Expression Logique et Fonctionnelle . . . Évidemment

CTD1 : Introduction à CAML et Core ML

1 Introduction à la programmation

Premier langage fonctionnel : LISP (fin des années 1950). Les langages fonctionnels ont pour fondement théorique le λ-calcul introduit par Alonzo Church dans les années 1930 pour définir la notion de *calculabilité*.

Nombreux langages fonctionnels : LISP, SCHEME, HASKELL, SML, OCAML, . . . qui se distinguent selon différents caratéristiques :

Langages fonctionnels pur vs. impurs. Un langage fonctionnel sans effet de bord est dit langage *fonctionnel pur*. Par exemple, dans de tels langages il n'y a pas d'opération d'affection. Les langages purs restent assez rares, citons HASKELL parmi eux. LISP, SCHEME, la famille des langages ML sont impurs.

Typage statique vs. dynamique. Les arguments d'une fonction doivent avoir un type compatible avec celui de la fonction pour que l'évaluation d'une application puisse être réalisée. La vérification du type, ou typage, peut se faire à l'exécution du programme, on parle alors de *typage dynamique*. C'est le cas de langages comme LISP ou SCHEME. Inversement, le typage peut avoir lieu avant l'exécution du programme, lors de la compilation, le typage est alors qualifié de *statique*. C'est le cas des langages de la famille ML.

Vérification vs. inférence de types. La plupart des langages de programmation impératifs typés s'appuie sur un *typage explicite* : le programmeur donne toutes les informations de types liées aux déclarations d'identificateurs. Le compilateur, ou machine d'execution, s'appuie alors sur cette information pour *vérifier* (statiquement ou dynamiquement) que les identificateurs sont utilisés conformément au type déclaré. La plupart des langages fonctionnels, dont OCAML, libèrent le programmeur des annotations de types, les types étant automatiquement *inférés* par le compilateur.

Distinction entre programmation impérative et programmation fonctionnelle :

Paradigme impératif. Celui-ci repose sur les notions d'*état d'une machine* (i.e. état de la mémoire), d'*instructions*, de *séquence* (ordonnancement) et d'*itération*. Il nécessite d'avoir une connaissance minime du modèle de machine sur laquelle on programme (modèle de *Von Neumann*).

Paradigme fonctionnel. Celui-ci repose sur les notions *valeurs*, *expressions* et *fonctions*. L'exécution d'un programme est l'*évaluation* d'une ou plusieurs expressions, expressions qui sont souvent des applications de fonctions à des valeurs passées en paramètre[1]. Ne nécessite de modèle de machine.

Le langage utilisé dans ce cours est Objective-Caml (OCAML), dialecte de la famille ML. OCAML est développé à l'INRIA[2] et librement distribué (`http ://caml.inria.fr`). Ce langage est très riche et offre

- un noyau fonctionnel (Core ML) ;
- des structures de contrôle impératives, ainsi que des types de données mutables ;
- un système de *modules* ;
- la possibilité de programmation par *objets*.

Les modules et les objets d'OCAML ne seront pas abordés dans ce cours et nous nous limiteront à l'apprentissage de CAML et de son «noyau théorique» Core ML.

2 CAML, un tour d'horizon (partiel)

2.1 Phrases du langage

Un programme fonctionnel écrit en CAML est une suite de *phrases*, *déclarations* de variables ou de types, ou bien *expressions* à évaluer. L'exécution d'un programme consiste en l'évaluation de toutes les phrases.

[1] On dit aussi *programmation applicative*

[2] Institut National de Recherche en Informatique et Automatique

Les phrases du langage sont terminées par un double point-virgule (`;;`). Ci-dessous quelques exemples de phrases évaluées par l'interprète du langage. Le symbole # est le prompt de la boucle d'interaction, ce qui suit est la phrase à évaluer. Les lignes débutant par le symbole – sont les résultats de l'évaluation de la phrase.

```
        Objective Caml version 3.11.1

# 132 ;;
- : int = 132
# 2*(45+10) ;;
- : int = 110
# 7/4 ;;
- : int = 1
# 7 mod 4 ;;
- : int = 3
# 2. ;;
- : float = 2.
# 2. +. 3. ;;
- : float = 5.
# 7./.4. ;;
- : float = 1.75
# float_of_int 2 ;;
- : float = 2.
# true ;;
- : bool = true
# true & false ;;
- : bool = false
# 1 = 2-1 ;;
- : bool = true
# "Hello" ;;
- : string = "Hello"
# "Hello " ^ "le monde " ;;
- : string = "Hello le monde "
# 'A' ;;
- : char = 'A'
# int_of_char 'B' ;;
- : int = 66
```

2.2 Types de base

CAML, comme tous les langages de la famille ML, est un langage fortement typé, à typage statique. Avant d'évaluer une phrase, le type de l'expression est calculé, c'est l'*inférence de type* : contrairement à des langages comme C, PASCAL, ADA, le programmeur n'a pas besoin de préciser le type de ces expressions, variables ou fonctions. Le type d'une expression est calculé à partir de ses composants. Si l'inférence de type échoue, l'expression n'est pas évaluée. Si en revanche elle réussit, l'expression peu être évaluée.

Voici les types de base du langage :

`int` : les entiers. L'intervalle du type `int` dépend de l'architecture de la machine utilisée. C'est l'intervalle $[\![2^{30}, 2^{30}-1]\!]$ sur les machines 32 bits.

Opérateur	signification
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division entière
`mod`	reste de la division entière

`float` : les nombres flottants. Ils respectent la norme IEEE 754. OCAML distingue les entiers des flottants : `2` est de type `int`, tandis que `2.` est de type `float`, et ces deux valeurs ne sont pas égales. Même les opérateurs arithmétiques sur les flottants sont différents de ceux sur les entiers.

Opérateur	signification
`+.`	addition
`-.`	soustraction
`*.`	multiplication
`/.`	division
`**`	exponentiation

On ne peut pas additionner un `int` et un `float`

```
# 2 + 3. ;;
Characters 4-6:
  2 + 3. ;;
      ^^
This expression has type float but is here used with type int
# 2. +. 3 ;;
Characters 6-7:
  2. +. 3 ;;
       ^
This expression has type int but is here used with type float
```

Fonction	signification
`float_of_int`	conversion d'un `int` en `float`
`ceil`	partie entière supérieure (plafond)
`floor`	partie entière inférieure (plancher)
`sqrt`	racine carrée
`exp`	exponentielle
`log`	logarithme népérien
`log10`	logarithme décimal
...	...

`string` : les chaînes de caractères (longueur limitée à $2^{64} - 6$).

Opérateur	signification
`^`	concaténation

Fonction	signification
`string_of_int`	conversion d'un `int` en un `string`
`int_of_string`	conversion d'un `string` en un `int`
`string_of_float`	conversion d'un `float` en un `string`
`float_of_string`	conversion d'un `string` en un `float`
`String.length`	longueur

`char` : les caractères. Au nombre de 256, les 128 premiers suivent le code ASCII.

Fonction	signification
`char_of_int`	conversion d'un `int` en un `char`
`int_of_char`	conversion d'un `char` en un `int`

`bool` : les booléens pouvant prendre deux valeurs **`true`** et **`false`**.

Opérateur	signification
`not`	négation
`&&` ou `&`	et séquentiel
`||` ou **`or`**	ou séquentiel

Les opérateurs de comparaison sont *polymorphes*, ils sont utilisables pour comparer deux `int`, ou deux `float`, ou deux `string`,.... Mais ils ne permettent pas la comparaison d'un `int` et un `float`.

Opérateur	signification
`=`	égalité (structurelle)
`<>`	négation de =
`<`	inférieur
`<=`	inférieur ou égal
`>`	supérieur
`>=`	supérieur ou égal

`unit` : le type `unit` est un type ne possédant qu'une seule valeur notée `()`. Ce type est utilisé dans la partie impérative du langage OCaml, et par les différentes procédures d'affichage.

Procédure	signification
`print_int`	affichage d'un entier
`print_float`	affichage d'un flottant
`print_string`	affichage d'une chaîne de caractères
`print_char`	affichage d'un caractère
`print_newline`	passage à une nouvelle ligne

```
# print_int 12 ;;
12- : unit = ()
# print_float 3.14 ;;
3.14- : unit = ()
# print_string "Hello" ;;
Hello- : unit = ()
# print_char 'A' ;;
A- : unit = ()
# print_newline () ;;

- : unit = ()
```

2.3 Déclaration de variables

En programmation fonctionnelle, une *variable* est une liaison entre un nom et une valeur. Les variables peuvent être *globales*, et elles sont alors connues de toutes les expressions qui suivent la déclaration, ou bien *locales* à une expression, et dans ce cas elles ne sont connues que dans l'expression pour laquelle elles ont été déclarées.

L'ensemble des variables connues d'une expression est appelé *environnement* de l'expression.

La syntaxe d'une déclaration globale est de la forme

```
let <nom> = <expr> ;;
```

où `<nom>` est le nom de la variable (en CAML les noms de variables doivent obligatoirement commencer par une lettre minuscule), et `<expr>` une expression décrivant la valeur de la variable.

```
# let n = 12 ;;
val n : int = 12
# n ;;
- : int = 12
# 2 * n;;
- : int = 24
# let m = 2 * n ;;
val m : int = 24
# m + n;;
- : int = 36
# m + p;;
Characters 2-3:
m+p;;
  ^
Unbound value p
```

Pour être typable, et donc évaluable, toutes les variables d'une expression doivent être définies dans l'environnement de l'expression, sous peine d'avoir le message d'erreur `Unbound value`.

On peut aussi déclarer simultanément plusieurs variables.

```
let <nom1> = <expr1>
and <nom2> = <expr2>

...
and <nomn> = <exprn> ;;
```

2.4 Déclaration locale de variables

Syntaxe des déclarations locales

```
let <nom> = <expr1>
in <expr2> ;;
```

Cette forme syntaxique permet de déclarer la variable nommée <nom> dans l'environnement d'évaluation de l'expression <expr2>. En dehors de l'environnement de cette expression, la variable <nom> n'existe pas.

Il est possible de faire des déclarations simultanées de variables locales à une expression.

```
let <nom1> = <expr1>
and <nom2> = <expr2>

...
and <nomn> = <exprn>
in <expr> ;;
```

```
# let pi = 3.141592
  and r = 2.0
  in 2. *. pi *. r ;;
- : float = 12.566368
# pi ;;
Characters 0-2:
pi ;;
^^
Unbound value pi
```

2.5 Expression conditionnelle

La syntaxe d'une expression conditionnelle est

```
if <expr1> then <expr2> else <expr3> ;;
```

dans laquelle

- <expr1> est une expression booléenne (type bool),
- <expr2> et <expr3> ont le même type.

```
# if 1=1 then "egaux" else "non egaux" ;;
- : string = "egaux"
# if 1=2 then "egaux" else "non egaux" ;;
- : string = "non egaux"
# if 1=2 then 0 else "non egaux" ;;
Characters 19-30:
if 1=2 then 0 else "non egaux" ;;
                   ^^^^^^^^^^^
This expression has type string but is here used with type int
```

Remarques :

- en CAML, la structure de contrôle conditionnelle est une expression et non une instruction, c'est la raison pour laquelle les deux expressions des parties **then** et **else** doivent être du même type.
- en CAML, il est possible d'omettre la partie **else** d'une expression conditionnelle. Dans ce cas, elle est considérée comme étant la valeur () (type unit), et la partie **then** doit elle aussi être du type unit.

2.6 Fonctions

En CAML, comme dans tous les langages fonctionnels, les fonctions sont des valeurs comme toutes les autres. Une variable peut donc avoir une fonction pour valeur.

```
# let racine_carree = sqrt ;;
val racine_carree : float -> float = <fun>
# racine_carree 2. ;;
- : float = 1.41421356237309515
```

2.6.1 Type d'une valeur fonctionnelle

Le type d'une fonction est déterminé par le type de son (ou ses) argument(s) et celui des valeurs qu'elle renvoie. Ainsi le type de la fonction `sqrt` est `float -> float` puisqu'elle prend un `float` pour paramètre et retourne un `float`.

2.6.2 Définition d'une fonction

On peut déclarer une variable fonctionnelle de la façon suivante :

```
let <fonct> <param> = <expr> ;;
```

où `<fonct>` est le nom donné à la fonction de paramètre `<param>` et `<expr>` est l'expression à évaluer lors d'un appel à la fonction.

Par exemple, la fonction `carre` peut être déclarée ainsi :

```
# let carre x = x*x ;;
val carre : int -> int = <fun>
```

2.6.3 Appel à une fonction

Pour appliquer une fonction à une valeur `<arg>`, il suffit d'écrire le nom de la fonction suivi de l'expression

```
<fonct> <arg>;;
```

```
# carre 2 ;;
- : int = 4
# carre (2 + 2) ;;
- : int = 16
```

Attention aux parenthèses qui permettent de contrôler la priorité des opérateurs

```
# carre 2+2 ;;
- : int = 6
```

2.6.4 Fonctions à plusieurs paramètres

On peut déclarer des variables dont la valeur est une focntion à plusieurs variables sous la forme

```
let <fonct> <arg1> ... <argn> = <expr> ;;
```

Voici la déclaration de la variable `add` désignant la fonction calculant la somme de deux entiers.

```
# let add x y = x+y ;;
val add : int -> int -> int = <fun>
# add 8 2 ;;
- : int = 10
```

Dans cet exemple, on peut noter que

- le type d'une fonction à plusieurs arguments se note
  `type_arg_1 -> type_arg_2 -> ... -> type_arg_n -> type_resultat`
- un appel à une fonction à plusieurs paramètres se fait en notant successivement le nom de la fonction et ses arguments.

Si le nombre de paramètres effectifs passés à l'appel d'une fonction est inférieur au nombre de paramètres formels, la valeur du résultat est une fonction.

```
# let add_8 = add 8 ;;
val add_8 : int -> int = <fun>
# add_8 11 ;;
- : int = 19
```

2.7 Déclarations récursives

Envisageons la définition de la fonction calculant la factorielle de l'entier passé en paramètre. Une expression récursive classique de $n!$ est

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)!\ \forall n \in \mathbb{N}^* \end{aligned}$$

On peut être tenté de traduire cette formulation par la déclaration

```
# let fact n =
  if n=0 then 1 else n*fact(n-1) ;;
```

mais on a tort car on obtient le message

```
Characters 37-41:
  if n=0 then 1 else n*fact(n-1) ;;
                       ^^^^
Unbound value fact
```

qui indique que la variable `fact` apparaissant dans l'expression définissant `fact n` n'est pas liée dans l'environnement de déclaration de la variable ... `fact`. On tourne en rond.

Pour remédier à cela, il faut utiliser la forme syntaxique **`let rec`**

```
let rec <nom> = <expr>
```

dans laquelle toute référence à la variable `<nom>` dans l'expression `<expr>` est une référence à la variable en cours de déclaration.

```
# let rec fact n =
if n=0 then 1 else n*fact(n-1) ;;
val fact : int -> int = <fun>
# fact 5 ;;
- : int = 120
```

3 Core ML

3.1 Des fonctions nommées aux termes fonctionnels

En mathématiques, une fonction d'incrémentation peut être définie de la façon suivante :

$$inc(x) = x + 1 \tag{1}$$

ou, alternativement, ainsi :

$$\begin{aligned} inc : \mathbb{Z} &\rightarrow \mathbb{Z} \\ x &\mapsto x + 1 \end{aligned} \tag{2}$$

Cette deuxième forme est obtenue par la donnée de trois éléments :

- un *nom* (ou identificateur) ;
- un *type* (i.e. son domaine et son codomaine) ;
- et d'un *terme fonctionnel* représentant le graphe de la fonction (i.e. l'association d'une expression mathématique à une variable).

En CAML, le type d'une fonction est automatiquement *inféré*. CAML admet la définition de fonctions suivant l'une ou l'autre de ces deux formes. Ainsi, la définition (1) sera traduite en CAML par

```
# let inc1 x = x+1;;
val inc1 : int -> int = <fun>
```

et la définition (2), sans l'information de type, par

```
# let inc2 = function x -> x+1;;
val inc2 : int -> int = <fun>
```

Les fonctions `inc1` et `inc2` sont strictement équivalentes et implémentent la fonction mathématique *inc* :

```
# inc1 13;;
- : int = 14
# inc2 13;;
- : int = 14
```

En CAML, les termes fonctionnels peuvent s'employer comme n'importe quelle autre expression et, en particulier, indépendamment de tout identificateur. Ainsi,

```
# (function x -> x+1) 13;;
- : int = 14
```

Considérons la fonction d'addition suivante :

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

Comme le type inféré l'indique, `add_paire` s'applique à une paire d'entiers. En CAML, on préfère généralement écrire les fonctions sous forme *curryfiée* :

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

Cette déclaration est presque identique à la précédente mais avec un type[3] indiquant que `add` est une fonction d'une variable dont le résultat est encore une fonction d'une variable. Cette déclaration est strictement équivalente à la suivante qui explicite les termes fonctionnels *imbriqués* :

```
# let add = function x -> function y -> x+y;;
val add : int -> int -> int = <fun>
```

on comprend mieux son type !

La déclaration suivante illustre aussi la bonne lisibilité obtenue par l'utilisation des termes fonctionnels :

```
# let compose f g = function x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Question 1 Écrivez un terme fonctionnel réalisant la conjonction de deux booléens. □

Question 2 Que calculent les expressions suivantes :

```
(function x -> x * 2) 3
(function x -> function y -> x+y) ((function x -> x+x) 1) 1
(function x -> x 3) (function x -> x * 2)
(function x -> x) (function x -> x) 1
(function x -> x 1) (function x -> x)
```

□

[3] notez que les types fonctionnels sont implicitement associatifs à droite, i.e. `int->int->int` se lit `int->(int->int)`.

3.2 Syntaxe

Soient $x, y, \ldots, f, g, \ldots \in \mathcal{V}$ une collection de variables, $\mathtt{f} \in \mathcal{F} = \{+, -, *, \ldots\}$ une collection de *fonctions primitives* et $\mathtt{c} \in \mathcal{C} = \{\mathtt{0}, \mathtt{1}, \mathtt{true}, \mathtt{cons}, \mathtt{pair}, \ldots\}$ une collection de *constructeurs de données*. La fonction *arité* donne pour chaque fonction primitive et constructeur de données son arité (par exemple *arité*(1) = 0, *arité*(pair) = *arité*(+) = 2, etc.). Les programmes écrits en Core ML sont appelés **λ-termes**.

Les λ-termes, notés $t, u, \ldots$, de Core ML sont définis par la grammaire BNF[4] suivante :

t, u	$::=$	x	*variable*
	$\mid$	$\lambda x.t$	*abstraction (terme fonctionnel* **function** x -> t*)*
	$\mid$	tu	*application (d'une fonction t à un argument u)*
	$\mid$	$\mathtt{f}$	*fonction primitive*
	$\mid$	$\mathtt{c}$	*constructeur de donnée*
	$\mid$	**let** $x = t$ **in** u	*liaison (d'un λ-terme t à une variable x)*

On autorise l'utilisation des parenthèses pour lever d'éventuelles ambigüités. Pour éviter la surchage de parenthèses et de λ, on adoptera les conventions suivantes :

- $tu_1 \ldots u_n$ est interprété comme $(\ldots((t)u_1)\ldots u_n)$, i.e. t appliqué à u_1, le tout appliqué à u_2, etc. ;
- $\lambda x.tu$ dénote le λ-terme $\lambda x.(tu)$;
- $\lambda x_1, x_2, \ldots, x_n.t$ est une abréviation pour $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.t$.

Question 3 Réécrivez les programmes de la Question 2 en λ-termes.

3.3 Notions de liaison et α-equivalence

Il est clair que les définitions suivantes

$$inc(x) = x + 1 \qquad \text{et} \qquad inc(y) = y + 1$$

bien que *syntaxiquement* différentes, sont *sémantiquement* équivalentes, c'est-à-dire elle définissent la même fonction mathématique. Il en est bien sûr de même pour ces définitions à base de termes fonctionnels :

$$inc : \begin{array}{l}\mathbb{Z} \to \mathbb{Z} \\ x \mapsto x + 1\end{array} \qquad \text{et} \qquad inc : \begin{array}{l}\mathbb{Z} \to \mathbb{Z} \\ y \mapsto y + 1\end{array}$$

et des programmes CAML !

```
# let inc x = x+1 in inc 4;;
- : int = 5
# let inc y = y+1 in inc 4;;
- : int = 5
# let inc = function x -> x+1 in inc 4;;
- : int = 5
# let inc = function y -> y+1 in inc 4;;
- : int = 5
```

Dans Core ML ces programmes s'écrivent :

$$\begin{array}{rl} & \textbf{let } inc = \lambda x.+\ x\ \mathtt{1}\ \textbf{in } inc\ \mathtt{4} \\ \text{ou} & \textbf{let } inc = \lambda y.+\ y\ \mathtt{1}\ \textbf{in } inc\ \mathtt{4} \end{array}$$

Question 4 Donnez les variables, les fonctions primitives et les constructeurs de ces deux derniers λ-termes. □

Considérons le λ-terme suivant :

$$\begin{array}{l}\textbf{let } carre = \lambda x.*\ x\ x\ \textbf{in} \\ \quad \textbf{let } inc = \lambda x.+\ x\ \mathtt{1}\ \textbf{in} \\ \quad\quad carre\ (inc\ \mathtt{4})\end{array}$$

dont l'évaluation est 25. À nouveau, Ce λ-terme est strictement équivalent au suivant :

$$\begin{array}{l}\textbf{let } square = \lambda x.*\ x\ x\ \textbf{in} \\ \quad \textbf{let } plusOne = \lambda x.+\ x\ \mathtt{1}\ \textbf{in} \\ \quad\quad square\ (plusOne\ \mathtt{4})\end{array}$$

[4] Backus Naur Form

qui s'évalue également à 25. Ceci nous amène à conclure que le *renommage* de variables abstraites par un terme fonctionnel (i.e. un λ) ou liées par un `let` ne change pas le sens (i.e. la sémantique) d'un λ-terme. On appelle ces variables des **variables liées** (*bound variables*). Les variables qui ne sont pas liées sont dites **libres** (*free variables*). Deux λ-termes qui ne diffèrent que par le renommage de leurs variables liées sont dits **α-équivalents**.

Définition 1 (Variables libres et liées) *L'ensemble des **variables libres** et **liées** d'un λ-terme t sont respectivement notés $fv(t)$ et $bv(t)$, et sont définis inductivement sur les λ-termes :*

$$\begin{array}{rcl} fv(x) & = & \{x\} \\ fv(t\ u) & = & fv(t) \cup fv(u) \\ fv(\lambda x.t) & = & fv(t) \backslash \{x\} \\ fv(\texttt{c}) & = & fv(\texttt{f}) = \emptyset \\ fv(\textbf{\texttt{let}}\ x = t\ \textbf{\texttt{in}}\ u) & = & fv(t) \cup (fv(u) \backslash \{x\}) \end{array} \qquad \begin{array}{rcl} bv(x) & = & \emptyset \\ bv(t\ u) & = & bv(t) \cup bv(u) \\ bv(\lambda x.t) & = & bv(t) \cup \{x\} \\ bv(\texttt{c}) & = & bv(\texttt{f}) = \emptyset \\ bv(\textbf{\texttt{let}}\ x = t\ \textbf{\texttt{in}}\ u) & = & bv(t) \cup bv(u) \cup \{x\} \end{array}$$

*L'ensemble de toutes les variables d'un λ-terme t est noté $var(t)$ et défini par $var(t) = fv(t) \cup bv(t)$. Dans les λ-termes $\lambda x.u$ et $\textbf{\texttt{let}}\ x = t\ \textbf{\texttt{in}}\ u$, la **portée** de la variable liée x est le λ-terme u.*

Question 5 Montrez que $fv(t)$ et $bv(t)$ peuvent avoir une intersection non vide. □

Question 6 Pour le λ-terme suivant

$$\begin{array}{l} \texttt{let}\ inc = \lambda x.\texttt{+}\ x\ \texttt{2}\ \texttt{in} \\ \quad \texttt{let}\ inc = \lambda x.\texttt{+}\ (inc\ x)\ \texttt{1}\ \texttt{in}\ inc\ \texttt{4} \end{array}$$

donnez la portée des deux déclarations de la variable *inc*. À quelle valeur s'évalue ce λ-terme ? Réécrivez ce λ-terme pour le rendre plus «lisible». □

Avant de définir l'α-équivalence il est nécessaire de définir la notion de substitution qui est aussi essentielle à l'étude de l'évaluation des λ-termes. Celle-ci substitue des occurences *libres* de variables dans les λ-termes.

Définition 2 (Substitution) *Une **substitution** $[t/x]$ est une application des variables dans les λ-termes de Core ML. On étend son domaine d'application à l'ensemble des λ-terme de la façon suivante :*

$$\begin{array}{rcl} [t/x]y & = & \begin{cases} t & si\ x = y \\ y & sinon \end{cases} \\ [t/x](u_1\ u_2) & = & ([t/x]u_1)([t/x]u_2) \\ [t/x]\lambda y.u & = & \begin{cases} \lambda y.u & si\ x = y \\ \lambda y.[t/x]u & si\ x \neq y\ et\ y \notin fv(t) \end{cases} \\ [t/x]\texttt{f} & = & \texttt{f} \\ [t/x]\texttt{c} & = & \texttt{c} \\ [t/x]\textbf{\texttt{let}}\ y = u_1\ \textbf{\texttt{in}}\ u_2 & = & \begin{cases} \textbf{\texttt{let}}\ y = [t/x]u_1\ \textbf{\texttt{in}}\ u_2 & si\ x = y \\ \textbf{\texttt{let}}\ y = [t/x]u_1\ \textbf{\texttt{in}}\ [t/x]u_2 & si\ x \neq y\ et\ y \notin fv(t) \end{cases} \end{array}$$

□

Notez la condition $y \notin fv(t)$ pour l'application d'une substitution à une abstraction ou à une construction `let`. Elle implique qu'une substitution est une application partielle sur les termes : par exemple, $[y/x]\lambda y.x$ n'est pas défini. Si on levait cette condition on aurait $[y/x]\lambda y.x = \lambda y.y$ et ainsi une *capture de variable* ce qui n'est généralement pas désirable.

Proposition 1 $\forall x, t, u.bv([u/x]t) = bv(t) \cup bv(u)$.

Définition 3 (α-équivalence) *L'α-**équivalence**, notée $=_\alpha$, est la plus petite relation d'équivalence[5] telle que :*

$$\lambda x.t =_\alpha \lambda y.[y/x]t$$
$$\textbf{\texttt{let}}\ x = t\ \textbf{\texttt{in}}\ u =_\alpha \textbf{\texttt{let}}\ y = t\ \textbf{\texttt{in}}\ [y/x]u$$

et compatible avec l'abstraction, l'application et `let`, c'est-à-dire si $t_1 =_\alpha t_2$ et $u_1 =_\alpha u_2$ alors

$$\lambda x.t_1 =_\alpha \lambda x.t_2 \qquad t_1\ u_1 =_\alpha t_2\ u_2$$
$$\textbf{\texttt{let}}\ x = t_1\ \textbf{\texttt{in}}\ u_1 =_\alpha \textbf{\texttt{let}}\ x = t_2\ \textbf{\texttt{in}}\ u_2$$

□

[5] rappel : c'est une relation réflexive, symétrique et transitive.

Question 7 Appliquez la substitution $[\lambda x.\texttt{+}\ x\ \texttt{2}/inc]$ au λ-terme $\texttt{let}\ \ inc = \lambda x.\texttt{+}\ (inc\ x)\ \texttt{1}\ \texttt{in}\ \ inc\ \texttt{4}$. □

Proposition 2 *Si* $t =_\alpha u$ *alors* $fv(t) = fv(u)$. □

Question 8 En déterminant les variables libres et liées, confirmez ou infirmez les équivalences suivantes :

- $(\lambda x.x)y =_\alpha (\lambda y.y)y$
- $(\lambda x.x)y =_\alpha (\lambda x.x)z$
- $(\texttt{let}\ \ x = y\ \ \texttt{in}\ \ x) =_\alpha (\texttt{let}\ \ y = x\ \ \texttt{in}\ \ y)$
- $(\texttt{let}\ \ x = y\ \ \texttt{in}\ \ x) =_\alpha (\texttt{let}\ \ y = y\ \ \texttt{in}\ \ y)$
- $(\texttt{let}\ \ x = \lambda x.x\ \ \texttt{in}\ \ x(\lambda y.y)\texttt{1}) =_\alpha (\texttt{let}\ \ y = \lambda y.y\ \ \texttt{in}\ \ x(\lambda x.x)\texttt{1})$
- $(\texttt{let}\ \ x = \lambda x.x\ \ \texttt{in}\ \ x(\lambda y.y)y) =_\alpha (\texttt{let}\ \ x = \lambda x.x\ \ \texttt{in}\ \ x(\lambda x.x)y)$