L'inférence de types

on veut, étant donné un programme implicitement typé, être capable de dire s'il est typable...et, en général, en donner le type ...le ou un type, suivant les cas.

Remarque: des langages comme Pascal, C ou Java demandent que les arguments des fonctions soient explicitement typés, ainsi que le résultat

Les étapes de la compilation

programme source

↓ analyse lexicale, analyse syntaxique

arbre de syntaxe abstraite

↓ analyses statiques (typage,...)

arbre décoré

↓ traductions, optimisations

code exécutable (langage machine)

- on analyse le programme sans l'exécuter
- on travaille sur un arbre représentant le code source

Premier exemple

```
let rec f x = fun y ->
    if x then (string_of_int (9*y)) else f (y>11) (y-3)
f : bool->int->string
```

ingrédients:

- on part des valeurs constantes
- on manipule des *contraintes* entre types

Inférence – esquisse

étapes de la méthode:

- partir du *programme*, et le parcourir en écrivant des contraintes de typage qui doivent être satisfaites suivant les différentes constructions du langage
  - constantes (`3`, `+`, `>`, `...`)
  - constructions du langage `if then else`
  - fonctions: déclaration, application
- "raisonner" sur les contraintes
  - propager l'information
  - arrêter en cas de conflit
    (p.ex. $\texttt{int} = \texttt{int} \to \texttt{int}$, $\texttt{'a} \to \texttt{bool} = \texttt{int}$,...)

Engendrer le problème

Récolter l'information

- on écrit des *contraintes* (égalités entre types) qui doivent être satisfaites pour que l'expression soit typable
- exemple:

$$\texttt{let f g x = if } \overbrace{\underbrace{\texttt{x}}_{A_1} > 0}^{A_x} \texttt{ then } \underbrace{3}_{A_3} \texttt{ else } \overbrace{\underbrace{\texttt{g}}_{A_g}\ \underbrace{\texttt{x}}_{A_x}}^{A_2}}^{A_0}$$

  - A_i: types pour toutes les sous-expressions *(il en manque ci-dessus)*
  - contraintes: $A_3 = \texttt{int},\ A_x = \texttt{int},$ $\overbrace{A_1 = \texttt{bool}, A_0 = A_3 = A_2}^{\texttt{if then else}}, A_g = A_x \to A_2, A_0 = \texttt{int}, \ldots$

    et le type de f? $\overbrace{\texttt{f}}^{A_f} = \texttt{fun } \overbrace{\texttt{g}}^{A_g} \texttt{ -> } \underbrace{\texttt{fun } \overbrace{\texttt{x}}^{A_x} \texttt{ -> } \overbrace{\ldots}^{A_o}}_{A_4}$

    $\rightsquigarrow \quad A_4 = A_x \to A_0,\ A_f = A_g \to A_4$

Engendrer les contraintes

- on associe une 'inconnue de type' (variable de type) à chaque sous-expression *(ou sous-arbre)*
- contraintes = équations entre types

`m= e1+e2`	$T_\texttt{m} = \texttt{int}, T_\texttt{e1} = \texttt{int}, T_\texttt{e2} = \texttt{int},$
`m = if e1 then e2 else e3`	$T_\texttt{e1} = \texttt{bool}, T_\texttt{e2} = T_\texttt{m}, T_\texttt{e3} = T_\texttt{m}$
`m = e1 e2`	$T_\texttt{e1} = T_\texttt{e2} \to T_\texttt{m}$
`m = fun x -> e`	$T_\texttt{m} = T_\texttt{x} \to T_\texttt{e}$

  ainsi, pour `fun x -> e`, on engendre $T_\texttt{x}$, et (en principe) à chaque occurrence de `x` dans `e`, on engendre T_i et on écrit $T_i = T_\texttt{x}$
- parcours récursif de l'arbre en appliquant ces règles

Engendrer les contraintes – exemples

- exemple: `let f = fun g -> fun x -> (g (x*2))-3`

  $A_f = A_g \to A_0,\ A_0 = A_x \to A_1,\ A_1 = \texttt{int},\ A_2 = \texttt{int},\ A_g = A_3 \to A_2,\ A_3 = \texttt{int},\ A_x = \texttt{int}$
- `fun x ->` $\overbrace{\texttt{fun f -> } \underbrace{\texttt{(f x)}}_{u}}^{v}$

  $T_\texttt{f} = T_\texttt{x} \to T_u \qquad T_v = T_\texttt{f} \to T_u \qquad T_0 = T_\texttt{x} \to T_v$

  (ce qui se résoud en $T_0 = T_\texttt{x} \to (T_\texttt{x} \to T_u) \to T_u$)

Le système de types

- la manière dont les contraintes sont engendrées découle de la définition du système de types, qui à son tour est décrit par des *règles de typage*

  on définit la relation $\Gamma \vdash \mathtt{e} : T$, où Γ est une liste d'*hypothèses de typage* de la forme $\mathtt{x} : T_{\mathtt{x}}$, "pour x variable libre de e"

$$\textsc{TCst}_0 \quad \Gamma \vdash \mathtt{0} : \mathtt{int} \qquad \textsc{TCst}_1 \quad \Gamma \vdash \mathtt{1} : \mathtt{int} \qquad \textsc{TCst}_+ \quad \Gamma \vdash (+) : \mathtt{int} \to \mathtt{int} \to \mathtt{int} \qquad \textit{etc.}\ldots$$

$$\textsc{TIf}\ \frac{\Gamma \vdash \mathtt{b} : \mathtt{bool} \qquad \Gamma \vdash \mathtt{e} : T \qquad \Gamma \vdash \mathtt{e}' : T}{\Gamma \vdash \mathtt{if\ b\ then\ e\ else\ e}' : T}$$

$$\textsc{TApp}\ \frac{\Gamma \vdash \mathtt{f} : T \to U \qquad \Gamma \vdash \mathtt{e} : T}{\Gamma \vdash \mathtt{f\ e} : U} \qquad \textsc{TFun}\ \frac{\Gamma, \mathtt{x} : T \vdash \mathtt{e} : U}{\Gamma \vdash \mathtt{fun\ x\ ->\ e} : T \to U} \qquad \textsc{TVar}\ \frac{\mathtt{x} : T \in \Gamma}{\Gamma \vdash \mathtt{x} : T}$$

Dérivation de typage – exemple

$$\textsc{TCst}_0\ \Gamma \vdash \mathtt{0} : \mathtt{int} \qquad \textsc{TCst}_1\ \Gamma \vdash \mathtt{1} : \mathtt{int} \qquad \textsc{TCst}_+\ \Gamma \vdash (+) : \mathtt{int} \to \mathtt{int} \to \mathtt{int}$$

$$\textsc{TIf}\ \frac{\Gamma \vdash \mathtt{b} : \mathtt{bool} \qquad \Gamma \vdash \mathtt{e} : T \qquad \Gamma \vdash \mathtt{e}' : T}{\Gamma \vdash \mathtt{if\ b\ then\ e\ else\ e}' : T}$$

$$\textsc{TApp}\ \frac{\Gamma \vdash \mathtt{f} : T \to U \qquad \Gamma \vdash \mathtt{e} : T}{\Gamma \vdash \mathtt{f\ e} : U} \qquad \textsc{TFun}\ \frac{\Gamma, \mathtt{x} : T \vdash \mathtt{e} : U}{\Gamma \vdash \mathtt{fun\ x\ ->\ e} : T \to U} \qquad \textsc{TVar}\ \frac{\mathtt{x} : T \in \Gamma}{\Gamma \vdash \mathtt{x} : T}$$

- ces règles permettent de construire des *dérivations de typage* (arbres dont la conclusion est un *jugement de typage*)
- exemple:

  $\emptyset \vdash$ `fun g -> fun x -> (g (x*2))-3 : (int->int)->int->int`

  DÉMO au tableau

- une règle par construction du langage

  $\hookrightarrow$ pour l'inférence, on raisonne par cas

  "on fait un `match with`*"*

Retour de l'unification

- les contraintes engendrées par un programme sont vues comme un **problème d'unification**

  on résoud des équations symboliques sur les *types* de Caml

  p.ex. $A_1 \to (\mathtt{int} \to A_2) \stackrel{?}{=} (A_3 \to \mathtt{bool}) \to A_1$

  ou si on préfère

  $fleche(A_1, fleche(\mathtt{int}, A_2)) \stackrel{?}{=} fleche(fleche(A_3, \mathtt{bool}), A_1)$

  - si l'unification donne une substitution S, on renvoie le type $S(A_f)$ *(on est en train de typer* `let f = ...`*)*
  - sinon, on proteste *(Caml raconte où l'unification a planté)*
- et voilà

Inférence de types – propriétés

terme $\mathtt{m} \longrightarrow$ système d'équations $\mathcal{C}(\mathtt{m}) \xrightarrow{\text{unification}}$ unificateur S

Propriétés:

- correction: un unificateur S de $\mathcal{C}(\mathtt{m})$ permet d'inférer $\emptyset \vdash m : S(A_{\mathtt{m}})$
- complétude: si l'on peut dériver $\emptyset \vdash \mathtt{m} : T$, alors $\mathcal{C}(\mathtt{m})$ admet une solution S t.q. $S(A_{\mathtt{m}}) = T$

Déroulons un exemple

```
let h f b = if b then 52 else (f b)+32
```

on engendre le problème d'unification

$A_h \stackrel{?}{=} A_f \to A_1,\ A_1 \stackrel{?}{=} A_b \to A_2,\ A_2 \stackrel{?}{=} \texttt{int},\ A_b \stackrel{?}{=} \texttt{bool},\ A_3 \stackrel{?}{=} \texttt{int},\ A_f \stackrel{?}{=} A_b \to A_3$

$\Rightarrow \quad A_1 \stackrel{?}{=} A_b \to A_2,\ A_2 \stackrel{?}{=} \texttt{int},\ A_b \stackrel{?}{=} \texttt{bool},\ A_3 \stackrel{?}{=} \texttt{int},\ A_f \stackrel{?}{=} A_b \to A_3,\ \{A_h \leftarrow\!\!\shortmid A_f \to A_1\}$

$\Rightarrow \quad A_2 \stackrel{?}{=} \texttt{int},\ A_b \stackrel{?}{=} \texttt{bool},\ A_3 \stackrel{?}{=} \texttt{int},\ A_f \stackrel{?}{=} A_b \to A_3,\ \{A_h \leftarrow\!\!\shortmid A_f \to (A_b \to A_2),\ A_1 \leftarrow\!\!\shortmid A_b \to A_2\}$

$\Rightarrow \quad A_b \stackrel{?}{=} \texttt{bool},\ A_3 \stackrel{?}{=} \texttt{int},\ A_f \stackrel{?}{=} A_b \to A_3,\ \{A_h \leftarrow\!\!\shortmid A_f \to (A_b \to \underline{\texttt{int}}),\ A_1 \leftarrow\!\!\shortmid A_b \to \underline{\texttt{int}},\ A_2 \leftarrow\!\!\shortmid \texttt{int}\}$

$\Rightarrow \quad A_3 \stackrel{?}{=} \texttt{int},\ A_f \stackrel{?}{=} \underline{\texttt{bool}} \to A_3,\ \{A_h \leftarrow\!\!\shortmid A_f \to (\underline{\texttt{bool}} \to \texttt{int}),\ A_1 \leftarrow\!\!\shortmid \underline{\texttt{bool}} \to \texttt{int},\ A_2 \leftarrow\!\!\shortmid \texttt{int},\ A_b \leftarrow\!\!\shortmid \texttt{bool}\}$

$\Rightarrow \quad A_f \stackrel{?}{=} \texttt{bool} \to \underline{\texttt{int}},\ \{A_h \leftarrow\!\!\shortmid A_f \to (\texttt{bool} \to \texttt{int}),\ A_1 \leftarrow\!\!\shortmid \texttt{bool} \to \texttt{int},\ A_2 \leftarrow\!\!\shortmid \texttt{int},\ A_b \leftarrow\!\!\shortmid \texttt{bool},\ A_3 \leftarrow\!\!\shortmid \texttt{int}\}$

$\Rightarrow \quad \emptyset,\ \{A_h \leftarrow\!\!\shortmid \underline{(\texttt{bool} \to \texttt{int})} \to (\texttt{bool} \to \texttt{int}),\ A_1 \leftarrow\!\!\shortmid \texttt{bool} \to \texttt{int},\ A_2 \leftarrow\!\!\shortmid \texttt{int},\ A_b \leftarrow\!\!\shortmid \texttt{bool},\ A_3 \leftarrow\!\!\shortmid \texttt{int},\ A_f \leftarrow\!\!\shortmid \texttt{bool} \to \texttt{int}\}$

Typage des termes *"purs"*

un type pour `g = fun x f -> (f x)` ?

- si on déroule l'algorithme d'inférence, on trouve $A_g = A_1 \to (A_2 \to A_3)$ avec la contrainte $A_2 = A_1 \to A_3$, d'où le type $A_1 \to (A_1 \to A_3) \to A_3$
- *qui sont ces A_1 et A_3 qui 'restent'?*
  - des variables de type non contraintes
    exemple encore plus évident: `let f x y = y`, $\rightsquigarrow A_x \to A_y \to A_y$
  - si `g` avait été appliqué à des arguments, A_1 et A_3 auraient pu subir d'autres contraintes

Limites du typage envisagé

- intéressons-nous à

  `(fun f -> (` $\underbrace{\texttt{f}}_{T_1}$ `(32,"hop"))*(` $\underbrace{\texttt{f}}_{T_2}$ `(52,false)))` $\underbrace{\texttt{(fun (u,v) -> u)}}_{T_0}$

- on engendre les contraintes, on mélange un peu:

  $T_1 = \texttt{int} * \texttt{string->int} \quad T_1 = T_0$
  $T_2 = \texttt{int} * \texttt{bool->int} \quad T_2 = T_0$
  $T_0 = T_u * T_v \to T_u$

  - conflit de ressource: T_1 et T_2 'veulent' instancier T_u et T_v
  - d'ailleurs ça ne type pas en Caml
- on voudrait avoir le droit de donner un type *générique* que l'on puisse *instancier* plusieurs fois
  - une instanciation par utilisation de `f` sur l'exemple
- jusque là les types étaient *monomorphes*, on veut le **polymorphisme**