Résumé de l'épisode précédent

- les étapes de l'inférence de types
  1. introduire, pour chaque sous-expression e de l'expression considérée, une variable de type T_e
  2. écrire des égalités entre types *(contraintes)* découlant des constructions du langage
     $T_f = T_x \to T_e$, $T_e =$ `bool`
  3. utiliser l'algorithme d'*unification* pour résoudre les contraintes
     - en cas de succès, renvoyer le type de l'expression de départ
     - en cas d'échec, le programme n'est pas typable
- le typage est jusqu'ici *monomorphe*, car on ne peut instancier une variable de type qu'au plus *une* fois
  - `(fun f -> (f f)) (fun x -> x)` avec T_f, T_3, T_1, T_2, $T_4 \to T_4$
  - $T_f = T_1 = T_2 \to T_3$, $T_1 = T_2 \to T_3$, $T_f = T_4 \to T_4$
  - on aimerait instancier T_1 et T_2 de deux manières différentes, ou si on veut instancier deux fois T_f

Types polymorphes

Schémas de types

- pour donner un sens générique au typage, on veut disposer de *schémas de types* $\boxed{\forall a.T}$
  - ainsi $\forall a.\forall b.\,(a \times b) \to a$ (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 ::= `int` | `bool` | $T \to T'$ | $\forall a.\,T$ | a

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

T	::=	`int` | `bool` | $T \to T'$ | a	types
τ	::=	T | $\forall a.\,\tau$	schémas de types

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 $T_1[\vec{T_i}/\vec{a}]$)
- 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

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`

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

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

- on généralise par rapport aux variables libres de T *qui ne sont pas dans* Γ *(méfions-nous des généralisations hâtives)*

Polymorphisme: typage

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

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

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

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

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 =` $\overbrace{\mathtt{fun\ y\ ->\ y}}^{a_v}$ $\underbrace{\phantom{\mathtt{y}}}_{a_u}$ `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 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
- 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 $\Gamma, \mathtt{x} : a \hookrightarrow \phi$ (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 $\hookrightarrow \phi$
  - 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 $\Gamma, x : T_1' \hookrightarrow \phi$
  - si `m` est `x`, $(\mathtt{x} : \forall \vec{a}.T) \in \Gamma$, instancier T avec des variables de type *nouvelles* $\hookrightarrow \phi$
- 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` $\rightsquigarrow$ $a_1 = \mathtt{int}$, $a_2 = \mathtt{bool}$

Propriétés de $\mathcal{W}$

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

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

(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,
$\Gamma \vdash \mathtt{let\ x = m\ in\ n} : T$ ssi
$\exists\, T'.\, \Gamma \vdash \mathtt{m} : T'$ et $\Gamma \vdash \mathtt{n}\{\mathtt{x} \leftarrow \mathtt{m}\} : T$

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

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
  $T \;=\; a \mid T \to T' \mid \forall a.T$
- un formalisme expressif:
  - on va au-delà de ML $(\forall a. a \to (\forall b.\, b \to b) \to a)$
  - 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*

Mélanger fonctionnel et impératif

Polymorphisme du polymorphisme

polymorphisme: une valeur peut avoir plusieurs types

- polymorphisme de sous-typage
  - "tout T peut être vu comme un U": $T \leq U$
  - 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
- 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

Vers le "vrai" ML: aspects impératifs

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

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

$$\ldots$$

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

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

Inférence de types en présence de `ref`

- 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 $\forall a.\ ((a \to a)\ \texttt{ref})$ 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'
- 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:` $\forall a.\ a \to (a \to \texttt{unit} * \texttt{unit} \to a)$

  pas de `ref` dans le type: on est polymorphe

- 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`)