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. $\texttt{int} = \texttt{int} \to \texttt{int}$, ou $\texttt{'a} \to \texttt{bool} = \texttt{int}, \ldots$)

Engendrer le problème

Récolter l'information

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

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

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

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

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 $T_i = T_{\texttt{x}}$

  autant associer directement $T_{\texttt{x}}$ 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`

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

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

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

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_x
  - à chaque fois qu'on tombe sur `x` dans `e2`, renvoyer comme type T_x

  environnement: on associe aux variables (`x`) des types (T_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 $\Gamma \vdash e : T$, où Γ est une liste d'*hypothèses de typage* de la forme $x : T_x$, "pour x variable libre de e"

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

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

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

Dérivation de typage

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

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

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

- 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 $T_x \to T_1 = T_2$"
- les contraintes engendrées sont vues comme un **problème d'unification**

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

  p.ex. $A_1 \to (\texttt{int} \to A_2) \stackrel{?}{=} (A_3 \to \texttt{bool}) \to A_1$
  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 `m` $\longrightarrow$ système d'équations $\mathcal{C}$(`m`) $\xrightarrow{\text{unification}}$ unificateur $\mathcal{S}$

Propriétés:

- correction: un unificateur $\mathcal{S}$ de $\mathcal{C}$(`m`) permet d'inférer
  $\emptyset \vdash m : \mathcal{S}(A_{\texttt{m}})$

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

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

Déroulons un exemple

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

on engendre le problème d'unification

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

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

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

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

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

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

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

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 = fun 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**

Types polymorphes

Schémas de types

- pour donner un sens *générique* au typage, on veut disposer de *schémas de types* $\boxed{\forall a.T}$
  - ainsi $\forall a.\forall b.(a*b)\to a$ (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?

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

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

  $T ::= \texttt{int} \mid \texttt{bool} \mid T\to T' \mid a$ types

  $\tau ::= T \mid \forall a.\tau$ schémas de types

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

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`: $\forall\vec{a}.T_{e1}$ 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 $a_1\to(a_1\to a_3)\to a_3$,

  que l'on peut *généraliser* en

  $\forall a_1\forall a_3.\, a_1\to(a_1\to a_3)\to a_3$
- ainsi, pour typer

  `let m = let f x = x in (f f)`
  - il suffit d'associer à `f` le schéma de type $\forall a.a\to a$,
  - que l'on instancie avec $b\to b$ et $(b\to b)\to(b\to b)$,
  - et l'on trouve `m` : $b\to b$ puis `m`: $\forall b.b\to b$ *(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 $\forall a_0.a_0\to a_0$ à `(fun t->t)`
  - et donner le type $(\forall a_0.a_0\to a_0)\to$`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: $T ::= \texttt{int} \mid \texttt{bool} \mid T_1\to T_2 \mid a$ a variable de type
- polymorphisme: quantifier sur les variables de type
  - types en ML: quantification *prénexe*

    $T ::= \texttt{int} \mid \texttt{bool} \mid T_1\to T_2 \mid a$ $\tau ::= T \mid \forall a.\tau$
  - le polymorphisme va de pair avec la construction `let...in`

    `let x = e1 in e2`:
    - on infère le type de `e1` T
    - on généralise $\forall\vec{a}.T$
    - on infère le type de `e2` avec l'hypothèse `x`:$\forall\vec{a}.T$

      chaque `x` qui apparaît dans `e2` : une instanciation de $\forall\vec{a}.T$
- système F: $T ::= \texttt{int} \mid \texttt{bool} \mid T_1\to T_2 \mid a \mid \forall a.T$
  - nécessaire pour typer

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

    car on veut donner le type $(\forall a_0.a_0\to a_0)\to$`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

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

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

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

- $\mathrm{Gen}(T,\Gamma)$: 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

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

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

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

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

Typer les `let`

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

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

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

  $$\frac{\texttt{u} : a\vdash \texttt{u} : a \qquad \texttt{u} : a, \texttt{x} : \forall\mathbf{a}.\mathbf{a}\vdash \texttt{x} : b}{\texttt{u} : a\vdash \texttt{let x} = \texttt{u in x} : b}$$

  $$\frac{}{\texttt{fun u -> let x} = \texttt{u in x} : a\to b}$$

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

  cf.

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

- 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:

  $\vdash$ `let x =` $\overbrace{\texttt{fun y ->}\ \underbrace{\texttt{y}}_{a_u}}^{a_v}$ `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 $\Gamma, \mathtt{x} : a \hookleftarrow \mathcal{S}$ (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 $\hookleftarrow \mathcal{S}$
  - si `m` est `let x = m1 in m2`, typer `m1` (renvoie T_1), calculer $T_1' = \mathrm{Gen}(T_1, \Gamma)$, et typer `m2` dans $\Gamma, x : T_1' \hookleftarrow \mathcal{S}$
  - si `m` est `x`, $(\mathtt{x} : \forall \vec{a}.\, T) \in \Gamma$, instancier T avec des variables de type *nouvelles* $\hookleftarrow \mathcal{S}$
- ainsi, pour `let id = fun x -> x in (id 1, id true)`
  - on infère `id`:$\forall a.\, a \to a$ pour `id = fun x -> x`
  - puis on instancie *(ici, deux fois)*: `id`:$a_1 \to a_1$ pour `id 1`, et `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. $T = T'[\vec{a} \hookleftarrow \vec{u}]$
- 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

  $T \quad = \quad a \mid T \to T' \mid \forall a.\, T$
- un formalisme expressif:
  - on va au-delà de ML $(\forall a.\, a \to (\forall b.\, b \to a) \to a)$
  - les types de données usuels (entiers, listes, arbres,...) peuvent être définis
    - entiers: `int`=$\forall a.\, (a \to a) \to a \to a$
      `f : int->int`, ...
    - listes: $\forall a. \forall b.\, a \to (a \to b \to a) \to a$
    - ...
  - 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'
```

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

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

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

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'
- première 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;

  ↪ 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`)