GADTs meet Subtyping

Gabriel Scherer, Didier Rémy

Gallium – INRIA

2014

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?

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

Variance for GADT: harder than it seems

Ok?

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

Variance for GADT: harder than it seems

Ok?

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

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)
```

Breaks abstraction!

```
let project : ∀α. α t → (α → file_descr) = function
  | File _ -> (fun x -> x)
project o' : int -> file_descr
```

Proving an example correct

```
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.

Proving an example correct

```
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? *)
```

Proving an example correct

```
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))
```

Proving an example correct

```
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 = private int`.

In the general case

Consider a GADT α `t` with a constructor of the form

$\quad$ `| K of` $\exists\overline{\beta}$`[`α `=` $T[\overline{\beta}]$`].` $\tau[\overline{\beta}]$

Imagine I have a value v of type σ `t`, and I know $\sigma \leq \sigma'$. Can I convert this σ `t` into a σ' `t`? Let's write the coercion code again:

```
match v : σ t with
...
| K arg -> (arg : τ[ρ̄] :> τ[?])
```

We can type-check this coercion term when $\sigma \leq \sigma'$ if and only if

$$\forall \overline{\rho}, \quad \sigma = T[\overline{\rho}] \implies \exists \overline{\rho}',\ \sigma' = T[\overline{\rho}'] \ \wedge\ \tau[\overline{\rho}] \leq \tau[\overline{\rho}']$$

This extends both upward-closure and the usual variance check on τ.

The semantic criterion

A GADT declaration for $\overline{v\alpha}$ t is correct if, for each constructor K of type $\exists\overline{\beta}[D[\overline{\alpha},\overline{\beta}]].\tau[\overline{\beta}]$, we have

$$\forall \overline{\sigma}\overline{\sigma}'\overline{\rho}, \quad \overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}\ \wedge\ D[\overline{\sigma},\overline{\rho}] \implies \exists\overline{\rho}',\ D[\overline{\sigma}',\overline{\rho}']\ \wedge\ \tau[\overline{\rho}] \leq \tau[\overline{\rho}']$$

How can we check this?
What does it even mean?

Our job: get something *syntactic* out of this semantic criterion, that compilers *and* humans can understand and use.

The plan

We will first explain how to check variance of type variables by a judgment $\Gamma \vdash \tau : v$.
Resembles previous work with a twist.

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

From there it's not too hard (but not too easy either) to derive the final syntactic formulation of the correctness criterion.

Variances

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

$$
\begin{tikzcd}
 & = & \\
+ \arrow[ur] & & - \arrow[ul]
\end{tikzcd}
$$

- $=$: 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

$$
\begin{tikzcd}
 & = & \\
+ \arrow[ur] & & - \arrow[ul] \\
 & \bowtie \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

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

Variances

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

$$\begin{tikzcd}
& = & \\
+ \arrow[ur] & & - \arrow[ul] \\
& \bowtie \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

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

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

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

$\Gamma \vdash \tau : v$

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".

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

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

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

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 $\alpha\ \mathtt{t} \to (\beta * \gamma)$ 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.

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

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

How do you check those rules?

Theorem

$\Gamma \vdash \tau : v$ *holds if and only if* $\forall \overline{\sigma}, \overline{\sigma}',\ \overline{\sigma} \prec_\Gamma \overline{\sigma}' \implies \tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$

Simple.

How do you check those rules?

Theorem

$\Gamma \vdash \tau : v$ *holds if and only if* $\forall \overline{\sigma}, \overline{\sigma}',\ \overline{\sigma} \prec_\Gamma \overline{\sigma}' \implies \tau[\overline{\sigma}] \prec_v \tau[\overline{\sigma}']$

Simple.

Theorem

$\Gamma \vdash \tau : v \Rightarrow (=)$ *holds if and only if* $\Gamma \vdash \tau : v$ *and*
$\forall \overline{\rho}, \tau',\ \tau[\overline{\rho}] \prec_v \tau' \implies \exists \overline{\rho}',\ \tau[\overline{\rho}'] = \tau'$

Soundness (syntactic $\implies$ semantic): routine.
Completeness (semantic $\implies$ syntactic): surprisingly hard.

Back to the check

From these primitives, we can devise a syntactic check for our initial semantic criterion. Assume we have the variances $\overline{v\alpha}$ and a constructor declaration of the form $(\exists\overline{\beta}\left[\bigwedge_{i \in I} \alpha_i = T_i[\overline{\beta}]\right].\tau[\overline{\beta}])$. Remember the *sophisticated* semantic criterion:

$$\forall\overline{\sigma}, \overline{\sigma}', \overline{\rho}, \quad \overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}\ \wedge\ (\sigma_i = T_i[\overline{\rho}])_{i \in I} \implies \exists\overline{\rho}',\ (\sigma'_i = T_i[\overline{\rho}'])_{i \in I}\ \wedge\ \tau[\overline{\rho}] \leq \tau[\overline{\rho}']$$

Back to the check

From these primitives, we can devise a syntactic check for our initial semantic criterion. Assume we have the variances $\overline{v\alpha}$ and a constructor declaration of the form $(\exists\overline{\beta}\left[\bigwedge_{i\in I}\alpha_i = T_i[\overline{\beta}]\right].\tau[\overline{\beta}])$. Remember the *sophisticated* semantic criterion:

$$\begin{aligned}\forall\overline{\sigma},\overline{\sigma}',\overline{\rho},\quad \overline{\sigma}\ \mathtt{t} \leq \overline{\sigma}'\ \mathtt{t}\ \wedge\ (\sigma_i = T_i[\overline{\rho}])_{i\in I} \implies \\ \exists\overline{\rho}',\ (\sigma'_i = T_i[\overline{\rho}'])_{i\in I}\ \wedge\ \tau[\overline{\rho}] \leq \tau[\overline{\rho}']\end{aligned}$$

Theorem (Algorithmic criterion)

The soundness criterion above is equivalent to

$$\exists\Gamma,(\Gamma_i)_{i\in I},\quad \Gamma\vdash\tau:(+)\quad\wedge\quad \Gamma = \curlywedge_{i\in I}\Gamma_i\quad\wedge\quad \forall i\in I,\Gamma_i\vdash T_i:v_i\Rightarrow(=)$$

(Oral explanation)

Phew !

That was the formal side of things.

Phew !

That was the formal side of things.

This criterion raises interesting design issues: `private` definitions make all OCaml types non-downward-closed.
Should we restrict those? The opposite of OOP's `final` keyword.

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.

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.

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* covariant data is immutable, so covariant type variables may be safely generalized. Very useful in practice.

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