Union de variants abstraits

Jacques Garrigue
Nagoya University, Grad. Sch. of Mathematics
avec Romain Bardou, E.N.S. Cachan

Union concrète

```
type u = [‘A of int | ‘B of bool]
let show_u = function .......
val show_u : u -> unit

type v = [‘C of char | ‘D of string]
let show_v = function .......
val show_v : v -> unit

type t = [u | v]
type t = [‘A of int | ‘B of bool | ‘C of char | ‘D of string]
let show = function
    #u as x -> show_u x
  | #v as y -> show_v y
val show : t -> unit
```

Union concrète

```
type u = [‘A of int | ‘B of bool]
let show_u = function ......
val show_u : u -> unit

type v = [‘C of char | ‘A of string]
let show_v = function ......
val show_v : v -> unit

type t = [u | v]
This variant type contains a constructor
[ ‘A of string ] which should be [ ‘A of int ]
```

On vérifie que `u` et `v` sont compatibles

Union abstraite

Que se passe-t-il dans le cas abstrait?

```
module type T =
  sig type t = private [> ] val show : t -> string end
module Mix(X : T)(Y : T) =
  struct
    type t = [X.t | Y.t]
    let show : t -> string = function
        #X.t as e -> X.show e
      | #Y.t as e -> Y.show e
  end
```

Comment assurer la compatibilité de `X.t` avec `Y.t`?

Union abstraite

Idée: déclarer les compatibilités en même temps que les types

```
module type T =
  sig type t = private [> ] val show : t -> string end
module Mix(X : T)(Y : T with type t = private [> ] ~ [X.t]) =
  struct
    type t = [X.t | Y.t]
    let show : t -> string = function
        #X.t as e -> X.show e
      | #Y.t as e -> Y.show e
  end
```

Applications

Construction modulaire de programmes

- domaine d'une interprètation abstraite
- construction d'automates finis
- factorisation de code dans des bibliothèques: GUI, etc. . .
- construction de langages (problème de l'expression. . .)

Étendre le typage à $F_<$

- permet les variables (càd types abstraits) dans les bornes supérieures et inférieures

Synopsis

- Unions concrètes et abstraites
- Modèlisation avec compatibilités
- Vérification des compatibilités
- Union disjointe
- Union semi-disjointe
- Union avec borne supérieure
- Intégration à l'inférence de types

Compatibilités

On introduit 4 types de compatibilités:

$$\texttt{type t} = \texttt{private } [> D] \sim [C]$$

$$\begin{array}{lcll} D & ::= & A \text{ of } \tau \mid t & \\ C & ::= & A \text{ of } \tau & \text{si } A \text{ présent, alors de type } \tau \\ & \mid & t & \text{compatible avec } t \\ & \mid & \neg A & A \text{ absent} \\ & \mid & \neg t & \text{pas d'intersection avec } t \end{array}$$

Comment formaliser leur sémantique?

Modèlisation: union de fonctions

Une valuation assigne un type de variant à chaque déclaration de type.

$$\mathcal{V} = N \to \mathcal{L} \to T_\emptyset$$

$N = \{t_1, \ldots\}$: noms de types
$\mathcal{L} = \{A, B, \ldots\}$: noms de labels
$T = \{\tau_1, \ldots\}$: types
$T_\emptyset = T \cup \{\emptyset\}$: $\emptyset$ exprime l'absence d'un label

Deux types t_1 et t_2 sont compatibles pour une valuation $v \in \mathcal{V}$ si l'union $v(t_1) \cup v(t_2)$ est cohérente.

$$\forall l \in \mathcal{L}\ v(t_1)(l) = \emptyset \vee v(t_2)(l) = \emptyset \vee v(t_1)(l) = v(t_2)(l)$$

Un modèle W est un ensemble de valuations. t_1 et t_2 sont compatibles pour W, si ils sont compatibles pour toutes les valuations de W.

Modèlisation des définitions

On étend les valuations aux types et compatibilités.

$$\begin{aligned} \tilde{v}(t) &= v(t) \\ \tilde{v}(l \text{ of } \tau) &= \{l \mapsto \tau\} \\ \tilde{v}(\neg l) &= \{l \mapsto \Omega\} \\ \tilde{v}(\neg t) &= \{l \mapsto \Omega \mid v(t)(l) \neq \emptyset\} \end{aligned}$$

Partant de $W_0 = \mathcal{V}$, on raffine le modèle incrémentalement en ajustant la définition de chaque type. Pour ajouter une définition δ au modèle W, on applique la fonction $V_W(\delta)$.

Pour les types fermés, il suffit de ne garder que les valuations où la définition est la bonne.

$$V_W(\texttt{type t} = [\mathcal{D}]) = \{v \in W \mid v(t) = \bigcup_{D \in \mathcal{D}} \tilde{v}(D)\}$$

Modèlisation des définitions (2)

Pour les types privés

$$\delta = (\texttt{type t} = \texttt{private}[> \mathcal{D}]\tilde{}[\mathcal{C}])$$

on vérifie d'abord la définition,

$$V_W^0(\delta) = \{v \in W \mid v(t) \supseteq \bigcup_{D \in \mathcal{D}} \tilde{v}(D)\}$$

puis les compatibilités.

$$V_W(\delta) = \{v \in V_W^0(\delta) \mid \forall C \in \mathcal{C}, v(t) \cup \tilde{v}(C) \text{ is coherent}\}$$

Validité des définitions

Une définition δ est valide dans W si

$$\forall v \in W, \forall (D, C) \in \mathcal{D} \times (\mathcal{D} \cup \mathcal{C}), \tilde{v}(D) \cup \tilde{v}(C) \text{ is coherent}$$

Thm 1 *Si δ est valide dans W, alors $V_W(\delta)$ est conservatif.*

$$\forall v \in W, \exists v' \in V_W(\delta), \forall t' \neq t, v'(t') = v(t')$$

Thm 2 *Si $V_W(\delta)$ est conservatif, et δ ne fait référence qu'à des types définis, alors δ est valide dans W.*

Axiomatisation

Les information sur les types privés sont données par un environnement Δ.

$$\begin{array}{rcl} D & ::= & l \texttt{ of } \tau \mid t \\ R & ::= & D \mid ?D \mid \neg D \\ \Delta & ::= & l \mapsto R^*, \ldots \end{array}$$

$$\begin{array}{rl} D \in \Delta(t) & D \text{ inclus dans } t \\ ?D \in \Delta(t) & D \text{ compatible avec } t \\ \neg D \in \Delta(t) & D \text{ disjoint de } t \end{array}$$

Deux jugements:

$$\begin{array}{rl} \Delta \vdash R \Subset t & t \text{ hérite } R \\ \Delta \vdash D \odot D' & D \text{ compatible avec } D' \\ \Delta \vdash \neg D \odot D' & D \text{ disjoint de } D' \end{array}$$

Axiomatisation (2)

$$\frac{R \in \Delta(t)}{\Delta \vdash R \Subset t}\text{In1} \qquad \frac{t' \in \Delta(t) \quad \Delta \vdash D \Subset t'}{\Delta \vdash D \Subset t}\text{In2}$$

$$\frac{?t' \in \Delta(t) \quad \Delta \vdash D \Subset t'}{\Delta \vdash ?D \Subset t}\text{In3} \qquad \frac{\neg t' \in \Delta(t) \quad \Delta \vdash D \Subset t'}{\Delta \vdash \neg D \Subset t}\text{In4}$$

$$\frac{\Delta \vdash C \Subset t}{\Delta \vdash C \odot t}\text{E1} \qquad \frac{\Delta \vdash ?D \Subset t}{\Delta \vdash D \odot t}\text{E2} \qquad \frac{\Delta \vdash \neg D \Subset t}{\Delta \vdash D \odot t}\text{E3}$$

$$\frac{\Delta \vdash ?l \ \texttt{of}\ \tau_1 \Subset t \quad \Delta \vdash ?l \ \texttt{of}\ \tau_2 \Subset t \quad \tau_1 \neq \tau_2}{\Delta \vdash l \ \texttt{of}\ \tau \odot t}\text{LT}$$

$$\frac{l \neq l' \text{ or } \tau = \tau'}{\Delta \vdash l \ \texttt{of}\ \tau \odot l' \ \texttt{of}\ \tau'}\text{LL} \qquad \frac{\Delta \vdash D' \odot D}{\Delta \vdash D \odot D'}\text{Sym} \qquad \frac{\Delta \vdash \neg D \odot D'}{\Delta \vdash \neg D' \odot D}\text{Any}$$

Complètude de l'axiomisation

Thm 3 *L'axiomisation de la compatibilité est correcte et complète par rapport au modèle décrit précédemment.*

Ce résultat est important, car trouver les bonnes règles n'est pas facile. Par exemple, la règle suivant est fausse:

$$\frac{t' \in \Delta(t) \quad \Delta \vdash ?D \Subset t'}{\Delta \vdash ?D \Subset t}\text{In2}$$

N.B. La preuve a été faite pour une version ne contenant pas la négation de types ($\neg t$), seulement d'étiquette ($\neg l$).

“La” solution ?

- Compatibilités très expressives
- Sémantique bien définie

Mais

- Axiomisation compliquée (inévitable?)
- En présence de foncteurs, LT3 n’est pas correcte.
  - sans elle, on pert la complétude
  - difficile de la récupérer
- La vérification de compatibilités pendant l’inférence n’est pas principale.

Contre-exemple pour LT3

```
module M : sig
  type t
  type u = [> ] ~ [‘L of int | ‘L of t]
  val v : [> u]
end = struct
  type t = int
  type u = [‘L of t]
  let v = ‘L 1
end

let l : [u | ‘L of bool] list = [‘L true; v]
```

Restreindre LT3?

Le modèle choisi n'est pas assez précis. Ne pourrait-on pas avoir une définition précise des cas où deux types ne peuvent pas être rendus compatibles?

Difficile

- Nécessite une analyse globale pour les types abstraits:
  il ne suffit pas de s'attaquer aux paramètres de foncteurs.
- Ceux-ci sont utilisés pour modèliser les types de base:
  comment prouver `int` $\neq$ `bool`?
- Même sans ça, implémentation complexe: besoin d'une nouvelle infrastructure pour traiter l'égalité potentielle.

Perte de principalité

```
module M : sig
  type t
  type u = [> ] ~ [‘L of int | ‘L of t]
  val v : [> u]
end = struct
  type t = int
  type u = [‘L of t]
  let v = ‘L 1
end

let f x = [‘L x; v]
```

Quel type pour `x`? Aussi bien `int` que `t` sont valides.

Solution radicale

Exiger que les compatibilités soient compatibles entre elles.

$$\delta = (\texttt{type t} = \texttt{private}[> \mathcal{D}]\tilde{}[\mathcal{C}])$$

La validité originelle était

$$\forall C \in \mathcal{D} \cup \mathcal{C}, \forall D \in \mathcal{D}, \Delta \vdash C \odot D$$

Il faut ajouter

$$\forall D, D' \in \mathcal{C}, \Delta \vdash D \odot D' \vee \exists \neg D'' \in \mathcal{C}, (\Delta \vdash \neg D'' \odot D \vee \Delta \vdash \neg D'' \odot D')$$

Pas d'affaiblissement sensible, mais moins intuitif.

Alternatives

- Union disjointe
  équivalent des rangées à la Rémy

- Union semi-disjointe
  similaire, mais mieux adaptée aux types privés

- Union avec borne supérieure
  régle la question des compatibilités à l'avance

Union disjointe

On ne conserve que les compatibilités $\neg l$ et $\neg t$.

Modèle et axiomatisation simplifiés.

Difficulté pour exprimer des cas où les types ont une intersection.

```
module F(I:sig type t = [> ] type t1 = [> ]~[t]
               type t2 = [> ]~[t;t1] end)
        (X:sig type t = [I.t|I.t1] ... end)
        (Y:sig type t = [I.t|I.t2] ... end) =
struct
  type t = [X.t | Y.t]
  let show : t -> string =
    function #X.t as x -> X.show x | #Y.t as y -> Y.show y
end
```

Union semi-disjointe

Les compatibilités sont traitées comme disjointes pour leurs parties abstraites.

On peut donc écrire:

```
module F(I:sig type t = [> ] end)(X:sig type t = [> I.t] ... end)
  (Y:sig type t = [> I.t] ~ [X.t] ... end) =
struct
  type t = [X.t | Y.t]
  let show : t -> string =
    function #X.t as x -> X.show x | #Y.t as y -> Y.show y
end
```

Union semi-disjointe (2)

Intuition: on interprète

```
type t = [> t0 ] ~ [u1; u2]
```

comme

```
type t_row = [> ] ~ [~u1; ~u2; ~t0]
type t = [t0 | t_row] ~ [u1; u2]
```

Les compatibilités sur `t` sont vérifiées lors de la définition, donc elles ne sont pas nécessaires après. Par conséquent il ne reste que les compatibilités disjointes sur `t_row`.

Union semi-disjointe (3)

L'axiomatisation est très simple:

$$\frac{l \in \Delta(t)}{\Delta \vdash l \texttt{ of } \tau \odot t}\mathsf{LT} \qquad \frac{t = t' \text{ or } t' \in \Delta(t)}{\Delta \vdash t' \odot t}\mathsf{TT}$$

$$\frac{l \neq l' \text{ or } \tau = \tau'}{\Delta \vdash l \texttt{ of } \tau \odot l' \texttt{ of } \tau'}\mathsf{LL} \qquad \frac{\Delta \vdash D' \odot D}{\Delta \vdash D \odot D'}\mathsf{Sym}$$

La sémantique est plus subtile:

$$W \models D_1 \oplus D_2 \Leftrightarrow \exists \mathcal{D}, \forall v \in W, \begin{cases} \tilde{v}(D_1) \cup \tilde{v}(D_2) \text{ coherent} \\ \bigcup_{D \in \mathcal{D}} \tilde{v}(D) = \tilde{v}(D_1) \cap \tilde{v}(D_2) \end{cases}$$

Càd $\mathcal{D}$ est une description syntaxique de l'intersection de D_1 et D_2.

Union semi-disjointe

- Simple à axiomatiser et implémenter.
  (branche varunion du CVS ocaml)

- Intuitivement prôche des rangées à la Rémy,
  tout en étant adaptée aux rangées privées.

Mais

- Sémantique un peu subtile.

- Il faut déclarer explicitement les intersections.

Union avec borne supérieure

On introduit des sous-types abstraits de types abstraits:

```
module F(U:sig type t = [> ] end)(X:sig type t = [< U.t] ... end)
  (Y:sig type t = [< U.t] ... end) =
struct
  type t = [X.t | Y.t]
  let show : t -> string =
    function #X.t as x -> X.show x | #Y.t as y -> Y.show y
end
```

– Plus besoin de compatibilités: il faut juste vérifier l'existence d'un supertype commun

– Il est facile de construire ce supertype (l'union)

Union avec borne supérieure (2)

Difficulté: quel type donner aux unions, et comment garder la principalité de l'inférence?

```
type t = private [> `A of int]
type u = private [< t]
let f = function #u -> true | `A x -> x
val f : [< `A of int & bool | u] -> int
```

Comment savoir que `u` non plus n'est pas utilisable ici (puisque `u` inclus potentiellement `` `A ``)?

Le problème est que `[> u]` ne s'expanse pas en `[> `A of int | t]` (on ne sait pas si `` `A `` est là).

Contraintes locales

En OCaml, les variants polymorphes sont typés par des contraintes qui n'engendrent pas de dépendences.

Idée: un type de variant est décrit comme la paire d'une contrainte structurelle C et d'une relation de typage R:

$$k = (C, R)$$

- C décrit quels constructeurs peuvent ou doivent être présents
- R associe à chaque constructeur le type de son argument

C agit sur R par des contraintes de propagation, qui peuvent obliger tous les types associés à l dans R à être identiques.

$$(C \vdash \mathsf{unique}(l)) \wedge (l \mapsto \tau \in R) \wedge (l \mapsto \tau' \in R) \Rightarrow \tau = \tau'$$

Typage des variants polymorphes

Dans le cas des variants polymorphe usuels, la contrainte est représentée par une paire d'ensembles:

$$C = (L, U)$$

- L ensemble fini de constructeurs obligatoirement présents.
- U ensemble de constructeur potentiellement présents.
- C est valide ssi $L \subset U$.
- $C \vdash \text{unique}(l)$ ssi $l \in L$.

Typage des variants polymorphes (2)

```
[`A 1; `B true] : [> `A of int | `B of bool] list
```

$$\alpha :: (\{A, B\}, \mathcal{L}, \{A \mapsto int, B \mapsto bool\}) \triangleright \alpha\ \mathsf{list}$$

```
let f = function `A x -> x = 0 | `B y -> y
val f : [< `A of int | `B of bool] -> bool
```

$$\alpha :: (\emptyset, \{A, B\}, \{A \mapsto int, B \mapsto bool\}) \triangleright \alpha \to \mathsf{bool}$$

Types avec borne inférieure abstraite

– Les types abstraits sont traités comme des constructeurs

– Les compatibilités sont vérifiées par l'unification

```
type t = private [> `A of int] ~ [`B of bool; ~`C]
type 'a u = private [> `B of bool; `C of 'a] ~ [t]
val v : [> t | char u]
```

$\alpha_t :: (\{A, t\}, \mathcal{L}, 1, \{A \mapsto int, B \mapsto bool, t \mapsto []\})$
$\alpha_u :: (\{B, C, u\}, \mathcal{L}, 1,$
$\{B \mapsto bool, C \mapsto \alpha, t \mapsto [], A \mapsto int, u \mapsto [\alpha]\})$
$\alpha_v :: (\{A, B, C, t, u\}, \mathcal{L}, 0,$
$\{A \mapsto int, B \mapsto bool, C \mapsto char, t \mapsto [], u \mapsto [char]\})$

Types avec borne inférieure abstraite (2)

Les compatibilités des types abstraits sont vérifiés via la validité de la constrainte:

$C = (L, U, p)$ est valide si

- $L \subset U$
- $\forall t, t \in U \Rightarrow t \in L$
- $\forall t, t' \in L, \Delta \vdash t \odot t'$
- $\forall l, t \in L, \Delta \vdash l \odot t$

Dans la relation de compatibilité ci-dessus on peut ignorer les paramètres de types, qui sont traités par R.

Types avec borne supérieure abstraite

La même approche est applicable, mais cette fois-ci il faut étendre la définition de unique.

On introduit d'abord présence potentielle d'un constructeur dans un type abstrait.

$$\Delta \vdash l \in_? t \qquad\qquad l \text{ un constructeur ou un type abstrait}$$

Elle correspond au cas où $l \in U$ pour $\alpha_t :: (L, U, \ldots)$.

On peut alors ajouter

$$(C \vdash \mathsf{unique}(t)) \wedge (\Delta \vdash l \in_? t) \Rightarrow (C \vdash \mathsf{unique}(l))$$

Cela permet de déclancher l'unification sur un constructeur dès qu'un type le contenant potentiellement est présent.

Compilation

```
module type T = sig type t = private [> ] val show : t -> string end
module Mix(X : T)(Y : T) = struct
    type t = [X.t | Y.t]
    let show : t -> string = function
        #X.t as e -> X.show e
      | #Y.t as e -> Y.show e
end
```

On a besoin d'un moyen de savoir si une valeur fait partie de `X.t`.

Idée: on ajoute automatiquement `val mem_#t : [> t] -> bool` à la signature `T`, et on l'utilise pour le filtrage.

La fonction correspondante est automatiquement générée lorsque l'on coerce un module vers cette signature, par exemple lors d'une application de foncteur.

Conclusion

L'union de variants abstraits est utile, mais elle introduit des problèmes complexes.

Elle est aussi intéressante d'un point de vue théorique: elle permet d'obtenir l'expressivité de $\mathsf{F}_{<:}$.

Chaque solution a ses points forts, et il est possible de combiner plusieurs solutions pour améliorer l'expressivité.

Quel est le meilleur choix du point de vue d'un langage de programmation?