Singleton types et parametricity

Singleton types

- Les types dépendants des pauvres

Parametricity

- Théorèmes gratuits!

Singleton types partout $\sim$ parametricity partout

Problèmes avec les types dépendants

Incompatibles avec:

- Mutation: incohérence
- Exceptions
- Récursion: vérification de types indécidable
- Compilation: chaque phase de compilation est un problème ouvert
- ...

Attrayant pour un assistant à la preuve, mais:

Ça pue pour un langage de *programmation*

GADTs: types “dépendants” sans arrière goût

On remplace:

```
type Nat : Type | zero : Nat | succ : Nat -> Nat;
type NList (a : Type) (l : Nat)
  | nnil : NList a zero
  | ncons: a -> NList a l -> NList a (succ 1);
```

par

```
Zero : Type;
Succ : Type -> Type;
type NList a l
  | nnil : NList a Zero
  | ncons: a -> NList a l -> NList a (Succ 1);
```

GADTs

Préserve la “*phase distinction*”

Compatible avec les effets de bord, la compilation, ...

Disponible en Haskell, OCaml, Scala, ...

Nécessite une inférence/vérification de types plus sophistiquée

Se réduit fondamentalement aux preuves d’égalité d’Ωmega

La saveur des types dépendants, mais sans la dépendance

Typage des GADTs

Le problème du typage des GADTs:

Le raffinement des types dans les branches

```
case (ys : NList a l)
  | nnil => nnil
  | ncons x xs => ncons x xs
```

Il faut se souvenir que $l = \mathit{Zero}$ dans la première branche

Tenir compte des ces égalités lors de vérification d'égalité d'autres types

Plusieurs typages possibles $\Rightarrow$ annotations pour aider l'inférence

Le type de make-nlist?

Avec typage dépendent:

```
make-nlist : a -> (len : Nat)
             -> NList a len;
```

Avec GADTs:

```
make-nlist : a -> (len : ?Nat?)
             -> NList a ?len?;
```

Nat n'existe plus (ou du moins, pas comme type)

len n'existe pas dans le monde des types

Les types singleton à la rescousse

type singleton: un type qui n'a qu'une seule valeur possible

Par exemple *Snat* n, le type des nombres qui valent n:

```
make-nlist : ∀ len. a -> Snat len
                -> NList a len;
```

On a maintenant accès à *len* dans les 2 mondes: types et valeurs

Mais, toujours pas de types dépendents!

Mais il faut dupliquer certaines choses:

```
(+) : ∀n1, n2. Snat n1 -> Snat n2
        -> Snat (plus n1 n2);
```

Types singleton partout

Les types singleton sont *équivalents* aux types dépendants

Paramétricité

$\forall t.t \to t$ est un singleton type

De même, il n'y a pas de valeur de type $\forall t.\mathit{Nat} \to t$

Ceci peut se déduire donc directement du type

La *paramétricité* est une propriété générale à partir de laquelle on peut prouver ces autres propriétés

Exemples

$\mathit{head} : \forall t.[t] \to t$

$f \circ \mathit{head} \simeq \mathit{head} \circ \mathit{map}\ f$

$(\mathit{++}) : \forall t.[t] \to [t] \to [t]$

$\mathit{map}\ f\ (x\ \mathit{++}\ y) \simeq \mathit{map}\ f\ x\ \mathit{++}\ \mathit{map}\ f\ y$

$K : \forall t_1, t_2.t_1 \to t_2 \to t_1$

$f\ (K\ x\ y) \simeq K\ (f\ x)\ (g\ y)$

$\mathit{zip} : \forall t_1, t_2.([t_1], [t_2])$

$\to [(t_1, t_2)]$

$\mathit{zip}\ (\mathit{map}\ f\ x, \mathit{map}\ g\ y)$

$\simeq \mathit{map}\ (f \times g)\ \mathit{zip}\ (x, y)$

Modèles

Un *modèle* d'un langage est un mapping vers un formalisme standard

Exemple:

$$\Gamma \vdash e : \tau \quad \Rightarrow \quad [\![e]\!]_\Gamma \in [\![\tau]\!]_\Gamma$$

$$[\![n]\!]_\Gamma \quad \Rightarrow \quad n$$

$$[\![\texttt{Int}]\!]_\Gamma \quad \Rightarrow \quad \mathbb{Z}$$

Modèle pour System F

Pour la paramétricité, on défini une sorte de modèle

Les types sont traduits en *relations binaires*

$$e : \tau \quad \Rightarrow \quad e \, [\![\tau]\!] \, e$$

Pour un type de base τ, la relation est la relation identité

$$[\![Bool]\!] : Bool \Leftrightarrow Bool = I_{Bool}$$

Si $[\![\tau]\!] = \mathcal{R} : A \Leftrightarrow A'$

alors $[\![[\tau]]\!] = \mathcal{R}' : [A] \Leftrightarrow [A']$

tel que $[x_1, ..., x_n] \, \mathcal{R}' \, [y_1, ..., y_n]$ iff $\forall i \in \{1..n\}, x_i \, \mathcal{R} \, y_i$

Modèle de fonctions

Si $[\![\tau_1]\!] : A \Leftrightarrow A'$

et $[\![\tau_2]\!] : B \Leftrightarrow B'$

alors $[\![\tau_1 \to \tau_2]\!] : A \to B \Leftrightarrow A' \to B'$

tel que $f\ [\![\tau_1 \to \tau_2]\!]\ f'$ iff $\forall x, x'.\ x\ [\![\tau_1]\!]\ x' \Rightarrow f\ x\ [\![\tau_2]\!]\ f'\ x'$

Functions are related if they take

related input arguments to related output results

Modèle de polymorphisme

$[\![\forall t.F\ t]\!] : (\forall t.F\ t) \Leftrightarrow (\forall t'.F'\ t')$

soit $\mathcal{F}$ une fonction de relation à relation telle que pour toute relation $\mathcal{R} : A \Leftrightarrow A'$ il y a une relation $\mathcal{F}\,\mathcal{R} : F\ A \Leftrightarrow F'\,A'$.

alors $f\ [\![\forall t.F\ t]\!]\ f'$ iff $\forall A, A', \mathcal{R} : A \Leftrightarrow A'.\ \ f\,[A]\ (\mathcal{F}\,\mathcal{R})\ f'\,[A']$

Polymorphic functions are related if they take

related input types to related output values

Théorème de paramétricité

Le théorème de paramétricité, dit simplement:

$$\text{Si } \bullet \vdash e : \tau \text{ alors } e \; [\![\tau]\!] \; e$$

Cela revient à dire que le modèle est valide

Usage

Prenons notre premier exemple $f : \forall t.\, t \to t$

Le théorème nous dit

$$\forall x_1, x_2, \mathcal{R}. \quad x_1 \,\mathcal{R}\, x_2 \quad \Rightarrow \quad (f\ x_1)\,\mathcal{R}\,(f\ x_2);$$

On peut alors l'instancier avec:

$$x_1 = x, x_2 = x, \mathcal{R} = \{(x, x)\}$$

Ce qui nous donne:

$$(f\ x)\,\mathcal{R}\,(f\ x) \quad \Rightarrow \quad (f\ x, f\ x) \in \{(x, x)\} \quad \Rightarrow \quad f\ x = x$$

Avec Curry-Howard

alors $[\![\tau_1 \to \tau_2]\!] : A \to B \Leftrightarrow A' \to B'$

tel que $f\ [\![\tau_1 \to \tau_2]\!]\ f'$ iff $\forall x, x'.\ x\ [\![\tau_1]\!]\ x' \Rightarrow f\ x\ [\![\tau_2]\!]\ f'\ x'$

Devient:

$$f\ [\![\tau_1 \to \tau_2]\!]\ f' = (x : A) \to (x' : A') \to (P : x\ [\![\tau_1]\!]\ x') \to (f\ x\ [\![\tau_2]\!]\ f'\ x')$$

A.K.A

$$[\![\tau_1 \to \tau_2]\!] = \lambda f \to \lambda f' \to (x : A) \to (x' : A') \to (P : x\ [\![\tau_1]\!]\ x') \to (f\ x\ [\![\tau_2]\!]\ f'\ x')$$