Extensions de PTS

Entiers, floats, ...

Tuples, Structures, Records, ...

Algebraic data types

Types récursifs

Types existentiels

Types récursifs

Généralement dénotés:

least fixed point: $\mu f = \mu x.fx \simeq f(f(f(f(...(f(\bot))))))$

greatest fixed point: $\nu f = \nu x.fx \simeq f(f(f(f(...(f(\top))))))$

$$\texttt{List}\ \alpha = \mu\,(\lambda l \to \texttt{Unit} + (\alpha \times l))$$

$$\texttt{Stream}\ \alpha = \nu\,(\lambda l \to (\alpha \times l))$$

μ ne marche pas pour `Stream`!

μ ajoute jusqu'à obtenir un point fixe

ν enlève jusqu'à obtenir un point fixe

Égalité des types récursifs

```
type List1 α | nil | cons α (List1 α);
type List2 α | nil | cons α (List1 α);
type List3 α | nil | cons α (List4 α);
type List4 α | nil | cons α (List3 α);
```

Ces types sont-ils tous égaux?

Iso- vs Équi- récursion

Deux sémantiques possibles:

equirecursive types: $\mu f = f(\mu f)$

isorecursive types:

$$\frac{\Gamma \vdash e : \mu f}{\Gamma \vdash \texttt{unroll}\, e : f(\mu f)}$$

$$\frac{\Gamma \vdash e : f(\mu f)}{\Gamma \vdash \texttt{roll}\, e : \mu f}$$

`roll`/`unroll` ne peuvent convertir `List1` à `List3`

`roll`/`unroll` facilitent grandement la vérification des types

`roll`/`unroll` sont habituellement cachés

Récursion "impropre"

$$\texttt{type Tree } \alpha \texttt{ | leaf } \alpha \texttt{ | node (Tree } (\alpha \times \alpha)\texttt{)}$$

$$\texttt{Tree} = \mu\,(\lambda(t : \texttt{Type} \to \texttt{Type}) \to \lambda(\alpha : \texttt{Type}) \to \alpha + t\,(\alpha \times \alpha))$$

Complique encore plus l'implantation des types équirécursifs

$$\frac{\Gamma \vdash e : (\mu\, f)\, \tau_1 \ldots \tau_n}{\Gamma \vdash \texttt{unroll}\, e : f\, (\mu\, f)\, \tau_1 \ldots \tau_n}$$

$$\frac{\Gamma \vdash e : f\, (\mu\, f)\, \tau_1 \ldots \tau_n}{\Gamma \vdash \texttt{roll}\, e : (\mu\, f)\, \tau_1 \ldots \tau_n}$$

Quantification existentielle

Utile quand un type ne sera connu qu'à l'exécution:

$$\mathit{filter} : \mathit{NList}\ \alpha\ n \to \exists n'.\mathit{NList}\ \alpha\ n;$$

Essayons d'encoder la quantification universelle: $\exists t.\tau \quad \simeq \quad \neg\forall t.\neg t$

$$\exists t : \kappa.\tau \quad \simeq \quad ((t : \kappa) \to t \to \mathit{False}) \to \mathit{False}$$

Utilisable pour des preuves

Renvoyer toujours *False* est problématique dans des programmes

$\Rightarrow$ Généralisé à n'importe quel type

Types existentiels

$$\exists t{:}\kappa.\tau \quad \simeq \quad (\alpha{:}\mathtt{Type}) \to ((t{:}\kappa) \to \tau \to \alpha) \to \alpha$$

Introduction (parfois dénoté $\langle t = \tau, e : f \rangle$)

$$\frac{\Gamma \vdash e : f\ \tau}{\Gamma \vdash \mathtt{pack}\ \tau\ e : \exists t{:}\kappa.f\ t}$$

Élimination (parfois dénoté $\mathtt{open}\ e_1\ \ldots\ \mathtt{in}\ e_2$)

$$\frac{\Gamma \vdash e_1 : \exists t{:}\kappa.\tau_1 \quad \Gamma, t{:}\kappa, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let\ pack}\ t\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

Types existentiels (exemples)

Une liste de taille non-spécifiée:

$$\texttt{List}\,\alpha \;\simeq\; \exists n{:}\texttt{Nat}.\texttt{NList}\,\alpha\,n$$

Une représentation de fermeture:

$$\tau_1 \to \tau_2 \;\simeq\; \exists \mathit{ctx} : \texttt{Type}.\,\texttt{Pair}\,((\tau_1, \mathit{ctx}) \to \tau_2)\,\mathit{ctx}$$

Où l'appelle à une telle fermeture f devient:

```
let pack t x = f in
let code = x.1          // code : (τ1, t) → τ2
let env  = x.2          // env  : t
code (arg, env)
```

Existentiels = paires dépendantes

$$\exists t : \kappa . \tau \quad \simeq \quad (\alpha : \mathtt{Type}) \to ((t : \kappa) \to \tau \to \alpha) \to \alpha$$

Rappelons l'encodage de Church des paires:

$$\mathit{Pair}\ \tau_1\ \tau_2 \quad \simeq \quad (\alpha : \mathtt{Type}) \to (\tau_1 \to \tau_2 \to \alpha) \to \alpha$$

Différences:

- Premier champ vient d'une autre *sorte*
- Type du deuxième champ peut dépendre de la valeur du premier

Généralisation

Existentiels deviennent un cas particulier:

```
type Exists k f
  | pack (t : k)
         (f t);

type Closure t₁ t₂
  | closure (ctx : Type)
            ((t₁ × ctx) → t₂)
            ctx;
```