Programmation Avancée

Types algébriques généralisés

David Baelde

ENS Paris-Saclay, L3 2020–2021

Échauffement : types fantômes

Idée : un paramètre de type calculatoirement inutile
pour exprimer des contraintes/invariants supplémentaires

Exemple : prévention d'injections de code

Échauffement : types fantômes

Idée : un paramètre de type calculatoirement inutile
pour exprimer des contraintes/invariants supplémentaires

Exemple : prévention d'injections de code

```
type clean
type dirty
type 'a str (* = string *)
val input : string -> dirty str
val sanitize : dirty str -> clean str
val eval : clean str -> result
val length : 'a str -> int
```

Note : cela ne fonctionne pas sans type abstrait !

Arbres rouge et noir

Interdire deux noeuds rouges consécutifs :

```
type red
type black
type ('a,'c) t
val leaf : ('a,black) t
val red :
  'a -> ('a,black) t -> ('a,black) t -> ('a,red) t
val black :
  'a -> ('a,'c1) t -> ('a,'c2) t -> ('a,black) t
```

Arbres rouge et noir

Interdire deux noeuds rouges consécutifs :

```
type red
type black
type ('a,'c) t
val leaf : ('a,black) t
val red :
  'a -> ('a,black) t -> ('a,black) t -> ('a,red) t
val black :
  'a -> ('a,'c1) t -> ('a,'c2) t -> ('a,black) t
```

Critique

- Pas de pattern matching sur un type abstrait.
- La couleur de l'arbre produit dépend du constructeur : inexpressible avec des variants.

Types algébriques

Le mieux qu'on puisse écrire avec un type standard :

```
type ('a,'c) tree =
  | Leaf
  | Red of 'a * ('a,black) tree * ('a,black) tree
  | Black of 'a * ('a,'c) tree * ('a,'c) tree
```

Types algébriques

Le mieux qu'on puisse écrire avec un type standard :

```
type ('a,'c) tree =
  | Leaf
  | Red of 'a * ('a,black) tree * ('a,black) tree
  | Black of 'a * ('a,'c) tree * ('a,'c) tree
```

Limitations

- Tous les constructeurs produisent un `('a,'c)tree`, pas de spécialisation possible.
- Les seules variables de types permises sont `'a` et `'c`.

Types algébriques

Le mieux qu'on puisse écrire avec un type standard :

```
type ('a,'c) tree =
  | Leaf
  | Red of 'a * ('a,black) tree * ('a,black) tree
  | Black of 'a * ('a,'c) tree * ('a,'c) tree
```

Limitations

- Tous les constructeurs produisent un `('a,'c) tree`, pas de spécialisation possible.
- Les seules variables de types permises sont `'a` et `'c`.

Ce qu'on veut, avec des constructeurs vus comme des fonctions :

```
Leaf : ('a,black) tree
Red : 'a * ('a,black) tree * ('a,black) tree
      -> ('a,red) tree
Black : 'a * ('a,'c1) tree * ('a,'c2) tree
      -> ('a,black) tree
```

Types algébriques généralisés

```
type (_,_) tree =
  | Leaf  : ('a,black) tree
  | Black : ('a *
             ('a,'c1) tree *
             ('a,'c2) tree) -> ('a,black) tree
  | Red   : ('a *
             ('a,black) tree *
             ('a,black) tree) -> ('a,red) tree
```

Exemples

- `l = Leaf` est un `('a,black)tree`
- `r = Red (1,l,l)` est un `(int,red)tree`
- `Red (2,r,r)` ne type pas car `r` est rouge

Types algébriques généralisés

```
type z                                          (* Voir fichier redblack4.ml *)
type 'a s
type (_,_,_) tree =
  | Leaf : ('a,black,z) tree
  | Black :
      ('a * ('a,'c1,'h) tree * ('a,'c2,'h) tree) -> ('a,black,'h s) tree
  | Red :
      ('a * ('a,black,'h) tree * ('a,black,'h) tree) -> ('a,red,'h) tree
```

Exemples

- `l = Leaf` et `r = Red (1,l,l)` ont pour hauteur noire `z`
- `b = Black (2,Leaf,r)` est un `(int,black,z s)tree`
- `Black (3,r,b)` ne type pas car `r` et `b` n'ont pas même hauteur
- Aucune contrainte d'ordre n'est exprimée.
  Les mondes du calcul et des types sont encore disjoints.

Pattern matching

Deux nouveaux phénomènes dans les cas d'un pattern matching :

1. Les types existants sont raffinés.
2. De nouveaux types sont introduits.

Exemple :

```
let rec get_black :
 type a c h. (a,c,h) tree -> (a,black,h) tree =
  function
  | Leaf -> Leaf                    (* c = black, h = z *)
  | Black _ as t -> t               (* c = black, h = h' s *)
  | Red (x,a,b) -> get_black a      (* c = red *)
```

Pattern matching

Deux nouveaux phénomènes dans les cas d'un pattern matching :

1. Les types existants sont raffinés.
2. De nouveaux types sont introduits.

Exemple :

```
let rec get_black :
 type a c h. (a,c,h) tree -> (a,black,h) tree =
  function
  | Leaf -> Leaf                    (* c = black, h = z *)
  | Black _ as t -> t               (* c = black, h = h' s *)
  | Red (x,a,b) -> get_black a      (* c = red *)
```

L'exhaustivité dépend du type (concision, optim runtime) :

```
let black_get : type a h. (a,black,h s) tree -> a =
  function Black (x,_,_) -> x       (* Exhaustif! *)
```

Exemple : printf.ml

Chaînes de format simples :

```
type _ f =
  | I : 'a f -> (int->'a) f
  | S : 'a f -> (string->'a) f
  | L : string*'a f -> 'a f
  | N : unit f
```

```
(* Exemple: "%d\n" *)
let exemple : (int -> unit) fmt =
  I (L ("\n", N))
```

Exemple : printf.ml

Chaînes de format simples :

```
type _ f =
  | I : 'a f -> (int->'a) f
  | S : 'a f -> (string->'a) f
  | L : string*'a f -> 'a f
  | N : unit f
```

```
(* Exemple: "%d\n" *)
let exemple : (int -> unit) fmt =
  I (L ("\n", N))
```

Exercice

Coder la fonction d'affichage :

- `val printf : 'a fmt -> 'a.`

Exemple : printf.ml

Chaînes de format simples :

```
type _ f =
  | I : 'a f -> (int->'a) f
  | S : 'a f -> (string->'a) f
  | L : string*'a f -> 'a f
  | N : unit f
```

```
(* Exemple: "%d\n" *)
let exemple : (int -> unit) fmt =
  I (L ("\n", N))
```

Exercice

Coder la fonction d'affichage :

- `val printf : 'a fmt -> 'a.`

Pour aller plus loin :

- Implémenter `val scanf : 'a fmt -> 'a -> unit.`
- chaînes de format avec une opération de concaténation.

Exemple : printf.ml

Chaînes de format simples :

```
type _ f =
  | I : 'a f -> (int->'a) f
  | S : 'a f -> (string->'a) f
  | L : string*'a f -> 'a f
  | N : unit f
```

```
(* Exemple: "%d\n" *)
let exemple : (int -> unit) fmt =
  I (L ("\n", N))
```

Exercice

Coder la fonction d'affichage :

- `val printf : 'a fmt -> 'a.`

Pour aller plus loin :

- Implémenter `val scanf : 'a fmt -> 'a -> unit.`
- chaînes de format avec une opération de concaténation.

Quelques techniques importantes

Type existentiel

On appelle ainsi un type “caché” par un constructeur de GADT.
Exemple : listes hétérogènes.

```
type etree =
  | ETree : ('a,'c,'h) tree -> etree
let l : etree list = [ ETree Leaf ; ETree (Red (1,Leaf,Leaf)) ]
```

Type singleton

On appelle ainsi un type qui n'est habité que par une valeur.
Exemple : `assoc.ml`.

```
type env
type _ sort = String : string sort | Int : int sort
val add : string -> 'a sort -> 'a -> env -> env
val lookup : string -> 'a sort -> env -> 'a
```

Types existentiels

Typage de l'analyse d'un `etree`, où α, β et γ sont des variables de type fraîches :

$$\frac{\Gamma \vdash e : \mathsf{etree} \quad \Gamma, x : (\alpha, \beta, \gamma)\ \mathsf{tree} \vdash e' : \tau'}{\Gamma \vdash \big(\mathsf{match}\ e\ \mathsf{with}\ \mathsf{ETree}\ x \mapsto e'\big) : \tau'}$$

Types existentiels

Typage de l'analyse d'un `etree`, où α, β et γ sont des variables de type fraîches :

$$\frac{\Gamma \vdash e : \mathsf{etree} \quad \Gamma, x : (\alpha, \beta, \gamma)\ \mathsf{tree} \vdash e' : \tau'}{\Gamma \vdash \big(\mathsf{match}\ e\ \mathsf{with}\ \mathsf{ETree}\ x \mapsto e'\big) : \tau'}$$

Variante pour un type existentiel tout nu, où α est fraîche :

$$\frac{\Gamma \vdash e : \exists \alpha.\tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \big(\mathsf{match}\ e\ \mathsf{with}\ x \mapsto e'\big) : \tau'}$$

Types existentiels

Typage de l'analyse d'un `etree`, où α, β et γ sont des variables de type fraîches :

$$\frac{\Gamma \vdash e : \mathsf{etree} \quad \Gamma, x : (\alpha, \beta, \gamma)\ \mathsf{tree} \vdash e' : \tau'}{\Gamma \vdash \big(\mathsf{match}\ e\ \mathsf{with}\ \mathsf{ETree}\ x \mapsto e'\big) : \tau'}$$

Variante pour un type existentiel tout nu, où α est fraîche :

$$\frac{\Gamma \vdash e : \exists \alpha.\tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \big(\mathsf{match}\ e\ \mathsf{with}\ x \mapsto e'\big) : \tau'}$$

Une autre utilisation des types existentiels en programmation :
les types abstraits dans les modules.

Typage des GADTs

En pratique avec OCaml

On ne s'en sort pas sans écrire des annotations de typage,
mais en général il suffit de spécifier le type des fonctions.

Typage des GADTs

En pratique avec OCaml

On ne s'en sort pas sans écrire des annotations de typage,
mais en général il suffit de spécifier le type des fonctions.

En théorie

Aucune difficulté pour vérifier les types. Inférer est délicat.
Quel type donner à `f` dans l'exemple suivant ?

```
type _ t = I : int t
let f = function I -> 3
```

Typage des GADTs

En pratique avec OCaml

On ne s'en sort pas sans écrire des annotations de typage,
mais en général il suffit de spécifier le type des fonctions.

En théorie

Aucune difficulté pour vérifier les types. Inférer est délicat.
Quel type donner à `f` dans l'exemple suivant ?

```
type _ t = I : int t
let f = function I -> 3
```

On a le choix, mais pas de type principal :

- `int t -> int`
- `'a t -> 'a`
- `'a t -> int`

Contraintes de types

Inférerence de types = génération puis résolution de contraintes de type.

ML standard

- Les contraintes sont des équations.
- Résolution par unification.
- Propriété de mgu $\Rightarrow$ principalité.

GADTs

- Contraintes de la forme "equation $\Rightarrow$ equation".
- Exemple précédent : $X = \texttt{int} \Rightarrow Y = \texttt{int}$ a deux solutions : $Y = X$ et $Y = \texttt{int}$.
- En général, on peut perdre la décidabilité du typage...
  mais pas avec les schémas de types à la ML.

Conclusion sur les GADTs

Ce qu'il faut retenir

- Des constructeurs avec des types arbitraires. Annotations de type nécessaires.
- Raffinement de types par pattern-matching.
- Types existentiels quand il faut cacher de l'information.
- Types singletons quand il faut calculer sur les types

Conclusion sur les GADTs

Ce qu'il faut retenir

- Des constructeurs avec des types arbitraires. Annotations de type nécessaires.
- Raffinement de types par pattern-matching.
- Types existentiels quand il faut cacher de l'information.
- Types singletons quand il faut calculer sur les types

Quand les utiliser

Le typage fort peut guider le développement de vos programmes, éviter des bugs.

- ⊕ Typed parsers, e.g. menhir. Tezos smart contract evaluation.
- ⊕ Statically typed lambda-calculus interpreter.
- ⊖ Most logical systems do not use the technique.

Conclusion sur les GADTs

Ce qu'il faut retenir

- Des constructeurs avec des types arbitraires. Annotations de type nécessaires.
- Raffinement de types par pattern-matching.
- Types existentiels quand il faut cacher de l'information.
- Types singletons quand il faut calculer sur les types

Quand les utiliser

Le typage fort peut guider le développement de vos programmes, éviter des bugs.

- ⊕ Typed parsers, e.g. menhir. Tezos smart contract evaluation.
- ⊕ Statically typed lambda-calculus interpreter.
- ⊖ Most logical systems do not use the technique.

Les GADTs peuvent être utilisés pour écrire des preuves qui sont des programmes...
mais pour ça, on se tourne généralement vers des concepts plus expressifs.

Ouverture : preuve et vérification

Types dépendants (Coq, Adga, etc.)

Types dépendent de valeurs, e.g. `list A 3` est le type des listes de 3 éléments de type `A`.
Exemples : $\forall x : \texttt{nat}.\ A \to \texttt{list A x}$ mais aussi $\forall x, y : \texttt{nat}.\ \texttt{eq (plus } x\ y\texttt{) (plus } y\ x\texttt{)}$.

- Les preuves sont des programmes purs : système formel autonome.
- La vérification de type implique d'exécuter les programmes.

Types raffinés (Liquid Haskell, Dsolve, etc.)

On contraint les valeurs d'un type par un énoncé logique, e.g. $x : \text{int}\{2 < x\}$.

- Relation de sous-typage basée sur la conséquence logique : vérification externe.
- L'inférence de type peut rester décidable si on restreint le langage de raffinement.

Mélange de genres

Dans un langage avec types dépendants, on peut définir la monade de Hoare `ST P A Q`.
Elle décrit les calculs qui,

- si ils sont exécutés sur un état initial s_1 tel que $P\ s_1$,
- renverront une valeur $v : A$ avec un état final s_2,
- tel que $Q\ v\ s_1\ s_2$.

Mélange de genres

Dans un langage avec types dépendants, on peut définir la monade de Hoare `ST P A Q`.
Elle décrit les calculs qui,

- si ils sont exécutés sur un état initial s_1 tel que $P\ s_1$,
- renverront une valeur $v : A$ avec un état final s_2,
- tel que $Q\ v\ s_1\ s_2$.

On peut donc appliquer la traduction monadique à un programme impératif (d'ordre supérieur !) pour obtenir un terme pur très fortement typé, dont le type contient des pré/post-conditions correctes par construction.
Il restera à vérifier que ces conditions sont compatibles avec la spécification attendue : cela peut typiquement être déchargé à un vérificateur externe (typiquement un SMT solver).

Mélange de genres

Dans un langage avec types dépendants, on peut définir la monade de Hoare `ST P A Q`.
Elle décrit les calculs qui,

- si ils sont exécutés sur un état initial s_1 tel que $P\ s_1$,
- renverront une valeur $v : A$ avec un état final s_2,
- tel que $Q\ v\ s_1\ s_2$.

On peut donc appliquer la traduction monadique à un programme impératif (d'ordre supérieur !) pour obtenir un terme pur très fortement typé, dont le type contient des pré/post-conditions correctes par construction.
Il restera à vérifier que ces conditions sont compatibles avec la spécification attendue : cela peut typiquement être déchargé à un vérificateur externe (typiquement un SMT solver).

Quelques détails dans `hoare.v`.

Un langage construit sur de telles idées : F$\star$ [tutoriel en ligne].