Introduction à la sémantique et au typage de ML

M.V. Aponte

Année 2007/2008

Le langage ML

- Langage fonctionnel fortément typé, d'une grande généralité,
- décliné en plusieurs dialectes: Standard ML, Ocaml, Moscow ML, etc.

Nous travaillerons sur un sous-ensemble très réduit: mini-ML, avec les traits fondamentaux communs à tous les langages issus de ML.

Un dialect ML: Ocaml

- C'est un langage complet, concis et puissant:
    - traits fonctionnels, et *impératifs*,
    - système de *modules* génériques,
    - Programmation *objet* sophistiquée,
    - nombreuses librairies

- C'est un langage sûr et facile à employer:
  typage *fort* et *inférence des types.*

Le langage Ocaml

- Il possède un noyau fonctionnel simple et un mécanisme de *programmation par filtrage* qui généralise la programmation pas cas.
- Ocaml est langage *compilé* qui possède une *mode intéractif*, ce qui facilite les tests.
- *Gestion mémoire automatique* (comme en Java).
- La sémantique d'Ocaml est bien définie: il n'y a pas des points obcurs ou mystérieux!

Documentation, distribution et plus de détails:
`http://caml.inria.fr/ocaml`.

Un peu d'histoire

Ocaml est un langage de la famille ML, développé depuis les années 80.

1975 : Robin Milner propose ML comme mèta-langage (langage de script) pour l'assistant de preuve LCF. Devient rapidement un langage de programmation à part entière.

1985 : Développement de Caml à l'INRIA, de Standard ML (Edinburgh), de "SML of New-Jersey", de Lazy ML(Chalmers), de Haskell (Glasgow), etc.

1990-95 : Implantation de Caml-Ligh, compilateur vers du code natif + système de modules.

1996 : Objets et classes (OCaml).

2001-... : Arguments étiquettés, Méthodes polymorphes, librairies partagées, Modules récursifs, private types, ...

Domaines d’utilisation

Un langage d’usage général avec des **domaines de prédilection**:

- Calcul symbolique: Preuves mathématiques, compilation, interprétation, analyse de programmes.
- Prototypage rapide. Langage de script. Langages dédiés.
- Programmation distribuée (bytecode rapide).

...et utilisé en **enseignement et recherche**:

- Classes préparatoires.
- Utilisé dans de grandes universités (Europe, US, Japon, etc.).

Domaines d'utilisation

Mais aussi...

- en Industrie: Startups, CEA, EDF, France Telecom, Simulog,..
- des Gros Logiciels: Coq, Ensemble, ASTREE, (voir la bosse d'OCaml)

Plus récemment, outils système:

- Unison,
- MLdonkey (peer-to-peer),
- Libre cours (Site WEB),
- Lindows (outils système pour une distribution de Linux)

Le mode compilé

Éditer le source

```
hello.ml
print_string  "Hello World!\n";;
```

Compiler, lier et exécuter (sous le Shell d'Unix):

```
ocamlc -o hello hello.ml
./hello
Hello World!
```

Le mode intéractif

Ocaml possède un mode intéractif où il analyse et répond à chaque phrase entrée par l'utilisateur.

On tape la commande `ocaml`:

```
% ocaml
        Objective Caml version 3.06

#
```

Le caractère `#` invite l'utilisateur à entrer une phrase écrite dans la syntaxe Ocaml.

Le mode intéractif(suite)

- Ocaml analyse chaque phrase:
  - calcule son type (*inférence des types*),
  - la traduit en langage exécutable (*compilation*)
  - et enfin l'*exécute* afin de fournir la réponse demandée.

- La réponse donnée en mode intéractif contient:
  - Le nom de la variable déclarée s'il y en a.
  - Le type trouvé pendant le typage.
  - La valeur calculée après exécution.

Exemple

```
# let x = 4+2;;
val x : int = 6
```

La réponse d'Ocaml signale :

- l'identificateur `x` est déclaré `(val  x)`,
- avec le type des entiers `(:int)`,
- et la valeur 6 `(=6)`.

L'identificateur `x` est lié à la valeur 6 de type `int`:

```
# x;;
val x : int = 6

# x * 3;;
- : int = 18
```

Les types de base

<table>
<tr><th>Type</th><th>Constantes</th><th>Primitives</th></tr>
<tr><td>unit</td><td>()</td><td>pas d'opération!</td></tr>
<tr><td>bool</td><td>true false</td><td>&& || not</td></tr>
<tr><td>char</td><td>'a' '\n' '\097'</td><td>code, chr</td></tr>
<tr><td>int</td><td>1 2 3</td><td>+ - * / max_int</td></tr>
<tr><td>float</td><td>1.0 2. 3.14</td><td>+. -. *. /. cos</td></tr>
<tr><td>string</td><td>"a\tb\010c\n"</td><td>^ s.[i] s.[i] <- c</td></tr>
</table>

Comparaison (pour tous les types) `=, > , < , >=, <=, <>`.

Exemples

```
# 1+ 2;;
- : int = 3

# 1.5 +. 2.3;;
- : float = 3.8

# let x = "cou" in x^x;;
- : string = "coucou"

# 2 > 7;;
- : bool = false

# "bonjour" > "bon";;
- : bool = true
```

Les programmes

Un programme est une suite de phrases:

définition de valeur	`let` x = e
définition de fonction	`let` f x1 ... xn = e
définition de fonctions	`let [ rec ]` f1 x1 ... = e1 ...
(mutuellement récursives)	`[ and` fn xn ... = en]
définition de type(s)	`type` q1 = t1... [`and` qn = tn]
expression	e

Les phrases se terminent par ;; optionnel entre des déclarations, mais obligatoires sinon.

Exemples de programmes

```
# let baguette = 4.20;;                    (* declaration
val baguette : float = 4.2

# let euro x =  x /. 6.55957;;             (* declaration
val euro : float -> float = <fun>

# euro baguette;;                          (* expression
- : float = 0.640285872397123645
```

Les expressions

définition locale	`let` x = e1 in e2
fonction anonyme	`fun` x1 ... xn `->` e
appel de fonction	f e1 ... en
variable	`x`
	(`M.x` si x est défini dans le module M)
valeur construite	(e1, e2)
(dont les constantes)	1, 'c', "aa"
analyse par cas	`match` e `with` p1 `->` e1 ... | pn `->` en
boucle for	`for` i = e0 `to` ef `do` e `done`
boucle while	`while` e0 `do` e `done`
conditionnelle	`if` e1 `then` e2 `else` e3
une séquence	e; e '
parenthèses	(e) `begin` e `end`

Remarques

- Il n'y a ni intructions, ni procédures.
- Tout sous-programme est une fonction.
- Toute expression retourne une valeur.
- En dehors des déclarations (des valeurs, des classes, des modules, des types), toute phrase Ocaml est une expression et dénote ainsi une valeur résultat.
- Les fonctions sont des valeurs comme les autres.

Les valeurs en Ocaml

- objets,
- valeurs de base (de type int, bool, string, . . .),
- fonctions,
- valeurs de types construits (tuples, enregistrements, listes, variants, . . .).

Ocaml: expressions

```
# 3;;                 (* constante * )
- : int = 3

# 3+4;;
- : int = 7

# (1, 3+4);;          (* paire  *)
- : int * int = (1, 7)

# (true, 1+4);;       (* paire  *)
- : bool * int = (true, 5)

# fst(true,2);;       (* 1ere projection *)
- : bool = true

# snd(true,2);;
- : int = 2
```

Ocaml

Définitions locales:

```
# let x = 3 in x+2;;
- : int = 5

# let x = 3 in let y = 4 in (x,y);;
- : int * int = (3, 4)
```

Ocaml: fonctions

Un identificateur de fonction est déclaré comme tout autre, à l'aide d'un `let`.

```
# let double y = y*2;;
val double : int -> int = < fun >
```

Une syntaxe alternative pour <u>la même fonction</u>:

```
# let double (y) = y*2;;
val double : int -> int = < fun >
```

La première est la syntaxe la plus utilisée en Ocaml.

Le type des fonctions (un argument)

```
# let double y = y*2;;
val double : int -> int = < fun >
```

Le type d'une fonction est noté `t -> q`, où `t`: type de l'argument, `q` type du résultat.

L'identificateur `double` est lié à une *valeur fonctionnelle*:

```
# double;;
val double : int -> int = < fun >
```

Application de fonctions

On **applique** une fonction en la faisant suivre de son argument (éventuellement entre parenthèses). Elle **retourne** un résultat:

```
# double 9;;
- : int = 18

# double(9);;
- : int = 18
```

Ocaml: fonctions anonymes

Construction d'une fonction: le mot-clé `fun` introduit chaque argument.

```
# fun x -> x+1;;
- : int -> int = <fun>
```

Construction puis application

```
# (fun x -> x+1) 2;;
- : int = 3

# let f = (fun x -> x+1) in f 2;;
- : int = 3
```

Ocaml: fonctions anonymes

Construction d'une fonction: le mot-clé `fun` introduit chaque argument. Fonctions à 2 arguments

```
# fun x -> fun y -> x+y;;
- : int -> int -> int = <fun>

(* application partielle *)

# (fun x -> fun y -> x+y) 3 ;;
- : int -> int = <fun>

(* application totale *)
# (fun x -> fun y -> x+y) 3 4;;
- : int = 7
```

Ocaml: fonctions en argument

```
#  fun f -> fun x -> 2* f(x);;
- : ('a -> int) -> 'a -> int = <fun>

# (fun f -> fun x -> 2*f(x)) String.length;;
- : string -> int = <fun>

# (fun f -> fun x -> 2*f(x)) String.length "coucou";;
- : int = 12
```

Expression conditionnelle

Une conditionelle a toujours deux branches: les expressions dans ces branches doivent être de même type.

```
# if true then "vrai" else "faux";;
- : string = "vrai"
```

```
# let x = 7 in if x>2 then 1 else true;;
Characters 32-36:
  let x = 7 in if x>2 then 1 else true;;
                                  ^^^^
This expression has type bool but is here used
 with type int
```

N-uplets

Un n-uplet est un “paquet” de n valeurs $v_1, v_2, \ldots v_n$ separées par des virgules.

Exemples: (1,true) est un 2-uplet; ("bonjour", 5, ’c’) est un triplet.

```
# let a = (1, true);;
val a : int * bool = (1, true)

# let livre = ("Paroles", "Prevert, Jacques", 1932);;
val livre : string * string * int =("Paroles", "Prevert,
```

N-uplets (suite)

Un n-uplet permet de mettre dans un “paquet” autant de valeurs que l’on veut. Cela est pratique, si une fonction doit renvoyer “plusieurs” résultats:

```
# let f x y = (x+y, x*y);;
val f : int -> int -> int * int = <fun>

# f 2 3;;
- : int * int = (5, 6)

# let division_euclidienne x y = (x/y, x mod y);;
val division_euclidienne : int -> int -> int * int = <fu

# division_euclidienne 5 2;;
- : int * int = (2, 1)
```

Type des n-uplets

Le type d'un n-uplet $(v_1, v_2, \ldots, v_n)$ est $t_1 * t_2 \ldots * t_n$,
où t_i est le type de la composante v_i.

```
# let a = (1, true);;
val a : int * bool = (1, true)

# let b = ("bonjour", 5, 'c');;
val b : string * int * char = ("bonjour", 5, 'c')
```

Type de fonctions avec n-uplet d'arguments

$$\texttt{let f}(x_1, x_2, \ldots, x_n) = \textit{corps}$$

possède 1 argument n-uplet $(x_1, x_2, \ldots\ x_n)$ et un résultat calculé par *corps*.

Le type du n-uplet $(x_1, x_2, \ldots, x_n)$ est $t_1 * t_2 \ldots * t_n$,
où t_i est le type de chaque x_i. Donc, le type de f est:

$$t_1 * t_2 \ldots * t_n \rightarrow t_{corps}$$

où t_{corps} est le type du résultat calculé par *corps*.

Inférence de types

- En Ocaml, il n'est pas nécessaire de déclarer les types des identificateurs ou des arguments d'une fonction, intervenant dans une définition.
- Le typeur infère leur type d'après leur contexte.

Exemple d'inférence de types

```
let f(x) = x + 1
```

Sachant que $+$ est définit uniquement sur les entiers, et son type est $+$: $\mathtt{int} * \mathtt{int} \rightarrow \mathtt{int}$, le typeur déduit les contraintes:

- le type de `f` est de la forme:

  $$\mathtt{f} : t_x \rightarrow t_{corps}$$

  où t_x est le type de x argument de la fonction, et t_{corps} est le type du corps.

- de `x+1` on déduit que `x` doit être de type `int` $\Rightarrow \boxed{t_x = \mathtt{int}}$,
- si ces contraintes sont respectées, alors le corps `x+1` est de type `int` $\Rightarrow \boxed{t_{corps} = \mathtt{int}}$.

Conclusion: $t_x = \mathtt{int}$ et $t_{corps} = \mathtt{int}$, donc

$$\mathtt{f} : \mathtt{int} \rightarrow \mathtt{int}$$

$\square$

Polymorphisme

Certaines définitions n'imposent aucune contrainte (ou peu) sur les types des objets manipulés. Dans ce cas, leur type peut-être quelconque. On parle alors de *polymorphisme paramétrique*.

```
let premier (x,y) = x
```

- `premier` : $t_x * t_y \rightarrow t_{corps}$, où t_x, t_y et t_{corps} sont les types de `x`, de `y` et du corps de la fonction.
- La seule contrainte imposée par le corps, est que son type soit le même que celui de la première composante de la paire argument, i.e: $\boxed{t_{corps} = t_x}$,

Polymorphisme(suite)

Aucune autre contrainte ne se dégage de l'examen du code ici. D'après les hypothèses de départ, on obtient:

$$\texttt{premier} : t_x * t_y \rightarrow t_x$$

t_x, t_y ne sont pas de véritables types mais des *variables de type*.

Polymorphisme (suite)

Puisqu'aucune contrainte ne pèse sur t_x, t_y, on pourrait choisir n'importe quel type à leur place et on obtiendrait un typage cohérent pour la fonction `premier`. Par exemple, on peut choisir parmi les typages:

$$\texttt{premier} : \begin{cases} \texttt{int} * \texttt{bool} \to \texttt{int} \\ (\texttt{int} \to \texttt{int}) * \texttt{string} \to (\texttt{int} \to \texttt{int}) \\ \texttt{couleur} * \texttt{int} \to \texttt{couleur} \\ \vdots \end{cases}$$

Polymorphisme

Un type polymorphe n'est pas un type “passe-partout”. Par exemple, les utilisations suivantes de `premier` sont mal typées:

```
# let premier (x,y) = x;;
val premier : 'a * 'b -> 'a = <fun>

# premier 1;;
This expression has type int but is here used
with type 'a * 'b

# (premier(true,2)) + 3;;
This expression has type bool but
is here used with type int

# premier (1,2,3);;
This expression has type int * int * int but
is here used with type 'a * 'b
```

Exemples de constructions polymorphes prédéfinies

```
#fst;;
- : 'a * 'b -> 'a = <fun>

# (=);;
- : 'a -> 'a -> bool = <fun>

# (>);;
- : 'a -> 'a -> bool = <fun>
```

Fonctions avec types polymorphes

```
# let identite = fun x -> x;;
val identite : 'a -> 'a = <fun>

# identite 1;;
- : int = 1

# identite true;;
- : bool = true

# let id = fun x -> x in (id 1, id true);;
- : int * bool = (1, true)
```

Fonctions avec types polymorphes

```
# let first = fun (x,y) -> x;;
val first : 'a * 'b -> 'a = <fun>

# first("coucou",1);;
- : string = "coucou"

# let double = fun f -> fun x -> f(f x);;
val double : ('a -> 'a) -> 'a -> 'a = <fun>

# let succ = fun x -> x+1 in double succ 3;;
- : int = 5

# let cou = fun x -> x^"cou" in double cou "le ";;
- : string = "le coucou"
```

Le langage Mini-ML

Les expressions de Mini-ML sont notées $a, a_1, \ldots$. Leur syntaxe est:

Expressions:

a	$::=$	c	constante
	$\mid$	x	nom de variable
	$\mid$	op	opérateur primitif
	$\mid$	$\mathtt{fun}\ x \rightarrow a$	construction d'une fonction
	$\mid$	$a_1\ a_2$	application de fonction
	$\mid$	(a_1, a_2)	création d'une paire
	$\mid$	$\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$	déclaration locale

Syntaxe de Mini-ML

Les constantes sont celles des types de base (`int, bool, unit` ...). Les opérateurs *op* sont les symboles d'opérations primitives (+,-, etc.).

Exemples d'expressions:

+ (1,2)	calcul de 1 plus 2 (en notation infixe
`fun` y → + (y,1)	la fonction $y \rightarrow y + 1$
(`fun` y → + (y,1)) 3	construction puis application à 3
`fun` x → `fun` y → x+y	fonction à 2 arguments
(`fun` x → `fun` y → x+y) 2	application partielle
(`fun` x → `fun` y → x+y) 2 3	application totale
`fun` f → `fun` y → f (+ (y,1))	la fonction $f \rightarrow y \rightarrow f(y + 1)$
`let p = fun` x → +(x,2) `in p` 5	la fonction $\mathtt{p}(x) \rightarrow x + 2$ appliquée

Etude de Mini-ML

1. La sémantique de l'évaluation de mini-ML,
2. Le typage de mini-ML, monomorphe dans un premier temps, puis polymorphe.
3. La sûreté du typage décrit précedemment (si possible).
4. L'algorithme d'inférence de types polymorphes pour mini-ML (si possible).

Sémantique d'évaluation de Mini-ML

- Nous utilisons un système de règles d'inférence pour définir une *relation entre expressions et leurs résultats*.
- Ce sera une *relation de valuation* permettant de dire si oui ou non tel expression peut s'évaluer en tel résultat.

Exemple: nous voudrons associer l'expression 3+1 avec sa valuation 4. Nous l'appelerons *jugement d'évaluation*.

Jugement d'évaluation

- Les expressions de Mini-ML ne sont pas *closes*: elles contiennent des variables, mais aussi des symboles d'opérations.
- Les valeurs à donner à de tels symboles, se trouve dans un contexte *extérieur* à l'expression.

Un jugement d'évaluation fait ainsi intervenir:

- le contexte d'évaluation (valeurs des symboles externes), noté e,
- l'expression évaluée, notée a,
- et la valeur obtenue en résultat, notée v (pour les valeurs) ou plus génŕalement r (résultats).

Jugement d'évaluation

Un *jugement d'évaluation* aura la forme:

$$e \vdash a \Rightarrow r$$

qu'on lira: *dans l'environnement d'évaluation e, l'expression a s'évalue dans le résultat* r.

Les valeurs

Trois sortes de valeurs possibles:

- Les valeurs de base (types de base de la syntaxe: entiers, boléens, etc).
- Les paires de valeurs, correspondant à l'évaluation d'une paire d'expressions,
- La valeur d'une fonction, qu'on appele *fermeture*.

Un contexte prédéfini, appelée Predef, fournit les défitions des opérateurs primitifs, tels que $+$

Exemples de valuations:

Predef $\vdash$ +(3,4)	$\Rightarrow$ 7
[x $\leftarrow$ 3] $\vdash$ (x,4)	$\Rightarrow$ (3,4)
Predef $\{y \leftarrow 3\}$ $\vdash$ +(y,2)	$\Rightarrow$ 5
Predef $\{y \leftarrow 3\}$ $\vdash$ (`fun` x $\rightarrow$ +(x,y))1	$\Rightarrow$ 4

Les fonctions

- Une fonction en mini-ML est définie par le nom d'une variable (paramètre formel), et par une expression (corps).
- tout paramètre x est introduit par la construction `fun` $x \to c$, et x est dite *liée* dans le corps c;
- le corps peut contenir des variables autres que les paramètres formels;
- ces variables ne sont pas liées par aucune construction `fun`. On dit qu'elles sont *libres dans la fonction*, et la fonction est dite *non close*.

Les variables libres

`fun` x $\rightarrow$ x+1	fonction close, x est lie dans le corps
`fun` x $\rightarrow$ x+y	fonction non close, y est libre

- La dernière fonction est non close: son corps contient une variable y “externe” à la fonction.
- Afin de l’évaluer, on doit donner une une valeur éffective à y.
- Le résultat rendu par la fonction dépendra de cette valeur.

La liaison des variables

En programmation, deux disciplines majeures determinent la manière de lier les variables libres.

- la Liaison statique,
- la Liaison dynamique.

La liaison statique

- En *liason statique*, une variable libre *y* dans une fonction *f*, est *liée à sa valeur* d'après le contexte statique.
- Il est formé de toutes les définitions qui précédent (et qui sont dans la portée de) la déclaration de *f* .
- C'est cette forme de liaison qui adoptent la plupart de langages actuellement, dont ML.

Quelles sont les résultats de `f(2)` ?

```
let y = 3;;
let f = function x -> x+y;;
f(2);;
let y = 0;;
f(2);;
```

La liaison statique

```
let y = 3;;
let f = function x -> x+y;; (* liaison statique y: y <-
f(2);;
let y = 0;;
f(2);;
```

Les deux appels `f(2)` donnent le même résultat:

- La valeur correspondant a y dans `f` ([y $\leftarrow$ 3]) est celle *au moment de la la définition* de `f` (moment “statique”);
- Cette liaison *statique* est capturée une fois pour toutes par la fonction `f`, et sera employée pour chaque appel de la fonction.
- Ainsi, la valeur de y libre dans `f` est determinée une fois pour toutes lors de la la définition de la fonction (moment “statique”).

La liaison dynamique

```
let y = 3;;
let f = function x -> x+y;; (* liaison statique y: y <-
f(2);;
let y = 0;;
f(2);;
```

- Dans cette discipline, c'est dans le contexte des définitions présent au moment de l'exécution de l'appel qu'il faut chercher la valeur des variables libres.
- Dans ce cas, la valeur de chaque appel peut changer selon la valeur courante de y.
- Dans l'exemple, le premier appel `f(2)` s'évalue en 5, le deuxième en 2.

Les fermetures

Pour garantir le comportément selon la liaison statique, la valeur d'une fonction est representée par une fermeture, notée

$$\texttt{fun}\ x.a\ e$$

composée de trois éléments:

- le nom du paramètre, x
- l'expression correspondant au corps a,
- et l'environnement e en place au moment d'introduction de la fonction.

Environnement dans une fermeture

Un *environnement d'évaluation* contient des liaisons entre identificateurs et leurs valeurs.

Exemple: $[y \leftarrow 3]$ est l'environnment où y est lié à 3.

Une fermeture "capture" l'environement présent lors de sa définition (en pratique, seulement la partie qui correspond aux variables libres dans la fonction).

Exemple: la valeur de la fonction `f = function` $x \rightarrow x + y$ définie dans l'environnement où $y = 3$ est la fermeture

$$\texttt{fun}\ x\,.\,x + y[y \leftarrow 3]$$

Valeurs de Mini-ML

v	$::=$	b	valeur de base
	$\mid$	$\mathtt{fun}\ x.a\,[\,e\,]$	fermeture de fonction, *e* environnement capturé
	$\mid$	$(v_1\,,\,v_2)$	paire de valeurs

Valeurs résultat d’une évaluation

Afin de tenir compte des erreurs éventuels pendant l’évaluation, le *résultat* r de l’évaluation d’une expression a est:

- soit une valeur notée v,
- soit `err` si l’évaluation correspond à une erreur pendant l’exécution.

$$\text{Résultats:} \quad r \; ::= \; v \quad \text{evaluation normale}$$
$$\qquad\qquad\quad \mid \; \mathtt{err} \quad \text{erreur d'exécution}$$

Les erreurs à l'exécution

Les erreurs qu'il nous intéresse d'identifier sont ceux qu'un système de types doit éviter. Il s'agit d'erreurs pendant l'exécution correspondant à:

- l'application d'une expression qui n'est ni une opération primitive ni une fonction,
- l'application d'une fonction à un argument du mauvais type,
- l'évaluation d'une variable libre (indéfinie)

Environnements ou contextes d'évaluation

Pour donner un sens aux identificateurs pouvant apparaître *librement* dans une expression, l'évaluation se fait dans un *environnement d'évaluation* e qui associe identificateurs avec leurs valeurs.

$$\text{Environnements} \quad e \quad ::= \quad [\, x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n \,]$$

L'extension d'un environnement d'évaluation e par une liaison $[x \leftarrow v]$ est notée $e\{x \leftarrow v\}$.

Les règles d'évaluation

- La sémantique que nous donnons ici décrit une relation entre une expressions a, un résultat v, et un environnement e.
- Elle est notée $e \vdash a \Rightarrow v$, qu'on lit: dans l'environnement e initial, l'expression a est évaluée dans le résultat v.

Règles d'évaluation normale

Variables (1), constantes (2), fonctions (3)

$$e \vdash x \Rightarrow e(x) \quad (1) \qquad\qquad e \vdash c \Rightarrow c \quad (2)$$

$$e \vdash \texttt{fun } x \rightarrow a \Rightarrow \texttt{fun } x.a[e] \quad (3)$$

Application de fonctions

$$\frac{e \vdash a_1 \Rightarrow \texttt{fun}\ x.a'[e'] \qquad e_1 \vdash a_2 \Rightarrow v_2 \qquad e'\{x \leftarrow v_2\} \vdash a' \Rightarrow r}{e \vdash a_1\ a_2 \Rightarrow r} \qquad (4)$$

Création d'une paire, Let

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e_1 \vdash a_2 \Rightarrow v_2}{e \vdash (a_1, a_2) \Rightarrow (v_1, v_2)} \quad (5)$$

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad e\{x \leftarrow v_1\} \vdash a_2 \Rightarrow r}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \Rightarrow r} \quad (6)$$

Règles de d'évaluation d'opérateurs (exemple)

$$\frac{e \vdash a \Rightarrow (c_1, c_2) \qquad c_1 \in Int \qquad c_2 \in Int \qquad c_1 + c_2 = c}{e \vdash +(a) \Rightarrow c} \tag{7}$$

Règles de détection d'erreurs

$$e \vdash x \Rightarrow \texttt{err} \qquad \text{si } x \notin Dom(e) \quad (8)$$

$$\frac{e \vdash a_1 \Rightarrow v_1 \qquad v_1 \text{ n'est ni une fermeture ni un opérateur}}{e \vdash a_1\ a_2 \Rightarrow \texttt{err}} \quad (9)$$

$$\frac{e \vdash a \Rightarrow v \qquad v \text{ n'est pas de la forme } (c_1, c_2) \text{ ou } c_1 \notin Int \text{ ou } c_2 \notin Int}{e \vdash +(a) \Rightarrow \texttt{err}}$$

Règles de propagation d'erreurs

$$\frac{e \vdash a_1 \Rightarrow v_1 \quad e_1 \vdash a_2 \Rightarrow \texttt{err}}{e \vdash a_1\ a_2 \Rightarrow \texttt{err}} \quad (11)$$

$$\frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \quad (1$$

$$\frac{e \vdash a_1 \Rightarrow v_1 \quad e_1 \vdash a_2 \Rightarrow \texttt{err}}{e \vdash (a_1, a_2) \Rightarrow \texttt{err}} \quad (13)$$

$$\frac{e \vdash a_1 \Rightarrow \texttt{err}}{e \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 \Rightarrow \texttt{err}}$$

Exemples

Exemple 1: Construction de fermeture pour l'expression
$\overline{\texttt{let } x = 2}\texttt{ in } (\texttt{fun } y \to y + x)$

$$\dfrac{[] \vdash 2 \Rightarrow 2 \qquad \dfrac{\text{(3:fun)}}{[x \leftarrow 2] \vdash \texttt{fun } y \to y + x \Rightarrow \texttt{fun } y. + (y, x)[x \leftarrow 2]}}{[] \vdash \texttt{let } x = 2 \texttt{ in } (\texttt{fun } y \to y + x) \Rightarrow \texttt{fun } y. + (y, x)[x \leftarrow 2]} \text{(6:let}$$

Exemples

Exemple 2: Evaluation d'une application:
$(\texttt{let}\ x = 2\ \texttt{in}\ (\texttt{fun}\ y \rightarrow y + x))\ 3$. Sachant qu'une dérivation existe pour évaluer l'exemple précédent, nous aurons:

$$\dfrac{\begin{array}{cc} \begin{array}{c} [] \vdash \texttt{let}\ x = 2\ \texttt{in}\ (\texttt{fun}\ y \rightarrow y + x) \\ \Rightarrow \texttt{fun}\ y. + (y, x)[x \leftarrow 2] \end{array} & [] \vdash 3 \Rightarrow 3 \\ \\ [x \leftarrow 2; y \leftarrow 3] \vdash \quad y + x \Rightarrow 5 & \text{(4:app)} \end{array}}{[] \vdash (\texttt{let}\ x = 2\ \texttt{in}\ (\texttt{fun}\ y \rightarrow y + x))\ 3 \Rightarrow 5}$$

Exemples

Exemple 3: Expressions qui s'évaluent en `err`:

$$
\begin{array}{lcl}
[] & \vdash & +\,(1, \texttt{true}) \\
[] & \vdash & +\,2 \\
[] & \vdash & x \\
[] & \vdash & 1\ 2
\end{array}
$$

Exemples

Exemple 5: $[] \vdash +(1, \texttt{true}) \Rightarrow \texttt{err}$.

$$\dfrac{\dfrac{[] \vdash 1 \Rightarrow 1 \quad [] \vdash \texttt{true} \Rightarrow \texttt{true}}{[] \vdash (1, \texttt{true}) \Rightarrow (1, \texttt{true})}\text{(5:pair)} \quad \texttt{true} \notin \mathit{Int}}{[] \vdash +(1, \texttt{true}) \Rightarrow \texttt{err}}\text{(10:Op\_err)}$$

Typage de Mini-ML

- Le typage est l'analyse statique des programmes visant à détecter des expressions incohérentes, telles que:
  (1 2)
  ou (1+true)
  ou encore (`fun` x $\rightarrow$ x+1) true.
- Pendant le typage, chaque sous-expression se voit assigner un type, et l'on vérifie leur cohérence
- Le typage et l'évaluation sont deux manière de décrire la sémantique des programmes. Le résultat d'une évaluation est une valeur, celui du typage est un type.
- Nous donnons un système de *règles de typage* pour des types aux expressions du langage.

Les types

On se donne:

- un ensemble *TyBase* pour les types de base tels que `int`, `bool`, etc;
- et un ensemble infini *TyVar* de variables de types (intervenant dans les types *polymorphes*).

Les expressions des types sont notées τ:

$$
\underline{\text{Types:}} \quad \tau \quad ::= \quad \begin{array}{lll} & \iota & \text{type de base} \\ | & \alpha & \text{variable de type} \\ | & \tau_1 \rightarrow \tau_2 & \text{type de fonction} \\ | & \tau_1 \times \tau_2 & \text{type produit} \end{array}
$$

Environnements de typage

Les expressions sont typées dans un environnement de typage E associant identificateurs avec leurs types.

Environnements de typage: $\quad E \quad ::= \quad [\, x_1 \leftarrow \tau_1, \ldots, x_n \leftarrow \tau_n \,]$

Un environnement E pourra être étendu par une liaison entre une variable x et son type τ, noté $E\{x \leftarrow \tau\}$.

Relation de typage

La relation de typage décrite par les règles d'inférence fait intervenir:

- un contexte d typage E,
- une expression a du langage
- et un type τ.
- Elle est notée $E \vdash a : \tau$
- et se lit: "selon les hypothèses de typage de l'environnement E, l'expression a a le type τ".
- Nous considérons donnée une fonction $Type(c)$ qui renvoie le type d'une constante c ou d'un opérateur primitif op.

Les règles de typage monomorphe

$$E \vdash c : \mathit{Type}(c) \quad (1) \qquad \frac{E(x) = \tau}{E \vdash x : \tau} \quad (2) \qquad E \vdash op : \mathit{Type}(op) \quad (3)$$

Fonctions, applications

$$\frac{E\{x \leftarrow \tau'\} \vdash a : \tau}{E \vdash \texttt{fun}\ x \rightarrow a : \tau' \rightarrow \tau} \quad (4) \qquad \frac{E \vdash a_1 : \tau' \rightarrow \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau} \quad (5)$$

Construction paires

$$\frac{E \vdash a_1 : \tau_1 \quad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \quad (6)$$

Let

$$\frac{E \vdash a_1 : \tau_1 \quad E\{x \leftarrow \tau_1\} \vdash a_2 : \tau_2}{E \vdash (\texttt{let } x = a_1 \texttt{ in } a_2) : \tau_2} \quad (7)$$

Exemples

Exemple 1: Dérivation de typage pour le jugement
$[] \vdash (\texttt{fun}\ x \rightarrow x) : \texttt{bool} \rightarrow \texttt{bool}$

$$\dfrac{\dfrac{}{[x \leftarrow \texttt{bool}] \vdash x : \texttt{bool}}\ (2{:}\mathrm{Var})}{[] \vdash (\texttt{fun}\ x \rightarrow x) : \texttt{bool} \rightarrow \texttt{bool}}\ (4{:}\mathrm{Fun})$$

Exemples

Exemple 2: Dérivation de typage pour le jugement
$[] \vdash (\texttt{fun}\ x \to x)\ 1 : \texttt{int}$

$$\dfrac{\dfrac{[x \leftarrow \texttt{int}] \vdash x : \texttt{int} \quad (4{:}\mathrm{Fun})}{[] \vdash (\texttt{fun}\ x \to x) : \texttt{int} \to \texttt{int}} \qquad \dfrac{[] \vdash \mathit{Type}(1) = \texttt{int} \quad (1{:}\mathrm{Const})}{[] \vdash 1 : \texttt{int}}}{[] \vdash (\texttt{fun}\ x \to x) 1 : \texttt{int}} \quad (5$$

Exemples

Exemple 4: Autres exemples d'expressions que l'on peut typer:

$$
\begin{array}{lcl}
[] & \vdash & (\texttt{fun}\ x \rightarrow (x, 1))\ 2 : \texttt{int} \times \texttt{int} \\
[] & \vdash & (\texttt{fun}\ x \rightarrow (x, 1)) : \texttt{bool} \rightarrow \texttt{bool} \times \texttt{int} \\
[] & \vdash & (\texttt{fun}\ f \rightarrow f\ 2) : (\texttt{int} \rightarrow \texttt{int}) \rightarrow \texttt{int}
\end{array}
$$

Exemples

Exemple 6: Jugements de typage que l'on ne peut pas obtenir:

$$\begin{array}{lll} [] & \vdash & x : \mathtt{int} \\ [] & \vdash & (\mathtt{fun}\ x \to (x, 1)) : \mathtt{int} \\ [] & \vdash & (\mathtt{fun}\ x \to +(x, 1))\ \mathtt{true} : \mathtt{int} \\ [] & \vdash & (\mathtt{fun}\ x \to +(x, 1)) : \mathtt{bool} \times \mathtt{int} \to \mathtt{int} \end{array}$$

Sûreté du typage

- Un des buts du typage est d'exclure les programmes pouvant provoquer des erreurs de typage à l'exécution.
- Les règles d'évaluation caractérisent ces erreurs.
- Il reste à montrer que les règles du typage sont sûres vis-à-vis de celles d'évaluation,
- autrement dit, que toute expression *a* close et bien typée selon ces règles, et dont l'évaluation *termine*, ne produit pas de résultat `err`.

Sûreté du typage

Exemples d'expressions qui s'évaluent en `err` du fait des incohérences entre types.

On suppose que l'environnement d'évaluation e ne contient que le code des primitives:

$1\ 2$	1 ne s'évalue pas en une fermeture
$+(1, \mathtt{true})$	$\mathtt{true} \notin \mathit{Int}$
$+\ 2$	$2 \notin \mathit{Int} \times \mathit{Int}$
$(x, 1)$	$x \notin e$
$\mathtt{let}\ x = 2\ \mathtt{in}\ x := 3$	x ne s'évalue pas en une adresse l
$\mathtt{let}\ x = 2\ \mathtt{in}\ !x$	x ne s'évalue pas en une adresse l

Aucune de ces expressions n'est typable dans les systèmes de règles étudiés.

Résultat de sûreté du typage

- Nous présentons ici le résultat qui permet de garantir la *sûreté du typage monomorphe de mini-ML*.
- Il nous permet d'affirmer qu'un programme qui termine et qui est bien typé exclut toute erreur d'exécution due aux incohérences entre types.

Théorème 1 (Sûreté faible du typage) *Soient a une expression, τ un type, et r une réponse. Si on a $[] \vdash a : \tau$ et $[], [] \vdash a \Rightarrow r$, alors r n'est pas* `err`.

La démonstration se fait sur une proposition plus forte (sûreté forte) par récurrence sur la longueur de la dérivation d'évaluation. Elle s'appuie sur une relation de "typage sémantique" qui permet d'associer à chaque valeur (obtenue par évaluation) un type (obtenu par le typage) obtenus pour chaque expression.

Polymorphisme: schémas de types

Un **schéma de type** est:

- une representation finie de tous les types pouvant être donnés à une expression très générale;
- c'est une expression de types contenant n variables de type ($n \geq 0$) quantifiées universellement.

$$\text{Schémas de types} \qquad \sigma \quad ::= \quad \forall \alpha_1, \ldots \alpha_n . \tau$$

Lorsque $n = 0$, on notera ce schéma de types par τ.

Schémas de types et instances polymorphes

Un schéma de types peut être vu comme l'ensemble des types obtenus en spécialisant ses variables de types par des types particuliers.

$$\forall\alpha.\alpha \to \alpha \qquad \text{vu comment l'ensemble } \{\tau \to \tau \mid \tau \text{ est un type}\}$$

- $\texttt{int} \to \texttt{int}$ et $\texttt{bool} \to \texttt{bool}$ appartiennent à cet ensemble,
- mais pas $\texttt{bool} \to \texttt{int}$ ou $\texttt{int}$.

Les deux premiers types sont des instances polymorphes du type $\forall\alpha.\alpha \to \alpha$.

Substitution de variables de types

C’est une application finie des variables de types dans les expressions de types. On les notes φ.

Substitutions	φ	$::=$	$[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$

On lit: “la variable de type α_1 est remplacée par le type τ_1, …”

Substitution sur une expression de types

La substitution φ appliquée à une expression de type τ, notée $\varphi(\tau)$ est définie récursivement sur la structure de τ. Il s'agit de remplacer dans τ, chaque occurrence de la variable de type α_i par le type τ_i lui correspondant selon la substitution φ:

$$
\begin{array}{rcl}
\varphi(\iota) & = & \iota \\
\varphi(\alpha) & = & \varphi(\alpha) \text{ si } \alpha \in \mathit{Dom}(\varphi) \\
\varphi(\alpha) & = & \alpha \text{ si } \alpha \notin \mathit{Dom}(\varphi) \\
\varphi(\tau_1 \to \tau_2) & = & \varphi(\tau_1) \to \varphi(\tau_2) \\
\varphi(\tau_1 \times \tau_2) & = & \varphi(\tau_1) \to \varphi(\tau_2)
\end{array}
$$

Substitutions (suite)

Une substitution $\varphi = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ appliquée sur une expression de type τ sera notée indistinctement $\tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ ou $\varphi(\tau)$.

Exemple: Application de la substitution $\varphi = [\alpha \mapsto \mathtt{int}, \beta \mapsto \gamma]$ à l'expression $\tau = (\alpha \times \beta) \to \alpha$:

$$\begin{array}{rcl} \varphi((\alpha \times \beta) \to \alpha) & = & (\mathtt{int} \times \gamma) \to \mathtt{int}) \\ ((\alpha \times \beta) \to \alpha)[\alpha \mapsto \mathtt{int}, \beta \mapsto \gamma] & = & (\mathtt{int} \times \gamma) \to \mathtt{int}) \end{array}$$

Variables de type libres

On note:

- $L(\tau)$ l'ensemble des variables de type libres dans un type τ,
- $L(\sigma)$ l'ensemble des variables de type libres dans un schéma de types,
- et $L(E)$, celui d'un environnement de type E.

$$
\begin{array}{rcl}
L(\iota) & = & \emptyset \\
L(\alpha) & = & \{\alpha\} \\
L(\tau_1 \to \tau_2) & = & L(\tau_1) \cup L(\tau_2) \\
L(\tau_1 \times \tau_2) & = & L(\tau_1) \cup L(\tau_2) \\
L(\forall \alpha_1 \ldots \alpha_n . \tau) & = & L(\tau) \setminus \{\alpha_1, \ldots \alpha_n\} \\
L(E) & = & \bigcup_{x \in Dom(E)} L(E(x))
\end{array}
$$

Instance polymorphe

Un type τ *est une instance polymorphe d'un schéma de type* $\sigma = \forall \alpha_1, \ldots \alpha_n . \tau'$, noté $\tau \leq \sigma$:

$\tau \leq \forall \alpha_1, \ldots \alpha_n . \tau'$ ssi il existe $\tau_1, \ldots, \tau_n$ t.q. $\tau = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]\tau'$

Exemple

Le type $(\mathtt{int} \times (\mathtt{bool} \to \gamma)) \to \mathtt{int}$ est une instance polymorphe de $\forall\alpha, \beta.(\alpha \times \beta) \to \alpha$, noté:
$(\mathtt{int} \times (\mathtt{bool} \to \gamma)) \to \mathtt{int} \leq \forall\alpha, \beta.(\alpha \times \beta) \to \alpha$, car:

- il existe $\tau_1 = \mathtt{int}$ et $\tau_2 = (\mathtt{bool} \to \gamma)$,
- tels que $(\mathtt{int} \times (\mathtt{bool} \to \gamma)) \to \mathtt{int} = (\alpha \times \beta) \to \alpha[\alpha \mapsto \tau_1, \beta \mapsto \tau_2]$
- autrement dit: $(\mathtt{int} \times (\mathtt{bool} \to \gamma)) \to \mathtt{int} = (\alpha \times \beta) \to \alpha[\alpha \mapsto \mathtt{int}, \beta \mapsto (\mathtt{bool} \to \gamma)]$

Généralisation polymorphe

Permet d'introduire du polymorphisme dans un type ayant des variables de type libres, par quantification universelle de ces variables.

Exemple: la généralisation de $\alpha \to \alpha$ résulte dans le schéma de type $\forall \alpha . \alpha \to \alpha$.

La généralisation permet d'obtenir l'effet inverse de l'instantiation.

La généralisation se fait par rapport à un environnement de typage E: les variables de types généralisées dans un type τ sont celles libres dans τ mais qui ne sont pas libres dans E. La génŕalisation de τ dans E est notée:

$$Gen(\tau, E) = \forall \alpha_1, \ldots, \alpha_n . \tau \text{ si } \alpha_i \in L(\tau) \setminus L(E)$$

Règles de typage polymorphe

Les règles sont celles de son typage monomorphe avec quelques changements:

- l'environnement de typage E: liaisons entre identificateurs et schémas de types (et non pas avec des types simples).
- la fonction *Type*(c) associe des schémas de types aux constantes et opérateurs primitifs.
  Exemple: $\mathit{Type}(\texttt{fst}) = \forall\alpha,\beta.(\alpha\times\beta)\to\alpha$.
- la règle de typage des identificateurs et opérateurs éffectue une étape d'instantiation polymorphe.
- la règle du `let` introduit du polymorphisme dans le type de l'expression liée localement. Son type est généralisé puis elle peut être utilisée de manière polymorphe dans la partie `in`.

Les règles de typage polymorphe

La relation de typage fait intervenir un contexte d typage E, une expression a et un schéma de type σ.

- Ces deux règles réalisent une étape d'instantiation (notée $\sigma \leq \tau$). On instantie le type σ obtenu pour la variable ou la constante typée.
- le résultat est le type τ

Constantes, variables:

$$\frac{Typ(c) = \sigma \quad \text{et} \quad \tau \leq \sigma}{E \vdash c : \tau} \quad (1) \qquad \frac{E(x) = \sigma \quad \text{et} \quad \tau \leq \sigma}{E \vdash x : \tau} \quad (2)$$

Opérateurs

$$\frac{\mathit{Typ}(\mathit{op}) = \sigma \qquad \text{et} \qquad \tau \leq \sigma}{E \vdash \mathit{op} : \tau} \quad (3)$$

Fonctions, applications

$$\frac{E\{x \leftarrow \tau'\} \vdash a : \tau}{E \vdash \texttt{fun}\ x \to a : \tau' \to \tau} \quad (4) \qquad \frac{E \vdash a_1 : \tau' \to \tau \qquad E \vdash a_2 : \tau'}{E \vdash a_1\ a_2 : \tau} \quad (5)$$

Paires

$$\frac{E \vdash a_1 : \tau_1 \quad E \vdash a_2 : \tau_2}{E \vdash (a_1, a_2) : \tau_1 \times \tau_2} \tag{6}$$

Let

$$\frac{E \vdash a_1 : \tau_1 \quad E\{x \leftarrow Gen(\tau_1, E)\} \vdash a_2 : \tau_2}{E \vdash (\texttt{let}\ x = a_1\ \texttt{in}\ a_2) : \tau_2} \qquad (7)$$

- Cette règle introduit le polymorphisme,
- le type τ_1 de l'expression locale a_1 est généralisé dans $Gen(\tau_1, E)$ afin de typer la suite a_2

Exemples

<u>Exemple 1</u>: Typons l'expression $\mathtt{let}\ f = \mathtt{fun}\ x \rightarrow x\ \mathtt{in}\ (f\ 1, f\ \mathtt{true})$.
La seule règle applicable dans ce cas est (7:Let).

$$\dfrac{\dfrac{\dfrac{\alpha \leq \alpha}{[x \leftarrow \alpha] \vdash x : \alpha}}{[] \vdash \mathtt{fun}\ x \rightarrow x : \alpha \rightarrow \alpha} \quad \dfrac{\begin{array}{c}[f \leftarrow \forall\alpha.(\alpha \rightarrow \alpha)] \vdash f\ 1 : \mathtt{int} \\ [f \leftarrow \forall\alpha.(\alpha \rightarrow \alpha)] \vdash f\ \mathtt{true} : \mathtt{bool}\end{array}}{[f \leftarrow Gen(\alpha \rightarrow \alpha, [])] \vdash (f\ 1, f\ \mathtt{true}) : \mathtt{int} \times}}{[] \vdash \mathtt{let}\ f = \mathtt{fun}\ x \rightarrow x\ \mathtt{in}\ (f\ 1, f\ \mathtt{true}) : \mathtt{int} \times \mathtt{bool}}$$

Exemple 1 (suite)

Détaillons le typage des deux hypothèses du jugement $[f \leftarrow \forall\alpha.(\alpha \to \alpha)] \vdash (f\ 1, f \mathtt{true}) : \mathtt{int} \times \mathtt{bool}$. En utilisant la règle (2) d'instantiation polymorphe des identificateurs, pour l'identificateur f, nous obtenons:

$$\dfrac{\dfrac{\mathtt{int} \to \mathtt{int} \leq \forall\alpha.(\alpha \to \alpha)}{[f \leftarrow \forall\alpha.(\alpha \to \alpha)] \vdash f : \mathtt{int} \to \mathtt{int}} \qquad [f \leftarrow \forall\alpha.(\alpha \to \alpha)] \vdash 1 : \mathtt{int}}{[f \leftarrow \forall\alpha.(\alpha \to \alpha)] \vdash f\ 1 : \mathtt{int}}$$

Par une dérivation analogue, la deuxième hypothèse peut être également typée.

Quelques résultats

- Les système de règles de typage présenté correspond à un algorihtme non detéministe.
- Un algorithme W déterministe existe permettant d'inférer le type le plus général pour toute expression typable. Il mélange étapes d'instatiation, d'unification et de généralisation.
- L'unification est un procédé permettant de résoudre un système contraintes sur des expressions et variables de types.
- L'algorithme W a été prouvé correct et complet.
- Le résultat de sûreté du typage s'étend au typage polymorphe.