L'inférence de types

Principe

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 et le résultat des fonctions soient explicitement typés

on s'intéresse au typage *statique* (avant l'exécution du programme)

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* (un terme), 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: définition, application
- "raisonner" sur les contraintes
  - propager l'information
  - arrêter en cas de conflit
    (p.ex. $\mathtt{int} = \mathtt{int} \rightarrow \mathtt{int}$, ou $'\mathtt{a} \rightarrow \mathtt{bool} = \mathtt{int}, \ldots$)

Engendrer le problème

Récolter l'information

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

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

  - 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? $\overbrace{\texttt{f}}^{A_f} = \texttt{fun} \overbrace{\texttt{g}}^{A_g} \texttt{->} \underbrace{\texttt{fun} \overbrace{\texttt{x}}^{A_x} \texttt{->} \overbrace{\texttt{...}}^{A_0}}_{A_4}$

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

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

  ainsi, pour `fun x -> e`, on engendre $T_{\mathtt{x}}$, et (en principe) à chaque occurrence de `x` dans `e`, on engendre T_i et on écrit $T_i = T_{\mathtt{x}}$

  autant associer directement $T_{\mathtt{x}}$ à toutes les occurrences

- parcours récursif de l'arbre en appliquant ces règles

Engendrer les contraintes – exemples

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

  $A_f = A_g \rightarrow A_0,\ \ A_0 = A_x \rightarrow A_1,\ \ A_1 = \texttt{int},\ \ A_2 = \texttt{int},\ \ A_g = A_3 \rightarrow A_2,\ \ A_3 = \texttt{int},\ \ A_x = \texttt{int}$

- $\texttt{fun x ->} \overbrace{\texttt{fun f -> } \underbrace{\texttt{(f x)}}_{u}}^{v}$

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

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

Variables et environnements

- évaluateur du premier TP

  pour *évaluer* `let x = e1 in e2`

  - évaluer `e1` $\rightsquigarrow$ `v1`
  - évaluer `e2`, à chaque fois qu'on tombe sur `x`, renvoyer `v1`

  environnement: on associe aux variables (`x`) des valeurs (`v1`)

- pour *typer* `fun x -> e2`

  - créer une nouvelle variable de type $T_{\mathtt{x}}$
  - à chaque fois qu'on tombe sur `x` dans `e2`, renvoyer comme type $T_{\mathtt{x}}$

  environnement: on associe aux variables (`x`) des types ($T_{\mathtt{x}}$)

- à chaque fois, un **environnement** pour savoir gérer les variables libres de l'expression que l'on examine

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

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

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

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

Dérivation de typage

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

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

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

- ces règles permettent de construire des *dérivations de typage* (arbres dont la conclusion est un *jugement de typage*)
- typage du λ-calcul: la dernière ligne

  un λ-terme typable par ces règles est terminant
- 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

- engendrer des contraintes en se fondant sur les règles de typage *(qui définissent "être bien typé")*

  "telle expression est typable à condition que $T_x \rightarrow T_1 = T_2$"
- les contraintes engendrées sont vues comme un **problème d'unification**

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

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

  ou si on préfère

  $\mathit{fleche}(A_1, \mathit{fleche}(\texttt{int}, A_2)) \stackrel{?}{=} \mathit{fleche}(\mathit{fleche}(A_3, \texttt{bool}), A_1)$
  - si le processus d'unification aboutit à une substitution $\mathcal{S}$, on renvoie le type $\mathcal{S}(A_f)$ *(on est en train de typer `let f = ...`)*
  - sinon, on proteste *(Caml raconte où l'unification a planté)*
- et voilà

Inférence de types – propriétés

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

Propriétés:

- correction: un unificateur $\mathcal{S}$ de $\mathcal{C}(\mathtt{m})$ permet d'inférer

$$\emptyset \vdash m : \mathcal{S}(A_{\mathtt{m}})$$

$A_{\mathtt{m}}$: variable de type associée à $\mathtt{m}$

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

Déroulons un exemple

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

on engendre le problème d'unification

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

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

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

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

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

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

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

Typage des termes *"purs"*

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

- si on déroule l'algorithme d'inférence, on trouve $A_g = A_1 \rightarrow (A_2 \rightarrow A_3)$ avec la contrainte $A_2 = A_1 \rightarrow A_3$, d'où le type $A_1 \rightarrow (A_1 \rightarrow A_3) \rightarrow A_3$
- *qui sont ces A_1 et A_3 qui 'restent'?*
  - des variables de type non contraintes

    exemple encore plus évident:
    `let f = fun x y -> y`, $\rightsquigarrow$ $A_x \rightarrow A_y \rightarrow A_y$
  - 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

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

- on engendre les contraintes, on mélange un peu:

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

  - 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**

Types polymorphes

Schémas de types

- pour donner un sens *générique* au typage, on veut disposer de *schémas de types* $\boxed{\forall a. T}$
  - ainsi $\forall a. \forall b. (a * b) \rightarrow a$ (noté `('a * 'b) -> 'a` en Caml) peut s'instancier en un nombre infini de types
    - `(int*bool)->int, (bool*string)->bool,...`
    - $\forall$ est un lieur
- ajouter les schémas de types aux types?

$$T ::= \texttt{int} \mid \texttt{bool} \mid T \rightarrow T' \mid \forall a.\, T \mid a$$

  a: variable de type

- **non**, on impose une restriction: *forme prénexe pour les* $\forall$

$$\begin{array}{lll} T & ::= & \texttt{int} \mid \texttt{bool} \mid T \rightarrow T' \mid a \qquad \text{types} \\ \tau & ::= & T \mid \forall a.\, \tau \qquad \text{schémas de types} \end{array}$$

on n'a pas droit p.ex. à $\forall b.\, b \rightarrow \forall a.\, (a \rightarrow b)$

Polymorphisme et `let`

- les schémas de types n'apparaissent que dans une situation bien particulière: dans un `let...in...`
  pour typer `let x = e1 in e2`
  1. inférer le type de `e1` $\rightsquigarrow \; T_1$
  2. généraliser $t_1 \;\rightsquigarrow \forall\vec{a}.\, T_1$
  3. inférer le type de `e2`, environnement enrichi avec `x`: $\forall\vec{a}.\, T_1$

     ce faisant, dans `e2`, quand on rencontre `x`, on a le droit d'instancier $\forall\vec{a}.\, T_1$ (en $T_1[\vec{U_i}/\vec{a}]$)
- remarques
  - le polymorphisme arrive par les `let` et s'en va dans les variables
  - le contexte de typage (Γ) est enrichi
    - par `x`: $\forall\vec{a}.T_{\mathtt{e1}}$ lorsqu'on type `let x = e1 in e2`,
    - par `x`: a lorsqu'on type `fun x -> e`

Généralisation – exemples

- le typage de `let g = fun x f -> (f x)`

  renvoie un type $a_1 \rightarrow (a_1 \rightarrow a_3) \rightarrow a_3$,

  que l'on peut *généraliser* en

  $$\forall a_1 \forall a_3 .\, a_1 \rightarrow (a_1 \rightarrow a_3) \rightarrow a_3$$

- ainsi, pour typer

  `let m = let f x = x in (f f)`

  - il suffit d'associer à `f` le schéma de type $\forall a.a \rightarrow a$,
  - que l'on instancie avec $b \rightarrow b$ et $(b \rightarrow b) \rightarrow (b \rightarrow b)$,
  - et l'on trouve `m` : $b \rightarrow b$ puis `m`: $\forall b.b \rightarrow b$ *(oui enfin bon,... cf. + tard)*

- reste que pour espérer typer

  `(fun f -> ((f 1),(f "un")))` `(fun t->t)`

  il faut pouvoir *"généraliser en cours de route"* pour pouvoir donner à `f` **les** types `int->int` et `string->string`

  - donner le type $\forall a_0.a_0 \rightarrow a_0$ à `(fun t->t)`
  - et donner le type $(\forall a_0.a_0 \rightarrow a_0) \rightarrow$`int*string` à `(fun f -> ((f 1),(f "un")))`

    $\hookrightarrow$ pas possible

Inférence, suite

- inférence de types
  - on engendre des *contraintes de typage* en explorant un terme
  - on résoud ces contraintes par *unification*
  - types: $T \quad ::= \quad \texttt{int} \mid \texttt{bool} \mid T_1 \to T_2 \mid a$ $\qquad a$ variable de type
- polymorphisme: quantifier sur les variables de type
  - types en ML: quantification *prénexe*
    $T \quad ::= \quad \texttt{int} \mid \texttt{bool} \mid T_1 \to T_2 \mid a \qquad \tau \quad ::= \quad T \mid \forall a.\tau$
  - le polymorphisme va de pair avec la construction `let...in`
    `let x = e1 in e2`:
    - on infère le type de `e1` $\qquad T$
    - on généralise $\qquad \forall \tilde{a}.T$
    - on infère le type de `e2` avec l'hypothèse `x`:$\forall \tilde{a}.T$
      chaque `x` qui apparaît dans `e2`: une instanciation de $\forall \tilde{a}.T$
- système F: $\quad T \quad ::= \quad \texttt{int} \mid \texttt{bool} \mid T_1 \to T_2 \mid a \mid \forall a.\, T$
  - nécessaire pour typer
    `(fun f -> ((f 1),(f "un")))    (fun t->t)`
    car on veut donner le type $(\forall a_0.a_0 \to a_0) \to \texttt{int*string}$
    à la fonction de gauche
  - interdit en ML: pas d'arguments polymorphes
    `let f = fun t -> t in ((f 1), (f "un"))` : ok

Polymorphisme: typage

$$\frac{\Gamma \vdash \mathtt{m} : T \to T' \qquad \Gamma \vdash \mathtt{n} : T}{\Gamma \vdash (\mathtt{m}\ \mathtt{n}) : T'} \qquad \frac{\Gamma, \mathtt{x} : T \vdash \mathtt{m} : T'}{\Gamma \vdash \mathtt{fun\ x\ \text{->}\ m} : T \to T'}$$

$$\frac{\Gamma \vdash \mathtt{m} : T \qquad \Gamma, \mathtt{x} : \mathrm{Gen}(T, \Gamma) \vdash \mathtt{n} : T'}{\Gamma \vdash \mathtt{let}\ \mathtt{x} = \mathtt{m}\ \mathtt{in}\ \mathtt{n} : T'}$$

$$\frac{}{\Gamma, \mathtt{x} : \forall \vec{a}. T \vdash \mathtt{x} : T[\vec{T'}/\vec{a}]}$$

- $\mathrm{Gen}(T, \Gamma)$: généralisation de T *(dépend de Γ)*
- dans les règles de typage, Γ associe des *schémas de types* aux variables
- le système de types avec polymorphisme est défini pour un langage *incluant* `let...in`

Typage avec polymorphisme – exemple

$$\frac{\Gamma \vdash \texttt{m} : T \to T' \qquad \Gamma \vdash \texttt{n} : T}{\Gamma \vdash (\texttt{m n}) : T'} \qquad \frac{\Gamma, \texttt{x} : T \vdash \texttt{m} : T'}{\Gamma \vdash \texttt{fun x -> m} : T \to T'}$$

$$\frac{\Gamma \vdash \texttt{m} : T \qquad \Gamma, \texttt{x} : \mathrm{Gen}(T, \Gamma) \vdash \texttt{n} : T'}{\Gamma \vdash \texttt{let x = m in n} : T'}$$

$$\frac{}{\Gamma, \texttt{x} : \forall \vec{a}. T \vdash \texttt{x} : T[\vec{T'}/\vec{a}]}$$

$$\dfrac{\dfrac{\texttt{t} : a \vdash \texttt{t} : a}{\emptyset \vdash \texttt{fun t -> t} : a \to a} \qquad \dfrac{\begin{array}{c}\texttt{id} : \forall a.\, a \to a \vdash \texttt{id} : (b \to b) \to (b \to b) \\ \text{et} \qquad \texttt{id} : \forall a.\, a \to a \vdash \texttt{id} : b \to b\end{array}}{\texttt{id} : \forall a.\, a \to a \vdash \texttt{id id} : b \to b}}{\emptyset \vdash \texttt{let id = fun t -> t in id id} : b \to b}$$

Typer les let

$$\frac{\Gamma \vdash \mathtt{m} : T \qquad \Gamma, x : \mathrm{Gen}(T, \Gamma) \vdash \mathtt{n} : T'}{\Gamma \vdash \mathtt{let\ x = m\ in\ n} : T'}$$

- comment calcule-t-on $\mathrm{Gen}(T, \Gamma)$?

  considérons `fun u -> let x = u in x`

$$\dfrac{\dfrac{\mathtt{u} : a \vdash \mathtt{u} : a \qquad \mathtt{u} : a, \mathtt{x} : \forall \mathbf{a.a} \vdash \mathtt{x} : b}{\mathtt{u} : a \vdash \mathtt{let\ x = u\ in\ x} : b}}{\mathtt{fun\ u\ \text{->}\ let\ x = u\ in\ x} : a \rightarrow b}$$

le type $a \rightarrow b$ n'est pas correct!

  cf.

$$\dfrac{\dfrac{\mathtt{t} : a \vdash \mathtt{t} : a}{\emptyset \vdash \mathtt{fun\ t\ \text{->}\ t} : a \rightarrow a} \qquad \dfrac{\mathtt{id} : \forall a.\, a \rightarrow a \vdash \mathtt{id} : (b \rightarrow b) \rightarrow (b \rightarrow b) \qquad \mathtt{id} : \forall a.\, a \rightarrow a \vdash \mathtt{id} : b \rightarrow b}{\mathtt{id} : \forall a.\, a \rightarrow a \vdash \mathtt{id\ id} : b \rightarrow b}}{\emptyset \vdash \mathtt{let\ id = fun\ t\ \text{->}\ t\ in\ id\ id} : b \rightarrow b}$$

- on généralise par rapport aux variables libres de T *qui ne sont pas dans* Γ

Typer les `let` (2)

$$\frac{\Gamma \vdash \mathtt{m} : T \qquad \Gamma, x : \mathrm{Gen}(T, \Gamma) \vdash \mathtt{n} : T'}{\Gamma \vdash \mathtt{let\ x = m\ in\ n} : T'}$$

- on ne peut pas résoudre les contraintes dans n'importe quel ordre:

$$\vdash \quad \mathtt{let\ x} = \overbrace{\mathtt{fun\ y\ \text{->}\ } \underbrace{\mathtt{y}}_{a_u}}^{a_v} \mathtt{\ in\ (x\ 1)} \ : \ ?$$

  - on obtient en particulier les contraintes:
    (1) $a_x = \mathrm{Gen}(a_v, \emptyset)$ (2) $a_v = a_y \to a_u$ (3) $a_u = a_y$
  - si on commence par (1), on trouve $a_x = \forall a_v . a_v$:
    ça ne va pas, puisque `x` *a forcément un type fonctionnel*
- l'ensemble de contraintes que l'on manipule pour l'inférence est donc plus structuré que dans le cas monomorphe
  - on unifie 'bout par bout'
  - on intercale des généralisations et des instanciations
  - du *contrôle* dans l'inférence: ce n'est plus une grande soupe de contraintes *('goulots d'étranglement' dans la procédure d'inférence)*
- algorithme $\mathcal{W}$, dû à Damas et Milner (1982) *(cf. aussi Hindley)*

Algorithme $\mathcal{W}$ (esquisse)

- pour inférer le type de `m` dans Γ:
  - si `m` est `fun x -> m'`, appel récursif sur `m'` avec $\Gamma, \mathtt{x} : a$ $\hookrightarrow \mathcal{S}$ (a *nouvelle* variable de type)
  - si `m` est `(m₁ m₂)`, deux appels récursifs ($\rightsquigarrow$ T_1 et T_2), puis unifier T_1 et $T_2 \to a$, où a est nouvelle $\hookrightarrow \mathcal{S}$
  - si `m` est `let x = m₁ in m₂`, typer $\mathtt{m}_1$ (renvoie T_1), calculer $T_1' = \mathrm{Gen}(T_1, \Gamma)$, et typer $\mathtt{m}_2$ dans $\Gamma, x : T_1'$ $\hookrightarrow \mathcal{S}$
  - si `m` est `x`, $(\mathtt{x} : \forall \vec{a}. T) \in \Gamma$, instancier T avec des variables de type *nouvelles* $\hookrightarrow \mathcal{S}$
- ainsi, pour `let id = fun x -> x in (id 1, id true)`
  - on infère $\mathtt{id}{:}\forall a. a \to a$ pour `id = fun x -> x`
  - puis on instancie *(ici, deux fois)*: $\mathtt{id}{:}a_1 \to a_1$ pour `id 1`, et $\mathtt{id}{:}a_2 \to a_2$ pour `id true` $\rightsquigarrow$ $a_1 = \mathtt{int}$, $a_2 = \mathtt{bool}$
  - chaque usage d'une variable dont le type a été généralisé a un type potentiellement différent

Propriétés de $\mathcal{W}$

- correction:

  si $\mathcal{W}(\mathtt{m}, \Gamma)$ retourne un type T et une substitution $\mathcal{S}$, alors $\mathcal{S}(\Gamma) \vdash \mathtt{m} : T$

- complétude et principalité:

  si $\Gamma \vdash \mathtt{m} : T$, alors $\mathcal{W}$ renvoie T' t.q. $T = T'[\vec{a} \leftarrow \vec{u}]$

- c'est la fête

Et si on expansait les `let`?

$$\frac{\Gamma \vdash \mathtt{M} : t \quad \Gamma \vdash \mathtt{N[M/x]} : t'}{\Gamma \vdash \mathtt{let\ x = M\ in\ N} : t'}$$

(on doit typer `M` à cause de `let x = (5 "toto") in 3`)

plus besoin de la généralisation, on retrouve le monomorphisme

on a de plus:

Théorème: dans le langage avec polymorphisme,
$\Gamma \vdash \mathtt{let\ x = e1\ in\ e2} : T$ ssi
$\exists\, T'. \Gamma \vdash \mathtt{e1} : T'$ et $\Gamma \vdash \mathtt{e2[e1/x]} : T$

...oui mais $\mathcal{W}$ ne type `e1` qu'une seule fois, alors qu'ici on le type autant de fois qu'il y a d'occurrences de `x` dans `e2`

technique de compilation ("*inlining*"): parfois utile pour avoir du code plus efficace *(travailler avec des valeurs non boxées)*

Le système F Girard 72, Reynolds 74

- en ML, les types polymorphes ont une quantification *prénexe*
  - $\forall\vec{a}.T$, et T ne contient pas de $\forall$
- système F: on mélange tout

$$T \quad = \quad a \mid T \rightarrow T' \mid \forall a.T$$

- un formalisme expressif:
  - on va au-delà de ML $(\forall a.a \rightarrow (\forall b.\, b \rightarrow a) \rightarrow a)$
  - les types de données usuels (entiers, listes, arbres,...) peuvent être définis
    - entiers: `int`$=\forall a.\,(a \rightarrow a) \rightarrow a \rightarrow a$
      `f : int->int`, ...
    - listes: $\forall a.\forall b.\, a \rightarrow (a \rightarrow b \rightarrow a) \rightarrow a$
    - ...
  - motivations à la fois informatiques et logiques
    1. le polymorphisme pour typer des fonctions génériques
    2. toutes les fonctions prouvablement totales dans l'arithmétique de Peano du second ordre sont typables dans F

Mais surtout

- décidabilité du typage dans F?
  - J. Wells 1994: étant donné un terme m, on ne sait pas décider s'il existe Γ, T tels que $\Gamma \vdash m : T$ dans le système F
  - conséquence: pas d'inférence de type

    $\hookrightarrow$ d'où, après coup, la justification du fait qu'on travaille dans des sous-ensembles de F
- le polymorphisme à la ML, c'est bien
  - inférence *décidable* (algorithme $\mathcal{W}$)
  - *expressivité*: le polymorphisme à la ML est dans bien des cas suffisant
- pour aller plus loin
  - B. Pierce, *Types and Programming Languages*, MIT Press (disponible à la bibliothèque)
  - Girard, Lafont, Taylor, *Proofs and Types*, Cambridge University Press (trouvable sur le net)
  - D. Le Botlan et D. Rémy, *MLF*

Polymorphisme du polymorphisme

polymorphisme: une valeur peut avoir plusieurs types

- polymorphisme de sous-typage
  - "tout T peut être vu comme un U": $T \leq U$ 　 `'a->'a` $\leq$ `bool->bool`
  - cf. modules (structures) et signatures
- polymorphisme paramétrique
  - une fonction peut être appliquée à des arguments de types différents (c'est les `'a` de Caml)
  - c'est *contraignant* pour une fonction que d'avoir un type polymorphe
    `f: 'a -> ...`: `f` n'inspecte pas son premier argument
    - cf. types abstraits, protection
    - fonction polymorphe en Caml: déplace des mots mémoire
- polymorphisme ad hoc, surcharge
  - ce n'est pas toujours le même code qui est exécuté lorsqu'une fonction est appelée
  - `(=)` en Caml, et C++ de manière intensive

on a été jusqu'ici purement fonctionnel

Mélanger fonctionnel et impératif

Vers le "vrai" ML: aspects impératifs

```
m  ::=  x | fun x -> m | (m m') | 32 | (+) | ...
      | ref m | !m | m:=m' | m;m'
```

$$T \;=\; a \mid T \to T' \mid \mathtt{int} \mid \ldots \mid T\ \mathtt{ref} \qquad\qquad \tau \;=\; \forall \vec{a}.T$$

$$\ldots$$

$$\frac{\Gamma \vdash \mathtt{m} : T}{\Gamma \vdash \mathtt{ref}\ \mathtt{m} : T\ \mathtt{ref}} \qquad \frac{\Gamma \vdash \mathtt{m} : T\ \mathtt{ref}}{\Gamma \vdash \mathtt{!m} : T}$$

$$\frac{\Gamma \vdash \mathtt{m} : T\ \mathtt{ref} \quad \Gamma \vdash \mathtt{n} : T}{\Gamma \vdash \mathtt{m} := \mathtt{n} : \mathtt{unit}} \qquad \frac{\Gamma \vdash \mathtt{m} : S \quad \Gamma \vdash \mathtt{n} : T}{\Gamma \vdash \mathtt{m}\,;\mathtt{n} : T}$$

Polymorphisme et références

- jeux dangereux entre polymorphisme et références:

```
let r = ref (fun x -> x) in
  r := (fun x -> x+1);
  (!r) true;;
```

- si on donnait le type `('a -> 'a) ref` à `r`, on perdrait la sûreté du typage: on tombe sur `(fun x -> x+1) true`

  (les deux `r` 'se parlent' à travers l'effet de bord)

- ainsi, lorsqu'on type `let x = ref e1 in e2`, on ne généralise pas le type trouvé pour `ref e1`
  - retour au monomorphisme DÉMO `ref_mono.ml`
    `'_a`: 'polymorphisme faible'

- premiére idée: avoir peur lorsque le type que l'on généralise contient `ref`

  est-on sorti d'affaire?

Les applications

- regardons le code suivant:

```
let ref_fonctionnelle = fun x ->
    let r = ref x in
        ((fun newx -> r:= newx), (fun () -> !r));;
```

- on a

  `ref_fonctionnelle :  'a -> ('a->unit * unit->'a)`

  pas de `ref` *dans le type*: on est tenté par le polymorphisme

- et maintenant:

```
let écrire,lire = ref_fonctionnelle (fun x->x) in
  écrire (fun x -> x+1);
  lire() true;;
```

- . . . le danger, c'est de généraliser le type de

  `ref_fonctionnelle (fun x -> x)`

- les applications peuvent engendrer la création de références: elles sont a priori *dangereuses*

Expressions non expansives

- **dans `let x = e1 in e2`, la difficulté provient des références que pourrait créer l'évaluation de `e1`**

  dans le type de `e1`, on ne généralise pas si on craint que l'évaluation de `e1` ne crée des références

  - `e1 = ref e` est potentiellement dangereux
  - `e1 = (f e)` aussi

- on introduit la notion d'*expression non expansive*, i.e. dont l'évaluation ne risque pas de créer une référence;

  $\hookrightarrow$ ne sont pas expansives:

  - les constantes
  - les variables
  - les fonctions

(les valeurs: the value restriction)

Limites de l'analyse d'expansivité

l'analyse d'expansivité est une *approximation*: il peut arriver que l'on restreigne le type de certaines expressions non dangereuses

(mais pas dans l'autre sens)

- une application peut cacher une valeur non expansive

```
# let g = let id = fun x -> x in (id id);;
val g : '_a -> '_a = <fun>
# g 3;;
- : int = 3
# g;;
- : int -> int = <fun>
# g true;;
This expression has type bool but is here used with type int
```

  - mais on peut indiquer explicitement que l'on a affaire à une fonction *(η-expansion)*

    ```
    let id = fun x -> x in (fun z -> ((id id) z)) : 'a -> 'a
    ```

- autres phénomènes DÉMO `monopoly.ml`
- NB: en purement fonctionnel *(Haskell)*, on a *dans tous les cas* un type polymorphe (pas de `'_a`)