = struct type 'a r = unit end
module M = F (R)
let f x = let M.T y = M.T x in y
val f : 'a -> 'b
```

A simple solution to the above variant and injectivity problems is to assume that all abstract types, when used in parameter constraints or GADT return types, are potentially bivariant, and non injective. This basically means that, as long as we have no way to tell that an abstract type is injective, we will not be able to use it in GADT return types (at least, when it occurs in the body). This can be problematic when one uses a GADT to represent type witnesses:

```
type 'a u
type _ typ =
  | Tint  : int typ
  | Tlist : 'a typ -> 'a list typ
  | Tu    : 'a typ -> 'a u typ
```

The first two cases, for int and list, create no problem, but in the last case u is an abstract type, and the definition cannot be accepted.

There are two other place where injectivity matters for GADTs: one is type refinement, where locally abstract types can be refined during pattern matching; the other is detection of impossible cases in pattern matching, where one needs to know that a type is injective in order to eliminate the case when type parameters differ. Note however that this two problems do not even occur as long as we cannot use abstract types in GADT return types.

4 Solution

As mentioned just above, the solution to recover soundness looks simple enough: one must assume that nothing is known about abstract types, when they are used in parameter constraints or GADT return types.

However, life is always more complicated than that: we also have private types, for which we can declare a variance different from that of the body. Moreover, from the point of view of injectivity, it turns out to be useful to distinguish datatype definitions, in which a type parameter, even when it is bivariant, can only be modified through subtyping, and type alias, for which some type parameters may simply disappear through expansion.

As a result, OCaml 4.01 uses no less than 6 flags to describe the variance of type parameters (it actually uses 7, but the last one is not relevant here):

- `may_pos` and `may_neg` are upper bounds of the presence of positive or negative occurrences in the real definition; for abstract or private types, `+` indicates {`may_pos`}, `-` indicates {`may_neg`}, and no variance annotation means {`may_pos`, `may_neg`}.
- `pos` and `neg` are about known occurrences, they may only be true if the actual definition is visible, *i.e.* the type is not abstract.
- `inv` denotes a strong version of invariance: for datatypes, a parameter is invariant if it has both positive and negative occurrences; for aliases, it must explicitly contain invariant occurrences.
- `inj` tells that a parameter is injective, *i.e.* it cannot disappear through expansion. All datatype parameters are injective, *i.e.* the only way to change them

is through subtyping; but this is not true for type aliases and abstract types.

These flags are related by the following implications:

$$\begin{array}{lcl} \texttt{inv} & \Rightarrow & \texttt{pos} \wedge \texttt{neg} \\ \texttt{pos} & \Rightarrow & \texttt{inj} \wedge \texttt{may\_pos} \\ \texttt{neg} & \Rightarrow & \texttt{inj} \wedge \texttt{may\_neg} \end{array}$$

Variances are sets of flags, closed by these implications. The variance of an occurence is computed by composing the variance of the context, which starts as the closure of `inv`, `pos` or `neg` for parameter constraints, and `pos` for the definition. They are composed pointwise: $S_1 \circ S_2 = \{f_1 \circ f_2 \mid f_1 \in S_1, f_2 \in S_2\}$, using the following rules:

$$\begin{array}{ll} \texttt{inv} \circ \texttt{inj} = \texttt{inv} & \texttt{neg} \circ \texttt{pos} = \texttt{neg} \\ \texttt{inj} \circ \texttt{inj} = \texttt{inj} & \texttt{neg} \circ \texttt{neg} = \texttt{pos} \\ \texttt{pos} \circ \texttt{inv} = \texttt{inv} & \texttt{may\_pos} \circ \texttt{may\_pos} = \texttt{may\_pos} \\ \texttt{neg} \circ \texttt{inv} = \texttt{inv} & \texttt{may\_pos} \circ \texttt{may\_neg} = \texttt{may\_neg} \\ \texttt{pos} \circ \texttt{pos} = \texttt{pos} & \texttt{may\_neg} \circ \texttt{may\_pos} = \texttt{may\_neg} \\ \texttt{pos} \circ \texttt{neg} = \texttt{neg} & \texttt{may\_neg} \circ \texttt{may\_neg} = \texttt{may\_pos} \end{array}$$

Note that this relation is not strictly symmetric: while we have $\texttt{inv} \circ \texttt{inj} = \texttt{inv}$, we do not have $\texttt{inj} \circ \texttt{inv} = \texttt{inv}$.

The rationale is that, inside an invariant context, there is no way to change anything by subtyping. So all injective occurrences become invariant. This is particularly useful for GADTs, as index positions are always invariant, so that type constructors used in indices need only be injective. The opposite order is not true, as a context may be injective and bivariant, in which case we can use subtyping to remove an invariant occurrence:

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

While OCaml 4.01 implements this complex lattice to represent variance, it adds no new syntax to describe points in this lattice. That is, this lattice is exclusively used to represent inferred variance information. In particular the lack of new variance annotations for type parameters means that abstract types are never injective.

To avoid failing on existing code, particularly code using objects where constrained parameters are used for sharing, OCaml 4.01 implements a slightly more clever criterion for relating the variance of constrained parameters to the variance inside the definition. Namely, it detects identical types appearing in the parameter and in the definition, and in that case compares the variance of the root.

```
type 'a u
type 'a t = T of 'a * 'b u
  constraint 'a = <m : 'b u; ..>
```

This avoids failing due to the discrepancy between `may_pos` and `pos` for `u`.

5 Proposals

As we just stated, OCaml 4.01 adds no way to refine the variance of abstract types. To overcome this, in the presentation I would like to propose several extensions which may help to deal with these problems.

Injectivity annotations A first natural step is to allow declaring that an abstract type is injective. This would solve most of the limitations due to the loss of known variance information, in particular in connection with GADTs.

Some would argue that this should rather be a compiler flag, injectivity becoming progressively the default.

Newtypes A problem with injectivity annotations is that many types are actually not injective to start with. Namely, phantom types usually do not use their index in their implementation. This is fine if they are defined as datatypes, but in ML it is not unusual to use a type alias.

Converting to datatypes solves the problem, but this may require extensive changes in programs, and inefficiencies in the generated code.

Newtypes, as they exist already in Haskell, are just datatypes with a single constructor. In OCaml, we could go a bit further, to omit the constructor, and instead use subtyping. Leo White suggested that the subtyping could be done at the module interface level, which would greatly reduce the need for coercions.

Unique types Once the problem of injectivity solved, there remains another problem for GADTs: the exhaustiveness check cannot prove the incompatibility of two types when they differ only at the level of abstract types. This is because there is no way to known whether two abstract types could not be aliases for the same type. Unique type annotations could help solving that problem.