GADTs meet Subtyping

Gabriel Scherer, Didier Rémy

Gallium – INRIA

A reminder on GADTs

GADTs are algebraic data types that may carry *type equalities*. Think of the following simple type:

```
type expr =
  | Int of int
  | Bool of bool
```

A reminder on GADTs

GADTs are algebraic data types that may carry *type equalities*. Think of the following simple type:

```
type expr =
  | Int of int
  | Bool of bool
```

It can be turned into the more finely typed:

```
type α expr =
  | Int of int with α = int
  | Bool of bool with α = bool
```

```
type α expr =
  | Int : int -> int expr
  | Bool : bool -> bool expr
```

A reminder on GADTs

GADTs are algebraic data types that may carry *type equalities*. Think of the following simple type:

```
type expr =
  | Int of int
  | Bool of bool
```

It can be turned into the more finely typed:

```
type α expr =
  | Int of int with α = int
  | Bool of bool with α = bool
```

```
type α expr =
  | Int : int -> int expr
  | Bool : bool -> bool expr
```

We can now write the following:

```
let eval : ∀α. α expr → α = function
  | Int n -> n        (* α = int *)
  | Bool b -> b       (* α = bool *)
```

Motivating variance

Subtyping: $\sigma \leq \tau$ means "all values of σ are also values of τ".
Checked by set of decidable and incomplete inference rules.

$$\frac{\sigma_1 \geq \sigma_1' \qquad \sigma_2 \leq \sigma_2'}{(\sigma_1 \to \sigma_2) \leq (\sigma_1' \to \sigma_2')}$$

Variance annotations lift subtyping to type parameters.

$$\texttt{type (-}\alpha\texttt{, =}\beta\texttt{, +}\gamma\texttt{) t = } (\alpha * \beta) \to (\beta * \gamma)$$

$$\frac{\alpha \geq \alpha' \qquad \beta = \beta' \qquad \gamma \leq \gamma'}{(\alpha, \beta, \gamma)\ \texttt{t} \leq (\alpha', \beta', \gamma')\ \texttt{t}}$$

For simple types, this is easy to check.

Variance for GADT: harder than it seems

Ok?

type $+\alpha$ expr =
  | Val : $\forall\alpha.\ \alpha \to \alpha\ \texttt{expr}$
  | Prod : $\forall\beta\gamma.\ \beta\ \texttt{expr} * \gamma\ \texttt{expr} \to (\beta * \gamma)\ \texttt{expr}$

Variance for GADT: harder than it seems

Ok?

type $+\alpha$ expr =
  | Val : $\forall\alpha.\ \alpha \rightarrow \alpha$ expr
  | Prod : $\forall\beta\gamma.\ \beta$ expr $* \gamma$ expr $\rightarrow (\beta * \gamma)$ expr

And this one?

```
type file_descr = private int (* file_descr ≤ int *)
val stdin : file_descr

type +α t =
  | File : file_descr -> file_descr t
let o = File stdin in
let o' = (o : file_descr t :> int t)
```

Variance for GADT: harder than it seems

Ok?

type $+\alpha$ expr =
  | Val : $\forall\alpha.\ \alpha \rightarrow \alpha$ expr
  | Prod : $\forall\beta\gamma.\ \beta$ expr $* \gamma$ expr $\rightarrow (\beta * \gamma)$ expr

And this one?

type file_descr = private int (* file_descr $\leq$ int *)
val stdin : file_descr

type $+\alpha$ t =
  | File : file_descr -> file_descr t
let o = File stdin in
let o' = (o : file_descr t :> int t)

It's unsound!

let project : $\forall\alpha.\ \alpha$ t $\rightarrow (\alpha \rightarrow$ file_descr$)$ = function
  | File _ -> (fun x -> x)
project o' : int -> file_descr

Upward closure

```
type +α expr =
  | Val : ∀α. α → α expr
  | Prod : ∀βγ. β expr ∗ γ expr → (β ∗ γ) expr
```

When $\sigma \leq \sigma'$, I know it's safe to assume σ expr $\leq \sigma'$ expr.
Because I could *almost* write that conversion myself.

Upward closure

```
type +α expr =
  | Val : ∀α. α → α expr
  | Prod : ∀βγ. β expr ∗ γ expr → (β ∗ γ) expr
```

When $\sigma \leq \sigma'$, I know it's safe to assume σ expr $\leq \sigma'$ expr.
Because I could *almost* write that conversion myself.

```
let coerce (α ≤ α′) : α expr ≤ α′ expr = function
  | Val (v : α) -> Val (v :> α′)
  | Prod β γ ((b, c) : β expr ∗ γ expr) ->
      (* α = (β ∗ γ), α ≤ α′; Prod? *)
```

Upward closure

```
type +α expr =
  | Val  : ∀α. α → α expr
  | Prod : ∀βγ. β expr ∗ γ expr → (β ∗ γ) expr
```

When $\sigma \leq \sigma'$, I know it's safe to assume σ expr $\leq \sigma'$ expr.
Because I could *almost* write that conversion myself.

```
let coerce (α ≤ α′) : α expr ≤ α′ expr = function
  | Val (v : α) -> Val (v :> α′)
  | Prod β γ ((b, c) : β expr ∗ γ expr) ->
      (* α = (β ∗ γ), α ≤ α′; Prod? *)
      (* 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))
```

Upward closure

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

When $\sigma \leq \sigma'$, I know it's safe to assume σ expr $\leq \sigma'$ expr.
Because I could *almost* write that conversion myself.

```
let coerce (α ≤ α') : α expr ≤ α' expr = function
  | Val (v : α) -> Val (v :> α')
  | Prod β γ ((b, c) : β expr * γ expr) ->
      (* α = (β * γ), α ≤ α'; Prod? *)
      (* 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))
```

Upward closure for $\tau[\overline{\alpha}]$:

If $\tau[\overline{\sigma}] \leq \tau'$, then τ' is also of the form $\tau[\overline{\sigma}']$ for some $\overline{\sigma}'$.

Holds for $\alpha * \beta$, but fails for `file_descr`.

A conflict

Conversely, to be contravariant, a GADT parameter need be instantiated with types τ that are *downward-closed*: if $\tau' \leq \tau[\overline{\sigma}]$, then τ' is also of the form $\tau[\overline{\sigma}']$ for some $\overline{\sigma}'$.

But this will not work in presence of `private` types: for any τ we can define a distinct type $\tau' := \texttt{private}\ \tau$ with $\tau' \leq \tau$.

Are GADT contravariance and private types incompatible?

Closed-world vs. open-world

Think about a subtyping fact $\sigma \leq \tau$ as a *knowledge* about the world (of types). Private types allow to introduce, at any point in time, a new subtyping relation. You learn something new about the world.

Closure properties are a negative property: a bunch of subtyping relations *must not exist* for some variance annotation to be correct. Checking it makes a *closed world* assumption.

Closed-world vs. open-world

Think about a subtyping fact $\sigma \leq \tau$ as a *knowledge* about the world (of types). Private types allow to introduce, at any point in time, a new subtyping relation. You learn something new about the world.

Closure properties are a negative property: a bunch of subtyping relations *must not exist* for some variance annotation to be correct. Checking it makes a *closed world* assumption.

Types internal to a module live in a closed world. For the exposed ones, it is also possible to resolve this tension by letting the programmer *request* that some subtyping relations *will never hold*.

```
type t = downward-closed
  | Foo ...
  | Bar ...
```

A private synonym of `t` would be rejected by the compiler.
(In other words, `t` is not privatizable. Socialist France striking again.)

An interesting problem

We have to *restrict* subtyping of private types to *enrich* it for GADTs.
A difficult design tension.

It can be compared to the use of `final` classes in object-oriented programming languages.

Going technical : how we prove these things

We must reject unsound GADT definitions. We need an algorithm to tell whether a given type is {upward,downward}-closed.

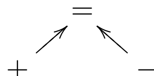
We will first explain how to check variance of type variables by a judgment $\Gamma \vdash \tau : v$.
Resembles previous work [Emir et al., 2006] and [Abel, 2006], with a twist.

We will then extend it to a judgment $\Gamma \vdash \tau : v \Rightarrow v'$ to check closure properties.

With all that and a little plumbing, we can use a soundness proof from the literature [Simonet and Pottier, 2007].
(We got the decomposability criterion by going backward from S&P.)

Variances

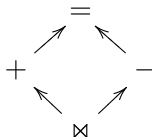
- $+$: only positive occurences
- $-$: only negative occurences
- $=$: both positive and negative



$$
\begin{array}{lcl}
\sigma \prec_{+} \tau & := & \sigma \leq \tau \\
\sigma \prec_{-} \tau & := & \sigma \geq \tau \\
\sigma \prec_{=} \tau & := & \sigma = \tau
\end{array}
$$

Variances

- $+$: only positive occurences
- $-$: only negative occurences

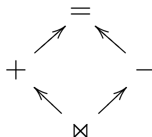


- $=$: both positive and negative
- $\bowtie$: no occurence at all

$$\begin{array}{lcl} \sigma \prec_{+} \tau & := & \sigma \leq \tau \\ \sigma \prec_{-} \tau & := & \sigma \geq \tau \\ \sigma \prec_{=} \tau & := & \sigma = \tau \\ \sigma \prec_{\bowtie} \tau & := & \texttt{true} \end{array}$$

Variances

- $+$: only positive occurences
- $-$: only negative occurences
- $=$: both positive and negative
- $\bowtie$: no occurence at all



$$
\begin{aligned}
\sigma \prec_{+} \tau &:= \sigma \leq \tau \\
\sigma \prec_{-} \tau &:= \sigma \geq \tau \\
\sigma \prec_{=} \tau &:= \sigma = \tau \\
\sigma \prec_{\bowtie} \tau &:= \texttt{true}
\end{aligned}
$$

If α has variance v in $(\alpha\ \texttt{t})$, and β variance w in $(\beta\ \texttt{u})$, what is the variance of α in $((\alpha\ \texttt{t})\ \texttt{u})$?

$v.w$	$=$	$+$	$-$	$\bowtie$
$=$	$=$	$=$	$=$	$\bowtie$
$+$	$=$	$+$	$-$	$\bowtie$
$-$	$=$	$-$	$+$	$\bowtie$
$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$	$\bowtie$

(rows: v; columns: w)

$\Gamma \vdash \tau : v$

We manipulate contexts Γ of variables with variances: $(-\alpha, =\beta, +\gamma)$.
$\Gamma \vdash \tau : v$ means that "if the variables vary according to their variance, τ varies along v".

$$-\alpha, =\beta, +\gamma \;\vdash\; (\alpha * \beta) \rightarrow (\beta * \gamma) \;:\; (+)$$

$$=\alpha, =\beta, =\gamma \;\vdash\; (\alpha * \beta) \rightarrow (\beta * \gamma) \;:\; (=)$$

$\Gamma \vdash \tau : v$

We manipulate contexts Γ of variables with variances: $(-\alpha, =\beta, +\gamma)$.
$\Gamma \vdash \tau : v$ means that "if the variables vary according to their variance, τ varies along v".

$$-\alpha, =\beta, +\gamma \;\vdash\; (\alpha * \beta) \to (\beta * \gamma) \;:\; (+)$$

$$=\alpha, =\beta, =\gamma \;\vdash\; (\alpha * \beta) \to (\beta * \gamma) \;:\; (=)$$

$$\frac{w\alpha \in \Gamma \qquad w \geq v}{\Gamma \vdash \alpha : v} \qquad\qquad \frac{\Gamma \vdash \overline{w\alpha}\;\mathtt{t} \qquad \forall i,\ \Gamma \vdash \sigma_i : v.w_i}{\Gamma \vdash \overline{\sigma}\;\mathtt{t} : v}$$

For instance, in the arrow case:

$$\frac{\Gamma \vdash \sigma_1 : v.- \qquad \Gamma \vdash \sigma_2 : v.+}{\Gamma \vdash (\sigma_1 \to \sigma_2) : v}$$

Closure properties in depth

In our system, $\alpha * \beta$ is upward-closed. This is because the head type constructor, $(*)$, is closed.

For $\alpha\ \mathtt{t} \rightarrow (\beta * \gamma)$ to be upward-closed, $\alpha\ \mathtt{t}$ must be downward-closed. In the general case, we recursively check closure, according to variance.

What about variables?

What about variables?

[Reminder] *Upward closure*: If $\tau[\overline{\sigma}] \leq \tau'$, then $\tau' = \tau[\overline{\sigma}']$ for some $\overline{\sigma}'$.

$\beta * \beta$ is not closed : $(\texttt{file_descr} * \texttt{file_descr}) \leq (\texttt{file_descr} * \texttt{int})$.
Repeating a variable twice is dangerous.

What about variables?

[Reminder] *Upward closure*: If $\tau[\overline{\sigma}] \leq \tau'$, then $\tau' = \tau[\overline{\sigma}']$ for some $\overline{\sigma}'$.

$\beta * \beta$ is not closed : $(\texttt{file_descr} * \texttt{file_descr}) \leq (\texttt{file_descr} * \texttt{int})$.
Repeating a variable twice is dangerous.

Yet, $(\beta\ \texttt{ref}) * (\beta\ \texttt{ref})$ is closed... because *all* occurences are invariant.

What about variables?

[Reminder] *Upward closure*: If $\tau[\overline{\sigma}] \leq \tau'$, then $\tau' = \tau[\overline{\sigma}']$ for some $\overline{\sigma}'$.

$\beta * \beta$ is not closed : $(\texttt{file_descr} * \texttt{file_descr}) \leq (\texttt{file_descr} * \texttt{int})$.
Repeating a variable twice is dangerous.

Yet, $(\beta\ \texttt{ref}) * (\beta\ \texttt{ref})$ is closed... because *all* occurences are invariant.

We capture those subtleties through a partial variance operation $v_1 \curlywedgeuparrow v_2$.
Defined only when two occurences at variances v_1 and v_2 can be soundly combined.

$v \curlywedgeuparrow w$ (rows: v, columns: w)	$=$	$+$	$-$	$\bowtie$
$=$	$=$			$=$
$+$				$+$
$-$				$-$
$\bowtie$	$=$	$+$	$-$	$\bowtie$

$\Gamma \vdash \tau : v \Rightarrow v'$

We can finally extend the judgment $\Gamma \vdash \tau : v$ to capture closure properties. We want to say that $\Gamma \vdash \tau$ is v-closed if:

$$\forall \tau', \overline{\sigma},\ \tau[\overline{\sigma}] \prec_v \tau' \implies \exists \overline{\sigma}',\ \tau[\overline{\sigma}'] = \tau'$$

We need a generalization:

$$\forall \tau', \overline{\sigma},\ \tau[\overline{\sigma}] \prec_v \tau' \implies \exists \overline{\sigma}',\ \tau[\overline{\sigma}'] \prec_{v'} \tau'$$

This is our $\Gamma \vdash \tau : v \Rightarrow v'$ judgment.

Inference rules for the show

They rely on closure information for type constructors, and $\curlywedge$ to merge contexts of subterms.

Inference rules for the show

They rely on closure information for type constructors, and $\curlywedge$ to merge contexts of subterms.

$$\dfrac{v \geq v' \qquad \Gamma \vdash \tau : v}{\Gamma \vdash \tau : v \Rightarrow v'}\;\textsc{Triv} \qquad \dfrac{w\alpha \in \Gamma \qquad w = v}{\Gamma \vdash \alpha : v \Rightarrow v'}\;\textsc{Var}$$

$$\dfrac{\Gamma \vdash \overline{w\alpha}\ \mathtt{t} : v\text{-closed} \qquad \Gamma = \curlywedge_i \Gamma_i \qquad \forall i,\ \Gamma_i \vdash \sigma_i : v.w_i \Rightarrow v'.w_i}{\Gamma \vdash \overline{\sigma}\ \mathtt{t} : v \Rightarrow v'}\;\textsc{Constr}$$

Another solution

We showed, through hard work, how to check that equality constraints are upward-closed.

With subtyping in constructor types, variance is easy to check

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

Now pattern-matching only knows a subtyping relation:

```
let eval : ∀α. α expr → α = function
  | Int n -> (n :> α)          (* α ≥ int *)
  | Bool b -> (b :> α)         (* α ≥ bool *)
```

Less convenient: use of subtyping must be annotated explicitly, while equations where implicit.

Not presented here:

- GADTs with subtyping rather than equalities, less closure constraints
- What if we had a symmetric for `private`?

Future work:

- Completeness of the S&P criterion: type inhabitation.
- Verify behavior through type abstraction.
- Does this also happen with inductive dependent types?

Conclusion

GADT variance checking: suprisingly less obvious than we thought.

Not anecdotal: raises deeper design questions (open vs. closed world).

We have a sound criterion that can be implemented easily in a type checker.

Bonus Slide: Variance and the value restriction

```
type (='a) ref = { mutable contents : 'a }
```

In a language with mutable data, generalizing any expression is unsafe (because you may generalize data locations):

```
# let test = ref [];;
val test : '_a list ref
```

Solution (Wright, 1992): only generalize values (`fun () -> ref []`, or `[]`).

Painful when manipulating polymorphic data structures:

```
let test = id [] (* not generalized? *)
```

OCaml relies on variance for the *relaxed value restriction* [Garrigue, 2004]: covariant data is immutable, so covariant type variables may be safely generalized. Very useful in practice.

```
# let test = id [];;
val test : 'a list = []
```

Abel, A. (2006).
Polarized subtyping for sized types.
Mathematical Structures in Computer Science.
Special issue on subtyping, edited by Healfdene Goguen and Adriana Compagnoni.

Emir, B., Kennedy, A., Russo, C., and Yu, D. (2006).
Variance and generalized constraints for C# generics.
In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06.

Garrigue, J. (2004).
Relaxing the value restriction.
In *In International Symposium on Functional and Logic Programming, Nara, LNCS 2998*.

Simonet, V. and Pottier, F. (2007).
A constraint-based approach to guarded algebraic data types.
ACM Transactions on Programming Languages and Systems, 29(1).