Résumé de LINFO1104

compilation du 19 mars 2023

Thomas Debelle

Juin 2023

Table des matières

Préface

Bonjour à toi !

Cette synthèse recueille toutes les informations importantes données au cours, pendant les séances de tp et est amélioré grâce au note du Syllabus. Elle ne remplace pas le cours donc écoutez bien les conseils et potentielles astuces que les professeurs peuvent vous donner. Notre synthèse est plus une aide qui on l'espère vous sera à toutes et tous utiles.

Elle a été réalisée par toutes les personnes que tu vois mentionné. Si jamais cette synthèse a une faute, manque de précision, typo ou n'est pas à jour par rapport à la matière actuelle ou bien que tu veux simplement contribuer en y apportant ta connaissance ? Rien de plus simple ! Améliore la en te rendant ici où tu trouveras toutes les infos pour mettre ce document à jour. (*en plus tu auras ton nom en gros ici et sur la page du github*)

Nous espérons que cette synthèse te sera utile d'une quelconque manière ! Bonne lecture et bonne étude.

Chapitre 1

Introduction

1.1 Les Paradigmes

Une paradigme, est une façon d'approcher et apporter une solution à un problème. De ce fait, chaque langage de programmation utilise 1 voir 2 paradigmes. Ce cours couvrira 5 paradigmes cruciaux qui sont :

1. "Functionnal Programming"
2. "Object Oriented Programming"
3. "Functional DataFlow Programming"
4. "Actor DataFlow Programming or Multi-Agent"
5. "Active Objects"

Et pour découvrir ces paradigmes, nous utiliserons les langages de programmations "Oz" qui est un langage de recherche multi paradigme ainsi que "Erlang" à la fin du cours.

Chapitre 2

Les différents Paradigmes

2.1 Functional Programming

Avec ce paradigme, on impose qu'une variable peut être nommé qu'une seule fois ! Donc : $X = 10$ mais on ne peut pas plus loin dire $X = 9$. X est déjà attribué. On peut penser que cela risque d'être handicapant alors qu'en réalité, cela rend notre code plus simple à débuguer. De plus, nombreux sont les langages et microservices utilisés qui implémentent la programmation fonctionnelle. Formellement, quand on déclare une variable et qu'on l'assigne à une valeur ceci se passe. Une chose importante à noter est que cette façon de programmer peut être réalisé dans n'importe quel langage de programmation. On peut également redéclarer un identificateur. C'est-à-dire écrire "$X = 42$" et plus loin en ayant redéclarer une variable "$X = 11$" car ces deux déclarations pointent à deux éléments totalement différents dans la mémoire et ont des *scopes* différents.

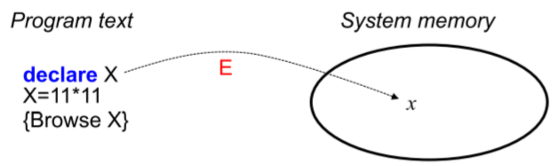


Figure 2.1 – Déclaration d'une variable

Un "Scope" ou portée est une propriété centrale en programmation. En effet, c'est le scope qui nous permet d'avoir différente valeur pour des variables qui ont le même nom. Naturellement, elle ne représente pas la même chose car elle diffère de leur scope. On peut déterminer le scope d'une variable sans même exécuter le code. Il nous suffit d'analyser le code qui comprend un "lexical scoping" ou un "static scoping".

```
local
   X
in
   X = 42 {Browse X}
   local
      X
   in
      X = 11 {Browse X}
   end
   {Browse X}
end
```

Figure 2.2 – Exemple de code avec des scope différents

Chapitre 3

Programmation symbolique

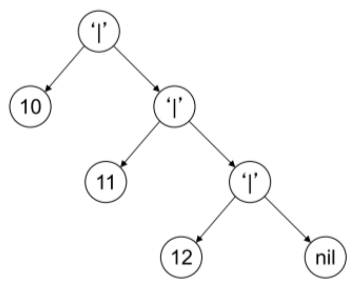
3.1 Listes

On dit d'une liste est **récursive** si elle se définit par elle-même. C'est-à-dire elle fait appel à elle-même. On utilise la récursion pour les calculs et pour stocker des données. Une liste est soit vide ou soit une pair *d'une valeur suivi par une autre liste.*

3.1.1 Définition formelle

En utilisant la notation **Extended Backus-Naur Form** ou *EBNF* pour les intimes, on écrit une liste comme : <List T> : := nil | T '|' <List T>. Une chose importante à noter est le deuxième "ou" qui s'écrit comme '|' signifiant qu'il n'appartient pas à la définition de List T mais plutôt à l'ensemble T '|' <List T>. Si on lit ceci, on dirait "*Une list d'élément représentant T correspond à un élément vide ou un élément représentant T suivi d'une autre Liste d'élément T.*

Donc une List d'entier se définit comme : <List <Int> >. Une chose importante à remarquer est que j'ai utilisé le mot "représentation" en effet <Int> n'est pas un entier mais une représentation d'entier.



Pour définir une liste en Oz, on utilise soit la notation [1 2 3] ou 1 | 2 | 3 | nil.(il existe d'autre manière semblable qu'on verra plus loin) C'est 2 déclarations reviennent à la même chose en mémoire. Une utilité des listes est leur facilité à être représenté sous forme d'arbre comme montré ci-contre. La *head* est accessible via list.1 et la *tail* est obtenu via list.2.

3.2 Pattern matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```

Grâce à cette représentation en arbre, il est facile de voir si une liste est bien une liste.
Ci-contre, on voit une fonction classique en Oz qui analyse une liste et détermine si elle est d'une structure correcte. Le [] correspond au cas où l'élément L est une liste avec une Head et une Tail. On appelle cela une *Clause* et H|T est le pattern de la clause. Le premier cas est définit par of.

3.3 Introduction au langage Kernel

Le langage Kernel est la première partie de la sémantique formelle d'un langage de programmation. Une règle importante est que tout programme écrit en programmation fonctionelle *peut être traduit en langage kernel*. Les grands principes du langage Kernel sont :

— Tous les résultats intermédiaires de calculs sont visibles. Donc on a 1 opération par ligne et la déclaration en locale de toutes les variables.

— Toutes les fonctions deviennent des *procédures* avec un argument en plus. Cet argument donne le résultat de la fonction.

— Les fonctions dans une fonction sont sortis de leur fonction et on leur donne un nouvel identificateur.

Les résultats de la traduction : Les programmes Kernel sont plus longs mais on voit facilement comment un programme s'exécute et on voit si il est *tail-recursive*

3.4 Les arbres

Les arbres sont des structures de données extrêmement utiles et utilisées. On peut y stocker des données spécifiques, faire des calculs, ... Les arbres illustrent bien *la programmation orienté but*. Par le standard *EBNF*, on définit un arbre comme suit : <tree T> : := leaf | t(T <tree T> ... <tree T>). Donc un arbre est une feuille ou *leaf* qui est suivie par un ensemble de *sous-arbres*. Les arbres sont forts similaires au liste si ce n'est que les listes n'ont qu'une sous-listes alors qu'un arbre peut avoir plusieurs sous-arbres.

3.4.1 Ordered Binary tree

Un arbre de ce type a 2 particularités :

— Binary : toutes les éléments hors les feuilles possèdes 2 sous-arbres.

— Ordered : pour chaque arbre, la clé à gauche est plus petite que la clé de l'arbre et la clé à droite est plus grande.

Ce type d'arbre est très utile pour ; par exemple, effectuer des recherches binaires et permet de facilement et rapidement trouver des données.

Lookup K T

Nous permet de trouver une valeur. Ce programme est plutôt simple et il nous suffit de regarder la clé de l'arbre où on est. Puis on compare avec notre recherche, si on est plus grand, on va à droite sinon à gauche. On répète le processus jusqu'à trouver la clé.
Lookup est très efficace car il s'exécute en $log_2 n$, le pire cas est si l'arbre n'est pas équilibré et il ressemble à une liste. Mais en général, en ayant un nombre suffisant de données, il est très rare d'avoir un arbre non équilibré.

Insert K W T

Il existe 4 possibilités.

1. remplace une feuille.
2. on remplace un noeud.
3. on remplace un sous-arbre à gauche.
4. on remplace un sous-arbre à droite.

Le premier cas est le plus simple car on créé simplement un nouvel sous-arbre avec 2 feuilles. Si on remplace un noeud, on change la clé et la valeur du noeud. Pour remplacer un sous-arbre, on garde les mêmes clés et valeur de Y pour le noeud mais on change le sous-arbre à gauche ou à droite en fonction.

Delete K T

Celle-ci est plus compliqué, on a 4 possibilités

1. La valeur qu'on veut supprimer n'existe pas
2. On supprime une feuille.
3. on supprime un sous-arbre à gauche.
4. on supprime un sous-arbre à droite.

TODO

3.5 Tuples et Records

3.5.1 Tuples

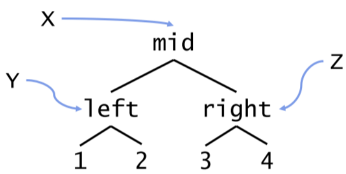
Un tuple est une manière de stocker des données de différents tuples, l'*ordre* est *important* dans un tuple. On doit également donné un nom, un **label**.

```
X = state(1 b 2)
{Browse {Label X}}
{Browse {Width X}}
```

La première ligne défini un tuple ayant pour *label* "state". La seconde ligne imprime le label du tuple. La dernière affiche sa taille. (c'est donc un entier toujours positif ou 0)
Les champs dans les tuples sont numérotés de 1 **à width** X. On appelle aussi le champ (field) une "*feature*". Un tuple possède toutes ces features de manière consécutives.

On peut donc ainsi construire des structures de données plus compliqués comme des arbres :

```
declare
Y = left(1 2) Z = right(3 4)
X = mid(Y Z)
```



Comparaison

Il est très simple de comparer des tuples via "==", il faut simplement comparer leur value à chaque champ. Attention au *loop* causé par les approches naïves.

3.5.2 Similitude Tuples et liste

En effet, une liste qui n'est autre que "H|T" peut facilement être traduit en tuple '|' (H T). Quand on peut déterminer un même élément via différentes manières, on appelle ça du *sucre syntaxique*. Dans le *kernel*, on fait au plus simple donc que des tuples.

```
List1 = [1 2 3]
List2 = (1:1 2:(1:2 2:(1:3 2:nil)))
List1 == List2 //Vrai
```

3.5.3 Les Records

Les "*records*" sont une **généralisation** des tuples. La différence avec les tuples est que le *field* peut être n'importe quel valeur et ne doit pas être consécutif. Donc ceux-ci sont des *records* corrects :

```
X = state(a:1 2:a b:2)
Y = inv(3:a 2:b 1:c)
```

Donc la position d'une valeur et son *field* n'importe plus et on peut déclarer dans le sens qu'on veut. Si on ne nomme pas un *field* dans un *record*, Oz va attribuer un nombre commençant à 1 et qui n'est pas utilisé par un autre champ.

3.5.4 Résumé

- Un *atom* est un record de width 0.
- Un tuple est un record avec des champs étant numéroté de manière consécutive de 1 à width X. (consécutive, donc on skip pas. pas forcément dans l'ordre dans la déclaration)
- Une liste est réalisé avec des *tuples* et des $(X \quad Y)$
- **1** seule *structure de donné* dans le kernel pour rester simple.

3.6 Sémantique Formelle

3.6.1 Les environnements

Un environnement est une fonction qui passe des *identifieurs* aux *variables en mémoire* autrement dit : $E_1 = (X \to x, Y \to y)$

Environnement contextuel

Un *environnement contextuel* d'une fonction contient tous les *identificateurs* qui sont utilisés dans la fonction mais déclarés *en dehors*. Donc ce sont des fonctions qui lorsqu'on appelle une variable va pointer en dehors du scope de la fonction.

Stocker une Procédure

Les procédures sont stocker dans la mémoire sous le forme de procédure anonyme symboliser par le "$".

```
local P Q in
        {Browse 'do something'}
        proc {Q}
                {P}
        end
        {Browse 'another something'}
end
```

Notre "proc Q" sera stocker comme : "q = (proc{$}{P} end, $\{P \to p\}$)". On lit donc, la procédure *anonyme* ($), fais un appel à P ({P}) et finit (end), son *environnement contextuel* fait que lorsqu'on appelle "P" on va récupérer la valeur "p" en mémoire ($\{P \to p\}$). Donc on voit que l'*environnement contextuel* est stocké avec le code de procédure.
On appelle également la valeur d'une procédure une "*closure*" ou une "*lexically scoped closure*" car elle ferme les identificateurs libres quand définis.
Donc l'avantage d'un environnement contextuel est d'être sûr qu'on appellera la bonne valeur même si elle est déclaré en dehors de la fonction.

Un *identificateur libre* est un identificateur utilisé dans une *fonction* qui est déclaré *en dehors* de la fonction.
Les arguments d'une procédure **ne sont pas** des identificateurs libres car l'argument défini l'identificateur.

3.6.2 Sémantique

Il est important de comprendre le fonctionnement même d'un programme car si on ne comprend pas comment celui-ci fonctionne, il nous domine. *If you do not understand something, then you do not master it – it masters you !*

Définition

La *sémantique* d'un langage de programmation est une explication *précise* de comment un programme s'exécute. Nous verrons la sémantique pour tous les paradigmes. Il en existe 4 types :

1. Sémantique opérationnelle : explique un programme sur base d'*exécution* sur un PC simplifié appelé *la machine abstraite.* → Fonctionne pour tous les paradigmes.
2. Sémantique axiomatique : explique un programme sur base d'*implication.* C'est-à-dire que certaines *propriétés* présent avant l'exécution, et d'autres seront présentes après. → très utilisé pour la programmation orienté objet comme *Java.*
3. Sémantique de notation : explique un programme comme une *fonction* sur un domaine abstrait. Donc simplifie l'analyse mathématique d'un programme. (utilisé dans *Haskell* et *Scheme*)
4. Sémantique logique : explique un programme comme étant un *modèle logique* basé sur des *axiomes logiques.* Le résultat est une propriété correcte dérivé des axiomes. (cela est implémenté par exemple dans *Prologue* ou dans la *programmation sous contrainte*)

3.6.3 Sémantique opérationnelle

Ce type de sémantique à 2 parties majeures :
— Langage Kernel : traduit le programme en langage Kernel.
— Machine abstraite : puis exécute le programme sur la machine abstraite.

1. Langage Kernel complet

Pour définir correctement une sémantique, il faut tout d'abord s'intéresser à son langage Kernel complet. On peut également prouver qu'un programme est correct en analysant son kernel. Par exemple, prenons ce code kernel :

```
<s> ::= skip
      | <s>_1<s>_2
      | local <x> in <s> end
      | <x>_1=<x>_2
      | <x>=<v>
      | if <x> then <s>_1 else <s>_2 end
      | {<x> <y>_1,...,<y>_n}
      | case <x> of <p> then <s>_1 else <s>_2 end

<v> ::= <number> | <procedure> | <record>
<number> ::= <int> | <float>
<procedure> ::= proc {$ <x>_1,...,<x>_n} <s> end
<record>, <p> ::= <lit> | <lit>(<f>_1:<x>_1,...,<f>_n:<x>_n)
```

donc "<s>" contient le programme exécuté, "<v>" est une structure de donné contenant différent type de structure de donné qui sont définis juste en dessous.

2. La machine abstraite

Voici ci-dessous comment s'exécute un programme initialement.

```
local X in
  local B in
    B=true
    if B then X=1 else skip end
  end
end
```

```
([(local X in
    local B in
      B=true
      if B then X=1 else skip end
    end
  end, {})],
 {})
```

instruction stack [...] → ensemble vide

empty environment

empty memory

FIGURE 3.1 – à gauche : programme écrit en Oz   à droite : état initiale d'exécution

Au début, l'environnement et la mémoire sont vides. L'état d'exécution est écrit typiquement comme :

([(<s>,E)] , σ)

Sur la machine abstraite, on va d'instructions en instructions. C'est-à-dire on descend petit à petit. donc on a pour la suite :

```
([(local B in
      B=true
      if B then X=1 else skip end
    end,
    {X → x})],
 {x})
```

environnement créé

```
([((B=true
      if B then X=1 else skip end),
   {B → b, X → x})],
 {b,x})
```

environnement et

FIGURE 3.2 – à gauche : on descend de 1 cran   à droite : on descend encore de 1 cran

On voit que au fur et à mesure qu'on descend, la pile de mémoire et d'environnement s'agrandit. Ensuite on va *séparer* la *composition séquentielle* comme suit :

```
([(B=true,{B → b, X → x}),
  (if B then X=1
   else skip end,{B → b, X → x})],
 {b,x})
```

```
([(if B then X=1
   else skip end,{B → b, X → x})],
 {b=true, x})
```

FIGURE 3.3 – à gauche : on sépare en deux   à droite : on attribue à b la valeur définit à gauche

Une nouvelle instruction va s'ajouter à cause du "then" de notre condition :

```
([(X=1,{B → b, X → x})],
 {b=true, x})
```

```
([],
 {b=true, x=1})
```

FIGURE 3.4 – à gauche : la nouvelle instruction   à droite : les instructions sont vides, c'est fini

3. Définir la machine abstraite

- Pour chaque instructions dans le langage Kernel, on associe sa règle dans la machine abstraite
- Chaque instructions prends un état d'exécution en entrée et sort un état d'exécution en sortie $\rightarrow (ST, \sigma)$.

L'instruction la plus simple est "skip" car il fonctionne comme $([(skip, E), S_2, ..., S_n], \sigma)$ et renvoie $([S_2, ..., S_n], \sigma)$

Instructions	entrée	sortie
skip	1	2
$(< s >_1 < s >_2)$	$([(S_a S_b), S_2, ..., S_n], \sigma)$	$([S_a, S_b, S_2, ..., S_n], \sigma)$
local in <x> in <s> end	$([($local $< x >$ in $< s >$ end $, E)$ $, S_2, ..., S_n], \sigma)$	$([(< s >, E + \{< x > \rightarrow x\})$ $, S_2, ..., S_n], \sigma)$

Il y a également d'autres types d'instructions dont on détaillera pas le langage en machine abstraite :
- <x>=<v> (crée et assigne une valeur) : quand <v> est une procédure, on **doit** crée un environnement contextuel.
- if <x> then $< s >_1$ else $< s >_2$ end (condition) : si <x> n'est pas attribué, l'instruction va attendre ("*block*") jusqu'à ce que <x> soit attribuer à une valeur.
- case <x> of <p> then $< s >_1$ else $< s >_2$ end : Le system de "*case*" se construit en combinant des structures de données Kernel.
- $\{< x >< y >_1, ..., < y >_n\}$: ceci est la base de l'abstraction de donnée

Par ailleurs, voici d'autres concepts de machine abstraite :
- Single-assignment memory $s = \{x_1 = 10, x_2, x_3 = 20\}$: Définition d'une variable et la valeur associée.
- Environnement $E = \{X \rightarrow x, Y \rightarrow y\}$: Lien entre un identificateur et son lien dans la mémoire
- Instruction sémantique $(< s >, E)$: Une instruction avec son environnement.
- Stack Sémantique $ST = [(< s >_1, E_1), ..., (< s >_n, E_n)]$: Un stack d'instructions sémantiques.
- État d'exécution (ST, σ) : Une paire d'un stack sémantique et sa mémoire.
- Execution $(ST_1, s_1) \rightarrow (ST_2, s_2) \rightarrow (ST_3, s_3) \rightarrow ...$: Une séquence d'état d'exécution.

4. Programme correct

grâce à la sémantique, on sait prouver qu'un programme est correct. On dit qu'un programme produit une solution correcte, on l'appelle une *spécification*.
Donc on prouve qu'un programme satisfait la *spécification* quand on utilise une certaine *sémantique*.
La sémantique lie le *programme* à un résultat mathématique appelé *spécification*.

Donc on lie une vérité mathématique à un programme. Et on prouve cela via ces différentes étaps : (exemple avec une factorielle)

1. On commence avec la spécification du programme.
2. Notre programme est *récursif* donc on va utiliser une preuve mathématique par *induction*.
3. On doit prouver le cas de base et le cas général.
4. On utilise la sémantique pour prouver la véracité de notre programme.

5. Procédures

Les procédures sont la base de toutes abstractions de données.
Il y a deux choses importantes dans une *procédure* : sa définition et son appel.

Définition : on crée l'environnement *contextuel*. Puis, on stocke le code de la procédure et son environnement.

Appel : on crée un nouvel environnement combinant l'environnement *contextuel* de la procédure et les variables *formelles*. Ensuite, le tout est exécuté.

```
local Z in
        Z=1
        proc{P X Y}Y=X+Z end
end
```

Ici, le seul identificateur *libre* est **Z** qui est donc déclaré en dehors de la *procédure*. Donc à l'exécution de **P**, **Z** est connu donc **Z** fait partie de l'environnement contextuel de la *procédure*.

```
local P in
        local Z in
                Z=1
                proc{P X Y}Y=X+Z end
        end
        local A B in
                A=10
                {P A B}
                {Browse B}
        end
end
```

Ici, à la ligne de la création de la procédure P, son environnement contextuel est $E_c = \{Z \to z\}$. Au moment de l'exécution de P avec les valeurs A et B, on va donc ajouter un environnement qui est de la sorte : $E_P = \{Y \to b, X \to a, Z \to z\}$ Donc en langage *sémantique*, la définition d'une procédure ressemble à cela :

- **Instruction sémantique** : (<x>=proc{$\$ < x >_1, ..., < x >_n$} <s> end, E)
  - **Arguments formels** : $< x >_1, ..., < x >_n$
  - **Identificateurs libres de** <s> : $< z >_1, ..., < z >_k$
  - **Environnement contextuel** : $E_C = E_{|<z>_1,...n<z>_k}$ (que les identificateurs libres)
- Cela crée une liaison en mémoire de la forme : x(proc{$\$ < x >_1, ..., < x >_n$}<s>, end, E_C)

Maintenant, voyons pour un *appel sémantique* :

- **Instruction sémantique** : ($\{\langle x\rangle\langle y\rangle_1 ... \langle y\rangle_n\}$, E)
  - Si la condition est *false* donc $E(\langle x\rangle)$ n'est pas lié.
  - Si $E(\langle x\rangle)$ n'est **pas** une procédure, on a une erreur de *condition*.
  - Si $E(\langle x\rangle)$ est une procédure *mais* avec le mauvais nombre d'argument, on a aussi une erreur de *condition*.

Une chose primordiale à comprendre est comment sont stocké les instructions. Elles sont stocké sur une **pile** (*stack*). Donc on a l'instruction sémantique sur le stack : ($\{\langle x\rangle\langle y\rangle_1 ... \langle y\rangle_n\}$, E) avec la définition de procédure dans la *mémoire* comme cela : $E(\langle x\rangle) = proc\{\$\langle z\rangle_1, ..., \langle z\rangle_n\}\langle s\rangle \text{end}, E_c)$
Ensuite, on met ces instructions sur la *pile* $(\langle s\rangle, E_C + \{\langle z\rangle_1 \to E(\langle y\rangle_1), ..., \langle z\rangle_n \to E(\langle y\rangle_n)\})$

La machine abstraite fait 2 choses :

1. Adjonction : $E_2 = E_1 + \{X \to y\}$ Donc ajoute une paire (identificateur $\to$ variable) à l'environnement. Ré-écrit par dessus E_1 si existe déjà. Utile pour local $< x >$ in $< s >$ end.
2. Restriction : $E_C = E_{|\{X,Y,Z\}}$ Donc limite les *identificateurs* dans un environnement. On a besoin de cela pour calculer l'environnement *contextuel*.

Une adjonction :

```
local X in
        (E1) X=1
        local X in
                (E2) X=2
                {Browse X}
        end
end
E1 = {Browse → b, X → x}
E2 = E1 + {X → y} = {Browse → b, X → y}
```

Une restriction :

```
local A B C AddB in
        A=1 B=2 C=3 (E)
        fun {AddB X} (EC : contextual environment)
                X+B
        end
end
```

E = {A → a, B → b, C → c, AddB → a' }

E_C = $E_{|\{B\}}$ = {B → b }

3.6.4 Résumé

Définir la sémantique permet de relier les programmes au mathématique. On donne des instructions *sémantique* au *kernel* pour qu'il sache comment exécuter dans la *machine abstraite*. La sémantique nous permet de prouver qu'un programme est *correct*.
La sémantique est au cœur de la programmation. Une nouvelle librairie est comme si on ajoutait des instructions au programme donc on augmente sa sémantique.
Quand on écrit un programme, il faut comprendre la sémantique (l'utilisateur n'a pas besoin de savoir). La sémantique doit être simple et complète.
On peut voir la sémantique comme le langage de programmation *ultime*.
Il ne faut pas oublier que les pc sont baser sur les mathématiques *discrètes*.

3.7 Rappel procédure sémantique

Tout d'abord, en programmation nous avons différentes étapes qui reposent chacun sur les précédentes. Fermeture → Programmation d'ordre supérieur → Abstraction des données → Technologie de l'information.

Rappel sur l'exécution d'un programme :

```
{Browse {Inc 10}}        #Langage pratique (classique)
```

```
local M in                      #Langage Kernel
        local N in
                M=10
                {Inc M N}
                {Browse N}
        end
end
```

A l'exécution,$[(\{IncMN\},\{M \to m, N \to n, Inc \to i, Browse \to b\}),(\{BrowseN\},\{M \to m, N \to n, Inc \to i, Browse \to b\})],\{m = 10, n, i = (proc\{\$XY\}Y = X + Aend,\{A \to a\}), a = 1, b = (...browsercode\ldots)\}$ et Inc va référencer cela :
$[(Y = X+A,\{A \to a, X \to m, Y \to n\}),(\{BrowseN\},\{M \to m, N \to n, Inc \to i, Browse \to b\})],\sigma$
Il est important de remarquer que dans la mémoire, quand on stocke une procédure, on stocke le tout donc avec son environnement contextuel.

Chapitre 4

Programmation d'ordre supérieur

Ce concept découle directement du concept d'*environnement contextuel*. Dans un *procédure* ou *fonction* (les mêmes pour un langage kernel) peuvent prendre des valeurs ou des fonctions en arguments.

Définition

— Une fonction est dit de premier ordre si elle ne prend et ne ressort aucune fonction.
— Une fonction est N+1 si son entrée et sortie prennent en tout N fonctions en argument.

Nomenclature des différentes fonctions :

— **Une génératrice** est le fait de prendre une fonction en entrée d'une fonction.
— **Une instantiation** est le fait de retourner une fonction en sortie d'une fonction.
— **Une composition de fonctions** est le fait de prendre 2 fonctions en entrée et on retourne leur composition.

Utilisation

Via la programmation d'ordre supérieure, on peut cacher un accumulateur. On dit qu'on fait une *abstraction d'accumulateur*.
Une fonction type est la fonction **FoldL** (*reduce*). En effet, la fonction FoldL fait :

```
declare
fun {FoldL L F U}
        case L
        of nil then U
        [] H|T then {FoldL T F {F U H}}
        end
end
```

```
{FoldL LIST Function Acc}
```

On peut, un peu dans le même style, faire de l'encapsulation afin de cacher sa valeur à l'intérieur.
→ C'est la base de *l'abstraction de donnés*.

Il faut faire attention à l'**exécution retardé**. En effet, si on ne stocke pas le résultat d'une fonction, elle ne sera exécuté que quand on appellera la valeur. Donc cela peut prendre beaucoup de place en mémoire de stocker une fonction plutôt que son résultat.

Chapitre 5

Lambda Calcul

1. C'est un modèle de calcul qui est Turing complete.
2. **Tous** les types de données peuvent être encodé en lambda calcul.
3. Par le théorème de **Church-Rosser**, le lambda calcul est confluent. Même résultat peu importe l'ordre de réduction
4. C'est la base de la programmation ordre supérieur et de la programmation formelle.

5.1 Introduction

Le *lambda calcul* est une manière mathématique formelle pour représenter des calculs informatiques. Cela a été créé avant l'arrivée des ordinateurs. Cela ne contient que des **définitions**, **appels** et utilise des **liens de variables** et de la **substitution**.

Le *lambda calcul* est une manière universelle de calcul. On peut l'utiliser pour simuler des **machines de Turing**.

5.1.1 Fonctionnement

Le lambda calcul n'a que des *fonctions anonymes d'un seul argument*. Donc si on veut en avoir plusieurs, il faut combiner les fonctions :

sum_square(x,y) = $x^2 + y^2$ //fonction classique
(x,y) $\rightarrow x^2 + y^2$ //fonction anonyme
x $\rightarrow$ (y $\rightarrow x^2 + y^2$) //d'une seul argument
$\lambda x.\lambda y.x^2 + y^2$ //en lambda calcul

Le fait de combiner et de "*nest*" des fonctions s'appellent le currying.

5.1.2 Syntaxe

Les expressions lambdas sont composées de :
— Variables (x,y, ...)
— Du symboles d'abstractions (λ) et de point (.)
— Et des parenthèses

En syntaxe EBNF c'est :

t ::= x | (λx.t) | t_1 t_2

(λx.t) : est appelé une abstraction (*définition de fonction*)
t_1 t_2 : c'est *l'appel de fonction.*

5.1.3 En Oz

1. La définition de fonction (λx.t)

   ```
   fun {$ X} T end
   ```

2. L'appel de fonction $(t_1 \quad t_2)$

   $\{T_1 \quad T_2\}$

Le currying en Oz :

1. Définition

   ```
   F = fun {$ X} fun {$ Y} T end end
   ```

2. Appel

   ```
   {{F X} Y}
   ```

5.1.4 Sémantique des expressions lambdas

Le sens d'une expression lambda dépend de comment on peut la réduire. Il en existe 3 types.

1. α-renaming : change le nom des variables liés
2. β-reduction : applique une fonction à un argument
3. η-reduction : enlève les variables inutilisés

Variables libres et liées

Si nous avons : $\lambda x.t$ on dit que *l'opérateur* λx lie la variable x à t. Mais la variable t est libre car n'est lié à *aucune* fonction. Si x est libre dans t alors on dit qu'on capture x.
On dénote $FV(t)$ l'ensemble des variables libres :
— $FV(x) = \{x\}$ où x est une variable
— $FV(\lambda x.t) = FV(t) \backslash \{x\}$
— $FV((t_1 \quad t_2)) = FV(t_1) \cup FV(t_2)$

α-renaming

Ainsi, on peut changer le nom d'une variable lié :

$\lambda . x \quad x \quad \rightarrow_\alpha \lambda . y \quad y$

Il faut faire attention à ce qu'on ait pas de *conflit de nom* et de *capture de variable*.
Des termes qui diffèrent d'un α-renaming sont dits α-equivalent.

Substitution

La substitution de $t_1[x := t_2]$ remplace toutes les occurrences libres de x dans t_1 par t_2.
On a parfois recourt à l'*α-renaming*. En effet, la substitution ne peut pas *capturer* des variables libres. Donc $(\lambda x.y)[y := x]$ peut être transformé en $(\lambda z.y)[y := x]$. La définition :

$$x[x := t] = t \qquad y[x := t] = y, \text{ if } x \neq y \qquad (t_1 \quad t_2)[x := t] = (t_1[x := t])(t_2[x := t])$$
$$(\lambda x.t_1)[x := t_2] = \lambda x.t_1 \qquad (\lambda y.t_1)[x := t_2] = \lambda y.(t_1[x := t_2]), \qquad \text{if } x \neq y \wedge y \notin FV(t_2)$$

β-reduction

C'est une application de fonction et se décrit via la substitution. (cfr 5.1.4)
Définition : $(\lambda x.t_1)t_2 \rightarrow t_1[::= t_2]$
Exemple : $(\lambda x.(x \quad x))y \rightarrow (y \quad y)$

η-reduction

C'est l'idée que 2 fonctions sont les *mêmes* si elles produisent le *même résultat* pour tous les arguments possibles. On appelle cela extensibilité. Donc 2 fonctions sont les mêmes si elles ont les mêmes propriétés extérieures.
Définition : $\lambda x.(t \quad x) \rightarrow t$ if $x \notin FV(t)$

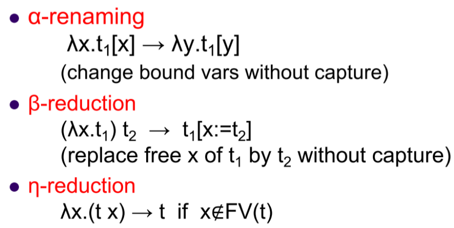


FIGURE 5.1 – Résumé

Convention de notation

Quand on manipule des expression lambda, il est important de suivre quelques règles :
— Enlever les parenthèses les plus extérieur. ex : $(t_1 \quad t_2) \rightarrow t_1 \quad t_2$
— Les applications sont associatives depuis la gauche. ex : $t_1 \quad t_2 \quad t_3 \rightarrow ((t_1 \quad t_2) \quad t_3)$
— On étend vers la droite ex : $\lambda x.t_1 t_2 \rightarrow \lambda x.(t_1 t_2)$
— On peut simplifier les séquences d'abstractions. ex : $\lambda x.\lambda y.\lambda z.t \rightarrow \lambda xyz.t$

5.2 Types de données

Le lambda calcul peut faire tout type de calcul, il est Turing complete.
Pour le démontrer, on va encoder des chiffres, opérations, ... en lambda calcul. Donc tous les chiffres, booléens, ... seront encodés en lambda calcul

5.2.1 Nombres

On utilise une notation appelée Church Numerals :
— $0 \triangleq \lambda f.(\lambda x.x)$
— $1 \triangleq \lambda f.(\lambda x.(f \quad x))$
— $2 \triangleq \lambda f.(\lambda x.(f(f \quad x))$
— $3 \triangleq \lambda f.(\lambda x.(f(f(f \quad x))$
On peut remarquer que ces fonctions dont d'ordre supérieur. En effet, elles prennent en argument une fonction et ressort une fonction. (d'un seul argument)
Le nombre n retourne donc la fonction qui est une composition de n fois de f. Donc on dit que **f est appliqué n fois**.

5.2.2 Opérations

Fonction successeur

Prend le nombre de Church n et retourne $n + 1$.

$$SUCC \triangleq \lambda n.\lambda f.\lambda x.f((n \quad f)x) \qquad SUCC \triangleq \lambda n.\lambda f.\lambda x.f(n \quad f \quad x)$$

Addition

Retourne donc la composition de $m + n$.

$$PLUS \triangleq \lambda m.\lambda n.\lambda f.\lambda x.(m \quad f)((n \quad f)x) \qquad PLUS \triangleq \lambda m.\lambda n.\lambda f.\lambda x.m \quad f(n \quad f \quad x)$$

Multiplication

$$MULT \triangleq \lambda m.\lambda n.\lambda f.m(n \quad f) \qquad MULT \triangleq \lambda m.\lambda n.m(PLUS \quad n) \quad 0$$

La deuxième façon de faire $MULT$ est comme si on disait, "fait une somme m fois en commençant à 0.

Exponentielle

$$POW \triangleq \lambda b.\lambda e.e \quad b$$

Prédécesseur

$$PRED \triangleq \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.u)$$

Soustraction

Grâce à la méthode du prédécesseur, on peut définir la soustraction :

$$SUB \triangleq \lambda m.\lambda n.n \quad PRED \quad m$$

Donc pour $m - n$ on effectue la fonction prédécesseur n fois en commençant à m.

5.2.3 Opération logique

Booléens

$$TRUE \triangleq \lambda x.\lambda y.x \qquad FALSE \triangleq \lambda x.\lambda y.y$$

Opérateur logique

$$AND \triangleq \lambda p.\lambda q.p \quad q \quad p \qquad OR \triangleq \lambda p.\lambda q.p \quad p \quad q$$
$$NOT \triangleq \lambda p.p \quad FALSE \quad TRUE \qquad IFTHENELSE \triangleq \lambda p.\lambda a.\lambda b.p \quad a \quad b$$

Prédicat

Ce sont des fonctions qui retournent une valeur booléenne :

$$ISZERO \triangleq \lambda n.n(\lambda x.FALSE)TRUE \qquad LEQ \triangleq \lambda m.\lambda n.ISZERO(SUB \quad m \quad n)$$

Paire

Ce sont des tuples de $width = 2$ et peuvent être définis en terme de booléens :

$$PAIR = (x, y) \text{ le tuple} \qquad PAIR \triangleq \lambda x.\lambda y.\lambda f.f \quad x \quad y$$
$$FIRST \triangleq \lambda p.p \quad TRUE \qquad SECOND \triangleq \lambda p.p \quad FALSE$$
$$NIL \triangleq \lambda x.TRUE \qquad NULL \triangleq \lambda p.p(\lambda x.\lambda y.FALSE)$$

Ceci nous introduit à l'abstraction de donnée. Une chose cruciale à comprendre est que depuis les 3 principes simples du lambda calcul, (définition de fonction, utilisation de fonction, des variables en arguments) on peut tout réaliser !

Liste

En sachant qu'une liste est un ensemble de valeur de nil, on sait utiliser des listes en lambda calcul. On peut même les utiliser pour "*simplement*" définir la fonction PRED via SHIFTINC :

$$SHIFTINC \triangleq \lambda x.PAIR(SECOND \quad x)(SUCC(SECOND \quad x)) \quad (m,n) \rightarrow (n, n+1)$$
$$PRED \triangleq \lambda n.FIRST(n \quad SHIFTINC(PAIR \quad 0 \quad 0))$$

En effet cela fonctionne car le premier élément sera $n-1$ par rapport au second.

5.2.4 Fonctions récursives

Comme les fonctions sont anonymes, on ne peut pas faire d'appel direct récursif. On va faire en sorte qu'une fonction devient l'argument d'une expression lambda. Par exemple : $G \triangleq \lambda f.\lambda n.$(if n = 0 then 1 else n(f n-1))

Y combinator

Cela nous permet de passer de $Y(g)$ à $g(Y \quad g)$. Ainsi, on sait faire des factorielles de manière récursive :

$$Y \triangleq \lambda g.(\lambda x.g(x \quad x))(\lambda x.g(x \quad x))$$

5.2.5 Théorème de Church-Rosser

Ce théorème nous dit que l'**ordre de réduction** n'a pas d'importance.
Concrètement, si a se réduit à b avec aucune ou plusieurs étapes et que si a se réduit à c avec aucune ou plusieurs étapes. Alors, il existe un d tel que b et c peuvent se réduire à ce dernier.
Le programme est confluent ou possède la *propriété de Church-Rosser*.

5.2.6 Le lambda calcul et les langages de programmation

La *programmation fonctionnelle* peut être comprise en terme de lambda calcul.
— Les valeurs de procédures (donc *lexically scope closures*) **sont** des fonctions lambda.
— l'*Eager et lazy evaluation* est sont des stratégies de réduction différentes.

2 approches

Eager évaluation méthode de réduction qui commence par l'intérieur donc on exécute un programme comme suit : {Double {Average 5 7}} → {Double ((5+7)/2)} → {Double (12/2)} → {Double 6} → 6+6 → 12	La lazy évaluation elle, va calculer que ce qui est nécessaire et réduit si nécessaire pour l'exécution. On va de l'extérieur à l'intérieur {Double {Average 5 7}} → {Average 5 7}+{Average 5 7} → ((5+7)/2)+{Average 5 7} → (12/2)+{Average 5 7} → 6+{Average 5 7} → 6+((5+7)/2) → 6+(12/2) → 6+6 → 12

If statement

Dans la plupart des langages de programmation, la condition if est réalisé de manière applicative. La partie de l'action then est réalisé de manière lazy. Donc si une action produit une erreur mais quelle est dans une clause qui ne serait pas atteint, alors le programme ne crashe pas.

5.2.7 Astuces pour le Lambda Calcul

Voici quelques astuces pour vous aider à maitriser le lambda calcul qui peut sembler de prime abord barbare :

1. Ajouter des parenthèse ! ! Ex : $\lambda x.\lambda y.x + y \quad 1 \quad 2 \rightarrow (\lambda x.(\lambda y.x + y)1)2$ Cela peut sembler anodin mais cela permet de mieux comprendre et lire les expressions qui peuvent vite devenir lourdes. Aussi, garder les parenthèses ça peut aider à voir plus clair.
2. Utiliser des abréviations. Car on n'a pas toujours besoin de savoir comment cela fonctionne et permet de faire partie par partie une fonction.
3. Pour bien réaliser un *η-reduction*, commencer par faire une *β-reduction*. Vous voyez ce qu'il vous reste. Vous faites une *α-renaming* pour retrouver une expression qui correspond à une partie de l'expression de base.
4. Cet article est très utile pour comprendre le *y-combinator* et mieux disséquer les opérations en Lambda Calcul.

5.2.8 Variation et extension

Le lambda calcul est une base fondamentale dans la théorie de l'informatique. Dans les extensions du lambda calcul, on retrouve :

— Lambda calcul typé : donc avec des variables et fonctions
— System F : avec des variables de **types**
— Constructions de calcul : les types sont les valeurs de classes premières
— Combinateur logique : logique sans variables
— Calcul de combinateur SKI : comme le lambda calcul mais sans substitution de variables. On utilise les combinateurs S,K et I.
— Langage Kernel d'OZ : le lambda calcul avec des variables *dataflow*(une seule attribution), exécution *dataflow*, *threads* et evaluation *lazy* explicite.

Chapitre 6

État mutable et abstraction des données

Une chose importante à réaliser, il n'y a pas de notion de temps en programmation. Toutes les fonctions qu'on a sont mathématiques et ne change pas en fonction du temps. Un programme *ne peut observer* son évolution dans le temps sur l'ordinateur.

6.1 Motivation

Une des solutions possibles, on définit le temps abstrait comme une suite de valeurs. On appelle cela un **état**.
Formellement, un state est une séquence calculé de manière *progressive* et contient les résultats *intermédiaire*. Le paradigme fonctionnel peut utiliser les états. ci-dessous la définition de **Sum** comme un état :

```
fun {Sum Xs A}
        case Xs
        of nil then A
        [] X|Xr then
                {Sum Xr A+X}
        end
end
{Browse {Sum [1 2 3 4] 0}}
```

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

Mais cela n'est pas suffisant. En effet, on voudrait que le programme remarque *lui-même* ses propres changements. Il faut faire une **extensions**, on rentre dans un nouveau **paradigme**.

6.2 État explicite

On va essayer de montrer les changements en rendant les états *explicites*. On appelle l'extension une **cellule** ou *cell*(c'est l'équivalent d'un *pointeur* en C).
Donc tout comme en C, on peut changer la référence en changeant le pointeur ou en remplaçant la variable par une autre valeur.

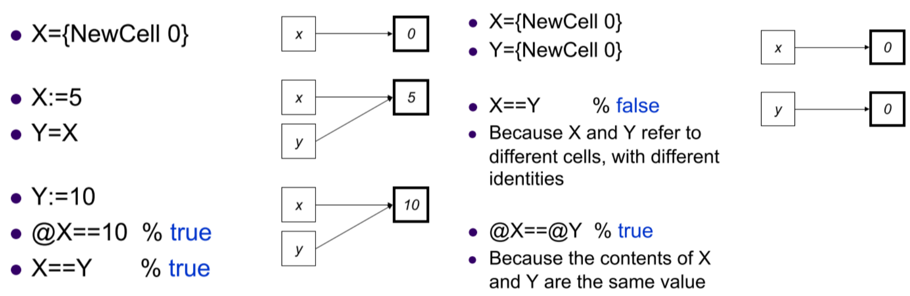
Cell

Une cellule à une identité et un contenu :
— identité : c'est constant et correspond au nom/pointeur de la cellule
— contenu : c'est le contenu stocké qui lui peut varier.

```
A=5; B=6
C={NewCell A}      // on crée une nouvelle cellule
{Browse @C}        // on montre le contenu avec @ ici 5
C:=B               // on ré-attribue le pointeur qui pointe vers B maintenant
{Browse @C}        // vaut 6
```

6.2.1 Exemple

Une chose à bien comprendre est que lorsqu'on réalise l'opération :=, on remplace le contenu de la *box*. Si on lie 2 cell via = on relie notre variable vers la même cell. Donc on peut modifier le contenu de la box via une des deux variables. (6.2.1)



6.3 Sémantique de cellules

On va maintenant étendre la machine abstraite afin d'expliquer comment les *cells* fonctionnent. Tout d'abord, nous avons maintenant **2 types de données**. Les données immutables ou assignement unique et les mutables ou assignements multiples (les cellules).
Une cellule est une paire de 2 *variables*.
— Une variable constante qui est lié au nom de la *cell*
— Une variable qui est lié au contenu de la cellule
Quand on assigne du nouveau contenu à une *cell*, on change la paire (seulement la deuxième variable).

Pour stocker le tout, on a 2 parties comme $\sigma = \sigma_1 \cup \sigma_2$.
— Single assignement : $\sigma_1 = \{t, v, x = \xi, y = \zeta\}$
— Multiple assignement : $\sigma_2 = \{x : t, y : w\}$
Quand on réalise l'opération $X := Z$ on change σ_2. $x : t \rightarrow x : z$

6.3.1 Programmation impérative

La **programmation impérative** est le nom du nouveau *paradigme*. C'est donc le cumul du *functional paradigm* et du concept de cellules.
La programmation impérative est fondamentale en **programmation orienté objet**.

Langage Kernel de la programmation impérative

```
<s> ::= skip
      | <s>1 <s>2
      | local <x> in <s> end
      | <x>1=<x>2
      | <x>=<v>
      | if <x> then <s>1 else <s>2 end
      | {<x>,<y>1,...,<y>n}
```

| case <x> of <p> then $< s >_1$ else $< s >_2$ end
| {NewCell <y> <x>}
| <x>:=<y> _____ {Exchange <x> <y> <z>}
| <y>=@<x> __|
<v> ::= <number> | <procedure> | <record>
<number> ::= <int> | <float>
<procedure> ::= proc {$ $< x >_1, ..., < x >_n$} <s> end
<record>, <p> ::= <lit> | <lit>($< f >_1$:$< x >_1$ ... $< f >_n$:$< x >_n$)

On remplace souvent l'attribution et l'appel du contenu par la fonction *Exchange* qui est une opération atomique (c'est-à-dire 2 opérations indissociables en 1).

6.4 Nécessité de l'état mutable

On dit qu'un programme est modulaire pour une partie si on peut changer cette dernière sans changer le reste du programme.

Chapitre 7

Conseils pour la syntaxe d'Oz

Voici une liste d'astuces et de choses importantes à savoir sur Oz :

— Déclarer vos fonctions et variables avec une **majuscule** au début !

— Une procédure est une fonction qui ne retourne **rien**.

— Pour retourner une valeur dans une fonction, on écrit une ligne où on ne fait *aucune* assignation ou opération, ... (Ex : on veut retourner X on écrit sur une ligne X tout seul)

— Pour éviter d'avoir des surprises, toujours bien terminer sa liste par un $|nil$

— On peut toujours utiliser une fonction récursive avec **accumulateur** pour une fonction récursive. C'est même conseillé !

— On peut écrire en langage Kernel directement en oz

— On peut faire des listes de procédures via la notation Kernel d'une procédure

— Une liste est un tuple de type (1 :Num 2 :(1 :Num 2 :nil))

— Pour être plus "opti" utiliser les local in.

— Si notre code est le même que celui du voisin mais que vous avez des résultats différents, *relancez emacs ou VScode*. Nettoyer le buffer et feed tout le code dedans.

— Pour être plus *productif*, télécharger "Powertoys" sur Windows et activer l'option "**toujours afficher**" (**always on top**).