GADT meet subtyping
Gabriel Scherer, Didier Rémy

To cite this version:
Gabriel Scherer, Didier Rémy. GADT meet subtyping. ACM SIGPLAN Workshop on ML, Aug 2012, Copenhague, Denmark. hal-01093940

HAL Id: hal-01093940
https://inria.hal.science/hal-01093940
Submitted on 18 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
GADT meet subtyping

Gabriel Scherer
Inria
Gabriel.Scherer@inria.fr

Didier Rémy
Inria
Didier.Remy@inria.fr

Abstract

While generalized abstract datatypes are now considered well-understood, adding them to a language with a notion of subtyping comes with 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 allow variance annotations in GADT definitions, present a sound and effective algorithm to check such declarations, and describe its application in a real-world language.

Consider the following example of GADT definition:

```plaintext
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 β * γ 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, perhaps, the most important use of variance information. The current safe criterion is too restrictive to allow the use of a 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 to a value 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:

```plaintext
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 α = β * γ, so from α ≤ α′ we know that β * γ ≤ α′; we can deduce that α′ is itself a pair of the form β′ * γ′, and by covariance of the product, we know that β ≤ β′ and γ ≤ γ′, allowing to conclude by casting, recursively, at types β′ expr and γ′ expr.

Similarly, in the Int case, we know that α = int and an int expr is returned; this is because we know that, in OCaml, no type is above int: if int ≤ α′ then α′ is int.

What we use in both cases is reasoning of the form\(^1\): “if $T[β] \leq α′$, then α′ is of the form $T[β]$ for some types $β$.” We call this an upward closure property: when we “go up” from a $T[β]$, 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 $T[β] \geq α′$ entails that α′ is of the form $T[β]$.

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 β * γ, in a covariant GADT\(^2\):

```plaintext
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:

```plaintext
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.

```plaintext
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 this work, we prove that the notions of upward and downward closures are the key to a sound variance check for GADT. We start 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 are able to split it into three smaller criteria, which are simple to implement in a type-checker. One of them, the most delicate and important one, 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: they are not necessarily preserved by extensions of the type system.

\(^1\)We write $[β]$ for a sequence of type variables $β_1, \ldots, β_n$, $T[β]$ for a type with free type variables $β$ and $T[β]$ for the same type where $β$ have been replaced by $β$.

\(^2\)This counterexample is due to Jacques Garrigue.
subtyping lattice. For example, OCaml has a concept of
private types: a type specified by \texttt{type t = private \tau} is
a new semi-abstract type smaller than \tau (t \leq \tau but
t \not\leq \tau). As private types can be defined from any type,
no type is downward-closed: for any type \tau 1 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 \texttt{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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained. Thus, 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
must be retained.