Introduction à la Compilation
Cours 11 : Les fonctions de première classe

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

vendredi 9 avril 2010

Enquête

Boom !

```
let add1 :=
  let x := 1 in
    fun (y : int) ⇒ x + y
in
  add1 (1)
```

Exercice

Comment évaluer ce programme ?

La réponse « symbolique »

- On détermine la valeur associée à `add1` par substitution du `let` interne. :

$$\begin{array}{l}\textbf{let}\ \texttt{add1} := \\ \quad \textbf{fun}\,(y : \textbf{int}) \Rightarrow 1 + y \\ \textbf{in} \\ \quad \texttt{add1}\,(1)\end{array}$$

- On substitue cette valeur à l'unique occurrence de `add1` :

$$(\textbf{fun}\,(y : \textbf{int}) \Rightarrow 1 + y)\,(1)$$

- L'évaluation de l'appel de fonction anonyme consiste à substituer ses arguments e□ectifs aux arguments formels :

$$1 + 1$$

- On obtient finalement la valeur « 2 » par évaluation de « + ».

Problème d'un interprète de la sémantique à petits pas

- Cependant, l'évaluation à petits pas ne fournit pas un interprète e□cace : on réécrit le programme sans arrêt et on doit parcourir ce dernier pour appliquer les substitutions.
- Les interprètes e□caces des précédents cours produisaient des valeurs par une inspection linéaire de l'arbre de syntaxe.

$\Rightarrow$ Comment représenter du code (exécutable) sous la forme de valeurs ?

Deux approches

1. Une unification des valeurs et des programmes : les S-expressions de LISP.
2. Des valeurs pour représenter du code clos : les fermetures.

LISP

Les S-expressions

- La syntaxe des expressions de LISP :

$$\begin{array}{rcl} \mathit{Sexp} & ::= & \mathit{atom} \\ & | & (\mathit{Sexp}\ .\ \mathit{Sexp}) \end{array}$$

- Cette syntaxe est aussi celle des valeurs !

Type d'expression

- On classifie les expressions :
  - `cons` : une paire formée de deux expressions.
    - On appelle `car` la première composante ;
    - et `cdr` la seconde. [1]
  - `int` : un entier.
  - `sym` : un symbole.
  - `prim` : une fonction primitive.
  - `comp` : une fonction anonyme.
  - `spec` : une forme spéciale (une fonction qui n'évalue pas tout ses arguments).

[1] `car` pour *Contents of Address Register* et `cdr` pour *Contents of Decrement Register*

Boucle d'interaction de LISP

`read-eval-print`

- La boucle d'interaction de LISP se décompose en trois fonctions :
  - `read` : transforme une chaîne de caractères en S-expression ;
  - `eval` : transforme une S-expression en sa forme évaluée, une S-expression ;
  - `print` : affiche une S-expression sous la forme d'une chaîne de caractères.

$\Rightarrow$ "`read`" et "`print`" sont particulièrement simples à réaliser, reste "`eval`".

Exemples

```
;; Cette expression s'évalue en 3. "+" est une primitive.
(+ 1 2)
;; Cette expression définit un symbole x valant 1.
(define x 1)
;; Une fonction anonyme à deux arguments.
(lambda (x y) (+ x y))
;; La fonction identité appliquée à la liste vide.
((lambda (x) x) ())
;; "if" est une forme spéciale : seule une branche est évaluée.
(if (= x 2) (1 + 2) (3 + 4))
;; "quote" permet de retarder l'évaluation d'une expression
((lambda (x) (car (cdr x))) ('(1 2 3 4)))
```

Environnements d'interprétation des S-expressions

- LISP utilise deux environnements :
  - un environnement lexical Γ qui associe des S-expressions à des symboles ;
  - un environnement global Ξ qui enregistre les définitions de fonctions.

Interprétation des S-expressions

- L'évaluation est définie par cas sur le type de S-expressions à évaluer :

$$
\begin{array}{rl}
\forall e \in \{\texttt{int}, \texttt{spec}, \texttt{comp}, \texttt{prim}\} & \text{eval}\,(e) = e \\
\forall e \in \{\texttt{sym}\} & \text{eval}\,(e) = \Gamma(e) \\
\forall e \in \{\texttt{cons}\} & \text{eval}\,(e) = \text{apply}(\text{eval}(\texttt{car}(e)), \texttt{cdr}(e))
\end{array}
$$

où apply est aussi définie par cas sur le type de son premier argument :

$$
\begin{array}{rl}
\forall f \in \{\texttt{comp}, \texttt{prim}\} & \text{apply}(f, (e_1 \ldots e_n)) = \text{eval}(\text{body}(f)) \\
\\
\text{dans} & \Gamma' \equiv ((x_1\ \text{eval}(e_1)) \ldots (x_n\ \text{eval}(e_n))\ \Gamma) \\
\text{où} & \text{args}(f) = x_1 \ldots x_n \\
\\
\forall f \in \{\texttt{sym}\} & \text{apply}(f, (e_1 \ldots e_n)) = \text{apply}(\Xi(f), (e_1 \ldots e_n)) \\
\\
\forall f \in \{\texttt{spec}\} & \text{apply}(f, (e_1 \ldots e_n)) = \text{eval}(\text{body}(f)) \\
\\
\text{dans} & \Gamma' \equiv ((x_1\ e_1) \ldots (x_n\ e_n)\ \Gamma) \\
\text{où} & \text{args}(f) = x_1 \ldots x_n
\end{array}
$$

Interprète LISP en LISP (de Paul Graham)

```
(defun null. (x)
  (eq x '()))

(defun and. (x y)
  (cond (x (cond (y 't) ('t '())))
        ('t '())))

(defun not. (x)
  (cond (x '())
        ('t 't)))

(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons (car x) (append. (cdr x) y)))))

(defun list. (x y)
  (cons x (cons y '())))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list. (car x) (car y))
               (pair. (cdr x) (cdr y))))))

(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ;; Special forms
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom) (atom (eval. (cadr e) a)))
       ((eq (car e) 'eq) (eq (eval. (cadr e) a) (eval. (caddr e) a)))
       ((eq (car e) 'car) (car (eval. (cadr e) a)))
       ((eq (car e) 'cdr) (cdr (eval. (cadr e) a)))
       ((eq (car e) 'cons) (cons (eval. (cadr e) a) (eval. (caddr e) a)))
       ((eq (car e) 'cond) (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a) (cdr e)) a))))
    ;; Anonymous function
    ((eq (caar e) 'lambda)
     (eval. (caddar e) (append. (pair. (cadar e) (evlis. (cdr e) a)) a)))))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun evcon. (c a)
   (cond ((eval. (caar c) a)
          (eval. (cadar c) a))
          ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
   (cond ((null. m) '())
          ('t (cons (eval. (car m) a) (evlis. (cdr m) a)))))
```

Portée dynamique

```
> (define foo 1)
foo
> (define add-foo (lambda (x) (+ x foo)))
add-foo
> ((lambda (foo) (add-foo 3)) 5)
4
```

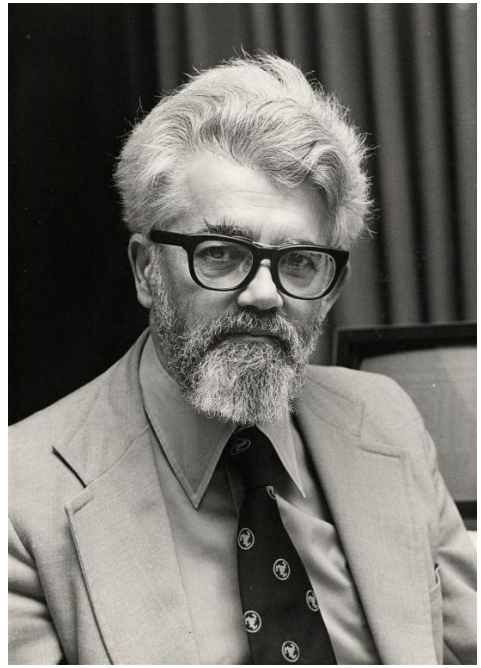
```
> (define foo 1)
foo
> (define add-foo (lambda (x) (+ x foo)))
add-foo
> ((lambda (foo) (add-foo 3)) 5)
8
```

Exercice

Parmi ces deux comportements, lequel correspond à notre interprète ?

Un peu d'histoire – retour en 1960

"Recursive Functions of Symbolic Expressions
and
Their Computation by Machine, Part I"



J. McCarthy

- LISP est le langage de haut niveau le plus ancien après FORTRAN.
- Il a été un véritable laboratoire pour les premières expérimentations des traitements « symboliques » de l'information (gestion de connaissance, logique, transformation et analyse de programmes . . .).

Les LISPs modernes

- LISP vit toujours !
- Ses descendants sont SCHEME, COMMON-LISP, E-LISP, CLOJURE...
- Les idées directrices de ces langages sont :
  - La dynamicité :
    Le programmeur peut toujours évaluer n'importe quelle S-expression.
    Aucun système de type ne restreint l'expressivité du langage. [2]
  - La reflexivité :
    Les programmes sont des S-expressions comme les autres.
    On peut certes les évaluer mais aussi les observer, transformer, calculer...
  - La généralité :
    Comme tous les langages fonctionnels, on cherche à écrire des fonctions les plus générales possibles pour augmenter la réutilisabilité.
- LISP a aussi évolué :
  - On utilise maintenant par défaut la **portée lexicale statique**.
  - Les implémentations modernes de LISP fournissent un **compilateur**.

[2] Cependant, des langages comme SCHEME proposent de plus en plus d'analyse statique pour aider le programmeur à ne pas faire d'erreurs stupides...

Du λ-calcul aux fermetures

Introduction

- Problème dans la théorie des ensembles (de l'époque) :

$$y = \{x \mid x \notin x\}$$

B. Russell

A. Church

- Autre théorie des ensembles centrée sur les fonctions :

Le λ-calcul

Syntaxe du λ-calcul

e	$::=$		***Expression***
	$\mid$	$x, f, \ldots$	*Variable*
	$\mid$	$\mathbf{c}$	*Constante* $\mathbf{c} \in \mathbb{C}$
	$\mid$	$e\ e$	*Application*
	$\mid$	$\lambda x.\ e$	*Fonction*
a, v	$::=$		***Valeurs***
	$\mid$	$x, f, \ldots$	*Variable*
	$\mid$	$\mathbf{c}$	*Constante* $\mathbf{c} \in \mathbb{C}$
	$\mid$	$\lambda x.\ e$	*Fonction*

- Le λ (le mot-clé `fun`) joue une rôle de **lieur** (comme le mot-clé `let`).

Exercice

Définissez l'ensemble des variables libres d'une expression e.

Syntaxe du λ-calcul

e	$::=$		***Expression***
	$\mid$	$x, f, \ldots$	*Variable*
	$\mid$	$\mathbf{c}$	*Constante* $\mathbf{c} \in \mathbb{C}$
	$\mid$	$e\ e$	*Application*
	$\mid$	$\texttt{fun}\ x \Rightarrow e$	*Fonction*
a, v	$::=$		***Valeurs***
	$\mid$	$x, f, \ldots$	*Variable*
	$\mid$	$\mathbf{c}$	*Constante* $\mathbf{c} \in \mathbb{C}$
	$\mid$	$\texttt{fun}\ x \Rightarrow e$	*Fonction*

- Le λ (le mot-clé `fun`) joue une rôle de **lieur** (comme le mot-clé `let`).

Exercice

Définissez l'ensemble des variables libres d'une expression e.

Associativité et priorité

- « $e_1\ e_2\ e_3$ » se lit « $(e_1\ e_2)\ e_3$ »
- « $\texttt{fun}\ x \Rightarrow e_1\ e_2$ » se lit « $\texttt{fun}\ x \Rightarrow (e_1\ e_2)$ ».

Définition des variables libres

$$
\begin{array}{rcll}
FV(x) & = & \{x\} & \\
FV(\mathtt{n}) & = & \emptyset & \mathtt{n} \in \mathbb{N} \\
FV(\mathtt{fun}\ x \Rightarrow e) & = & FV(e) \setminus \{x\} & \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) &
\end{array}
$$

Exercice

Comment définir le mécanisme de substitution sur ce langage ?

Substitution (fausse)

$$
\begin{array}{rcll}
x\{y \mapsto e\} & = & x & \text{si } x \neq y \\
x\{y \mapsto e\} & = & e & \text{si } x = y \\
\mathtt{n}\{y \mapsto e\} & = & \mathtt{n} & \mathtt{n} \in \mathbb{N} \\
(\mathtt{fun}\ x \Rightarrow e')\{y \mapsto e\} & = & \mathtt{fun}\ x \Rightarrow (e'\{y \mapsto e\}) & \\
(e_1\ e_2)\{y \mapsto e\} & = & e_1\{y \mapsto e\} \cup e_2\{y \mapsto e\} &
\end{array}
$$

Exercice

Par analogie avec la substitution déjà définie pour les expressions avec `lets`, pourquoi cette définition est-elle incorrecte ?

Premier exemple

$$(\texttt{fun}\ x \Rightarrow x)\{x \mapsto \texttt{1}\} = \texttt{fun}\ x \Rightarrow (x\{x \mapsto \texttt{1}\})$$

Premier exemple

$$(\texttt{fun}\ x \Rightarrow x)\{x \mapsto \texttt{1}\} = \texttt{fun}\ x \Rightarrow (x\{x \mapsto \texttt{1}\})$$

- À gauche : la fonction identité.
- À droite : la fonction constante égale à 1.

Second exemple

$$(\texttt{fun}\, y \Rightarrow x + y)\{x \mapsto y\} = \texttt{fun}\, y \Rightarrow ((x + y)\{x \mapsto y\})$$

Second exemple

$$(\texttt{fun}\, y \Rightarrow x + y)\{x \mapsto y\} = \texttt{fun}\, y \Rightarrow ((x + y)\{x \mapsto y\})$$

- À gauche : la fonction qui ajoute x à son argument.
- À droite : la fonction qui calcule le double de son argument.

Substitution sans capture

$$
\begin{array}{rcll}
x\{y \mapsto e\} & = & x & \text{si } x \neq y \\
x\{y \mapsto e\} & = & e & \text{si } x = y \\
\mathtt{n}\{y \mapsto e\} & = & \mathtt{n} & \mathtt{n} \in \mathbb{N} \\
(\mathtt{fun}\ x \Rightarrow e')\{y \mapsto e\} & = & \mathtt{fun}\ x \Rightarrow (e'\{y \mapsto e\}) & \text{si } x \neq y \text{ et } x \notin FV(e) \\
(e_1\ e_2)\{y \mapsto e\} & = & e_1\{y \mapsto e\} \cup e_2\{y \mapsto e\} &
\end{array}
$$

Sémantique à petits pas

$$(\beta\text{-réduction})\ \frac{}{(\texttt{fun}\ x \Rightarrow e_1)\ e_2 \longrightarrow e_1\{x \mapsto e_2\}}$$

$$(\delta\text{-réduction})\ \frac{}{\texttt{c}\ v_1 \ldots v_n \longrightarrow \delta_c(v_1, \ldots, v_n)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'} \qquad \frac{e \longrightarrow e'}{\texttt{fun}\ x \Rightarrow e \longrightarrow \texttt{fun}\ x \Rightarrow e'}$$

Exercice

Quelles sont les règles de calculs ? Les règles de passage au contexte ?

Codage du `let`

- On peut maintenant définir la construction « `let ... in ...` » comme un sucre syntaxique, c'est-à-dire une construction extérieur au langage noyau.

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \equiv (\texttt{fun}\ x \Rightarrow e_2)\ e_1$$

Quelques constantes utiles

$0, 1, \ldots$	
$+$	avec $\delta_+(n_1, n_2) = n_1 +_{\mathbb{N}} n_2, \ldots$
$\texttt{true}, \texttt{false}$	
$\texttt{if}$	avec $\delta_{\texttt{if}}(\texttt{true}, v_1, v_2) = v_1$ et $\delta_{\texttt{if}}(\texttt{false}, v_1, v_2) = v_2$
$\texttt{fix}$	avec $\delta_{\texttt{fix}}(v) = v\ (\texttt{fix}\ v)$

- En fait, ces constantes sont définissables en λ-calcul.
  (Cours de sémantique de M1)

Exemple

$$
\begin{array}{l}
mul \equiv \texttt{fun}\ m \Rightarrow \texttt{fix}\ (\\
\qquad \texttt{fun}\ mul' \Rightarrow \texttt{fun}\ n \Rightarrow \\
\qquad\quad \texttt{if}(m = 0, 0, \texttt{if}(n = 0, 0, mul'(n-1) + m)) \\
)
\end{array}
$$

Exercice

Comment s'évalue l'expression « *mul* 0 2 » ?

La nécessité d'une stratégie d'évaluation

- Le calcul défini par la sémantique à petits pas précédente est très général.

$\Rightarrow$ Ses propriétés seront l'objet du cours de sémantique de M1.

- Cependant, les langages de programmation fonctionnelle utilisent des règles de calcul moins générales, dirigées par stratégie d'évaluation dans le but d'améliorer son e□cacité (en pratique) et de simplifier la compilation.
- Par exemple, il est interdit d'évaluer sous le corps des fonctions en dehors des appels de fonctions. De cette façon, on peut compiler une fois le corps de chaque fonction.
- De plus, une stratégie d'évaluation fournit un procédé déterministe pour décider de façon univoque la façon de poursuivre l'évaluation d'un terme.

Quelques stratégies importantes

- Réduire le redex le plus externe.

$\Rightarrow$ Stratégie LMOM (*LeftMost OuterMost*) ou RMOM (*RightMost OuterMost*)

- Contraindre le terme de droite d'une β-réduction à être une valeur, ou non.

$\Rightarrow$ Stratégie d'appel par valeur ou par nom.

Exercice

Comment spécifier ces stratégies ?

Contexte d'évaluation

- Il est pratique de spécifier les contextes d'évaluation **syntaxiquement**.
- On définit des contextes d'évaluation à l'aide de grammaires.
- On s'assure que, pour tout contexte, il existe un unique sous-terme « **trou** ».
- Ce trou sert à spécifier où la réduction doit avoir lieu.
- On **décompose** toute expression e sous la forme d'un contexte d'évaluation $\mathcal{C}$ et d'un sous-terme réductible e' prenant la place du trou, ce que l'on note « $\mathcal{C}\,[e']$ ».

Sémantique à petits pas avec contexte d'évaluation

$(\texttt{fun}\ x \Rightarrow e)\ e'$	$\longrightarrow$	$e\{x \mapsto e'\}$	*(β-reduction)*
$\mathcal{C}\ [e]$	$\longrightarrow$	$\mathcal{C}\ [e']$	*(Réduction sous-contexte)* *si* $e \longrightarrow e'$

En appel par valeur, arguments évalués de droite à gauche

- On remplace la $\beta-$réduction arbitraire en :

$$(\texttt{fun}\ x \Rightarrow e)\ v \quad \longrightarrow \quad e\{x \mapsto v\} \quad (\beta_v\text{-reduction})$$

- On définit les contextes d'évaluation :

$\mathcal{C}$	::=		***Contexte d'évaluation***
	|	[]	*Trou*
	|	$\mathcal{C}\ e$	*Réduction à gauche*
	|	$v\ \mathcal{C}$	*Réduction à droite*

- Exemples de décomposition :
  - « $(\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1)$ »
  - « $((\texttt{fun}\ x \Rightarrow \texttt{fun}\ y \Rightarrow y)\ 1)\ 42$ »

En appel par valeur, arguments évalués de droite à gauche

- On remplace la $\beta-$réduction arbitraire en :

$$(\texttt{fun}\ x \Rightarrow e)\ v \quad \longrightarrow \quad e\{x \mapsto v\} \quad (\beta_v\text{-reduction})$$

- On définit les contextes d'évaluation :

$\mathcal{C}$	$::=$		***Contexte d'évaluation***
	$\mid$	$[]$	*Trou*
	$\mid$	$\mathcal{C}\ e$	*Réduction à gauche*
	$\mid$	$v\ \mathcal{C}$	*Réduction à droite*

- Exemples de décomposition :
  - « $(\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1)$ »

$$\begin{cases} \mathcal{C} \equiv (\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ [] \\ e \equiv (\texttt{fun}\ z \Rightarrow z)\ 1 \end{cases}$$

  - « $((\texttt{fun}\ x \Rightarrow \texttt{fun}\ y \Rightarrow y)\ 1)\ 42$ »

En appel par valeur, arguments évalués de droite à gauche

- On remplace la $\beta-$réduction arbitraire en :

$$(\texttt{fun}\ x \Rightarrow e)\ v \quad \longrightarrow \quad e\{x \mapsto v\} \quad (\beta_v\text{-reduction})$$

- On définit les contextes d'évaluation :

$\mathcal{C}$	::=		***Contexte d'évaluation***
	|	[]	*Trou*
	|	$\mathcal{C}\ e$	*Réduction à gauche*
	|	$v\ \mathcal{C}$	*Réduction à droite*

- Exemples de décomposition :
  - « $(\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1)$ »

$$\begin{cases} \mathcal{C} \equiv (\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ [] \\ e \equiv (\texttt{fun}\ z \Rightarrow z)\ 1 \end{cases}$$

  - « $((\texttt{fun}\ x \Rightarrow \texttt{fun}\ y \Rightarrow y)\ 1)\ 42$ »

$$\begin{cases} \mathcal{C} \equiv []\ 42 \\ e \equiv (\texttt{fun}\ x \Rightarrow \texttt{fun}\ y \Rightarrow y)\ 1 \end{cases}$$

En appel par nom

- Avec la $\beta-$réduction classique :

$\mathcal{C}$	::=		***Contexte d'évaluation***
	|	[]	*Trou*
	|	$\mathcal{C}\ e$	*Réduction à gauche*

- Exemples de décomposition :
  - « $(\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1)$ »

$$\begin{cases} \mathcal{C} \equiv [] \\ e \equiv (\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1) \end{cases}$$

  - « $((\texttt{fun}\ x \Rightarrow \texttt{fun}\ y \Rightarrow y)\ 1)\ 42$ »

En appel par nom

- Avec la $\beta-$réduction classique :

$\mathcal{C}$	::=		***Contexte d'évaluation***
	|	[]	*Trou*
	|	$\mathcal{C}\ e$	*Réduction à gauche*

- Exemples de décomposition :
  - « $(\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1)$ »

$$\begin{cases} \mathcal{C} \equiv [] \\ e \equiv (\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1) \end{cases}$$

  - « $((\texttt{fun}\ x \Rightarrow \texttt{fun}\ y \Rightarrow y)\ 1)\ 42$ »

En appel par nom

- Avec la $\beta-$réduction classique :

$\mathcal{C}$	$::=$		***Contexte d'évaluation***
	$\mid$	$[]$	*Trou*
	$\mid$	$\mathcal{C}\ e$	*Réduction à gauche*

- Exemples de décomposition :
  - « $(\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1)$ »

$$\begin{cases} \mathcal{C} \equiv [] \\ e \equiv (\texttt{fun}\ x \Rightarrow (\texttt{fun}\ y \Rightarrow y)\ 1)\ ((\texttt{fun}\ z \Rightarrow z)\ 1) \end{cases}$$

  - « $((\texttt{fun}\ x \Rightarrow \texttt{fun}\ y \Rightarrow y)\ 1)\ 42$ »

$$\begin{cases} \mathcal{C} \equiv []\ 42 \\ e \equiv (\texttt{fun}\ x \Rightarrow \texttt{fun}\ y \Rightarrow y)\ 1 \end{cases}$$

Pour la suite. . .

- Dans ce cours, nous nous focaliserons sur l'appel par valeur, plus courant.
- Vous étudierez l'appel par nom en :
  - Programmation fonctionnelle avancée (M1).
  - Programmation comparée (M1).
  - Sémantique (M1).

$\Rightarrow$ Un ra□nement, l'évaluation paresseuse, est à la base du langage HASKELL.

Sémantique à grands pas pour les valeurs closes

- On définit la relation « $e \Downarrow v$ » qui signifie :

  « L'expression close e s'évalue en la valeur v. »

$$\frac{}{\texttt{fun}\ x \Rightarrow e \Downarrow \texttt{fun}\ x \Rightarrow e} \qquad \frac{}{\texttt{n} \Downarrow \texttt{n}} \qquad \frac{e_1 \Downarrow \texttt{fun}\ x \Rightarrow e \qquad e_2 \Downarrow a}{e_1\ e_2 \Downarrow e\{x \mapsto a\}}$$

Sémantique à grands pas (tentative)

- On définit la relation « $\rho \vdash e \Downarrow v$ » qui signifie :
  « L'expression e s'évalue en la valeur v dans l'environnement ρ. »
- L'environnement ρ associe une valeur close à chaque variable libre.

$$\frac{}{\rho \vdash x \Downarrow \rho(x)} \qquad \frac{}{\rho \vdash \texttt{fun}\ x \Rightarrow e \Downarrow \texttt{fun}\ x \Rightarrow e} \qquad \frac{}{\rho \vdash \texttt{n} \Downarrow \texttt{n}}$$

$$\frac{\rho \vdash e_2 \Downarrow a \qquad \rho \vdash e_1 \Downarrow \texttt{fun}\ x \Rightarrow e \qquad \rho + x \mapsto a \vdash e \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v}$$

$\Rightarrow$ Est-ce correct ?

Sémantique à grand pas

- Il faut adjoindre l'environnement d'évaluation d'une fonction au moment de sa création pour pouvoir l'évaluer plus tard, quand le contexte d'évaluation aura éventuellement changé.
- On note "$e[\rho]$", l'objet clos obtenu, appelé fermeture.
- P. J. Landin, *"The mechanical evaluation of expressions"*, The Computer Journal, 1964.

$$\frac{}{\rho \vdash x \Downarrow \rho(x)} \qquad \frac{}{\rho \vdash \texttt{fun}\ x \Rightarrow e \Downarrow (\texttt{fun}\ x \Rightarrow e)[\rho]} \qquad \frac{}{\rho \vdash \texttt{n} \Downarrow \texttt{n}}$$

$$\frac{\rho \vdash e_2 \Downarrow a \qquad \rho \vdash e_1 \Downarrow (\texttt{fun}\ x \Rightarrow e)[\rho'] \qquad \rho' + x \mapsto a \vdash e \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v}$$

Capture de l'environnement

$$\frac{}{\rho \vdash \texttt{fun}\ x \Rightarrow e \Downarrow (\texttt{fun}\ x \Rightarrow e)[\rho]}$$

- On **étend la syntaxe des valeurs.**

a, v	$::=$		***Valeurs***
	$\mid$	$x, f, \ldots$	*Variable*
	$\mid$	$\mathbf{c}$	*Constante* $\mathbf{c} \in \mathbb{C}$
	$\mid$	$(\texttt{fun}\ x \Rightarrow e)[\rho]$	*Fermeture*

Sémantique à grand pas : règle de l'application

$$\frac{\rho \vdash e_2 \Downarrow a \qquad \rho \vdash e_1 \Downarrow (\texttt{fun}\ x \Rightarrow e)[\rho'] \qquad \rho' + x \mapsto a \vdash e \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v}$$

- Quand on veut évaluer le corps d'une fonction, on **rétablit** son environnement d'évaluation.

Expressions bloquées

Exercice

Énumérez quelques expressions qui ne sont pas des valeurs mais qui ne sont pas réductibles.

Système de type simple

- On étend la syntaxe des types du langage des expressions arithmétiques :

$$
\begin{array}{rcl}
\tau & ::= & \texttt{int} \\
 & | & \texttt{bool} \\
 & | & \tau \rightarrow \tau
\end{array}
$$

Système de type simple pour le λ-calcul

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathtt{i} : \mathtt{int}} \quad \mathtt{i} \in \mathbb{N}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma; (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}\ x \Rightarrow e : \tau_1 \to \tau_2}$$

Un algorithme d'inférence (partielle) de type

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x \Uparrow \tau} \qquad \frac{}{\Gamma \vdash \mathtt{i} \Uparrow \mathtt{int}} \quad \mathtt{i} \in \mathbb{N}$$

$$\frac{\Gamma \vdash e_1 \Uparrow \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \Downarrow \tau_1}{\Gamma \vdash e_1\ e_2 \Uparrow \tau_2} \qquad \frac{\Gamma; (x : \tau_1) \vdash e \Downarrow \tau_2}{\Gamma \vdash \mathtt{fun}\ x \Rightarrow e \Downarrow \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e \Uparrow \tau}{\Gamma \vdash e \Downarrow \tau} \qquad \frac{\Gamma \vdash e \Downarrow \tau}{\Gamma \vdash (e : \tau) \Uparrow \tau}$$

Un système trop restrictif ?

- Voici un programme :

```
let id = fun x ⇒ x in
  if id true then id 42 else 21
```

- Ce programme n'est pas bloqué.
- Ce programme est mal typé pour de mauvaises raisons.
- En e□et, quel type donner à id ?

⇒ Successivement, le type `bool` → `bool` et le type `int` → `int`.

Le polymorphisme à la ML étend
les types des valeurs nommées par des `lets`
à des familles de types, appelées schémas de type.

⇒ Cours de compilation de M1.

Synthèse

Synthèse

- Aujourd'hui : la représentation du code comme une valeur de première classe.
- La prochaine fois :
  - Comment compiler le $\lambda-$calcul vers une machine virtuelle ?
  - Comment compiler le $\lambda-$calcul vers un langage du premier ordre ?
  - Les fermetures pour les langages objets.