Paradigmes de programmation

Loïg JEZEQUEL

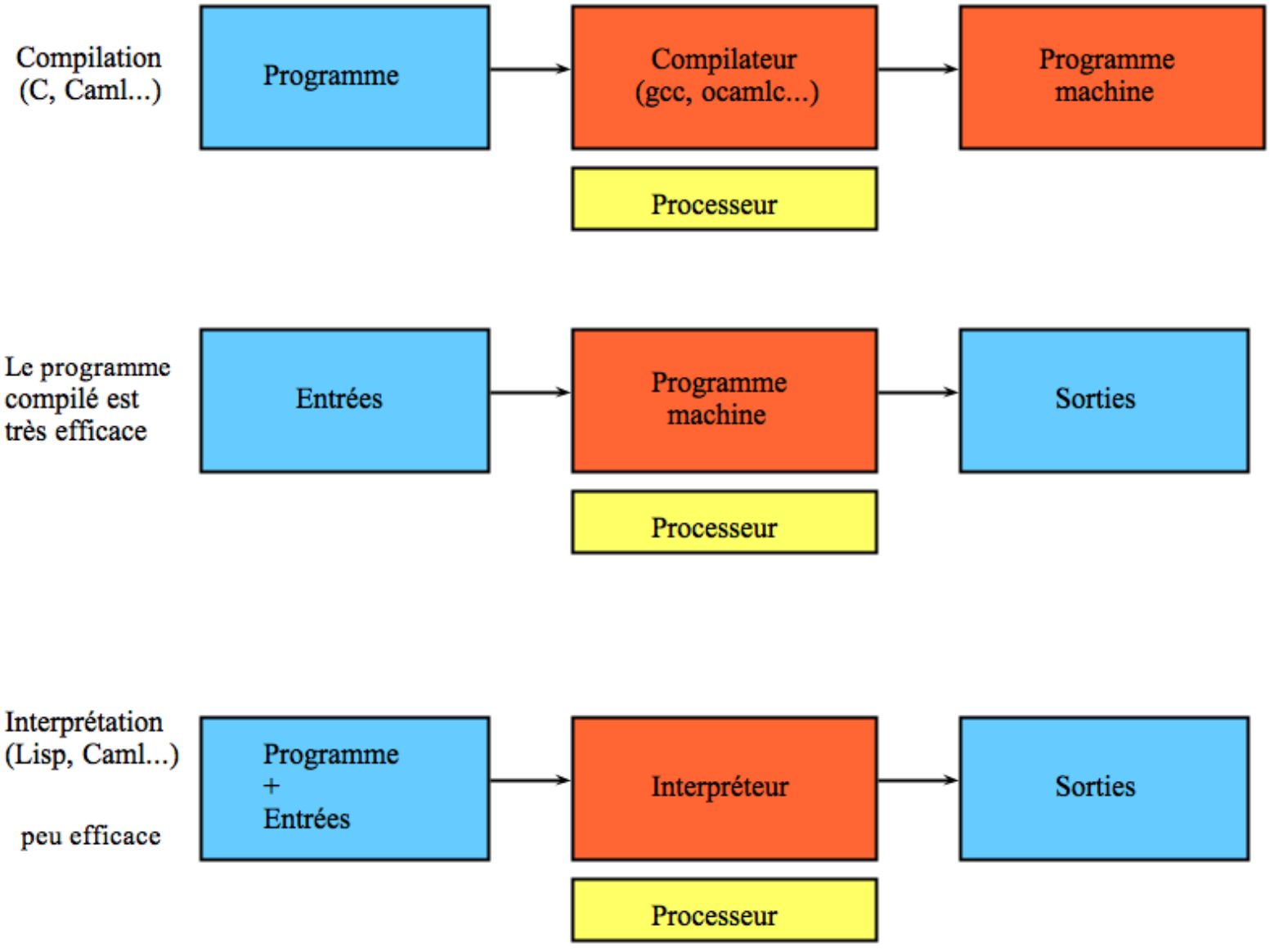
ENS Cachan Bretagne

16 novembre 2009

Introduction

- Les langages de programmation :
  - décrivent des calculs de façon plus abstraite qu'en langage machine
- Les programmes écrits dans ces langages sont :
  - soit compilés (= transformés en langage machine)
  - soit interprétés (= exécutés par un autre programme)
- Un compilateur :
  - génère un programme machine pour un processeur donné, dont la sémantique est donnée par le programme source
- Un interpréteur :
  - lit un programme et calcule directement sa sémantique pour une entrée donnée

Compilation et interprétation



Machine virtuelle

- Une machine virtuelle est un programme qui exécute des programmes (bytecode), c'est donc un interpréteur
- Le langage compris par une machine virtuelle est proche de celui compris par une machine réelle (c'est pourquoi on parle de machine virtuelle)
- Les programmes a exécuter sont obtenus par compilation d'un langage de haut niveau
- Il y a autant de versions de la machine virtuelle que de machines réelles différentes

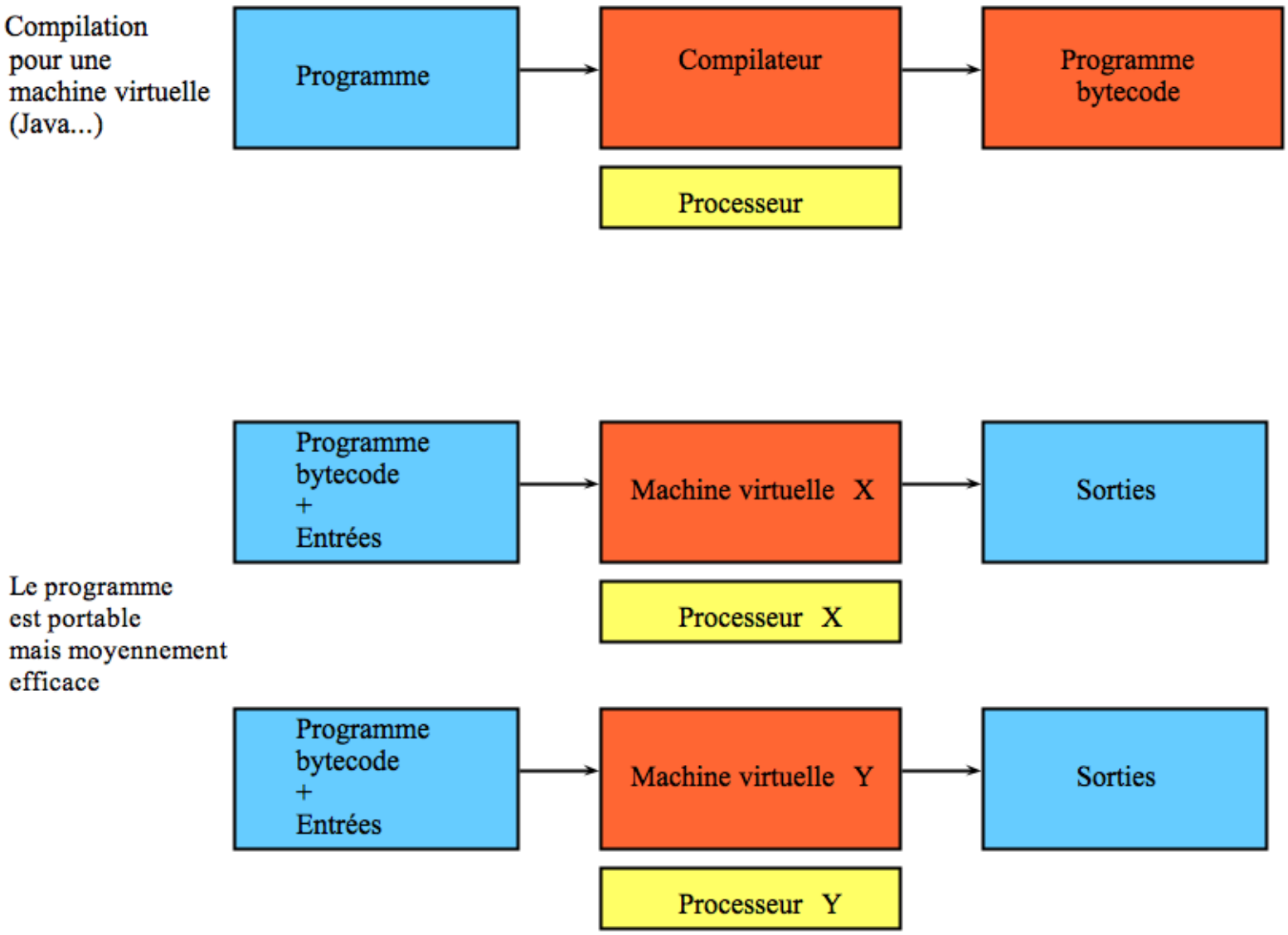
Intérêt

Les programmes compilés pour une machine virtuelle sont portables

Inconvénient

L'éxécution est plus lente que celle des programmes compilés pour une machine réelle

Machine virtuelle



Exemple de compilation : factorielle

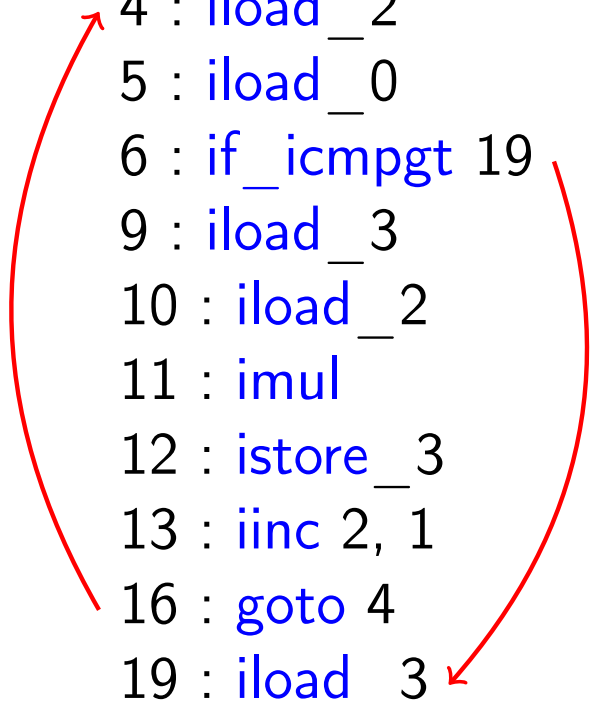
Java :

```
int k=1,l=1;
while( k<=i ){
   l=l*k;
   k++
}
return l;
```

$\longrightarrow$

Bytecode :

```
0 : iconst_1
1 : istore_2
2 : iconst_1
3 : istore_3
4 : iload_2
5 : iload_0
6 : if_icmpgt 19
9 : iload_3
10 : iload_2
11 : imul
12 : istore_3
13 : iinc 2, 1
16 : goto 4
19 : iload_3
20 : ireturn
```



En résumé

- Les compilateurs, interpréteurs, machines virtuelles se chargent :
  - de l'interface avec la machine
  - de la portabilité
  - des performances (en partie)
- Les langages de programmation offrent :
  - de l'expressivité (structuration des programmes, raisonement plus abstrait)
  - une spécialisation plus ou moins grandes pour certains problèmes (langages dédiés)
  - certaines garanties de correction, de sécurité, grâce à des concepts sémantiques forts (typage...)

Plan

Plan

Programmation impérative

Principe

Le calcul à effectuer est donné à l'aide de commandes qui changent l'état du programme : un programme impératif est une séquence de commandes à exécuter

- Structures de contrôle :
  - structurer le flôt d'exécution
- Types de données :
  - tableaux, structures modifiables « en place »
  - les fonctions ne sont pas des données (voir programmation fonctionnelle)
- Effets de bords :
  - certaines fonctions ne retourne rien, elles ne sont utiles que par leurs effets de bord (i.e. la façon dont elles modifient l'état du programme)
  - ces fonctions sont aussi appelées procédures

Structures de contrôle

- Les plus communes :
  - if c then e else e'
  - while c do e
  - for i=e1 to e2 do e
- Aujourd'hui le contrôle de bas niveau (goto) n'est plus utilisé, sauf dans quelques cas très spécifiques (code optimisé pour des pilotes...)
- Cependant les return, break, continue de Java peuvent simplifier certains algorithmes
- Les exceptions permettent de remonter brutalement dans la structure de contrôle et permettent, en plus, de retourner une valeur exceptionnelle, utilisable ensuite

Compilation des langages impératifs

- En simplifiant à l'extrême (on oublie les données et l'optimisation) la compilation d'un programme impératif consiste a transformer les structures de contrôle de haut niveau en branchements élémentaires compréhensibles par le processeur :

```
while condition
do corps
```

$\longrightarrow$

```
i0   : condition
...  : condition
i1   : jump_if_zero i2
i1+1 : corps
...  : corps
i2-1 : goto i1
i2   : suite
```

- Les tableaux sont représentés par des adresses contiguës et on calcule l'adresse d'une case en ajoutant son indice à l'adresse du tableau (de même pour les types structures)

Plan

Programmation objet : associer données et traitements

- Les objets sont des valeurs qui contiennent à la fois :
  - un état (des données mutables)
  - des méthodes qui font référence à l'état (de cet objet)
- On veut cacher les détails du fonctionnement interne des objets (encapsulation)

```
class Counter {
  private int i;
  public Counter {i = 0;}
  public int get() {return i;}
  public void incr() {i++;}
}
```

- Une classe définit à la fois :
  - un type d'objets
  - des moyens de construire des objets de ce type
  - leur comportement, de façon générique, au travers des méthodes

Éléments d'un langage objet

- Classes et objets : représentation de types de données abstraits
  - les classes sont des modèles d'objets
- Héritage
  - d'interfaces
  - d'implémentations
- Liaison dynamique et généricité
- Liaison dynamique et héritage

Composition des objets

- Des attributs d'instance (champs d'un enregistrement)

```
int n = 10; ...

Object x = ...
```

- Des méthodes d'instance (qui peuvent accéder aux attributs)

```
void change(int m) {n = m}

bool greater(Object o) {return (n >= o.n);}

int factorielle(int i) {
        if (i == 0) return 1;
        else return i*factorielle(i - 1);}
```

- Idée : rapprocher les données des opérations associées

Composition des classes

- Une classe est un type :
  - type des attributs d'instance
  - signature des méthodes
- Une classe est un modèle :
  - initialisation des attributs par un constructeur

```
MaClasse(int n, Object o)
          {this.n = n; this.o = o;}
```

  - corps des méthodes

En Java

Définition :

```
class Counter = {
 //attributs
 int x;
 //constructeurs
 Counter(x) {this.x = x;}
 Counter() {x = 0;}
 //méthodes
 int value() {return x;}
 void incr(int i) {x = x + i;}
}
```

Utilisation :

```
Counter c1, c2;
c1 = new Counter(3);
c2 = new Counter();
c1.incr(2);
int v;
v = c2.value();
```

Remarques

- Les méthodes sont mutuellement récursives
- dans une classe deux méthodes distinguables par leurs arguments peuvent avoir le même nom, on parle de surcharge
- tout objet est construit à partir d'une classe qui est son type

Attributs et méthodes d'instance et de classe

- Les attributs et méthodes d'instance existent en un exemplaire par objet :

```
class C { ... int x; ...}
//un attribut x par objet de classe C
class C { ... void m(); ...}
//une méthode m par objet de classe C
```

- Les attributs et méthodes de classe (appelés statiques en Java) existent en un seul exemplaire pour toute la classe :

```
class C { ... static int x; ...}
//un seul x pour tous les objets de classe C
class C { ... static void m(); ...}
//une seule méthode m pour tous les objets
```

Héritage d'interfaces

Interface

Une interface regroupe des signatures de méthodes qu'un objet doit posséder pour implémenter cette interface : c'est la notion de type pour les objets.

- L'héritage d'interface est un moyen d'étendre une interface en ajoutant des méthodes. Les objets qui implémentent une sous-interface sont compatibles avec la super-interface.

```
interface Collection {
 public void add(Object o);
 public bool isEmpty();
}

interface List extends Collection {
 public Object get(int index);
}
```

Héritage d'implémentations

- L'héritage d'implémentations permet d'étendre une classe en une autre en lui ajoutant des attributs et des méthodes. Il permet le partage de méthodes entre différentes classes.

```
class RazCounter extends Counter {
 void raz() {x = 0;}
}

//utilisation de méthodes de la super-classe
class UnCounter extends Counter {
 void decr(int i) {incr(-i);}
}

//redéfinition de méthodes de la super-classe
class CounterModulo extends Counter{
 void incr(int i) {x = x + i % 2;}
}
```

Liaison dynamique

Que se passe-t-il lorsqu'on redéfinit une méthode qui était utilisée par une autre classe (comme entre les classes Counter et CounterModulo) ?

- On ne connaît la méthode à appeler qu'à l'exécution ! C'est celle qui est définie dans la classe de l'objet appelant la méthode.
- Il est en général impossible (indécidable) de savoir quelle méthode sera appelée effectivement.

Héritage et liaison dynamique

L'héritage d'implémentations est un bon moyen de réutiliser du code : une méthode héritée est partagée avec la super-classe (rôle important de la liaison dynamique).

```
class Collection {
 public abstract void add(Object o);
 public void addAll(Collection c){
  for(...) add(...);
 }
}

class List extends Collection {
 public void add(Object o) {...}
}

class Set extends Collection {
 public void add(Object o) {...}
}
```

Compatibilité et liaison dynamique

- Des objets qui partagent un certain nombre de méthodes (même nom et même signature) ont une certaine compatibilité

```
Class List {
  ...
  public void add(Object o) {...}
}

Class Set {
  ...
  public void add(Object o) {...}
}
```

- L'intérêt est que l'on peut appeler la méthode add d'un objet sans connaître son type exacte. Il s'agit d'une liaison dynamique.

Conclusion : compilation et exécution

- Contrairement au cas des langages impératifs, un appel de méthode dans un langage à objets ne peut pas toujours être résolu à la compilation. Il faut donc, à l'exécution, chercher la bonne méthode, qui peut se trouver dans la classe de l'objet ou dans une super-classe.
- On peut considérer du point de vue du langage que chaque objet possède ses propres méthodes (dupliquées autant de fois qu'il y a d'objets différents). Cependant dans une implémentation efficace on ne crée qu'une seule fonction pour une classe et une méthode données, l'objet this devient un argument suplémentaire.

Plan

Syntaxe du lambda-calcul

$$
\begin{aligned}
e \quad = \quad & |x \text{ (identificateur)} \\
& |\lambda x.e \text{ (lambda-abstraction)} \\
& |ee \text{ (application)}
\end{aligned}
$$

Exemple :

- en lambda-calcul :

$$\lambda x.\lambda y.(xy)$$

- en Caml :

$$fun\ x \rightarrow fun\ y \rightarrow x\ y$$

Occurences libres/liées, remplacement

- Dans $(\lambda x.x)x$, le second x est une occurence liée de l'identificateur x ; le troisième est une occurence libre.
- On note $e[e'/x]$ le terme obtenu en remplaçant simultanément toutes les occurences libres de x par e' dans e.
  - exemple : $(\lambda x.x)x[(\lambda y.yy)/x] = (\lambda x.x)(\lambda y.yy)$
- Une substitution est licite si aucune variable libre du terme e' n'est capturée par un lambda de e.
  - exemple : $(\lambda x.y)[x/y]$ n'est pas licite

Alpha-équivalence (renommage)

1. si y n'a pas d'occurence libre dans e alors $\lambda x.e \sim \lambda y.e[y/x]$
2. si $e \sim e'$ alors $\lambda x.e \sim \lambda x.e'$
3. si $f \sim f'$ et $e \sim e'$ alors $fe \sim f'e'$

L'alpha-équivalence permet de renommer un identificateur, en évitant le phénomène de capture :

- $\lambda x.\lambda y.xy$
  - est équivalent à $\lambda x.\lambda z.xz$
  - n'est pas équivalent à $\lambda x.\lambda x.xx$

Bêta-réduction

1. si la substitution $e[e'/x]$ est licite, alors $(\lambda x.e)e' \longrightarrow e[e'/x]$
2. si $e \longrightarrow e'$, alors $\lambda x.e \longrightarrow \lambda x.e'$
3. si $e \longrightarrow e'$, alors $ee'' \longrightarrow e'e''$ et $e''e \longrightarrow e''e'$

- Exemple :
  - $((\lambda x.\lambda y.xy)(\lambda x.x))t$

Bêta-réduction

1. si la substitution $e[e'/x]$ est licite, alors $(\lambda x.e)e' \longrightarrow e[e'/x]$
2. si $e \longrightarrow e'$, alors $\lambda x.e \longrightarrow \lambda x.e'$
3. si $e \longrightarrow e'$, alors $ee'' \longrightarrow e'e''$ et $e''e \longrightarrow e''e'$

- Exemple :
  - $((\lambda x.\lambda y.xy)(\lambda x.x))t \longrightarrow (\lambda y.(\lambda x.x)y)t \longrightarrow (\lambda y.y)t \longrightarrow t$
  - $((\lambda x.\lambda y.xy)(\lambda x.x))t \longrightarrow (\lambda y.(\lambda x.x)y)t \longrightarrow (\lambda x.x)t \longrightarrow t$
- Confluence :
  - l'ordre de réduction ne change pas le résultat : il ne peut y avoir qu'une seule forme irréductible d'un lambda-terme. Par contre certains chemins peuvent terminer et d'autres non.

$$(\lambda x.xx)$$

Bêta-réduction

1. si la substitution $e[e'/x]$ est licite, alors $(\lambda x.e)e' \longrightarrow e[e'/x]$
2. si $e \longrightarrow e'$, alors $\lambda x.e \longrightarrow \lambda x.e'$
3. si $e \longrightarrow e'$, alors $ee'' \longrightarrow e'e''$ et $e''e \longrightarrow e''e'$

- Exemple :
  - $((\lambda x.\lambda y.xy)(\lambda x.x))t \longrightarrow (\lambda y.(\lambda x.x)y)t \longrightarrow (\lambda y.y)t \longrightarrow t$
  - $((\lambda x.\lambda y.xy)(\lambda x.x))t \longrightarrow (\lambda y.(\lambda x.x)y)t \longrightarrow (\lambda x.x)t \longrightarrow t$
- Confluence :
  - l'ordre de réduction ne change pas le résultat : il ne peut y avoir qu'une seule forme irréductible d'un lambda-terme. Par contre certains chemins peuvent terminer et d'autres non.
- Le lambda-terme suivant se réduit indéfiniment :

$$(\lambda x.xx)$$

Extensions du lambda-calcul et programmation fonctionnelle

- On peut enrichir le lambda-calcul avec :
  - des entiers et des opérateurs
  - des définitions de constantes locales (let in)
  - des conditionnelles (if then else)
  - la récursivité (let rec)
  - ...
- Tous ces ajouts sont représentables en lambda-calcul pur (cependant pour la récursivité la terminaison dépend de l'ordre de réduction)
- La programmation fonctionnelle pure se contente du lambda-calcul plus ou moins étendu (exemple : Haskell)

Programmation fonctionnelle

- Les lambda-termes clos représentent des fonctions au sens mathématique. Le résultat ne change pas d'un appel à l'autre, il ne dépend pas d'un contexte autre que les arguments.
- Propriété de transparence référentielle : on peut toujours remplacer une expression par sa valeur.
  - en particulier, une application de fonction peut être remplacée par son résultat, ou (par transitivité) par l'appel d'une autre fonction « équivalente »à la première (pas d'effets de bords, seul le résultat compte).
- Les fonctions sont des valeurs comme les autres : au cours de l'exécution on peut les créer, les appliquer, les retourner en résultat, les passer en argument d'autres fonctions.
- Les données ont le plus souvent une structure récursive (listes, arbres) ainsi que les fonctions qui les manipulent.

Ordre de réduction

- L'ordre d'évaluation n'a d'importance que pour :
  - La terminaison (c'est au programmeur d'écrire des programmes qui terminent pour l'ordre considéré)
  - L'efficacité (certains chemins sont moins coûteux que d'autres)
  - Les programmes impératifs (références et effets de bords, entrées/sorties...)
- En Caml : opérandes d'abord
  - toute lambda-abstraction est une valeur (ou forme normale) et n'est donc pas réduite
  - pour réduire (ee'), on réduit e' jusqu'à obtenir une lambda-abstraction, puis on réduit e, puis l'expression globale
  - cette réduction est plus faible que la bêta-réduction (on ne réduit pas sous les lambdas) mais plus naturelle s'il y a des effets de bords ($fun\ x \rightarrow y := 0$)
- Évaluation paresseuse (exemple : Haskell) : on ne réduit que les expressions indispensables, et au maximum une fois

Exemple

- En Caml :

$$(\lambda x.\lambda y.(+\ x\ x))(+\ 1\ 1)(+\ 2\ 2)$$

- Évaluation paresseuse :

$$(\lambda x.\lambda y.(+\ x\ x))(+\ 1\ 1)(+\ 2\ 2)$$

Exemple

- En Caml :

$$
\begin{aligned}
(\lambda x.\lambda y.(+\ x\ x))(+\ 1\ 1)(+\ 2\ 2) &\rightarrow \\
(\lambda x.\lambda y.(+\ x\ x))(+\ 1\ 1)4 &\rightarrow \\
(\lambda x.\lambda y.(+\ x\ x))2\ 4 &\rightarrow \\
(\lambda y.(+\ 2\ 2))4 &\rightarrow \\
+\ 2\ 2 &\rightarrow \\
4 &
\end{aligned}
$$

- Évaluation paresseuse :

$$
\begin{aligned}
(\lambda x.\lambda y.(+\ x\ x))(+\ 1\ 1)(+\ 2\ 2) &\rightarrow \\
(\lambda y.(+\ (+\ 1\ 1)(+\ 1\ 1)))(+\ 2\ 2) &\rightarrow \\
+\ (+\ 1\ 1)(+\ 1\ 1) &\rightarrow \\
+\ 2\ 2 &\rightarrow \\
4 &
\end{aligned}
$$

Un interpréteur très simple

```
let rec eval = function
    | e e' ->
       let e'r = eval e' in
          let Lx.e'' = eval e in
             eval e''[e'r/x]
    | e -> e (*c'est une valeur*)
```

Notion de contexte

- Pour éviter des remplacements coûteux, on préfère évaluer un terme dans un contexte, qui associe des termes à des identificateurs :

```
let rec eval c = function
    | e e' ->
        let e'r = eval c e' in
            let Lx.e'' = eval c e in
                eval [e'r/x]::c e''
    | x -> c(x)
    | e -> e (*c'est une valeur*)
```

où $[e'r/x] :: c$ est le contexte c enrichi par l'association de x à $e'r$. Tout identificateur x déjà défini dans c est masqué temporairement (liaison statique).

Plan

Lambda-calcul simplement typé (présentation infromelle)

- Types :
  - types de bases (int, bool) ou variables de types
  - types flèches : si t et t' sont des types, on forme $t \to t'$
- Lambda-termes typés
  - notation : $e : t$ pour e est de type t
- Jugement de type :
  - $\Gamma \vdash e : t$ où e est un terme, t est un type et Γ est un contexte

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t.e : t \to t'} \qquad \frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash ee' : t'}$$

Well-typed programs never go wrong

- Les types assurent une certaine cohérence aux programmes. Cela se traduit par un théorème sur la réduction dans un lambda-calcul enrichi :

  préservation un programme bien typé ne peut se réduire qu'en un programme bien typé

  progrès un programme bien typé ne peut être bloqué (exemple : (1 2), (plus true 0) et (1 := 2) sont bloqués)

- Dans la réalité le blocage correspond à des opérations absurdes consistant par exemple à confondre des entiers et des adresses mémoire.
- D'après la propriété de préservation, il suffit de vérifier le typage avant exécution, et on peut l'oublier ensuite : le typage est statique

Systèmes de types

- De nombreux langages possèdent :
  - des types de base : int, char, string, bool...
  - des types pointeur, tableau, fonction
  - la possibilité de définir de nouveaux types produits (structures)
- Un système de type est plus utile lorsqu'il est inviolable.
  - c'est le cas en Java, Caml, Haskell
  - ce n'est pas le cas en C (pour des raisons de performances)
  - Lisp est un langage non typé

Plan

Gestion de la mémoire

- Rappelons nous l'interpréteur du lambda-calcul (version avec contexte) vu plus tôt.
- Lorsque l'on doit évaluer x dans un environnement c, on ne recopie pas vraiment la valeur associée à x dans c : on se contente de retourner la même zone mémoire. . .en espérant qu'elle ne sera pas effacée.
- La durée de vie d'un objet peut dépasser la portée des identificateurs auquels il peut être associé lors de l'exécution.
- On ne peut pas décider statiquement si à tel point de programme un objet peut être libéré ou non.

Ramasse-miettes

Programme qui, de temps en temps au cours de l'exécution du programme principal, regarde quels objets sont devenus inaccessibles et les supprime pour récupérer la mémoire.

- Dans un langage muni d'un système de types sûrs (propriété de préservation) un objet inaccessible à un moment l'est définitivement.

Plan

Programme logique

- Une clause de Horn est une disjonction de formules atomiques dont au plus une est positive :

$$p(f(x)) \vee \neg q \vee \neg r(f(x), g(x, f(x)))$$

- Une telle formule est équivalente à :

$$p(f(x)) \Leftarrow q \wedge r(f(x), g(x, f(x)))$$

- Notée en Prolog :

$$p(f(x)) : -q, r(f(x), g(x, f(x)))$$

Un programme logique est un ensemble fini de clauses de Horn

Exemples

```
triee(cons(X,cons(Y,L))):-inf(X,Y),triee(cons(Y,L)).
triee(nil).
triee(cons(X,nil)).
```

```
pair(0).
impair(s(X)):-pair(X).
pair(s(X)):-impair(X).
```

```
plus(X,0,X).
plus(X,s(Y),s(Z)):-plus(X,Y,Z).
```

Exécution

- Pour exécuter un programme Prolog on se donne un but, qui est une conjonction de formules atomiques (non closes) :

```
plus(s(0),s(s(0)),X)
```

```
pair(X)
```

- L'interpréteur retourne alors une (ou des) substitution(s) qui rend(ent) ce but prouvable (s'il en existe) :

```
X = s(s(s(0)))
```

```
X=0, X=2, X=4, ...
```

Sémantique déclarative

- Un programme logique définit une théorie (l'ensemble des conséquences des formules du programme)
- Pour un but donné, un programme logique peut boucler, ou bien s'arrêter. Dans les deux cas, il peut rendre zéro, une, ou plusieurs substitutions.
  - toute substitution rendue par le programme (même s'il boucle après) fait du but une conséquence de la théorie
  - si le programme termine, alors toutes les conséquences qui ont le forme du but ont été énumérées

Sémantique opérationnelle

- Prolog procède par résolution linéaire directe pour les clauses de Horn. C'est une méthode correcte et complète, d'où les deux propriétés de la sémantique déclarative
- Principe :
  1. on cherche la première clause du programme dont la tête s'unifie avec le but courant (initialement le but visé)
  2. on applique la substitution obtenue à la queue de clause
  3. on cherche alors à prouver (de gauche à droite) les prémices de cette clause (qui deviennent tour à tour le but courant)
  4. lorsque l'on est bloqué dans une impasse on revient en arrière pour essayer une autre clause
- Ce faisant on construit (par unifications successives) une substitution qui rend le but prouvable

Intérêts et défauts

- Adapté à la recherche de solutions (Sudoku...)
- Les prédicats du programme jouent le rôle de fonctions, qu'on peut appeler de différentes façons (exemple : que se passe-t-il en utilisant le but $plus(X, 7, 14)$ au lieu de $plus(7, 7, X)$)

- L'impératif s'intègre plutôt mal (car l'ordre d'évaluation est difficile à prévoir), ce qui peut poser problème pour les entrées/sorties et l'interface avec d'autres langages
- L'absence de déclaration préalable des symboles de fonctions et de prédicats est une source importante d'erreurs

Les défis de la programmation aujourd'hui

Programmation parallèle : souvent très délicate et suscite d'autres paradigmes de programmation (passage de messages, partage de données, ...). Les besoins sont croissants : processeurs multi-cœurs, grappes de processeurs, ...

Génie logiciel : écrire de gros programmes pose depuis longtemps de nombreux problèmes (organisation, évolution, réutilisation).

Sûreté de fonctionnement : une question loin d'être résolue. Les techniques de test, d'analyse automatique, de preuve formelle avec assistant, sont des domaines de recherche importants.