Tracking injectivity and nominality beyond abstraction

Jacques Garrigue

Abstract

Seven years ago, a subtle unsoundness was discovered by Yallop in OCaml's type system [5]. It involved a combination of GADTs and abstraction, and uncovered how wrong assumptions about the injectivity of abstract types could lead to unsound type definitions. While it has been fixed by moving OCaml's variance analysis from a naive view to a more complex one, which involves six flags, including one for injectivity, the new flags have not been given surface syntax [1]. In this presentation, I would like to showcase two potential solutions to the problem of injectivity (two equal types must also have equal parameters), and the related problem of nominality (types whose implementation may only be a concrete type). The first one is very simple: it adds surface syntax for the injectivity flag, allowing one to stipulate that an abstract type is injective in one of its parameters. The second one is more involved, and attempts to track the nominality of type definitions. Currently, the only nominal type definitions in OCaml are concrete types (records and variants), built-in types, and types defined in the current module (excluding public type abbreviations). By tracking the nominality of types, it becomes possible to strengthen GADT unification, and infer more type equalities and incompatibilities.

1 Injectivity and GADTs

If you are an adept of OCaml GADTs, there is a reasonable probability that you have encountered the following problem.

```
module Vec : sig
  type +'a t
  val make : int -> (int -> 'a) -> 'a t
  val get : 'a t -> int -> 'a
end = struct
  type 'a t = int -> 'a
  let make n f = Array.get (Array.init n f)
  let get f = f
end

type _ ty =
  | Int : int ty
  | Vec : 'a ty -> 'a Vec.t ty
Error: In this definition, a type variable
    cannot be deduced from the type parameters.
```

In [1] we explained why the functorized version of this code would be unsound. But of course, the problem is not limited to functors, and a slight variation of this code exhibits this unsoundness.

```
type (_,_) eq = Refl : ('a,'a) eq

module Vec : sig type +'a t
                 val eq : ('a t, 'b t) eq end
= struct type 'a t = unit  let eq = Refl end

type _ ty = ... (* same as above *)

let coe : type a b. (a,b) eq -> a ty -> b ty =
  fun Refl x -> x

let eq_int_any : type a. (int, a) eq =
  let t : a Vec.t ty = coe Vec.eq (Vec Int) in
  let Vec Int = vec_ty in Refl
```

If the above code were typable, whereas `t` is not injective, we would have a proof that `int` is equal to any type. And since it is in general very difficult to check no equality such as `Vec.eq` is available through a side-channel, we need a way to protect ourselves against potentially non-injective definitions.

Oleg Kiselyov [2] recently described a classical workaround to allow the definition of `ty`, even without injectivity, through an existential encoding.

```
type _ ty =
  | Int : int ty
  | Vec : ('b, 'a Vec.t) eq * 'a ty -> 'b ty
```

The idea here is that `'a` is no longer a normal variable instantiated through unification, but an existential variable, which can only be deduced from `'b` through external means. This actually corresponds to the behavior of GADTs in GHC [4], where GADT cases are defined through type equations. However, this approach has several drawbacks. The first one is that it does not apply to constrained types [1], since they cannot contain existential variables. We will explain the other drawbacks at the end of this abstract.

2 Injectivity annotations

By observing a reasonable code sample, one can easily convince oneself that a large majority of non abstract type definitions are injective. This is the case of all concrete definitions (records or variants), and also of all type abbreviations for which the parameters occur at an injective position in the body. The only exception is phantom parameters in type abbreviations. So the real problem is

not so much a lack of injectivity, but the lack of a way to track it in abstract types.

This first extension[1] does just that.

```
module Vec : sig type +!'a t ... end = ...
```

The `!` annotation on type parameters indicates that they are injective. It is checked during module signature subtyping, and variance inference propagates it to other definitions, so that `ty` will be accepted.

Injectivity annotations are only required on abstract types and private row types. Injectivity is automatically inferred for type abbreviations, so that annotations will only enforce a check on the result of inference. And, as written above, concrete types are always injective.

While the above description may look trivial, the handling of constrained type parameters had to be improved to make inference work properly. The basic idea is similar to variance inference, a constrained type parameter should be injective if all type variables with an injective occurrence in the parameter have also an injective occurrence in the body of the type.

```
type !'a t = 'b constraint 'a = < b : 'b >
type !'a u = 'b constraint
             'a = < b : 'b; c : 'c >
Error: In this definition, expected parameter
       variances are not satisfied.
```

Here `t` is injective, since the only type variable in the constraint is `'b`, which appears in the body. But `u` is not injective, since `'c` does not appear in the body.

```
type _ v = unit
type !'a w = int constraint 'a = 'b v
```

A parameter containing only non-injective occurrences is injective. After expansion of `v`, `'a` contains no free variables, so assuming its injectivity generates no contradiction.

The problem is even more subtle when there are non-injective abstract types involved.

```
module M : sig type 'a t end =
  struct type 'a t = 'a list end
type !'a u = int constraint 'a = 'b M.t
Error: In this definition, expected parameter
       variances are not satisfied.
```

In the same way as we do for variance inference, inside constrained parameters we should think in terms of potential injectivity rather than verified injectivity. Since we do not track it explicitly, we consider all occurences to be potentially injective, except vanishing ones, that disappear when expanding type abbreviations. It is possible to work around the discrepancy between the handling of the parameters and body by using extra constraints.

```
type !'a u = 'b constraint 'a = <b : 'b>
                 constraint 'b = 'c M.t
```

[1] PR#9500: Add injectivity annotations, `https://github.com/ocaml/ocaml/pull/9500`

3 Nominal types

While tracking injectivity solves some limitations of the combination of GADTs and abstract types, it does not solve all of them. Namely, it says nothing about what to do with types whose head constructors are distinct, but for which we do not have a proof that they do not actually hide the same definition (so that there might exist a proof that they are actually compatible).

Again, due to the OCaml module system, this may occur in a variety of situations. A common one is when trying to unify an abstract type with the index of a GADT constructor.

```
module M : sig type t type !'a u end =
  struct type t = int type 'a u = 'a list end
let f : (int, M.t) eq -> M.t -> int =
  fun Refl x -> x
let g : (bool, M.t) eq -> 'a =
  function _ -> .
Warning 8: pattern-matching is not exhaustive.
This case is not matched: Some Refl
let h1 : (int list, int M.u) eq ->
         int M.u -> int list =
  fun Refl x -> x
Error: This expression has type int M.u
but an expression was expected of type int list
let h2 : (int list, M.t M.u) eq -> M.t -> int =
  fun Refl x -> x
Error: This expression has type M.t
    but an expression was expected of type int
```

In the case of non-parametric abstract types, the type checker is able to add local equations to the environment, so that `f` is typable. However, there is no way to track the fact `M.t` is incompatible with `bool`, so that `g`'s pattern-matching is deemed non-exhaustive. If we move to parametric abstract types, the situation is even worse. OCaml is unable to properly keep the information that `int list` and `int M.u` are equal, so that `h1` is untypable. In particular it is unable to infer that `M.u` itself should be equal to `list`[2], which combined with injectivity would allow to type `h2`.

Concerning the first problem of comparability or, more specifically, *separability*, OCaml is only able to distinguish builtin types, types defined in the current module (i.e., types that cannot hide any sharing of representation), and incompatible type definitions (according to the contents of the definition). In particular this means that even concrete type definitions may be compatible even if they were defined independently.

```
module N = struct type t = A type u = A end
let f : (N.t, N.u) eq option -> unit =
  function None -> ()
```

[2] Equating `u` and `list` would be wrong if for instance the actual definition was `type 'a u = int list`

```
Warning 8: pattern-matching is not exhaustive.
This case is not matched: Some Refl
```

The second problem is that the environment can only be extended by equations defining types, i.e. not equations on applied type constructors. Fixing this limitation would require major changes in the type inference algorithm, extending with a notion of equations derived from a set of equations. We will not consider this approach here.

The third problem is related to the ability to infer exact type definitions for parametric types. While this is impossible in general, as this corresponds to higher-order unification, which is known to be undecidable, the situation is different if we know that `M.u` actually abstracts a concrete type definition. This is because concrete type definitions are always *generative*, where generativity means that the equality of two generative types implies the equality of their head constructors. Since they can only be exported with exactly the same type parameters, it is sound to add the equation `type 'a M.u = 'a list` to the typing environment, which would allow to typecheck both `h1` and `h2`

We solve this by introducing a new concept of *nominal types*[3], through annotations on type declarations. They come in two forms:

```
type 'a u [@@nominal]
type 'a t [@@nominal "M.t"]
```

The first form says that `u` is nominal. `u` is allowed to be nominal if it is a new abstract definition (i.e., an abstract definition inside an implementation, which cannot hide another OCaml definition), or if it abstracts a nominal or concrete type definition. The second form adds an explicit name, here `"M.t"`, which allows to separate it from other nominal types (all built-in types being such named nominals). This name must be present since the original definition of the type, and can be forgotten in abstract exports, but not in concrete ones[4]. This allows us to remove the current special handling of built-in and local abstract types.

This notion of nominal type extends that of matchable type in Haskell [3]. A type is matchable if it is injective and generative, meaning that an unnamed nominal type is matchable. Due to abstraction, generativity alone only allows to deduce equalities, not incompatibilities. The addition of an explicit name allows to track separability, as two nominal types with different names (or a named type and an an unnamed concrete type) are now separable, which allows to refine the exhaustivity check of pattern-matching.

All our previous examples are now typable, without warning (i.e., the exhaustivity could be verified).

[3]OCaml RFC #4: Nominal (abstract) types, `https://github.com/ocaml/RFCs/pull/4`; PR#9042, `https://github.com/ocaml/ocaml/pull/9042`.

[4]This is needed to allow one to separate between unnamed concrete types and named abstract types.

```
module M : sig
  type t [@@nominal "int"]
  type 'a u [@@nominal]
end = struct
  type t = int
  type 'a u = 'a list
end
let g : (bool, M.t) eq option -> unit =
  function None -> ()
let h1 : (int list, int M.u) eq ->
         int M.u -> int list =
  fun Refl x -> x
let h2 : (int list, M.t M.u) eq -> M.t -> int =
  fun Refl x -> x
```

Type-level terms It is known that examples involving type-level terms can be handled by giving them a concrete representation. However, by using incompatibility, together with the implicit injectivity of nominal types, we need not rely on this representation.

```
module Nat = struct
  type zero [@@nominal "zero"]
  type 'n succ [@@nominal "succ"]
end
type ('a,'n) vec =
  | Nil : ('a, Nat.zero) vec
  | Cons : 'a * ('a,'n) vec ->
           ('a,'n Nat.succ) vec
let head : ('a,'n Nat.succ) vec -> 'a =
  function Cons (h,t) -> h
```

Contractiveness Nominality also implies contractiveness, which allows to define some fixpoints that couldn't be defined otherwise, as it is unsound to take the fixpoint of a non-contractive type[5]. A type $(\alpha_1, \ldots, \alpha_n)\ t$ is contractive in α_i if it does not act as a projection, i.e. $(\alpha_1, \ldots, \alpha_n)\ t \neq \alpha_i$. Here is an example where a nominal type is used to build a fixpoint.

```
#rectypes;; (* Use equi-recursive types *)
module Fixpoint
         (M : sig type 'a t [@@nominal] end) =
  struct type fix = fix M.t end
module Nat =
  Fixpoint(struct type 'a t = 'a option end)
module Nat : sig type fix = fix option end
```

The resulting type `Nat.fix` is isomorphic to the natural numbers.

Type witnesses We have already seen that an injectivity annotation on `Vec.t` lets us define the type `ty` of section 1. Using a nominality annotation is a bit heavier,

[5]See OCaml issue #5863, `https://github.com/ocaml/ocaml/issues/5863`.

as we need to attach the name to a concrete definition, but it also allows to define extractors.

```
module Vec : sig
  type +'a t [@@nominal "vec"]
  val make : int -> (int -> 'a) -> 'a t
  val get : 'a t -> int -> 'a
end = struct
  type 'a t =
    Vec of (int -> 'a) [@@nominal "vec"]
  let make n f =
    Vec (Array.get (Array.init n f))
  let get (Vec f) = f
end
type _ ty =
  | Int : int ty
  | Vec : 'a ty -> 'a Vec.t ty
let vec_arg = function Vec t -> t
val vec_arg : 'a Vec.t ty -> 'a ty
```

Here, the addition of a name allows to separate `Vec.t` from `int`, making the pattern-matching in `vec_arg` exhaustive, which would not be the case with injectivity alone.

Comparison with the existential encoding It turns out that, if we were to use the existential encoding of GADTs at the end of section 1, we wouldn't even be able to define `vec_arg` without a type annotation:

```
let vec_arg = function Vec (Refl, t) -> t
Error: ...
The type $Vec_'a would escape its scope
```

This is because the existential encoding does not allow to infer the type of GADT patterns, only to check it. Yet, this inference is used even inside the OCaml compiler to avoid explicit type annotations.

Not only do we need an annotation, but we also need an injectivity proof, which must be written inside the `Vec` module.

```
vec_inj : ('a Vec.t,'b Vec.t) eq -> ('a,'b) eq
```

Finally, with all those it only lets us define a non-exhaustive version of `vec_arg`:

```
let vec_arg : type a. a Vec.t ty -> a ty =
  function Vec (w, t) ->
    let Refl = vec_inj w in t
Warning 8: pattern-matching is not exhaustive.
This case is not matched: Int
val vec_arg : 'a Vec.t ty -> 'a ty = <fun>
```

References

[1] Jacques Garrigue. On variance, injectivity, and abstraction. In *OCaml Meeting*, Boston, September 2013. http://www.math.nagoya-u.ac.jp/~garrigue/papers/.

[2] Oleg Kiselyov. That dreaded 'a type variable cannot be deduced' GADT error. Mail to the caml-list, April 2020. https://sympa.inria.fr/sympa/arc/caml-list/2020-04/msg00010.html.

[3] Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. Higher-order type-level programming in Haskell. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[4] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X) Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5):333–412, September 2011.

[5] Jeremy Yallop. Unexpected interaction between variance and GADTs. OCaml bug report, April 2013. https://github.com/ocaml/ocaml/issues/5985.