

# Chapitre 9 - Typage

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



# Système de typage

# Système de typage

Un système de typage permet d'associer à chaque expression un *type*.

Une expression est *bien typée* si on peut lui associer un type; sinon elle est *mal typée*

Quel est le type des expressions suivantes ?

- $5 + 3$

# Système de typage

Un système de typage permet d'associer à chaque expression un *type*.

Une expression est *bien typée* si on peut lui associer un type; sinon elle est *mal typée*

Quel est le type des expressions suivantes ?

- `5 + 3 : int`
- `5 + true`

# Système de typage

Un système de typage permet d'associer à chaque expression un *type*.

Une expression est *bien typée* si on peut lui associer un type; sinon elle est *mal typée*

Quel est le type des expressions suivantes ?

- `5 + 3 : int`
- `5 + true` → expression mal typée
- `5 + (if false then true else 3)`

# Système de typage

Un système de typage permet d'associer à chaque expression un *type*.

Une expression est *bien typée* si on peut lui associer un type; sinon elle est *mal typée*

Quel est le type des expressions suivantes ?

- `5 + 3 : int`
- `5 + true` → expression mal typée
- `5 + (if false then true else 3) : int ...?`
- `5 + (if f 1 then true else 3)`

# Système de typage

Un système de typage permet d'associer à chaque expression un *type*.

Une expression est *bien typée* si on peut lui associer un type; sinon elle est *mal typée*

Quel est le type des expressions suivantes ?

- `5 + 3 : int`
- `5 + true` → expression mal typée
- `5 + (if false then true else 3) : int ...?`
- `5 + (if f 1 then true else 3) ???`

# Typage statique et dynamique

Un système de type peut être statique ou dynamique

- **typage statique** : on associe un type à chaque expression *avant* l'exécution du programme
  - cela requiert une approximation, on va refuser des programmes corrects  
`5 + (if false then true else 3)`
- **typage dynamique** : on associe un type à l'exécution
  - c'est ce qu'on a fait au chapitre précédent

# Typage dynamique

En **typage dynamique**, on associe un type au résultat de chaque évaluation

5 + 3

- on évalue le littéral 5 à la valeur `VInt 5`
- on évalue le littéral 3 à la valeur `VInt 3`
- on applique l'opérateur `+` et on obtient la valeur `VInt 8`

```
let int_binary_operator (op : int -> int -> int) (v1 : value) (v2 : value) : value =  
  match v1, v2 with  
  | VInt x, VInt y -> VInt (op x y)  
  | _ -> raise TypeError
```

```
let rec eval (env : env) (e : expr) : value =  
  match e with  
  ...  
  | Plus (e1, e2) ->  
    int_binary_operator (+) (eval env e1) (eval env e2)
```

# Typage dynamique

## 5 + true

- on évalue le littéral 5 à la valeur `VInt 5`
- on évalue le littéral `true` à la valeur `VBool true`
- on applique l'opérateur `+`, qui lève une erreur de typage

```
let int_binary_operator (op : int -> int -> int) (v1 : value) (v2 : value) : value =  
  match v1, v2 with  
  | VInt x, VInt y -> VInt (op x y)  
  | _ -> raise TypeError
```

```
let rec eval (env : env) (e : expr) : value =  
  match e with  
  ...  
  | Plus (e1, e2) ->  
    int_binary_operator (+) (eval env e1) (eval env e2)
```

# Typage dynamique

5 + (if false then true else 3)

- on évalue le littéral 5 à la valeur `VInt 5`
- on évalue l'expression `if ...` à la valeur `VInt 3`
- on applique l'opérateur `+` et on obtient la valeur `VInt 8`

# Typage statique

En **typage statique**, on associe un type à chaque expression *avant* d'évaluer quoi que ce soit

5 + 3

- on sait que 5 : int
- on sait que 3 : int
- l'addition de deux int donne un int, donc le type de l'expression est int

5 + **true**

- on sait que 5 : int
- on sait que true : bool
- l'addition d'un int et un bool lève une erreur de typage (avant l'exécution)

# Typage statique

5 + (if false then true else 3)

- on sait que 5 : int
- pour le if :
  - on sait que false : bool et on veut une condition booléenne : OK
  - on veut un accord entre les deux branches
    - le then a un type bool
    - le else a un type int
  - on a un désaccord : levée d'erreur de typage

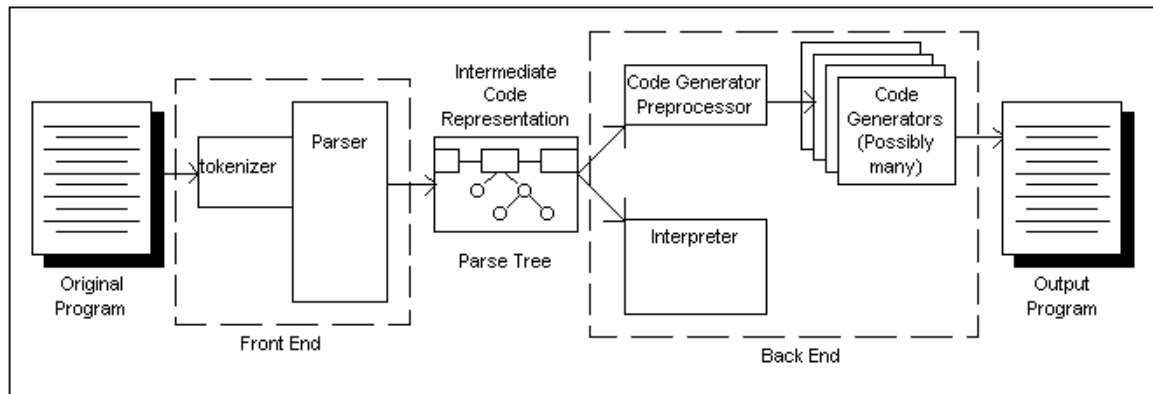
# Typage statique

Le typage statique s'effectue à travers un ensemble de *règles de typage*

Il existe plusieurs caractéristiques pour un système de typage statique, dont la différence entre :

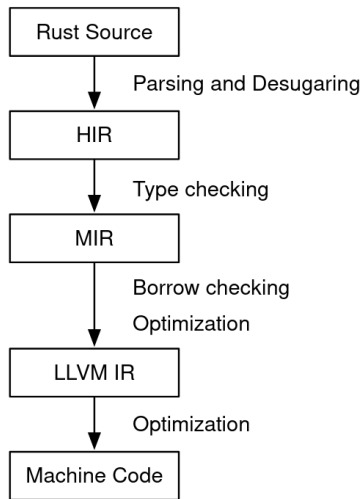
- vérification de type (*type checking*) :
  - le programme est annoté par la personne qui l'écrit
  - on vérifie que cela correspond aux règles de typage
- inférence de type :
  - le programme n'a pas besoin d'être annoté
  - on *infère* les types
  - on vérifie que cela correspond aux règles de typage

# Phases de compilation



Source

# Phases de compilation



Source

# Vérification de type

# Langage

Version sans annotation de types :

```
type binary_op = Add | And
```

```
type expr =  
  | Int of int  
  | Bool of bool  
  | Var of string  
  | BinOp of binary_op * expr * expr  
  | If of expr * expr * expr  
  | Let of string * expr * expr  
  | Function of string * expr  
  | Call of expr * expr
```

# Annotation de types

On pourrait annoter chaque expression de notre programme :

```
((5 : int) + (3 : int)) : int
```

```
((5 : int) + (true : bool)) : int
```

Mais c'est très verbeux...

# Annotation de types

On peut avoir des règles simples pour la plupart des constructions du langage.

- On sait que 5 est de type `int`, tout le temps
- On sait que 3 est de type `int`, tout le temps
- On sait que `true` est de type `bool`, tout le temps
- Si on sait que `(+)` : `int -> int -> int`, on n'a pas besoin d'annoter les appels à `(+)`

# Annotation de types

Le seul élément difficile est le typage des fonctions.

Quel est le type des fonctions suivantes ?

- `let f = fun x y -> x + y`

# Annotation de types

Le seul élément difficile est le typage des fonctions.

Quel est le type des fonctions suivantes ?

- `let f = fun x y -> x + y : int -> int -> int`
- `let g = fun a b -> a (b + 1)`

# Annotation de types

Le seul élément difficile est le typage des fonctions.

Quel est le type des fonctions suivantes ?

- `let f = fun x y -> x + y : int -> int -> int`
- `let g = fun a b -> a (b + 1) : (int -> 'a) -> int -> 'a`
- `let rec h = fun x -> h x`

# Annotation de types

Le seul élément difficile est le typage des fonctions.

Quel est le type des fonctions suivantes ?

- `let f = fun x y -> x + y : int -> int -> int`
- `let g = fun a b -> a (b + 1) : (int -> 'a) -> int -> 'a`
- `let rec h = fun x -> h x : 'a -> 'b`

# Annotation de types

On peut avoir des règles de bases :

- les littéraux n'ont pas besoin d'être annotés
- les résultats des opérateurs de base non plus
  - on définit par exemple que  $(a + b) : \text{int}$ , toujours
- si on sait déduire le type d'une valeur, on sait déduire le type d'une liaison
  - `let a = 5` : vu que `5 : int`, alors `a : int`
  - `let f = fun x -> ...` : si on sait déduire le type de la fonction, alors on connaît le type de `f`

On annote les fonctions pour connaître leur types :

- leurs paramètres sont annotés
- la valeur de retour est annotée

# Annotation de types

Quelques exemples :

```
let f = fun (x : int) (y : int) : int -> x + y
let g = fun (a : int -> int) (b : int) -> a (b + 1)
let g' = fun (a : int -> bool) (b : int) -> a (b + 1)
let rec h = fun (x : int) : int -> h x
let rec h' = fun (x : bool) : int -> h' x
let rec h'' = fun (x : int -> bool) : int -> h'' x
```

Il peut y avoir plusieurs types possibles pour une fonction !<sup>1</sup>

---

<sup>1</sup>Le polymorphisme permet d'avoir  $g : ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'b$ , mais on ne verra pas cela dans ce chapitre. Voir le cours de maîtrise [INF889J](#).

# Types

On a donc trois sortes de types :

```
type typ =  
  | TInt  
  | TBool  
  | TFun of typ * typ
```

## Extension du langage

On rajoute l'information des annotations à notre syntaxe :

```
type expr =  
  | Int of int  
  | Bool of bool  
  | Var of string  
  | BinOp of binary_op * expr * expr  
  | If of expr * expr * expr  
  | Let of string * expr * expr  
  | Function of string * typ * expr * typ (* seule modification *)  
  | Call of expr * expr
```

# Erreur de typage

Si on ne sait pas typer un programme correctement, on lève une exception

**exception** `TypeError`

# Sémantique statique et dynamique

Un interpréteur définit la *sémantique* d'un langage, plus précisément la **sémantique dynamique** :

- il répond à la question *quelle valeur l'expression  $e$  a-t-elle à l'exécution*

Un système de type statique définit une **sémantique statique** d'un langage :

- il répond, sans exécuter le programme, à la question *quel est le type de l'expression  $e$*

# Sémantique statique et dynamique

On veut que la sémantique statique représente une information sur la sémantique dynamique :

- si le type d'une expression (= la sémantique statique) est  $T$
- alors l'évaluation de cette expression (= la sémantique dynamique) donnera une valeur de type  $T$

Si c'est le cas, le système de typage est **correct** (*sound*)

## Environnement de typage

De manière similaire à nos environnements, on définit des *environnements de typage* :

```
type tenv = (string * typ) list
```

```
let tlookup (env : tenv) (var : string) : typ =  
    List.assoc var env
```

```
let textend (env : tenv) (var : string) (t : typ) : tenv =  
    (var, t) :: env
```

# Vérification de type

De manière similaire à la fonction `eval`, on définit une fonction de typage, `type_of` :

```
let rec type_of (env : tenv) (e : expr) : typ =  
  match e with  
    ...
```

## Vérification de type : constantes

Une constante est typée par le type de constante que c'est

```
let rec type_of (env : tenv) (e : expr) : typ =  
  match e with  
  | Int _ -> TInt  
  | Bool _ -> TBool
```

## Vérification de type : constantes

Mathématiquement, on note cela :

$$\frac{}{\Gamma \vdash \text{Bool } b : \text{TBool}}$$

À lire

- en français : “dans l’environnement de typage  $\Gamma$ , l’expression `Bool b` a le type `TBool`”
- en code : `type_of env (Bool b) = TBool`

(Similaire pour `Int` et `TInt`)

## Règle d'inférence : rappel

Rappel : une règle d'inférence s'écrit avec des conditions au dessus de la barre, et ce qu'on dérive en dessous. Par exemple :

$$\frac{A \implies B \quad B \implies C}{A \implies C}$$

À lire :

- Si  $A$  implique  $B$
- Et si  $B$  implique  $C$
- Alors  $A$  implique  $C$

## Vérification de type : opérations binaires

On a deux opérations binaires : Add et And

```
let rec type_of (env : tenv) (e : expr) : typ =  
  match e with  
  ...  
  | BinOp (op, x, y) ->  
    begin match (op, type_of env x, type_of env y) with  
      | (Add, TInt, TInt) -> TInt  
      | (And, TBool, TBool) -> TBool  
      | _ -> raise TypeError  
    end
```

## Vérification de type : opérations binaires

Mathématiquement :

$$\frac{\Gamma \vdash e_1 : \text{TInt} \quad \Gamma \vdash e_2 : \text{TInt}}{\Gamma \vdash e_1 + e_2 : \text{TInt}}$$

À lire :

- si, dans l'environnement  $\text{env}$ , l'expression  $e_1$  a le type  $\text{TInt}$
- et, dans le même environnement  $\text{env}$ , l'expression  $e_2$  a le type  $\text{TInt}$
- alors, dans le même environnement  $\text{env}$ , l'expression  $e_1 + e_2$  a le type  $\text{TInt}$

À noter : une erreur de type est représentée mathématiquement par l'absence de règle qui peut s'appliquer.

## Vérification de type : conditions

```
let rec type_of (env : tenv) (e : expr) : typ =  
  match e with  
  ...  
  | If (condition, consequence, alternative) ->  
    begin match (type_of env condition,  
                  type_of env consequence,  
                  type_of env alternative) with  
      | (TBool, tcons, talt) when tcons = talt -> tcons  
      | _ -> raise TypeError  
    end
```

## Vérification de type : conditions

$$\frac{\Gamma \vdash e_1 : \text{TBool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}$$

Donc :

- la condition doit être booléenne
- la conséquence et l'alternative doivent avoir le même type, peu importe celui-ci
- le type de l'expression en entier est le même que celui de la conséquence et l'alternative

## Vérification de type : liaisons

```
let rec type_of (env : tenv) (e : expr) : typ =  
  match e with  
  ...  
  | Var var -> tlookup env var  
  | Let (var, value, body) ->  
    let tvar = type_of env value in  
    type_of (textend env var tvar) body
```

## Vérification de type : liaisons

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

Ici,  $\Gamma(x)$  est la recherche dans notre environnement

Donc : une variable est typé par le type qui lui est assigné dans l'environnement

## Vérification de type : liaisons

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}$$

Ici,  $\Gamma, x : T_1$  est une extension de notre environnement de typage.

Donc :

- si  $e_1$  a un type  $T_1$
- et que  $e_2$  a un type  $T_2$ , en prenant un environnement étendu avec  $x$  ayant le type  $T_1$
- alors, l'expression en entier a le type  $T_2$

## Vérification de type : fonctions

```
let rec type_of (env : tenv) (e : expr) : typ =  
  match e with  
  ...  
  | Function (param, tparam, body, tret) ->  
    if type_of (textend env param tparam) body = tret then  
      TFun (tparam, tret)  
    else  
      raise TypeError  
  | Call (f, arg) ->  
    begin match (type_of env f, type_of env arg) with  
      | (TFun (tparam, tret), targ) when tparam = targ -> tret  
      | _ -> raise TypeError  
    end
```

## Vérification de type : fonctions

$$\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\text{fun } x : T_1 \rightarrow e : T_2) : T_1 \rightarrow T_2}$$

Donc :

- si  $e$  a le type  $T_2$  en étendant l'environnement avec  $x$  ayant pour type  $T_1$
- alors la fonction a le type  $T_1 \rightarrow T_2$

## Vérification de type : fonctions

$$\frac{\Gamma \vdash f : T_1 \rightarrow T_2 \quad \Gamma \vdash e : T_1}{\Gamma \vdash f \ e : T_2}$$

Donc :

- si la fonction a le type  $T_1 \rightarrow T_2$
- et son argument a le type  $T_1$  correspondant
- alors le résultat de son application a le type  $T_2$

# Système de typage au complet

$$\begin{array}{c} \frac{}{\Gamma \vdash \text{Bool } b : \text{TBool}} \quad \frac{}{\Gamma \vdash \text{Int } n : \text{TInt}} \quad \frac{\Gamma \vdash e_1 : \text{TInt} \quad \Gamma \vdash e_2 : \text{TInt}}{\Gamma \vdash e_1 + e_2 : \text{TInt}} \\[10pt] \frac{\Gamma \vdash e_1 : \text{TBool} \quad \Gamma \vdash e_2 : \text{TBool}}{\Gamma \vdash e_1 \text{ and } e_2 : \text{TInt}} \quad \frac{\Gamma \vdash e_1 : \text{TBool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \quad \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \\[10pt] \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2} \quad \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\text{fun } x : T_1 \rightarrow e : T_2) : T_1 \rightarrow T_2} \\[10pt] \frac{\Gamma \vdash f : T_1 \rightarrow T_2 \quad \Gamma \vdash e : T_1}{\Gamma \vdash f e : T_2} \end{array}$$

## Vérification de type : le tout ensemble

```
# type_of [] (Function ("x", TInt,  
                        (Function ("y", TInt,  
                                BinOp (Add, Var "x", Var "y"), TInt)),  
                                TFun (TInt, TInt))));;  
- : typ = TFun (TInt, TFun (TInt, TInt))
```

# Inférence de type

# Inférence de type

On a du annoter les types des fonctions, on souhaite maintenant *inférer* leur type.

## Inférence de type : exemple

Quel est le type de l'expression suivante ?

```
fun x -> fun y -> x + y
```

## Inférence de type : exemple

Quel est le type de l'expression suivante ?

```
fun x -> fun y -> x + y
```

- du +, on sait que  $x : \text{int}$  et  $y : \text{int}$
- on sait aussi que  $(x + y) : \text{int}$
- donc le type de `fun y -> x + y` est  $\text{int} \rightarrow \text{int}$
- donc le type de l'expression est  $\text{int} \rightarrow \text{int} \rightarrow \text{int}$

## Inférence de type : exemple

Quel est le type de l'expression suivante ?

```
fun f -> fun g -> (f 1) + (g true)
```

## Inférence de type : exemple

Quel est le type de l'expression suivante ?

```
fun f -> fun g -> (f 1) + (g true)
```

- f a la forme `int -> t1` pour un certain `t1`
- g a la forme `bool -> t2` pour un certain `t2`
- `t1` doit être égal à `int` pour le `+`
- `t2` doit être égal à `int` pour le `+`
- `fun g -> ...` est donc de type `(bool -> int) -> int`
- `fun f -> ...` est donc de type `(int -> int) -> (bool -> int) -> int`

# Inférence de type : algorithme

On souhaite rendre ce raisonnement exécutable.

- On va utiliser une structure similaire à une fonction d'évaluation : on descend récursivement dans l'arbre syntaxique
- On va se souvenir des contraintes à satisfaire
- Une fois fini de traverser l'expression, on peut résoudre les contraintes

## Inférence de type : exemple

```
fun f -> fun g -> (f 1) + (g true)
```

On considère `fun f -> ...`

- `fun f -> ...` est de type  $t_1 \rightarrow t_2$
- On ne connaît pas le type de `...`, donc on fait un appel récursif
  - Dans l'appel récursif, on retient que  $f : t_1$
  - Si le type de `...` est  $t$ , alors on a la contrainte  $t_2 = t$

## Inférence de type : exemple

```
fun f -> fun g -> (f 1) + (g true)
```

On considère `fun g -> ...`

- `fun g -> ...` est de type  $t_3 \rightarrow t_4$ 
  - Donc on aura la contrainte  $t_2 = t_3 \rightarrow t_4$
- On ne connaît pas le type de `...`, donc on fait un appel récursif
  - Dans l'appel récursif, on retient que  $g : t_3$
  - Si le type de `...` est  $t$ , alors on a la contrainte  $t_4 = t$

## Inférence de type : exemple

```
fun f -> fun g -> (f 1) + (g true)
```

On considère  $(f\ 1) + (g\ true)$

- On sait que  $(+): int \rightarrow int \rightarrow int$
- Si le type de  $(f\ 1)$  est  $t$ , on aura la contrainte  $t = int$
- Si le type de  $(g\ true)$  est  $t$ , on aura la contrainte  $t = int$
- Le type de l'expression en entier est  $int$ 
  - Donc on aura la contrainte  $t_4 = int$

## Inférence de type : exemple

```
fun f -> fun g -> (f 1) + (g true)
```

On considère  $f\ 1$

- On a une application de fonction, donc  $f$  doit être de type  $t5 \rightarrow t6$ 
  - On a retenu que  $f : t1$ , donc on ajoute la contrainte  $t1 = t5 \rightarrow t6$
- Appel récursif pour savoir le type de  $1$ , qui est `int`
  - On ajoute donc la contrainte  $t5 = \text{int}$
- Le type de l'expression est  $t6$ 
  - On a donc la contrainte  $t6 = \text{int}$

## Inférence de type : exemple

```
fun f -> fun g -> (f 1) + (g true)
```

On considère `g true`

- On a une application de fonction, donc `g` doit être de type `t7 -> t8`
  - On a retenu que `g : t3`, donc on a ajouté la contrainte `t3 = t7 -> t8`
- Appel récursif pour savoir le type de `true`, qui est `bool`
  - On ajoute donc la contrainte `t7 = bool`
- Le type de l'expression en entier est `int`
  - On a donc la contrainte `t8 = int`

## Inférence de type : exemple

On a visité l'expression en entier, qui est de type  $t1 \rightarrow t2$  avec les contraintes suivantes :

- $t1 = t5 \rightarrow t6$
- $t2 = t3 \rightarrow t4$
- $t3 = t7 \rightarrow t8$
- $t4 = \text{int}$
- $t5 = \text{int}$
- $t6 = \text{int}$
- $t7 = \text{bool}$
- $t8 = \text{int}$

## Inférence de type : exemple

On peut résoudre ces contraintes pour trouver

- $t1 = \text{int} \rightarrow \text{int}$
- $t2 = (\text{bool} \rightarrow \text{int}) \rightarrow \text{int}$
- $t3 = \text{bool} \rightarrow \text{int}$
- $t4 = \text{int}$
- $t5 = \text{int}$
- $t6 = \text{int}$
- $t7 = \text{bool}$
- $t8 = \text{int}$

Donc, le type de notre expression est  $(\text{int} \rightarrow \text{int}) \rightarrow (\text{bool} \rightarrow \text{int}) \rightarrow \text{int}$

# Langage

On va considérer le langage suivant (pas d'annotation de types) :

```
type expr =  
  | Int of int  
  | Bool of bool  
  | Var of string  
  | If of expr * expr * expr  
  | Let of string * expr * expr  
  | Function of string * expr  
  | Call of expr * expr
```

Note : on va considérer les opérations binaires comme définies dans l'environnement, comme si c'était des fonctions

# Types

On introduit un nouveau type, qui ne sera présent que pour l'inférence

```
type typ =  
  | TInt  
  | TBool  
  | TFun of typ * typ  
  | TFresh of int
```

TFresh 1 représente la variable de type t1

# Contraintes

Une contrainte est de la forme  $t1 = t2$  où  $t1$  et  $t2$  sont des types.

On va manipuler des ensembles de contraintes, représentés par des listes :

```
type constraints = (typ * typ) list
```

# Inférence

La signature de `type_of` est différente :

- on doit générer des nouvelles variables de type, `t1`, `t2`, ...
  - on va maintenir un état pour se souvenir du prochain nombre à utiliser
  - on prend donc un `int` en argument
  - et on renvoie un `int`
- on génère non seulement un type, mais aussi des contraintes

```
let rec type_of (env : tenv) (fresh_count : int) (e : expr)
: typ * constraints * int =
...
```

## Inférence : constantes

```
let rec type_of (env : tenv) (fresh_count : int) (e : expr)
  : typ * constraints * int =
match e with
| Int _ -> TInt, [], fresh_count
| Bool _ -> TBool, [], fresh_count
```

## Inférence : conditions

```
let rec type_of (env : tenv) (fresh_count : int) (e : expr)
: typ * constraints * int =
match e with
...
| If (condition, consequence, alternative) ->
  let tcond, c1, fresh_count = type_of env fresh_count condition in
  let tcons, c2, fresh_count = type_of env fresh_count consequence in
  let talt, c3, fresh_count = type_of env fresh_count alternative in
  let fresh_t, fresh_count = TFresh fresh_count, fresh_count + 1 in
  fresh_t,
  [(tcond, TBool); (fresh_t, tcons); (fresh_t, talt)] @ c1 @ c2 @ c3,
  fresh_count
```

## Inférence : liaisons

```
let rec type_of (env : tenv) (fresh_count : int) (e : expr)
  : typ * constraints * int =
  match e with
  ...
  | Var var -> tlookup env var, [], fresh_count
  | Let (var, value, body) ->
    let tvar, c1, fresh_count = type_of env fresh_count value in
    let tbody, c2, fresh_count =
      type_of (textend env var tvar) fresh_count body in
    tbody, c1 @ c2, fresh_count
```

## Inférence : définition de fonction

```
let rec type_of (env : tenv) (fresh_count : int) (e : expr)
  : typ * constraints * int =
  match e with
  ...
  | Function (param, body) ->
    let fresh_t, fresh_count = TFresh fresh_count, fresh_count + 1 in
    let tenv' = textend env param fresh_t in
    let tbody, c, fresh_count = type_of tenv' fresh_count body in
    TFun (fresh_t, tbody), c, fresh_count
```

## Inférence : appel de fonction

```
let rec type_of (env : tenv) (fresh_count : int) (e : expr)
  : typ * constraints * int =
  match e with
  ...
  | Call (f, arg) ->
    let fresh_t, fresh_count = TFresh fresh_count, fresh_count + 1 in
    let tf, c1, fresh_count = type_of env fresh_count f in
    let targ, c2, fresh_count = type_of env fresh_count arg in
    fresh_t, [(tf, TFun (targ, fresh_t))] @ c1 @ c2, fresh_count
```

## Inférence : résultat

```
# let expr =  
  (Function ("f",  
    (Function ("g",  
      Call (Call (Var "+",  
        (Call (Var "f", Int 1))),  
        Call (Var "g", Bool true)))))) in  
let env = [("+", TFun (TInt, (TFun (TInt, TInt)))] in  
let (t, cs, _) = type_of env 0 expr in  
(typ_to_string t), (constraints_to_string cs));  
  
(t0 -> (t1 -> t2)) with constraints  
t3 = (t5 -> t2),  
(int -> int) = (t4 -> t3),  
t0 = (int -> t4),  
t1 = (bool -> t5);
```

# Unification

# Unification

Comment résoudre automatiquement les contraintes ?

Exemple :

- $t1 = t5 \rightarrow t6$
- $t2 = t3 \rightarrow t4$
- $t3 = t7 \rightarrow t8$
- $t4 = \text{int}$
- $t5 = \text{int}$
- $t6 = \text{int}$
- $t7 = \text{bool}$
- $t8 = \text{int}$

On effectue simplement des substitutions !

# Unification : exemple

Un autre exemple :

- $t1 \rightarrow t2 = t3 \rightarrow \text{int}$
- $t3 \rightarrow t2 = \text{int} \rightarrow t1$

# Unification : exemple

Un autre exemple :

- $t1 \rightarrow t2 = t3 \rightarrow \text{int}$
- $t3 \rightarrow t2 = \text{int} \rightarrow t1$

Cela revient à résoudre :

- $t1 = t3$
- $t2 = \text{int}$
- $t3 = \text{int}$
- $t2 = t1$

Et on effectue ensuite des substitutions.

# Substitution

On note  $t[x \mapsto y]$  pour dire que  $x$  est substitué par  $y$  dans le terme  $t$  (un type par exemple).

Exemple :

$$(t1 \rightarrow t2)[t1 \mapsto \text{int}] = \text{int} \rightarrow t2$$

## Application de substitution

On peut définir l'application de substitution mathématiquement :

$$\text{int}[a \mapsto x] = \text{int}$$

$$\text{bool}[a \mapsto x] = \text{bool}$$

$$a[a \mapsto x] = x$$

$$b[a \mapsto x] = b$$

$$(t1 \rightarrow t2)[a \mapsto x] = (t1[a \mapsto x]) \rightarrow (t2[a \mapsto x])$$

# Composition séquentielle de substitutions

On peut séquencer plusieurs substitutions :

$$t([a \mapsto x]; [b \mapsto y]) = (t[a \mapsto x])[b \mapsto y]$$

## Application à des contraintes

On peut appliquer une substitution à une contrainte :

$$(t1 = t2)[a \mapsto x]$$

devient :

$$t1[a \mapsto x] = t2[a \mapsto x]$$

## Substitutions : implémentation

Au niveau de l'implémentation, on représente une substitution comme un environnement

```
type subst = (int * typ) list
```

On peut remplacer une variable avec une substitution

```
let subst_var (subst : subst) (x : int) : typ =  
  match List.assoc_opt x subst with  
  | Some y -> y  
  | None -> TFresh x
```

## Substitution : appliquer une substitution

On peut appliquer une substitution à un type

```
let rec apply_subst (subst : subst) (t : typ) : typ =  
  match t with  
  | TInt -> TInt  
  | TBool -> TBool  
  | TFun (t1, t2) -> TFun (apply_subst subst t1, apply_subst subst t2)  
  | TFresh x -> subst_var subst x
```

Et on peut appliquer une substitution à des contraintes

```
let apply_to_constraints (subst : subst) (cs : constraints) : constraints =  
  List.map (fun (t1, t2) -> (apply_subst subst t1, apply_subst subst t2))  
    cs
```

## Unification des contraintes

On dit qu'une substitution "unifie" un contrainte si on obtient une égalité syntaxique en l'appliquant à gauche et à droite :

Soit la contrainte

$$t1 \rightarrow (t1 \rightarrow int) = int \rightarrow t2$$

Elle est unifiée avec la substitution  $\rho = [t1 \mapsto int; t2 \mapsto int \rightarrow int]$ , car :

$$(t1 \rightarrow (t1 \rightarrow int)) \rho = (int \rightarrow t2) \rho$$

$$(int \rightarrow (int \rightarrow int)) = (int \rightarrow t2) \rho$$

$$(int \rightarrow (int \rightarrow int)) = (int \rightarrow (int \rightarrow int))$$

# Unification d'un ensemble de contraintes

Une substitution unifie un ensemble de contrainte si elle unifie chacune des contraintes dans l'ensemble

Le **but de l'unification** est de trouver une substitution qui unifie un ensemble de contraintes

# Algorithme d'unification

Une approche est la suivante : on traite les contraintes une à la fois, et on construit une substitution. Pour chaque contrainte  $t_1 = t_2$  :

- si on a  $\text{int} = \text{int}$  ou  $\text{bool} = \text{bool}$ , on passe à la suite
- si on a  $t_1 = \text{TFresh } x$ , avec  $x$  qui n'apparaît pas dans  $t_2$  :
  - on ajoute  $t_1 \mapsto x$  à la substitution et on continue
- similaire si  $t_2 = \text{TFresh } x$  avec  $\text{TFresh } x$  qui n'apparaît pas dans  $t_1$
- si on a  $a \rightarrow b = c \rightarrow d$  :
  - on sépare la contrainte en deux :  $a = c$  et  $b = d$ , et on continue
- sinon, les termes ne sont pas unifiables

À la fin, on retourne la substitution créée

On peut démontrer que cet algorithme termine toujours, et donne l'unificateur le plus général  
C'est l'algorithme de Martelli et Montanari (1982)

## Algorithme d'unification : exception

Si on n'arrive pas à unifier, on lèvera l'exception suivante :

**exception** `UnificationError`

## Algorithme d'unification : `appears_in`

On souhaite savoir si une variable de type apparaît dans un type :

```
let rec appears_in (var : int) (typ : typ) : bool =  
  match typ with  
  | TInt | TBool -> false  
  | TFun (t1, t2) -> (appears_in var t1) || (appears_in var t2)  
  | TFresh x -> x = var
```

## Algorithme d'unification : entre deux types

```
let rec unify1 (subst : subst) (left : typ) (right : typ) : subst =  
  match left, right with  
  | TInt, TInt -> subst  
  | TBool, TBool -> subst  
  | TFresh x, t2 | t2, TFresh x ->  
    if appears_in x t2 then  
      raise UnificationError  
    else  
      (x, t2) :: subst  
  | TFun (t1, t2), TFun (t3, t4) -> unify1 (unify1 subst t1 t3) t2 t4  
  | _ -> raise UnificationError
```

## Algorithme d'unification : pour un ensemble de contraintes

```
let rec unify (subst : subst) (cs : constraints) : subst =  
  match cs with  
  | [] -> subst  
  | (t1, t2) :: rest ->  
    let subst' = unify1 subst t1 t2 in  
    unify subst' (apply_to_constraints subst' rest)
```

## Le tout ensemble

```
(* function f -> function g -> (f 1) + (g true) *)
let expr =
  (Function ("f",
    (Function ("g",
      Call (Call (Var "+",
        (Call (Var "f", Constant (Int 1)))),
        Call (Var "g", (Constant (Bool true))))))))) in

(* (+) : int -> int -> int *)
let env = [("+", (TFun (TInt, TFun (TInt, TInt))))] in
(* inférence *)
let typ, constraints, _ = type_of env 1 expr in
(* résolution des contraintes *)
let subst = unify [] constraints in
(* application de la substitution *)
apply_subst subst typ
```

# Polymorphisme

Que se passe-t-il avec des fonctions polymorphes ?

```
let id = fun x -> x in  
id 3
```

On obtient `id : int -> int`

# Polymorphisme

En enlevant son appel :

```
fun x -> x
```

On obtient une erreur, car c'est sous-contraint !

# Polymorphisme

Autre variation :

```
let id = fun x -> x in  
let a = id 3 in  
id true
```

On obtient une erreur !

## En pratique

- OCaml utilise un système de typage à la Hindley-Milner (HM)
- Dans HM, le polymorphisme est restreint aux *let* (*let-bound polymorphism*)
- L'algorithme W permet d'inférer les types d'un système de typage HM, et est utilisé par OCaml
  - Il entrelace la génération de contrainte et leur résolution

# Concepts importants

- **Typage statique vs. dynamique** : en typage statique, les types sont vérifiés *avant* l'exécution
- **Environnement de typage** : comme un environnement pour l'évaluation, mais qui donne de l'information de typage
- **Vérification de type** : le fait de s'assurer que le programme est *bien typé*
- **Inférence de type** : le fait de déduire les types d'un programme
- **Substitution** : concept mathématique formalisant la notion de remplacement
- **Unification** : la recherche d'une substitution qui rend deux termes égaux
- **Résolution de contraintes de typage** : peut se faire par unification