UNIVERSITÉ TOULOUSE III PAUL SABATIER

Rédigé avec LaTeX
Version du 17 septembre 2013

Programmation fonctionnelle 1

Caml

L3 Informatique
Semestre 5

Cours donné par Christine Maurel
Rédigé par Antoine de Roquemaurel

2013

Chapitre 1

La programmation fonctionnelle

1.1 Différents paradigmes de programmation

- Impératif : C, Java, Ada, ...
- Objet : Java, C++, ...
- Fonctionnel : Lisp, Shceme, ML, Caml, Haskell, ...
- Déclarative ou logique : Prolog

1.2 Le fonctionnel

L'outil de base de la programmation fonctionnel est les fonctions. On peut les définir, les appliquer et les composer. Il n'y a pas d'affectation en fonctionnel.

Le fonctionnel est partis d'une base théorique avec le λ calcul en 1936,c'est un langage sûr. C'était d'abord non typé [1], les langages typés sont arrivés ensuite avec la famille Ocaml vers les années 2000.

Un langage fonctionnel typé possède plusieurs propriétés.

Inférence de type On ne déclare pas le type expressément.

Vérification de type Vérifier à la compilation, pas de risque de problème lors de l'execution

Polymorphe

Syntaxe simple Syntaxe non verbeuse, sémantique solide, environnement de développement solide, mise au point facilitée et programmation sûre

1.2.1 Mode de compilation

Le Caml peut être soit compilé soit interprété, l'avantage de la compilation étant l'efficacité et l'interprétation « convivial ». Historiquement ceux-ci étaient uniquement compilés.

1. Comme le lisp ou, Scheme

Chapitre 2

Syntaxe

2.1 Action

```
# expression ;;
-: valeur : type
#
```

Listing 2.1 – Syntaxe de base

- Lire l'expression jusqu'au ;;
- Typer
  - Si ko ⇒ Message d'erreur
  - Si ok ⇒ Évaluation ⇒ « Réduire, calculer » ⇒ Résultat / Valeur

2.2 Types de base

Type	Mot clé	Opération	Comparaison	Exemple
Entiers($\mathbb{Z}$)	int	`+, -, *, /, mod`	`=, >, <, >=, <=, <>`	2013
Flottants	float	`+., -., *., /., sqrt, **`	Polymorphe	2013.0
Chaines	string	`"__", ^`	Polymorphe	`"coucou"`
Caractères	char	`'_'`	Polymorphe	'c'
Booléens	bool	`true, false, &&, ||, not`	Polymorphe	

2.3 Structures de contrôles

```
# if condition then action else alternative ;;
```

Listing 2.2 – Syntaxe de la condition

R
- La condition doit être un booléen.
- L'action et l'alternative doit être du même type

2.4 Variables

Un définition peut être de plusieurs type :

– Globale
– Locale
– Simultanée

2.4.1 Définition globale

```
# let variable = expression;;
```

Listing 2.3 – Définition de variable

L'interpréteur va évaluer la valeur et donner un type à la variable, il effectue une liaison `<var,val>`, ceci peut aussi s'appeler une fermeture.

On ajoute la liaison à l'environnement, un environnement est donc un ensemble ordonné de liaisons.

2.4.2 Définition Locale

```
# let variable = expression 1
  in expression2 ;;
```

Listing 2.4 – Définition de variable

La définition et temporaire

1. Évaluer l'expression dans l'environnement ourant
2. Ajouter à l'environnement courantla nouvelle. Liaison `var,val1`
3. Évaluer l'expression 2 dans ce nouvel environnement augmenté ⇒ Résultat
4. Restituer environnement de départ

2.4.3 Définitions simultanées

```
# let var1 = expression1
and var2 = expression2
and var3 = expression3;;
```

Listing 2.5 – Définition de variable

2.5 Fonctions

2.5.1 Définition

```
# fun param -> corps;;
# function param -> corps;;
```

Listing 2.6 – Sytaxe d'une définition de fonction

R Il existe une différence entre `fun` et `function`, cette différence sera vu plus tard

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

Listing 2.7 – Sytaxe d'une définition de fonction

2.5.2 Application

2.5.2.1 Valeur d'une fonction dans un environnement Γ

Évaluer une fonction `fun x -> corps` dans Γ nous donne la fermeture suivante $< \Gamma, x, cors >$

2.5.2.2 Application d'une fonction à un argument dans Γ_2

- Évaluer f dans Γ_1
- Évaluer a dans Γ_1 Soit v la valeur de a dans Γ_1
- Soit $< \Gamma, x, corps >$ la valeur de f dans Γ_1
- On « branche x et v » et on évalue le corps de la fonction dans l'environnement où x est lié à v a été ajouté à Γ
- Résultat

```
# let x = 2013;;
val x : int = 2013
# let y = x + 10
  and z = x * 10;;
val y : int = 2013 z = int : 20130
# let f = func x -> x + z + y;;
val f: int -> int = <fun>
# f(y+1);;
```

Listing 2.8 – Exemple d'utilisation de fonctions

2.5.3 Fonction à n paramètres

```
# fun x1 -> fun x2 -> fun x3 -> ... -> fun xn -> corps;;
# fun x1 x2 x3 ... xn -> corps;;
```

Listing 2.9 – Sytaxe d'une définition de fonction à n paramètres