Polymorphisme en λ-calcul

Le λ-calcul typé avec polymorphisme s'appelle *System-F*

Cœur des langages OCaml/Haskell (en théorie et en pratique)

$$\textit{(types)} \quad \tau ::= \texttt{Int} \mid \tau_1 \to \tau_2 \mid t \mid \forall t.\tau$$

$$\textit{(exps)} \quad e ::= c \mid x \mid \lambda x\!:\!\tau \to e \mid e_1\, e_2 \mid \Lambda t \to e \mid e[\tau]$$

La fonction identité est en fait traduite comme suit:

$$\textit{id} :: \forall\alpha.\alpha \to \alpha$$

$$\textit{id} = \Lambda\alpha \to \lambda x \to x$$

$$\textit{réponse} = \textit{id}[\texttt{Int}]\, 42$$

Inventé en 1972/1974 par Girard/Reynolds

Sémantique statique de System-F

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \to e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda t \to e : \forall t.\tau} \qquad \frac{\Gamma \vdash e : \forall t.\tau_1}{\Gamma \vdash e[\tau_2] : \tau_1[\tau_2/t]}$$

Isomorphisme de Curry-Howard

Intime connection entre logique et langages de programmation

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad \text{vs} \quad \frac{P_1 \Rightarrow P_2 \qquad P_1}{P_2}$$

La règle de typage de l'application correspond au *modus ponens*

La β-réduction correspond au *cut-elimination*

Type = Proposition; Programme = Preuve

$$\frac{\Gamma \vdash e : \forall t.\tau_2}{\Gamma \vdash e[\tau_1] : \tau_2[\tau_1/t]} \quad \text{vs} \quad \frac{\forall x.P}{P[e/x]}$$

Il n'existe pas de fonction de type $\forall t_1, t_2.t_1 \to t_2$

System-F comme logique

$$(A \wedge B) \Rightarrow A$$

$$((A \wedge B) \Rightarrow C) \Leftrightarrow (A \Rightarrow (B \Rightarrow C))$$

$$A \Rightarrow (\neg(\neg A))$$

Curry-style System-F

$$\textit{(types)} \quad \tau ::= \texttt{Int} \mid \tau_1 \to \tau_2 \mid t \mid \forall t.\tau$$

$$\textit{(exps)} \quad e ::= c \mid x \mid \lambda x \to e \mid e_1\ e_2$$

Système *indécidable*!

$$\Gamma, x:\tau \vdash x : \tau \qquad \frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x \to e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \forall t.\tau} \qquad \frac{\Gamma \vdash e : \forall t.\tau_1}{\Gamma \vdash e : \tau_1[\tau_2/t]}$$

Terminaison

¡STLC et System-F sont *fortement normalisant*!

$\Rightarrow$ tous les programmes terminent!

$\Rightarrow$ Le combinateur Y ne peut pas être défini

Pas forcément désirable pour un langage de programmation

Important pour une logique:

$$\mathit{Explosion} = \forall\alpha.\alpha$$

$$\mathit{boom} = \Lambda\alpha.\mathit{boom}[\alpha]$$

L'auto-application est cependant possible: $\mathit{id}[\forall\alpha.\alpha \to \alpha]\mathit{id}$

Encodage imprédicatif de Church

Church numerals: $z = \lambda f\ x \to x \qquad s\ n = \lambda f\ x \to f\ (n\ f\ x)$

System-F type: *Nat* $= \forall \alpha.(\alpha \to \alpha) \to \alpha \to \alpha$

Se généralize à presque n'importe quel type algébrique:

$$
\begin{aligned}
\textit{Unit} &= \forall \alpha.\alpha \to \alpha \\
\textit{Bool} &= \forall \alpha.\alpha \to \alpha \to \alpha \\
\textit{Void} &= \forall \alpha.\alpha \\
\textit{Box}\ t &= \forall \alpha.(t \to \alpha) \to \alpha \\
\textit{Pair}\ t_1\ t_2 &= \forall \alpha.(t_1 \to t_2 \to \alpha) \to \alpha \\
\textit{Maybe}\ t &= \forall \alpha.(t \to \alpha) \to \alpha \to \alpha \\
\textit{Either}\ t_1\ t_2 &= \forall \alpha.(t_1 \to \alpha) \to (t_2 \to \alpha) \to \alpha
\end{aligned}
$$

Encodage imprédicatif de Church (simple)

```
%% type Bool = true | false
Bool  = ∀α. α → α → α;
true  = Λα → λt:α → λe:α → t;
false = Λα → λt:α → λe:α → e;
if c t e = c t e;

%% type Pair t1 t2 = pair t1 t2
Pair t1 t2 = ∀α. (t1 → t2 → α) → α;
pair = Λt1 → λx:t1 → Λt2 → λy:t2 →
       Λα → λf:(t1→t2→α) → f x y;
fst x = x[t1] (λx:t1 → λy:t2 → x);
snd x = x[t2] (λx:t1 → λy:t2 → y);
```

Encodage imprédicatif de Church (inductif)

```
%% type Nat = zero | succ Nat
Nat  = ∀α. α → (α → α) → α;
zero = Λα → λx:α → λf:(α→α) → x;
succ = λn:Nat →
       Λα → λx:α → λf:(α→α) → f (n[α] x f);

%% type List β = nil | cons β (List β)
List β = ∀α. α → (β → α → α) → α;
nil    = Λβ →
         Λα → λx:α → λf:(β→α→α) → x;
cons   = Λβ → λh:β → λt:List β →
         Λα → λx:α → λf:(β→α→α) → f h (t[α] x f);
```

Types inductifs

l'encodage de Church ne fonctionne pas pour les types non-inductifs

```
%% type U = ubase | ucons (U → U)
U     = ∀α. α → ((α → α) → α) → α;
ubase = Λα → λx:α → λf:((α→α)→α) → x;
ucons = λp:U→U →
        Λα → λx:α → λf:((α→α)→α) → f ?¿?;
```

La première occurrence récursive de U est *négative*!

C'est aussi une condition nécessaire pour la terminaison:

```
f u = case u | ubase ⇒ u
             | ucons f' ⇒ f' u;
infloop = f (ucons f);
```

Des fonctions dans les types

System F nous permet de définir $\texttt{List}\ \beta$ mais pas juste $\texttt{List}$

E.g. dans `cons`:

$$\texttt{cons} = \Lambda\beta \rightarrow \lambda\texttt{h}{:}\beta \rightarrow \lambda\texttt{t}{:}\texttt{List}\ \beta \rightarrow \Lambda\alpha \rightarrow \lambda\texttt{x}{:}\alpha \rightarrow \lambda\texttt{f}{:}(\beta{\rightarrow}\alpha{\rightarrow}\alpha) \rightarrow \texttt{f h (t}[\alpha]\ \texttt{x f)}$$

Il faut en réalité écrire:

$$\texttt{cons} = \Lambda\beta \rightarrow \lambda\texttt{h}{:}\beta \rightarrow \lambda\texttt{t}{:}(\forall\alpha.\alpha{\rightarrow}(\beta{\rightarrow}\alpha{\rightarrow}\alpha){\rightarrow}\alpha) \rightarrow \Lambda\alpha \rightarrow \lambda\texttt{x}{:}\alpha \rightarrow \lambda\texttt{f}{:}(\beta{\rightarrow}\alpha{\rightarrow}\alpha) \rightarrow \texttt{f h (t}[\alpha]\ \texttt{x f)}$$

$\texttt{List}\ \beta$ est en fait un “appel de fonction”

- $\texttt{List}$ est une fonction qui renvoie un type
- β est son argument (lui même un type)

System-F_ω

System-F_ω est une extension des types de System-F

Cœur des langages OCaml/Haskell (en théorie et en pratique)

$$\tau ::= \texttt{Int} \mid \tau_1 \to \tau_2 \mid t \mid \forall t.\tau \mid \lambda t \to \tau \mid \tau_1\ \tau_2$$

Certains types n'ont pas de sens, e.g. $\texttt{Int}\ \texttt{Int}$

$\texttt{List}$ a du sens, mais n'est pas un *proper type*

- Une valeur peut avoir type $\texttt{List}\ \beta$ mais pas juste $\texttt{List}$

Besoin de classifier les differentes *sortes de types*

Kinds*: types de types*

Les *types* classifient les *valeurs*

Les *kinds* classifient les *types*

(kinds) $\kappa ::= \texttt{Type} \mid \kappa_1 \to \kappa_2$

(types) $\tau ::= \texttt{Int} \mid \tau_1 \to \tau_2 \mid t \mid \forall t{:}\kappa.\tau \mid \lambda t{:}\kappa \to \tau \mid \tau_1\ \tau_2$

Et ainsi:

$$\texttt{Int} : \texttt{Type}$$

$$\texttt{List} : \texttt{Type} \to \texttt{Type}$$

$\texttt{Type}$ est le *kind* des *proper types*

Les types de System-F_ω forment une sorte de STLC!

Notations alternatives

Notation “traditionnelle”: $\texttt{List} :: * \to *$

- $*$ (ou parfois ω) pour le *kind* des *proper types*
- $::$ pour le *kind* d’un type: $\tau :: \kappa$

On peut aussi éliminer la syntaxe spéciale $\tau_1 \to \tau_2$:

$$(\to) :: * \to * \to *$$

$$\tau_1 \to \tau_2 \Rightarrow (\to)\, \tau_1\, \tau_2$$

On *presque* faire de même pour $\forall t : \kappa . \tau$ (avec HOAS):

$$\forall t : \kappa . \tau \Rightarrow \texttt{All}\, (\lambda t : \kappa \to \tau)$$

Règles de typage de System-F_ω

(kind ctxs) $\Delta ::= \bullet \mid \Delta, t : \kappa$

$$\Delta; \Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Delta \vdash \tau_1 : \mathtt{Type} \qquad \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1 \to e : \tau_1 \to \tau_2}$$

$$\frac{\Delta, t : \kappa; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda t : \kappa \to e : \forall t : \kappa . \tau}$$

$$\frac{\Delta; \Gamma \vdash e : \forall t : \kappa . \tau_1 \qquad \Delta \vdash \tau_2 : \kappa}{\Delta; \Gamma \vdash e[\tau_2] : \tau_1[\tau_2/t]}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \qquad \tau_1 \simeq \tau_2}{\Delta; \Gamma \vdash e : \tau_2}$$

Changements importants

Nouveaux jugements:

- Le jugement principal $\Gamma \vdash e : \tau$ est devenu $\Delta; \Gamma \vdash e : \tau$
- Nouveau jugement $\Delta \vdash \tau : \kappa$ qui vérifie le *kind* d'un *type*
- Nouveau jugement $\tau_1 \simeq \tau_2$ qui défini l'égalité des types

Nouvelle règle non-syntaxique: la règle de *conversion*

- Nécessaire pour que $\texttt{Int} = (\lambda t{:}\texttt{Type} \to t)\ \texttt{Int}$

En fait, il faut aussi vérifier que Γ est valide!

- Nouveau jugement $\Delta \vdash \Gamma$ qui vérifie les types de Γ

Règles de kinding *de System-F_ω*

$$\frac{}{\Delta \vdash \bullet} \qquad \frac{\Delta \vdash \Gamma \qquad \Delta \vdash \tau : \texttt{Type}}{\Delta \vdash \Gamma, x : \tau}$$

$$\frac{\Delta(t) = \kappa}{\Delta \vdash t : \kappa} \qquad \frac{\Delta \vdash \tau_1 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1\ \tau_2 : \kappa_2}$$

$$\frac{\Delta, t{:}\kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda t{:}\kappa_1 \to \tau : \kappa_1 \to \kappa_2} \qquad \frac{}{\Delta \vdash \texttt{Int} : \texttt{Type}}$$

$$\frac{\Delta \vdash \tau_1 : \texttt{Type} \qquad \Delta \vdash \tau_1 : \texttt{Type}}{\Delta \vdash \tau_1 \to \tau_2 : \texttt{Type}} \qquad \frac{\Delta, t{:}\kappa \vdash \tau : \texttt{Type}}{\Delta \vdash \forall t{:}\kappa.\tau : \texttt{Type}}$$

Règles de conversion de System-F_ω

$$\frac{}{\tau \simeq \tau} \qquad \frac{\tau' \simeq \tau}{\tau \simeq \tau'} \qquad \frac{}{(\lambda t\!:\!\kappa \to \tau_1)\tau_2 \simeq \tau_1[\tau_2/t]}$$

$$\frac{\tau_1 \simeq \tau_1' \qquad \tau_2 \simeq \tau_2'}{\tau_1\ \tau_2 \simeq \tau_1'\ \tau_2'} \qquad \frac{\tau \simeq \tau'}{\lambda t\!:\!\kappa \to \tau \simeq \lambda t\!:\!\kappa \to \tau'}$$

$$\frac{\tau \simeq \tau'}{\forall t\!:\!\kappa.\tau \simeq \forall t\!:\!\kappa.\tau'} \qquad \frac{\tau_1 \simeq \tau_1' \qquad \tau_2 \simeq \tau_2'}{\tau_1 \to \tau_2 \simeq \tau_1' \to \tau_2'}$$

Similaire à $e \rightsquigarrow e'$, sauf: *symétrique* et autorise la *réduction sous le* λ

Implémenté par *normalisation*: $$\frac{\tau_1 \rightsquigarrow^* \tau_1' \quad \tau_2 \rightsquigarrow^* \tau_2' \quad \tau_1' = \tau_2'}{\tau_1 \simeq \tau_2}$$

Pleins de λ partout

System-F$_\omega$ inclut 3 sortes de λ:

- $\lambda x : \tau \to e$: fonctions de valeurs à valeurs
- $\lambda t : \kappa \to \tau$: fonctions de types à types
- $\Lambda t : \kappa \to e$: fonctions de types à valeurs

Chacun vient avec une règle de typage un peu différente

Chacun vient avec une application

Chacun vient avec son type (pas toujours exactement le même)

¡Il serait bon d'unifier tout ça!

Unifier les λ

Syntaxe: remplacer $\Lambda t : \kappa \to e$ par $\lambda t : \kappa \to e$ et $e[\tau]$ par $e\ \tau$

Types: remplacer $\forall t : \kappa . \tau$ par $(t : \kappa) \to \tau$ (aka $\Pi t : \kappa . \tau$)

et décréter que $\tau_1 \to \tau_2$ est un abbréviation de $(x : \tau_1) \to \tau_2$

Règles?

$$\frac{\Delta ; \Gamma \vdash e_1 : (x : \tau_1) \to \tau_2 \qquad \Delta ; \Gamma \vdash e_2 : \tau_1}{\Delta ; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Delta ; \Gamma \vdash e : (t : \kappa) \to \tau_1 \qquad \Delta \vdash \tau_2 : \kappa}{\Delta ; \Gamma \vdash e\ \tau_2 : \tau_1[\tau_2/t]}$$

$$\frac{\Delta \vdash \tau_1 : (t : \kappa_1) \to \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1\ \tau_2 : \kappa_2}$$

Pure Type Systems (PTS)

Une seule catégorie syntaxique partagée pour les valeurs, types, ...:

$$\textit{(exps)} \quad e, \tau \ ::= \ c \mid x \mid \lambda x : \tau \to e \mid e_1 \ e_2 \mid (x : \tau_1) \to \tau_2$$

Les *sortes* classifient les expressions: valeurs, types, ...

Un PTS est défini par un triplet $(\mathcal{S}, \mathcal{A}, \mathcal{R})$:

- $\mathcal{S}$: ensemble des *sortes*, e.g. $\{*, \Box\}$ $*$=valeurs, $\Box$=types
- $\mathcal{A}$: axiomes, e.g. $\{* : \Box, \texttt{Int} : *, 0 : \texttt{Int}, ...\}$
- $\mathcal{R}$: règles de construction des λ

  (s_1, s_2, s_3) signifie un λ qui va de s_1 à s_2 et vit dans s_3

System-F$_\omega$ = $(\{*, \Box\}, \{* : \Box\}, \{(*, *, *), (\Box, *, *), (\Box, \Box, \Box)\})$

Règles de typage des PTS

$$\frac{\mathcal{A}(c) = \tau}{\Gamma \vdash c : \tau} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash (x{:}\tau_1) \to \tau_2 : s}{\Gamma \vdash \lambda x{:}\tau_1 \to e : (x{:}\tau_1) \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : (x{:}\tau_1) \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2[e_2/x]}$$

$$\frac{\Gamma \vdash \tau_1 : s_1 \qquad \Gamma, x{:}\tau_1 \vdash \tau_2 : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash (x{:}\tau_1) \to \tau_2 : s_3}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \simeq \tau_2}{\Gamma \vdash e : \tau_2}$$

Règles moins “sloppy” des PTS

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau : s}{\vdash \Gamma, x:\tau}$$

$$\frac{\mathcal{A}(c) = \tau}{\bullet \vdash c : \tau} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash \tau : s}{\Gamma, x:\tau \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 : s}{\Gamma, x:\tau_1 \vdash e : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \tau_2 : s \qquad \tau_1 \simeq \tau_2}{\Gamma \vdash e : \tau_2}$$

S’assurer que si on peut prouver $\Gamma \vdash e : \tau$, alors on a aussi:

- $\tau = s \vee \Gamma \vdash \tau : s$: pour exclure $\Gamma \vdash 3 : (\lambda x:5 \to \texttt{Int}) \; \square$
- $\vdash \Gamma$: exclure par exemple $\bullet, x:3 \vdash x : 3$

System-F_ω comme PTS

System-F_ω = $(\{*, \Box\}, \{* : \Box\}, \{(*,*,*), (\Box,*,*), (\Box,\Box,\Box)\})$

La syntaxe des PTS permet plus de liberté:

- Une variable “normale” pourrait apparaître dans un type
- Une fonction “normale” pourrait avoir un type $(x : \tau) \to \ldots x \ldots$
- Une fonction “de type” pourrait avoir un type $(t : \kappa) \to \ldots t \ldots$

À la place, les règles de typage les empêchent:

- $\Gamma \vdash x : \_ : * \wedge \Gamma \vdash \tau : \_ : \Box \Rightarrow x \notin \mathtt{fv}(\tau)$
- $\Gamma \vdash t : \_ : \Box \wedge \Gamma \vdash \kappa : \Box \Rightarrow t \notin \mathtt{fv}(\kappa)$

Vérification de types décidable

La vérification des types en System-F_ω est décidable

Dépend de la terminaison de la normalisation des types

Langage des types = STLC, qui est *strongly normalizing*!

Propriété moins importante qu'il n'y parait:

- la vérification des types de C++ n'est pas décidable!
- La vérification des types de SML est décicable mais:
  prend des siècles pour certains programmes de taille raisonnable