GADTs meet Subtyping

Gabriel Scherer, Didier Rémy

Gallium – INRIA

Motivation

GADTs were just added to OCaml.

OCaml also has limited, but useful, support for subtyping.
Type parameters have a variance.

How can we check the variances of GADT definitions?

Introduction to GADTs

With algebraic datatypes we often write things like:

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

```
let get_int : expr -> int = function
  | Int n -> n
  | Bool _ -> failwith "int excepted"
```

GADTs allow a more fine-grained typing

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

```
let get_int : int expr → int = function
  | Int n -> n
```

Internally, GADTs are data types that may carry **type equalities** (and existentials).

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

Equalities are used during pattern-matching: refinement and dead cases.

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

let get_int : int expr → int = function
  | Int n -> n
  (* case Bool is dead: (bool = int) unsatisfiable *)
```

Variance of type parameters

Subtyping: $\sigma \leq \tau$ means "all values of σ are also values of τ".
Equality $(\sigma = \tau)$ defined as $(\sigma \leq \tau) \wedge (\sigma \geq \tau)$.

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

Variance describes subtyping on a type from subtyping on its parametrs.

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

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

For simple types, this is easy to check. For GADTs?

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.

General framework with arbitrary constraints.
Generic **semantic** soundness criterion (hairy first-order formula).

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, 2006.

Distinct setting of **subtyping** constraints.
Simple syntactic soundness criterion.

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.

General framework with arbitrary constraints.
Generic **semantic** soundness criterion (hairy first-order formula).

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, 2006.

Distinct setting of **subtyping** constraints.
Simple syntactic soundness criterion.

We need : a **syntactic** criterion for **equality** constraints. New, hard.

Variance for GADT: harder than it seems

Is this definition correct?

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

The constructor Val could appear in a simple ADT:

  | Val of α

What about Prod?

Variance for GADT: harder than it seems (2)

And this one?

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

type +α t =
  | File : file_descr -> file_descr t
```

Variance for GADT: harder than it seems (2)

And this one?

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

type +α t =
  | File : file_descr -> file_descr t
```

Breaks abstraction!

```
let o = File stdin in
let o' = (o : file_descr t :> int t)

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

(Using polymorphic variants or object types, you could break soundness)

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 rec coerce (α ≤ α′) : α expr ≤ α′ expr = function
```

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 rec coerce (α ≤ α′) : α expr ≤ α′ expr = function
  | Val (v : α) -> Val (v :> α′)
```

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 rec coerce (α ≤ α') : α expr ≤ α' expr = function
  | Val (v : α) -> Val (v :> α')
  | Prod β γ ((b, c) : β expr ∗ γ expr) ->
```

$\alpha = (\beta * \gamma)\,, \qquad \alpha \;\leq\; \alpha'$

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

$\alpha = (\beta * \gamma)\,, \qquad \alpha \ \leq \ \alpha'$

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

$\alpha = (\beta * \gamma)$, $\qquad \alpha \leq \alpha'$

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

$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha'$

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

$\alpha = (\beta * \gamma)\,, \quad (\beta \boxed{*} \gamma) \boxed{\leq} \alpha' \qquad \Longrightarrow$

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

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \qquad \Longrightarrow \qquad \exists \beta', \gamma', \quad \alpha' = (\beta' * \gamma')$$

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

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \qquad \Longrightarrow \qquad \exists \beta', \gamma', \quad \alpha' = (\beta' * \gamma')$$

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

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \qquad \Longrightarrow \qquad \exists \beta', \gamma',\ \alpha' = (\beta' * \gamma')$$

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

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \quad \Longrightarrow \quad \exists \beta', \gamma', \quad \alpha' = (\beta' * \gamma')$$

$$(\beta * \gamma) \leq (\beta' * \gamma')$$

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

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \quad \Longrightarrow \quad \exists \beta', \gamma', \quad \alpha' = (\beta' * \gamma')$$

$$(\beta * \gamma) \leq (\beta' * \gamma') \quad \Longrightarrow \quad \beta \leq \beta', \quad \gamma \leq \gamma'$$

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 rec coerce (α ≤ α') : α expr ≤ α' expr = function
  | Val (v : α) -> Val (v :> α')
  | Prod β γ ((b, c) : β expr ∗ γ expr) ->
     Prod β' γ' (coerce (β ≤ β') b, coerce (γ ≤ γ') c)
```

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \implies \exists \beta', \gamma', \quad \alpha' = (\beta' * \gamma')$$

$$(\beta * \gamma) \leq (\beta' * \gamma') \implies \beta \leq \beta', \quad \gamma \leq \gamma'$$

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

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \implies \exists \beta', \gamma', \quad \alpha' = (\beta' * \gamma')$$

$$(\beta * \gamma) \leq (\beta' * \gamma') \implies \beta \leq \beta'\,, \quad \gamma \leq \gamma'$$

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

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \qquad \Longrightarrow \qquad \exists \beta', \gamma', \quad \alpha' = (\beta' * \gamma')$$

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 rec coerce (α ≤ α') : α expr ≤ α' expr = function
  | Val (v : α) -> Val (v :> α')
  | Prod β γ ((b, c) : β expr ∗ γ expr) ->
     Prod β' γ' (coerce ( β ≤ β' ) b, coerce ( γ ≤ γ' ) c)
```

$$\alpha = (\beta * \gamma)\,, \quad (\beta * \gamma) \leq \alpha' \qquad \Longrightarrow \qquad \exists \beta', \gamma', \quad \alpha' = (\beta' * \gamma')$$

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 $\beta * \gamma$, but fails for `file_descr = private int`.

In a more general case

```
type +α t =
  | ...
  | K of  ∃β̄[α = E[β̄]] .  A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

In a more general case

```
type +α t =
  | ...
  | K of  ∃β̄[α = E[β̄]] .  A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

```
match v :  σ t  with
| ...
|  K arg  -> K   arg
```

We can type-check this *partial* coercion term if and only if:

In a more general case

```
type +α t =
  | ...
  | K of ∃β̄[α = E[β̄]]. A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

```
match v : σ t with
| ...
| K arg -> K   arg
```

We can type-check this *partial* coercion term if and only if:

In a more general case

```
type +α t =
  | ...
  | K of  ∃β̄[α = E[β̄]] .  A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

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

We can type-check this *partial* coercion term if and only if:

$$\boxed{\sigma = E[\overline{\rho}]} \implies$$

In a more general case

```
type +α t =
  | ...
  | K of ∃β̄[α = E[β̄]] . A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

```
match v : σ t with
| ...
| K arg -> K ( arg : A[ρ̄] :> A[ρ̄'])
```

We can type-check this *partial* coercion term if and only if:

$$\sigma = E[\overline{\rho}] \implies \exists \overline{\rho}',$$

In a more general case

```
type +α t =
  | ...
  | K of  ∃β̄[α = E[β̄]] . A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

```
match v : σ t with
| ...
| K arg -> K ( arg : A[ρ̄] :> A[ρ̄'])
```

We can type-check this *partial* coercion term if and only if:

$$\sigma = E[\overline{\rho}] \implies \exists\, \overline{\rho}',\ \boxed{A[\overline{\rho}] \leq A[\overline{\rho}']}$$

In a more general case

```
type +α t =
  | ...
  | K of  ∃β̄[α = E[β̄]] .  A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

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

We can type-check this *partial* coercion term if and only if:

$$\sigma = E[\overline{\rho}] \implies \exists\, \overline{\rho}' ,\quad A[\overline{\rho}] \leq A[\overline{\rho}'] \;\wedge\; \sigma' = E[\overline{\rho}']$$

In a more general case

```
type +α t =
  | ...
  | K of  ∃β̄[α = E[β̄]] .  A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

```
match v :  σ t  with
| ...
|  K arg  -> K ( arg : A[ ρ̄ ]  :>  A[ ρ̄' ])
```

We can type-check this *complete* coercion term if and only if:

$$\forall \overline{\rho}, \quad \sigma = E[\overline{\rho}] \implies \exists\, \overline{\rho}', \quad A[\overline{\rho}] \leq A[\overline{\rho}'] \ \wedge\ \sigma' = E[\overline{\rho}']$$

This **semantic criterion** (also found in [SP07]) extends both upward-closure and the usual variance check on A.

In a more general case

```
type +α t =
  | ...
  | K of  ∃β̄[α = E[β̄]] .  A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ `t` into a σ' `t`?

```
match v :  σ t  with
| ...
|  K arg  -> K ( arg : A[ ρ̄ ]  :>  A[ ρ̄' ])
```

We can type-check this *complete* coercion term if and only if:

$$\forall \overline{\rho}, \quad \sigma = E[\overline{\rho}] \implies \exists\, \overline{\rho}'\,, \quad A[\overline{\rho}] \leq A[\overline{\rho}'] \;\wedge\; \sigma' = E[\overline{\rho}']$$

This **semantic criterion** (also found in [SP07]) extends both upward-closure and the usual variance check on A.

In a more general case

```
type +α t =
  | ...
  | K of  ∃β̄[α = E[β̄]] .  A[β̄]
```

Assume that $\sigma \leq \sigma'$. Can I convert any value $v : \sigma$ t into a σ' t?

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

We can type-check this *complete* coercion term if and only if:

$$\forall \overline{\rho}, \quad \sigma = E[\overline{\rho}] \implies \exists\, \overline{\rho}', \; \boxed{A[\overline{\rho}] \leq A[\overline{\rho}']} \;\wedge\; \sigma' = E[\overline{\rho}']$$

This **semantic criterion** (also found in [SP07]) extends both upward-closure and the usual variance check on A.

From semantics to syntax

The semantic criterion is a logic property (constraint entailment problem).

$$\forall \overline{\rho}, \quad (\forall i,\ \sigma_i = E_i[\overline{\rho}]) \implies \exists \overline{\rho}',\ \tau[\overline{\rho}] \leq \tau[\overline{\rho}'] \ \wedge\ (\forall i,\ \sigma'_i = E_i[\overline{\rho}'])$$

Our contribution: equivalent syntactic judgments.
Easier to implement and explain to users.

1. $\Gamma \vdash \tau : v$ — a variance check (well-known)
2. $\Gamma \vdash \tau : v \Rightarrow (=)$ — a v-closure check (new!)
3. $\Gamma_1 \curlywedge \Gamma_2$ — *an interference check (see the paper!)*

Correctness (syntax implies semantics) : easy with the right definitions.
Completeness (semantics implies syntax) : quite challenging.

From semantics to syntax: Some subtleties

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

$\beta * \gamma$ is upward-closed. What about $\beta * \beta$?
Repeating a variable twice is dangerous.

From semantics to syntax: Some subtleties

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

$\beta * \gamma$ is upward-closed. What about $\beta * \beta$?
Repeating a variable twice is dangerous.

$\beta * \beta$ is not closed : $(\texttt{file\_descr} * \texttt{file\_descr}) \leq (\texttt{file\_descr} * \texttt{int})$.

From semantics to syntax: Some subtleties

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

$\beta * \gamma$ is upward-closed. What about $\beta * \beta$?
Repeating a variable twice is dangerous.

$\beta * \beta$ is not closed : $(\mathtt{file\_descr} * \mathtt{file\_descr}) \leq (\mathtt{file\_descr} * \mathtt{int})$.

Yet, $(\beta\ \mathtt{ref}) * (\beta\ \mathtt{ref})$ is closed
Repeated variables are ok when all occurences are invariant.

From semantics to syntax: Some subtleties

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

$\beta * \gamma$ is upward-closed. What about $\beta * \beta$?
Repeating a variable twice is dangerous.

$\beta * \beta$ is not closed : $(\texttt{file\_descr} * \texttt{file\_descr}) \leq (\texttt{file\_descr} * \texttt{int})$.

Yet, $(\beta\ \texttt{ref}) * (\beta\ \texttt{ref})$ is closed
Repeated variables are ok when all occurences are invariant.

Some more subtleties in the paper (one extra variance), an operator $v_1 \curlywedge v_2$ defined when repeating a variable at variances v_1 and v_2 is ok.

Interesting design implications

This was the technical part of our work.

The closure conditions also raise interesting design questions.

Are GADT contravariance and private types incompatible?

We have discussed upward-closed types, but checking **contravariant** parameters naturally requires **downward-closed** types $\tau[\overline{\sigma}]$:

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

This never works in presence of `private` types: for any τ we can define a distinct type ($\tau' := \texttt{private}\ \tau$) with $\tau' \leq \tau$.

Closed-world vs. open-world

A subtyping fact $\sigma \leq \tau$ is **knowledge** about the world (of types).

Closure criterions make a **closed world** assumption.
However, introducing a private type adds a new subtyping fact.

To solve this tension, add `downward-closed` to forbid future extensions.

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

A private synonym of `t` would then be rejected.

Similar to `final` in object-oriented languages.

GADTs with subtyping constraints

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

This definition is obviously covariant:
If int $\leq \alpha$ and $\alpha \leq \alpha'$, then int $\leq \alpha'$.
No upward-closure issues.

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

But the following isn't obviously correct anymore:

```
let get_int : int expr -> int = function
| Int n -> n
(* bool ≤ int ? *)
```

GADTs with subtyping constraints

Constraints of the form $\alpha \geq \tau$ are obviously correct for a covariant $+\alpha$. Constraints of the form $\alpha \leq \tau$ are hard for $+\alpha$: like type equalities, they need reasoning on closure conditions.

Closure conditions (our work) and subtyping constraints (from [EKRY06]) are of incomparable expressivity:

- For `get_int` you really need equalities.
- For `file_descr`, only subtyping constraints can give you covariance.

In the general case you want to have both.

Conclusion

GADT variance checking: suprisingly less obvious than we thought.

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

Raises deeper design questions: open and closed worlds, GADTs with subtyping constraints.

Conclusion

GADT variance checking: suprisingly less obvious than we thought.

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

Raises deeper design questions: open and closed worlds, GADTs with subtyping constraints.

Thank you!

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

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

Triv
$$\frac{v \geq v' \qquad \Gamma \vdash \tau : v}{\Gamma \vdash \tau : v \Rightarrow v'}$$

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

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

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 (through module abstractions).

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