GADT meet subtyping

Didier Rémy
Inria
Didier.Remy@inria.fr

Gabriel Scherer
Inria
Gabriel.Scherer@inria.fr

Abstract

While generalized abstract datatypes are now considered well-understood, adding them to a language with a notion of subtyping reveals a few surprises. What does it mean for a GADT parameter to be covariant? The answer turns out to be quite subtle, and involves new semantic properties of types that raise interesting design questions. We prove a soundness theorem for GADT with variance annotations, and study its applicability in a real-world language.

Consider the following example of GADT definition:

```
type α expr =
| Val : α → α expr
| Int : int → int expr
| Prod : ∀βγ. (β expr * γ expr) → β * γ expr
```

Is it sound to claim that α `expr` is covariant? The variance checking algorithm currently implemented in OCaml would reject it, because it uses a simple conservative criterion: parameters that are instantiated with something other than a type variable (α is instantiated with `int` in the `Int` case and $\beta * \gamma$ in the `Prod` case) must be invariant.

In OCaml, covariance is used in particular with the relaxed value restriction [Gar04], which allows generalization of type variables appearing in covariant-only positions. This relaxation is crucial to work with polymorphic data structures, and is maybe the most important use of variance information. The current safe criterion is too restrictive to use GADT in this case.

Let's first show why it is reasonable to say that α `expr` is covariant. We will explain why, informally, if we are able to coerce an α into a α' (we write (`v` :> α') to explicitly cast a value `v` of type α), then we are also able to transform an α `expr` into a α' `expr`. Here is a pseudo-code for the coercion function:

```
let coerce : α expr ≤ α' expr = function
  | Val (v : α) -> Val (v :> α')
  | Int n -> Int n
  | Prod β γ ((b, c) : β expr * γ expr) ->
    (* if β * γ ≤ α', then α' is of the form
       β' * γ' with β ≤ β' and γ ≤ γ' *)
    Prod β' γ' ((b :> β' expr), (c :> γ' expr))
```

In the `Prod` case, we make an informal use of something we know about the OCaml type system: the supertypes of a tuple are all tuples. By entering the branch, we have gained the knowledge that $\alpha = \beta * \gamma$, so from $\alpha \leq \alpha'$ we know that $\beta * \gamma \leq \alpha'$; we can deduce that α' is itself a pair of the form $\beta' * \gamma'$, and by covariance of the product we know that $\beta \leq \beta'$ and $\gamma \leq \gamma'$, allowing to conclude by casting, recursively, at types β' `expr` and γ' `expr`.

Similarly, in the `Int` case, we know that $\alpha =$ `int` and returned an `int expr`; this is because we know that, in OCaml, no type is above `int`: if `int` $\leq \alpha'$ then α' equals `int`.

What we use in both cases is reasoning of the form: "if $T[\overline{\beta}] \leq \alpha'$, then I know that α' is of the form $T[\overline{\beta}']$ for some types $\overline{\beta}'$". We call this an *upward closure* property: when we "go up" from a $T[\overline{\beta}]$, we only find types that also have the structure of T. Similarly, for contravariant parameters, we would need a *downward closure* property: T is downward-closed if $T[\overline{\beta}] \geq \alpha'$ entails that α' is of the form $T[\overline{\beta}']$.

Not all types are upward closed: the object type `< m : int >`, which has one method `m` returning an integer, is smaller than the empty object type `< >`, which is not of the form `< m : int >`. For this reason, it would be unsound to use it, as we did for `int` and $\beta * \gamma$, in a covariant GADT[1]:

```
type +α wrong =
| K : < m : int > -> < m : int > wrong
```

To see why the type-checker must reject this definition, let's define the classic equality GADT and the corresponding casting function:

```
type (α, β) eq = Refl : ∀γ.(γ, γ) eq
let cast_eq : (α, β) eq → α → β = function
  | Refl -> (fun x -> x)
```

The type above leads to a way to cast the empty object of type `< >` into a `< n : int >`, which is clearly unsound.

```
let get_eq : α wrong → (α,<m:int>) eq = function
  | K _ -> Refl
let evil_cast : < > -> < m : int > =
  let obj = K (object method m = 0 end) in
  cast_eq (get_eq (obj :> < > wrong))
```

In the work we would like to present, we have proved that the notions of upward and downward-closure are the key to a sound variance check for GADT. We started from the formal development of Simonet and Pottier [SP07], which provides a general soundness proof for a language with subtyping and a very general notion of GADT expressing arbitrary constraints – rather than only type equalities. By specializing their correctness criterion, we were able to split it into three smaller criteria, that are simple to implement in a type-checker. One of them, the most delicate and important, is that instances of covariant (respectively contravariant) parameters should be upward-closed (resp. downward-closed).

The problem of non-monotonicity

We have a problem with those closure properties: while they hold naturally in a core ML type system with strong inversion theorems, they are non-monotonic properties:

[1] This counterexample is due to Jacques Garrigue.

they are not necessarily preserved by extensions of the subtyping lattice. For example, OCaml has a concept of *private types*: a type specified by `type t = private` τ is a new semi-abstract type smaller than τ (`t` $\leq \tau$ but `t` $\not\geq \tau$). As private types can be defined from any type, no type is downward-closed: for any type τ I may define a new, strictly smaller type.

This means that closure properties of the OCaml type system are relatively weak: no type is downward-closed (so instantiated GADT parameters cannot be contravariant), and arrow types are not upward-closed as their domain should be downward-closed. Only purely positive algebraic datatypes are upward-closed. The subset of GADT declarations that can be declared covariant is small, yet, we think, large enough to capture a lot of useful examples, such as α `expr` above.

Giving back the freedom of subtyping

It is disturbing that our type system would rely on non-monotonic properties: if we adopt the correctness criterion above, we must be careful in the future not to enrich the subtyping relations too much. This is contradictory to the general design aspects of subtyping, where decidability compromises may be made, but having *more* subtyping relations is always considered a good thing.

Consider for example `private` types: one could reasonably imagine a symmetric concept of a type that would be strictly *above* a given type τ; we will name those types `invisible` types (they can be constructed, but not observed). Invisible types and GADT covariance seem to be incompatible: the designer has to pick one, and cannot add the other.

A solution to this tension is to allow the user to *locally* guarantee negative properties about subtyping (what is *not* a subtype), at the cost of abandoning the corresponding flexibility. Just as object-oriented languages have `final` classes that cannot be extended, we would like to be able to define some types as `public` (respectively `visible`), that cannot later be made `private` (resp. `invisible`). Such declarations would be rejected if the defining type already has subtypes (eg. an object type), and would forbid further declarations of types below (resp. above) the declared, effectively guaranteeing downward (resp. upward) closure. Finally, upward or downward closure is a semantic aspect of a type that we must have the freedom to publish through an interface: abstract types could optionally be declared `public` or `visible`.

Another approach: subtyping constraints

The reason why getting fine variance properties out of GADT is difficult is because they correspond to type equalities which, to a first approximation, use their two operands both positively and negatively. One way to get an easy variance check is to encourage users to *change* their definitions into different ones that are trivial to check. Consider for example the following redefinition of α `expr`:

```
type α expr =
| Val : ∀α.α → α expr
| Int : ∀α[α≥int].int → α expr
| Prod : ∀αβγ[α≥β * γ]. (β expr * γ expr) → α expr
```

It is very simple to check that this definition is covariant, because all type equalities $\alpha = T_i[\overline{\beta}]$ have been replaced by inequalities $\alpha \geq T_i[\overline{\beta}]$ that are obviously preserved when replacing α by some α' such that $\alpha' \leq \alpha$. This variant of GADT, using subtyping rather than equality constraints, has been studied by Emir *et al.* [EKRY] in the context of the `C♯` programming language.

But isn't such a type definition less useful than the previous one, which had a stronger constraint? It actually appears that we have not lost much. In the examples we have studied, when a user considers a given parameter as "naturally covariant" (or contravariant), the uses he has in mind can be adapted to this weaker definition. Here is for example the classic `eval` : α `expr` $\to \alpha$ function on this weaker definition, using (v `:>` τ) to cast a value $v : \sigma$ when $\sigma \leq \tau$.

```
let rec eval : ∀α. α expr → α = function
  | Val α (v : α) -> v
  | Int α n -> (n :> α)
  | Prod α β γ (b, c) -> ((eval b, eval c) :> α)
```

As is manifest in this example, this approach could require more explicit annotations, at least with existing type system implementations relying on unification rather than implicit subtyping.

Work in progress: Completeness of variance annotations

For simple algebraic datatypes, variance annotations are "enough" to say anything we want to say about the variance of datatypes. Essentially all admissible variance relations between datatypes can be described by considering the pairwise variance of parameters separately.

This does not work anymore with GADT. For example, the type (α, β) `eq` cannot be accurately described by considering variation of each of its parameters independently. We would like to say that (α, β) `eq` $\leq (\alpha', \beta')$ `eq` holds as soon as $\alpha = \beta$ and $\alpha' = \beta'$. With the simple notion of variance we currently have, all we can soundly say about `eq` is that it must be invariant in both its parameters – which is considerably weaker. In particular, the well-known trick of "factoring out" GADT by using the `eq` type in place of equality constraint does not hold anymore: equality constraints allow fine-grained variance considerations based on upward or downward-closure, while the equality type instantly makes its parameters invariant.

We think it is possible to regain some "completeness", and in particular re-enable factoring by `eq`, by considering more information to decide subtyping between instances, in addition to individual parameter variances. We are considering using *domain information*, to know which instances of the type are inhabited: for example, `bool expr`, or (`int`, `float`) `eq`, are not inhabited.

References

[EKRY] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for C# generics. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06.

[Gar04] Jacques Garrigue. Relaxing the value restriction. In *In International Symposium on Functional and Logic Programming, Nara, LNCS 2998*, 2004.

[SP07] Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1), January 2007.